Abstraction for web programming

Jeremy Yallop

Doctor of Philosophy

Laboratory for Foundations of Computer Science

School of Informatics

University of Edinburgh

2010

Abstract

This thesis considers several instances of abstraction that arose in the design and implementation of the web programming language Links. The first concerns user interfaces, specified using HTML forms. We wish to construct forms from existing form fragments without introducing dependencies on the implementation details of those fragments. Surprisingly, many existing web systems do not support this simple scenario. We present a library which captures the essence of form abstraction, and extend it with more practical facilities, such as validation of the HTML a program produces and of the input a user submits.

An important part of our library is a simple semantics, given as the composition of three primitive "idioms", an interface to computation introduced by McBride and Paterson. In order to justify this approach we present a comparison of idioms with the related notions of monads and arrows, refining the informal claims in the literature.

Our library forms part of the Links framework for stateless web interactions. We describe a related aspect of this system, a preprocessor that derives generic instances of functions, which we use to serialise server state between client requests. The abstraction in this case involves the shape of datatypes: the serialisation operation is specified independently of the particular types involved.

Our final instance of abstraction involves abstract types. Functional programming languages typically offer one of two styles of abstract type: the abstraction boundary may be drawn using a private data constructor, or using a type signature. We show that there is a pair of semantics-preserving translations between these two styles. In the light of this, we revisit the decision of the Haskell designers to offer the constructor style, and define a library that supports signature-style definitions in Haskell by translation into the constructor style.

Acknowledgements

I would like to thank my supervisor, Philip Wadler, for encouraging me to come to Edinburgh, and for his support, guidance, and inspiration throughout my time here. His comments on earlier drafts of this thesis led to significant improvements.

I am grateful to my colleagues on the Links project, Sam Lindley and Ezra Cooper, for our enjoyable collaboration.

Discussions with visitors to the group also improved my understanding and led me to revise my views on more than one occasion. I particularly remember the visits of Thierry Martinez, Shriram Krishnamurthi, and Don Syme in this regard.

Conversations with Bob Atkey led to several improvements in the ideas presented here.

My time in Edinburgh has been enriched by the wide variety of talks and seminars on offer. In particular, the Scottish Programming Languages Seminar brought together researchers from across Scotland, both "squigglers" and "bodgers", and the result was invariably stimulating. The pl-read group offered many introductions to interesting work. Thanks to the organisers and participants.

Finally, I would like to thank Claire, Elizabeth, Amy, and Hannah, for unwavering love and support.

Declaration

I declare that this thesis was composed by myself and that the work contained therein is my own, except where explicitly stated otherwise below.

The paragraph describing Links in the introduction (Section 1.5) is taken from joint work with Ezra Cooper, Sam Lindley and Philip Wadler, previously published in the proceedings of Formal Methods for Components and Objects 2006 (Cooper, Lindley, Wadler, and Yallop, 2006).

Section 2.3 is based on joint work with Sam Lindley and Philip Wadler, previously published in the Journal of Functional Programming (Lindley, Wadler, and Yallop, 2010). An earlier version appeared as a technical report (Lindley, Wadler, and Yallop, 2008a).

Section 2.4 is based on joint work with Sam Lindley and Philip Wadler, to appear in the proceedings of Mathematically Structured Functional Programming 2008 (Lindley, Wadler, and Yallop, 2008b).

Chapter 3 is based on joint work with Ezra Cooper, Sam Lindley and Philip Wadler, previously published in the proceedings ofthe Asian Symposium on Programming Languages and Systems 2008 (Cooper, Lindley, Wadler, and Yallop, 2008a). An earlier version appeared as a technical report. (Cooper, Lindley, Wadler, and Yallop, 2008b).

Appendix B.2 is based on ideas from Sam Lindley's 2008 ML Workshop paper (Lindley, 2008); the presentation given here is original. It is included for completeness, since it is closely connected with the work described in Chapter 3.

(*Jeremy Yallop*)

Table of Contents

List of Figures

Chapter 1

Introduction

1.1 Abstraction

Abstraction makes programs more flexible by imposing a rigid separation between use and
definition. At the value level, lambda abstraction separates out the values used in an expression.
At the type level, an abstract type definition separates the parts of a program that make use of
a type from the definition of the representation of that type.

In each case we bind a concept (such as "a date", or "the type of dates") to a name (such
as `d`, or `Date`), and then use that name, rather than the definition of the concept, within the
remainder of the program. Abstraction isolates each definition within a single portion of the
program; if we later change our mind about the details of the definition then the changes to our
program will be confined to that portion.

Form abstraction

Can we apply these principles of abstraction to web programming? Let us briefly review the
interface for interaction on the web. A web page specified using the HTML markup language
may contain one or more *forms*. Each of these forms contains various *controls* (buttons, menus,
text input boxes, and so on), each of which has a name that is unique within the form. For
example, here is the HTML for a form that allows the user to enter arrival and departure dates
for booking a hotel:

1

```
<form action="book" method="post">
 <div>
  Arrival:<br/>
  <div>Month: <input name="m1"/>Day: <input name="d1"/></div>
  Departure:<br/>
  <div>Month: <input name="m2"/>Day: <input name="d2"/></div>
  <button type="submit">Book</button>
 </div>
</form>
```

A browser might render the form like this:

The user edits the values in the form, then submits the form to a server for processing; the values of the controls are sent as a sequence of pairs of names and values. If the user wishes to book a room for the first week of December then the body submission will contain the following lines:

```
m1=12
d1=1
m2=12
d2=8
```

Web programs, then, have to deal with each form on two occasions: when constructing HTML, and when processing the submitted values. There is therefore a need to ensure that the names used for controls when constructing the form correspond to the names expected by the part of the program that processes the submission, and indeed a number of existing web frameworks ensure that this need is met (Christensen, Møller, and Schwartzbach, 2003, Thiemann, 2005, Plasmeijer and Achten, 2006).

Proper abstraction of form controls requires more than this simple guarantee of correspondence: we must be able to separate the parts of the program which combine controls from the descriptions of any particular controls. Perhaps surprisingly, most existing web frameworks do not address this need.

To demonstrate, let us suppose that we have added a syntax for HTML literals to the Standard ML programming language; we will use braces within a literal to enclose an expression whose value should be inserted at that point. The control for entering a date contains two input fields, for the month and day components. The date control appears twice within the booking

form, but with distinct names for the fields in each case. We therefore define the date control as a function which accepts field names as arguments and returns the HTML for the control.

```
fun date m d =
  <div>
    Month: <input name="{m}"/> Day:<input name="{d}"/>
  </div>
```

The `date` control can now be used multiple times within a larger control, so long as fresh names are supplied as arguments each time. Of course, to make the larger control reusable in the same way we must abstract these names again. Here is part of our booking control again, built from two instances of the `date` control above:

```
fun date_range m1 d1 m2 d2 =
  <div>
    Arrival:<br/>   {date m1 d1}
    Departure:<br/> {date m2 d2}
    <button type="submit">Book</button>
  </div>
```

This passing around of names is rather inconvenient, not least since it is the responsibility of the caller to avoid name clashes. However, there is a more serious problem. Suppose that we wish to change the definition of the booking control to contain a single field for entering a free form date.

```
fun date d =
  <div>
    <input name="{d}"/>
  </div>
```

We now have to change the definition not only of `date`, but of `date_range`, and every other place in the program where the `date` control is used. We call this breakdown in modularity the *form abstraction* problem: the "clients" of a definition are written in a way that depends upon the internal details of that definition. The abstraction leaks.

1.2 Effects

Our solution to the form abstraction problem involves a domain-specific language for describing forms; we embed this language into a "host" functional programming language. The functional programming style discourages the use of side effects, but form processing naturally involves certain effects: for example, we must generate unique names for fields during form generation and look up values in the environment during processing of a submission. As we

have seen, it is possible to avoid these effects by threading state (such as the names needed by a component) throughout the program, but this can lead to an awkward programming style and a loss of abstraction.

There are several approaches to resolving the tension between the convenience of effects and the benefits of purity. An approach that has proven fruitful in Haskell (Peyton Jones and Hughes, 1999) is to shift the focus from *executing* effectful computations to *constructing* computations that can be executed later. Using this approach, computations are reified as regular values, and sequencing of computations is simply composition of values. It is this approach that we shall use in developing a solution to the form abstraction problem.

Monads (Moggi, 1989, Wadler, 1990) are the most popular and most powerful example of the computations-as-values approach. The monad interface is used for the standard I/O interface in Haskell, besides many other "notions of computation" ranging from parsers (Hutton and Meijer, 1998) to database queries (Leijen and Meijer, 1999). The interface provides great flexibility in constructing and sequencing computations; in particular, it enables higher-order programming, in which computations can be constructed and executed "on-the-fly". However, the power comes with a price: every implementation of the monad interface must offer the same flexibility. When the underlying notion of computation does not support the required degree of flexibility the monad interface cannot be used.

Two alternatives to the monad interface have been suggested. Hughes (2000) introduced *arrows* as a kind of first-order variant of monads. McBride and Paterson (2008) introduced *idioms* (also known as *applicative functors*) as an interface for writing effectful computations in an applicative style; as with arrows, the idiomatic interface is less powerful than the monadic. In order to choose the most suitable interface for defining composable form fragments we will make a careful investigation and comparison of idioms, monads and arrows.

1.3 Serialising continuations

Let us return to the HTML form example of Section 1.1:

```
<form action="book" method="post">
    ...
</form>
```

The `action` attribute of the `form` element is a URL indicating the program that will process the form submission. In our example the URL is relative, and specifies a program called `book`.

Indicating the program that will process the form submission in this way leads to a dis-

Figure 1.1: Constructing and processing a form
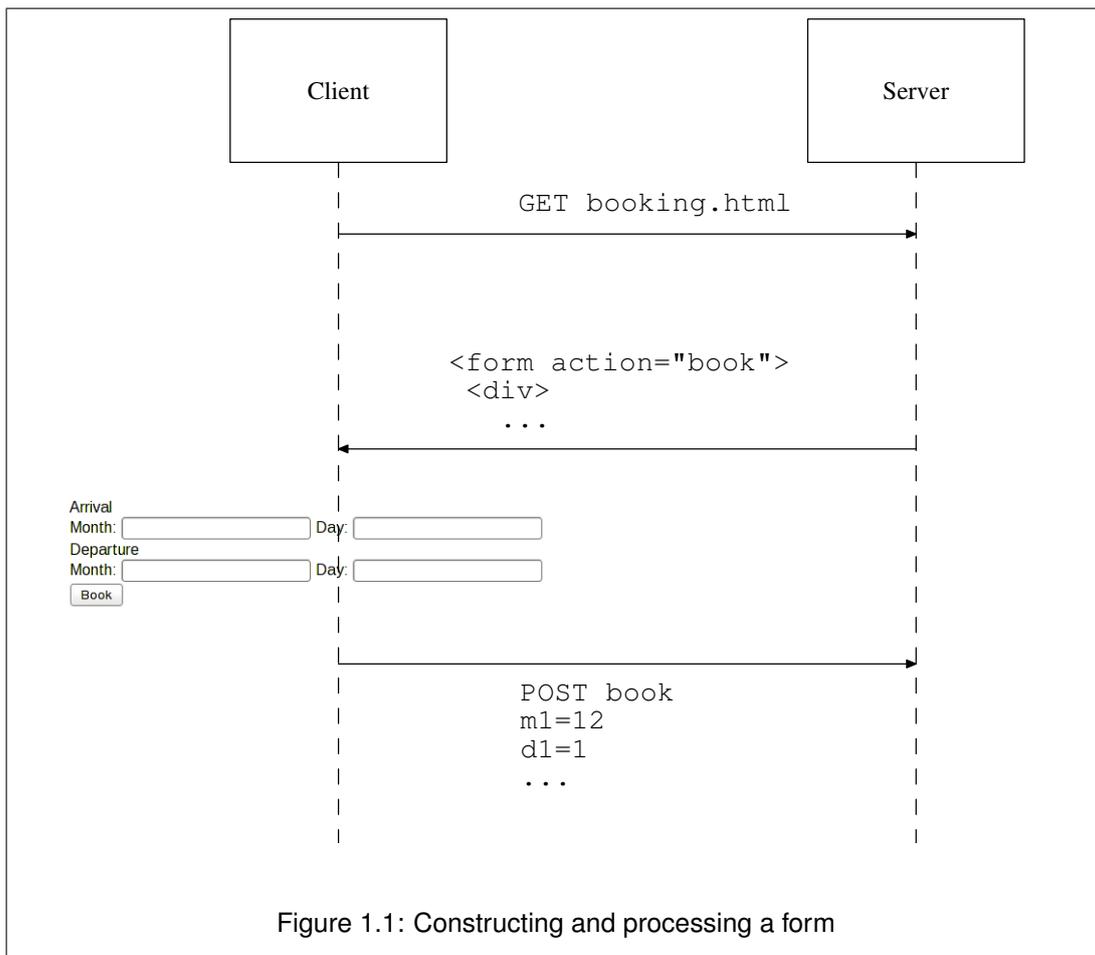
tinctive structure for web programs that is similar to the *continuation-passing style* familiar to functional programmers. Programs written in continuation-passing style pass an extra parameter, the *continuation*, to each function. When the function's work is finished, rather than returning a result to the calling context (as a program written in the regular, *direct* style does), it calls this continuation, passing the result as an argument. Similarly, a web program constructs a form, passing the name of a "continuation" program as the `action` attribute, then relinquishes control to the user. The user edits the fields, then "calls" the continuation program by submitting the form (Figure 1.1).

A significant difference between the continuation-passing style used in functional programs and the structure of web programs is that in the former continuations are often denoted by nested function expressions, whereas the latter requires that every "continuation" to a form is a program named at top level. This constraint is similar to the distinguishing feature of a second class of programs, those in *lambda-lifted* form (Johnsson, 1985). The lambda-lifting transformation replaces each nested function expression with a top-level function, and each free vari-

```
   fun book env =
    let () = register_booking
               (read_date "m1" "d1" env) (read_date "m2" "d2" env)
      in send_page <div>Booking confirmed!</div>

   val entry_points = [ ("book", book), ...]

   send_page
    (<form action="book">
       <div>Month: <input name="m1"/>
            Day: <input name="d1"/></div>
       <div>Month: <input name="m2"/>
            Day: <input name="d2"/></div>
       <button type="submit">Book</button>
     </form>)
```

Figure 1.2: A program in "web style"

able in the original expression with a parameter to the function. We refer to the continuation-passing lambda-lifted style used in web programs as "web style".

Both the continuation-passing style and the lambda-lifted style have interesting theoretical properties, but they are inconvenient for programming; they are more suitable for use in the internals of compilers than as source languages (Peyton Jones, 1986, Appel, 2007). Consequently, Queinnec (2000) and Graunke, Krishnamurthi, Hoeven, and Felleisen (2001b) advocate using a language with first-class continuations, such as Scheme (R. Kelsey, 1998), for writing web programs in a more direct style. A program written using Graunke et al.'s system uses a special procedure during form construction, `send/suspend`, which generates a string referring to its own continuation; this string is used as the `action` attribute of the form. When the form is submitted this reference is resolved, and the continuation is invoked with the form values entered by the user. To the programmer it appears that the form values are "returned" from `send/suspend`, in contrast to the regular style of web program, where they are passed as arguments to the entry point named in the `action` attribute.

Figures 1.2 and 1.3 illustrate the two styles (using Standard ML rather than Scheme, for continuity with the other examples in this chapter). The first program (Figure 1.2) is written in "web style". It creates and displays a form which names the function `book` as the `action`; we assume a function `send_page` that wraps the form in suitable boilerplate HTML to form a complete page, sends the page the client and exits the program. The `book` function accepts an environment containing the submitted field values, which it extracts using a function `read_date`. After registering the booking (in a database, say), it sends a confirmation page

```
   let env =
        (send_suspend
          (fn k_url ⇒
            (make_page
               <form action="{k_url}">
                 <div>Month: <input name="m1"/>
                       Day: <input name="d1"/></div>
                 <div>Month: <input name="m2"/>
                       Day: <input name="d2"/></div>
                 <button type="submit">Book</button>
               </form>)))
   in
    let () = register_booking
                (read_date "m1" "d1" env) (read_date "m2" "d2" env)
        in send_page <div>Booking confirmed!</div>
   end
```

Figure 1.3: A "direct style" version of Figure 1.2

to the client. The `entry_points` variable is bound to a table which maps each name used as the `action` of a form to the corresponding continuation function. (We do not show the code which performs this resolution.) The second program (Figure 1.3) uses `send_suspend` to generate a value for the `action` of the form and binds it to `k_url`. (We assume a function `make_page` which wraps the form in boilerplate HTML to form a complete page.) When the form is submitted this action is resolved to the continuation of the call to `send_suspend`, which is invoked with the submitted environment. This environment is bound to `env`, and the program resumes execution at the point where the call to `send_suspend` returns; at this point the program behaves like the `book` function of Figure 1.2, extracting the dates and passing the results the `register_booking`.

The direct style of Figure 1.3 uses the continuation of the program that generates a response as the entry point to the program that generates the next response. The question therefore arises as to where to store this continuation between requests. One approach, taken by Graunke et al. (2001b), Graham (1997), and others, is to store the continuation in a persistent table on the server between requests, treating the string used as the `action` as an index into this table. This implementation has the advantages of simplicity and efficiency: the index is small and simple compared to the continuation, and multiple continuations involving the same data can share structure, since they are stored in the same server. However, there are also serious shortcomings, most notably the difficulty in reclaiming the storage used by continuations. There are two potential strategies for reclaiming storage: either the continuations in the table are stored

indefinitely, in which case storage requirements on the server increase with every form sent to the client, or the continuations are removed from the table, invalidating forms previously sent to the client (which may be still open, or bookmarked for later use). Neither of these options is acceptable for a scalable, reliable web application.

Graunke, Findler, Krishnamurthi, and Felleisen (2001a) describe an alternative approach to writing direct-style web programs which requires neither first-class continuations in the source language, nor arbitrarily large storage capacity on the server. Starting with a source program written in the direct style of Figure 1.3, Graunke et al.'s (2001a) system applies a sequence of three transformations to obtain a program in web style, as in Figure 1.2. The first transformation, into continuation-passing style, reifies each resumption point in the program as a function. The second transformation, lambda-lifting, moves these continuations to top-level. The final transformation, defunctionalisation (Reynolds, 1972), gives higher-order values a first-order representation; it performs a function roughly analogous to that of the `entry_points` table in Figure 1.2. The purpose of the defunctionalisation step is to make it possible to *serialise* continuations — that is, to translate them into a format that can be stored outside the program, from which they can be recovered at a future point. Now the serialised continuations can be incorporated into forms — either as the `action`, or as hidden fields — and stored on the client, avoiding the problems with lifetime management which arise when continuations are stored on the server. Halls (1997) was perhaps the earliest proponent of this approach. As Halls notes, passing continuations between server and client in this way requires additional measures to avoid security problems; a naive approach risks inadvertently exposing secret data, or executing continuations constructed by a malicious client.

1.4   Abstract types

Up to this point we have been concerned with a proper separation between definition and use at the value level. Abstract type definitions enforce a similar separation at the type level, dividing a program into the region that defines an abstract type in terms of some existing representation type, and the region that uses the abstract type through an interface without making use the of its representation.

There are two common styles of abstract type. The first hides the definition behind an interface, using type signatures to conceal the representation. In Standard ML this is achieved using a module signature, as in the following simple definition for an abstract `date` type:

```
structure Date =
struct
  type date = { month : int, day : int }
  fun mkDate m d = { month = m, day = d }
  fun toString (d : date) = Int.toString (#day d) ^ "/"
                              ^ Int.toString (#month d)
end :>
sig
  type date
  val mkDate : int → int → date
  val toString : date → string
end
```

Now the functions `mkDate` and `toString` and the type constructor `date` are available for use in the remainder of the program, but the module signature conceals the representation of `date` from code outside the module, so that attempts to access the `month` and `day` fields are rejected. As a result, we can be sure that it is safe to change the type used as the representation of `date` without affecting the rest of the program.

With the second style of abstract type definition there is no need to write type signatures. Instead, a private data constructor conceals the representation from the rest of the program. This style exploits the scoping of identifiers to delimit that the constructor is only available within the code that defines the abstract type, and so cannot be used to access the representation type elsewhere in the program. Here is a definition of the `date` type in this style.

```
abstype
  date = Date of { month : int, day : int }
with
  fun mkDate m d = Date { month = m, day = d }
  fun toString (Date d) = Int.toString (#day d) ^ "/"
                              ^ Int.toString (#month d)
end
```

As before, the functions `mkDate` and `toString` are visible to the rest of the program. However, the data constructor `Date` is not: there is no way to access values of type `date` elsewhere in the program except via the two functions in the interface.

1.5 Links

The ideas presented in this dissertation were developed in the context of Links.

Links (Cooper et al., 2006) is a programming language for web applications that generates code for all three tiers of a web application from a single source, compiling into JavaScript to run on the client and into SQL to run on the database. Links supports rich clients running

in what has been dubbed *'Ajax'* style, and supports concurrent processes with statically-typed message passing. Links follows the scalable approach to form continuations, preserving session state in the client rather than the server. Client-side concurrency in JavaScript and transfer of computation between client and server are both supported by translation into continuation-passing style.

However, the contributions of this dissertation are not tied to Links. We will use a variety of functional programming languages to present our ideas, taking advantage of particular features of those languages. The investigation of idioms, arrows and monads in Chapter 2 benefits from Haskell's *type classes* (Hall, Hammond, Peyton Jones, and Wadler, 1996), and particularly from Jones's (1993) *constructor classes*. The metaprogramming facilities provided by Template Haskell (Sheard and Peyton Jones, 2002) are helpful in implementing a translation between the two styles of abstract type definition (Section 5.5). We necessarily use OCaml (Leroy, 2008), the language used for the Links implementation, in describing the techniques used in Links for serialising continuations in Chapter 4. OCaml's advanced module system and facility for syntactic extension are also helpful for presenting our new constructs for programming with form fragments in Chapter 3. Finally, for our more formal investigations in Chapters 2 and 5 we use variants of the lambda calculus and polymorphic lambda calculus.

1.6   Contributions

The key contributions of this dissertation are:

- A study of the idiom, arrow, and monad interfaces, including

    - A proof of completeness for a variant of Paterson's arrow notation

    - Presentation of the three interfaces as variations on a single calculus

    - An ordering of the three interfaces by expressive power

    - Transformers for each interface that convert computations to normal form

- A distilling of the form abstraction problem to its essence using composition of three primitive idioms, an argument that the idiom interface is a good match for our solution, and an exploration of various extensions.

- An approach to generating generic function instances in OCaml using the Camlp4 pre-processor and the correspondence between ML modules and Haskell type classes.

idioms   monads

static arrows   higher-order arrows

$(A \rightsquigarrow B \simeq 1 \rightsquigarrow (A \rightarrow B))$ $\longrightarrow$ arrows $\longrightarrow$ $(A \rightsquigarrow B \simeq A \rightarrow (1 \rightsquigarrow B))$

Figure 1.4: Idioms, arrows and monads
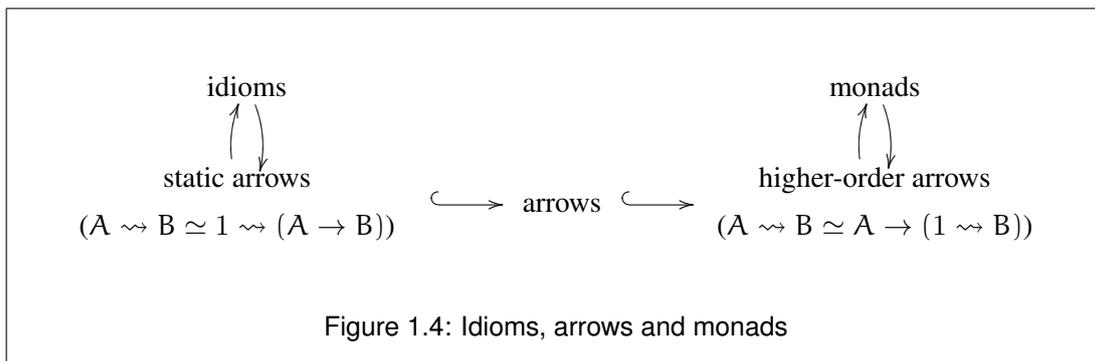
- A proof that the two common styles of abstract type definition are equivalent and inter-convertible; a library that uses the equivalence to extend Haskell with signature-based abstract type definitions.

1.7 Roadmap

Chapter 2 (*Three models for the description of computation*) sets up the theoretical background for the investigation of form abstraction in Chapter 3.

We explore the connection between Wadler's (1990) *monads*, Hughes's (2000) *arrows* and McBride and Paterson's (2008) *idioms* (also known as *applicative functors*). Existing work shows how arrows and monads are related, but there has been some confusion over the relation of idioms to the other two interfaces. McBride and Paterson (2008), introducing idioms, describe them informally as

> an abstract notion of effectful computation lying between Arrow and Monad in strength

but we will show that idioms should actually be ranked as the least powerful of the three interfaces.

Our approach is to use the *arrow calculus*, a variation on a notation introduced by Paterson (2001) that we show to be complete. We show that idioms and monads correspond to static and higher-order variations on this calculus and show that static arrows embed into arrows and that arrows embed into higher-order arrows, thus establishing the correct ordering of expressive power (Figure 1.4).

Each interface is accompanied by laws which equate certain computations; we show how to use these laws to construct normalizing transformers for idioms, arrows and monads in Haskell.

Chapter 3 (*Abstracting controls*) describes *formlets*, a solution to the form abstraction problem introduced in Section 1.1. Formlets allow true abstraction over form components, freeing the programmer from the need to worry about name collisions or type mismatch problems.

We argue that formlets are most naturally viewed as an idiom (rather than as an arrow or monad) and give a semantics based on the composition of the three standard idioms that capture the effects necessary for programming with forms. We also give a formal definition of the formlet syntactic sugar and an implementation in OCaml.

Although formlets capture only the essence of form abstraction it is easy to extend the basic definition with additional features. We describe extensions for statically checking the validity of generated XHTML and dynamically checking that user input meets specified constraints. We briefly discuss a more efficient desugaring translation.

Chapter 4 (*Serialising continuations*) describes *deriving*, an extension to OCaml for deriving generic function instances that we use to provide an essential part of the formlets implementation in Links: the serialisation of continuations.

The design of *deriving* is inspired by the Haskell keyword of the same name. As in Haskell, the programmer may list the names of classes after a type declaration to request that the implementation generate instances of those classes for the declared type. OCaml does not have type classes, but there is a well-known correspondence between modules and type classes that we use to guide the design of *deriving*.

One advantage of our approach is that, in contrast to the more common approach based on combinators, almost no effort is required on the part of the programmer in the common case where the generated instance is adequate; however, it is also straightforward to provide customised instances for particular types which integrate smoothly with the generated code.

Chapter 5 (*Signed and Sealed*) formally compares the two styles of abstract type definition, as introduced in Section 1.4, in which the representation type is concealed from the rest of the program using either a type signature or a private data constructor. We add constructs for both styles to a partial polymorphic lambda calculus introduced by Pitts (2000) and give a parametricity-based proof that the two are equivalent, together with an automatic translation from each style into the other.

In part, our motivation comes from addressing a problem encountered by the designers of Haskell: it was unclear how to combine the signing style of abstract type definition with unambiguous overloading (Hudak, Hughes, Peyton Jones, and Wadler, 2007). We show that this is possible by translating the signing style into the sealing style used by Haskell, and

describe a Template Haskell library that performs this translation. We demonstrate by examples that the signing style can lead to more elegant Haskell programs, removing the clutter of the constructors and destructors introduced by sealing.

Chapter 6 concludes.

Chapter 2

Three models for the description of computation

2.1 Introduction

The Internet Robustness Principle (RFC 793) states

> *Be conservative in what you do; be liberal in what you accept from others.*

In other words, robust systems make the weakest possible assumptions about input and give the strongest possible guarantees about output. Programs that accept only integers are less flexible than programs that accept all kinds of number. Contrariwise, programs that may output any kind of number are less flexible than programs that are guaranteed to output only integers.

To follow the principle we need to know which sets of values generalise which other sets. While there are certainly more numbers than integers, the ordering is not so obvious at higher-order types, such as function and computation types. Can a program that manipulates *arrow* computations be made more flexible by specifying that the input must be an *idiom* rather than an *arrow*? Can a library that exposes an *idiom* instance be made more flexible by exposing an *arrow* instance instead? In his original work on arrows, Hughes shows how each monad gives rise to an arrow, and gives an extended arrow interface, *ArrowApp*, that is equivalent to the monad interface[1] (Hughes, 2000). In their later work introducing idioms, McBride and Paterson show how to obtain an idiom from either a monad or an arrow, and how to combine an idiom and an arrow to yield another arrow (McBride and Paterson, 2008). However, the precise

---

[1] The title is a nod to Chomsky's seminal work on the relative power of three classes of formal language (Chomsky, 1956), but we shall use "interface" rather than "model" to refer to the concepts "monad", "arrow" and "idiom" in the remainder of this chapter.

Figure 2.1: Relating idioms, arrows and monads.
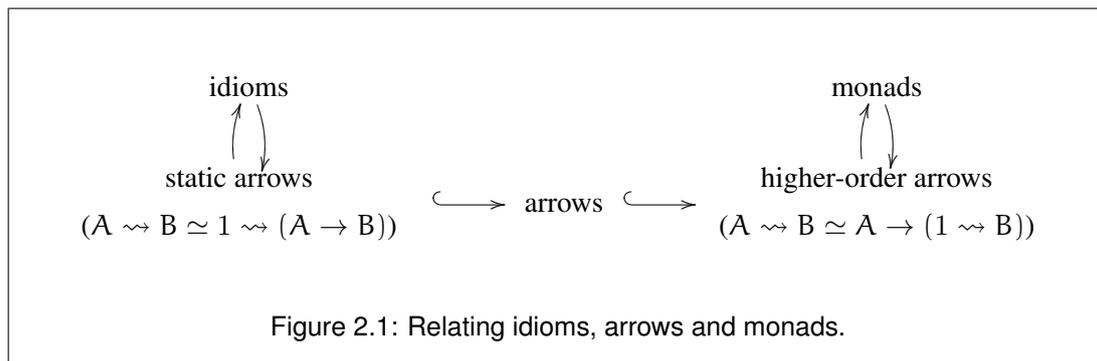
relationship between the three notions of computation has remained obscure. In particular, McBride and Paterson informally describe idioms as

> an abstract notion of effectful computation lying between Arrow and Monad in strength

whereas we show in the following pages that idioms are, in fact, weaker than both arrows and monads. The diagram in Figure 2.1 gives a high-level view of the situation: idioms correspond to a language which may be extended to obtain arrows; a further extension yields a language corresponding to monads. (As we shall see, the second of these extensions is based on a straightforward syntax inclusion, while the first involves a more subtle translation.)

In this chapter we provide first an informal and then a formal comparison between the three interfaces. This will serve as a solid basis both for the design of formlets (Chapter 3), and for resolving the similar questions that arise every time one wishes to embed a language which involves effects.

In Section 2.2 we begin our investigation with an informal comparison of the interfaces for programming with idioms, arrows and monads in Haskell. These interfaces take the form of mutually unrelated type classes whose methods are governed by somewhat ad-hoc laws. We give a number of examples that reveal the relative flexibility of the various type classes, and transformers for each class that put computations into normal form. These normal forms show clear differences in the expressive power of each interface.

Section 2.3 introduces a new presentation in which idioms, arrows and monads are simple variations on a common calculus, the first step on the road to formally establishing the order of the strength. This calculus, which we call *arrow calculus*, is closely related to a notation introduced by Paterson (2001). However, while Paterson's notation is a convenient abbreviation for arrow computations, we show that the arrow calculus is in fact complete: we are justified in using it as a full substitute for the classic formulation. The isomorphism between the two formulation takes the form of an *equational correspondence* (Sabry and Felleisen, 1993), in which the composition of the translations between the two is the identity, and in which the

laws of each follow from the laws of the other. The proof of the correspondence reveals that one of the laws in the standard presentation of arrows is redundant.

Section 2.4 extends the arrow calculus in two ways: first, by introducing either an additional operator or a type isomorphism to bring it into correspondence with idioms; next, by introducing either an additional operator or a type isomorphism to bring it into correspondence with monads. (These correspondences are weaker than the equational correspondence between arrow calculus and classic arrows; in order to characterise them we introduce the notion of *equational equivalence*.) The relationships that emerge invalidate the claim by McBride and Paterson quoted above. The extension of arrow calculus to support higher-order (i.e. monadic) computation also reveals that one of the laws in the standard presentation of higher-order arrows is redundant.

2.2   Arrows, idioms and monads

We begin by introducing monads, arrows and idioms as they are used in Haskell. Each interface makes computations available as first-class values within the language and provides operators for constructing and composing computations. Haskell provides a principled form of overloading, the *type class* (Hall et al., 1996), which is a good fit for programming with monads, arrows and idioms.

2.2.1   Monads

Wadler (1990) introduced *monads* to the functional programming community, drawing on work by Moggi (1991) on the semantics of effectful programs. Using monads, computations are presented as first-class values, with combinators for constructing and composing computations. Monads have proved remarkably successful as a program structuring technique: they are used, for example, as the sole means of constructing programs which perform I/O in Haskell (Peyton Jones and Hughes, 1999).

2.2.1.1   Monad operators

The Monad type class provides a common interface for various notions of effectful computation. Each instance of the class consists of a unary type constructor, m, for representing computations, and two computation-forming operations, return and $\gg\!\!=$ (pronounced "bind"). The

concrete definition of the Monad class is as follows[2] [3]:

```
class Monad (m :: * → *) where
    return :: α → m α
    (≫=)  :: m α → (α → m β) → m β
```

A value of type m τ, where m is a monad, represents a computation that performs some effects and then returns a result of type τ. The first operation of the class, return, accepts a value, v, and creates a trivial computation that simply returns v, performing no effects. The second operation, ≫=, creates a computation from two values: an existing computation, m and a continuation that accepts the result of m and creates a new computation.

2.2.1.2   Monad examples

The Monad type class defines an abstract *interface* to computation. In order to write programs which involve monadic computations we must provide instances, or *implementations* of the class; that is, we must instantiate the parameter m with a type constructor and supply corresponding definitions for the return and ≫= functions.

The monad interface encompasses a wide variety of notions of computation (Moggi, 1991). We will focus here on a single representative example: the addition of *state* to a pure language. (Further examples will arise in the development of *formlets* in Chapter 3.) We model computations involving updatable state as functions from an initial state to a result and an output state.

```
newtype State σ α = State { runState :: σ → (α, σ) }
```

Here σ is the type of the state in a computation, and α is the result type. Then State σ is an instance of the Monad class for any particular type σ:

```
instance Monad (State σ) where
    return v = State (λs → (v, s))
    State m ≫= k = State (λs → let (a, s') = m s
                                 in runState (k a) s')
```

The return function creates a computation which uses the unmodified input state as the output state. The ≫= function creates a computation whose input is passed to the computation m; the result and output state of m are passed to the continuation k to give the result and output state

---

[2] The Haskell 98 definition of Monad includes two further operators. The first, ≫, may be defined in terms of the ≫= operator. The second, fail, determines the action to take when pattern matching fails, and does not concern us here.

[3] In Haskell type expressions are classified by *kinds*, which are either of the form *, the kind of types, or $\kappa_1 \to \kappa_2$, the kind of type-level functions from kind κ_1 to kind κ_2. As an aid to comprehension we will include kind signatures for all type parameters that denote kinds other than *. (Kind signatures are an extension of the GHC Haskell implementation.) The use of Roman letters also distinguishes these *higher-kinded* type variables from type variables of kind *, for which we use Greek letters.

of the whole computation.

Most instances of `Monad` come with extra functions for creating computations of a particular type. For stateful computations it is useful to have computations that retrieve and set the state:

```
get :: ()  →  State σ σ
get () = State (λs → (s,s))

put :: σ  →  State σ ()
put s = State (λ_ → ((), s))
```

Using these constants and the operations of the `Monad` interface it is possible to express a wide range of computations.

Example 1 (sequencing).   We can *sequence* computations and process the results. For example, we can construct a computation that executes two computations `m` and `n` in order and returns the results as a pair[4]:

$$m \ \ggeq \ \lambda \mathtt{x} \ \rightarrow$$
$$n \ \ggeq \ \lambda \mathtt{y} \ \rightarrow$$
```
return (x,y)
```

Example 2 (dataflow).   We can use the result of a computation to influence *dataflow* in subsequent computations, using the result of one computation to compute the input to another. For example, we can construct a computation of type `State Bool ()` that negates the current state, setting a `True` state to `False` and a `False` state to `True`:

```
get () ⋙ λs →
put (not s)
```

Example 3 (control flow).   We can also use the result of a computation to determine *control flow* in subsequent computations. For example, we can write a second computation of type `State Bool ()` that decides whether to execute a computation m depending on whether the current state is `True` or `False` (i.e., depending on the result of running the computation `get`):

```
get () ⋙ λs →
if s then m else return ()
```

Example 4 (higher-order).   It is also possible to write *higher-order* computations. For example, we might use computations themselves as the result type, treating computation types

---

[4]The unconventional layout is intended to evoke assignment and sequencing in an imperative language: first set `x` to the result of `m`, then execute the next action with `x` in scope, and so on.

of the form

```
State σ (State σ τ)
```

Given a computation `m` of this type we can write a computation of type `State σ τ` that executes
the result of `m`:

```
m  ≫=  λn →
n
```

(This computation corresponds to the *multiplication* of the monad.)

As we shall see in Sections 2.2.2 and 2.2.3, the other computational interfaces — `Arrow`
and `Idiom` — are only sufficiently flexible to encode some, not all, of these computations.

2.2.1.3 Monad laws

We saw in 2.2.1.1 that the `Monad` class contains a type constructor and two operations. This
is not the whole story, however: a further aspect of monads is that each instance must satisfy
certain laws. These laws are not part of the Haskell specification of the class, and it is not
possible in general for a Haskell implementation to check that they are satisfied; instead, they
are left to a "gentlemen's agreement" wherein the author of an instance of `Monad` promises to
check that the laws are satisfied for that instance.

The three monad laws say that `return` is a kind of left and right unit for ≫= and that ≫=
is associative.

$$
\begin{aligned}
\texttt{return}\ v\ \ggg\!=\ f\ \ &\equiv\ \ f\ v \\
m\ \ggg\!=\ \texttt{return}\ \ &\equiv\ \ m \\
(m\ \ggg\!=\ j)\ \ggg\!=\ k\ \ &\equiv\ \ m\ \ggg\!=\ (\lambda\texttt{a} \to j\ \texttt{a}\ \ggg\!=\ k)
\end{aligned}
$$

It is easy to check that the `State` instance satisfies these laws. For example, we can demon-
strate that the first law holds as follows:

```
     return v ≫= f
=     (definition of return, ≫=)
     State (λs → let (a,s') = (λs → (v,s)) s
                 in runState (f a) s')
=     (β)
     State (λs → runState (f v) s)
=     (η)
     State (runState (f v))
=     (State (runState v) ≡ v)
     f v
```

It is similarly straightforward to demonstrate that the other monad laws hold for `State`.

2.2.2 Arrows

The examples in Section 2.2.1.2 illustrate the flexibility offered by the `Monad` interface. This flexibility is a boon when writing programs which use the `Monad` interface, but can become a burden when writing instances of the class.

Hughes (2000) gives a striking example of this phenomenon. *Parser combinators* embed parsers in functional languages as regular values (Wadler, 1985), commonly casting them as monads (Hutton and Meijer, 1998). Swierstra and Duponcheel (1996) present a variant of the parser combinator technique in which matching a string against a grammar is split into two phases. The first phase analyses the grammar specification to construct an efficient parser; the second phase uses this parser to convert the input string into a structured value. However, this two-phase implementation approach cannot be used for computations constructed using the `Monad` interface, since the $\gg\!\!=$ operator allows the user to delay construction of the grammar specification until part of the input string has been processed.

Hughes' response is to introduce *arrows*, an interface to *first-order* computation. The `Arrow` type class offers an interface to users that is more restrictive than `Monad`: in particular, arrows provide no way to construct a computation based on the result of an earlier computation. The benefit of relinquishing flexibility in the interface is that there is more freedom when writing `Arrow` instances: for example, Swierstra and Duponcheel's parsers, which cannot be implemented as monads, can be implemented as arrows.

2.2.2.1 Arrow operators

The `Arrow` type class, like `Monad`, provides an interface to effectful computation. Each instance of `Arrow` consists of a binary type constructor, $\rightsquigarrow$, for representing computations, and three computation-forming operations, `arr`, $\ggg$ (pronounced "compose") and `first`. The concrete definition of the `Arrow` class is as follows:

```
class Arrow ((⤳) :: ⋆ → ⋆ → ⋆) where
    arr   :: (α → β) → (α ⤳ β)
    (⫸) :: (α ⤳ β) → (β ⤳ γ) → (α ⤳ γ)
    first :: (α ⤳ β) → ((α,γ) ⤳ (β,γ))
```

For each instance $\rightsquigarrow$ of `Arrow`, a value of type $\sigma \rightsquigarrow \tau$ represents a computation that expects an input of type σ and, after performing some effects, returns a result of type τ. The fact that arrow computations have both an input and an output leads to an appealing graphical presentation, used in work by Paterson (2001) and others. The first operation of the `Arrow` class, `arr`, is

analogous to the monadic `return`: `arr` creates an arrow computation from a pure function. In the graphical presentation pure computations are denoted using circles:

$$\text{arr } f$$
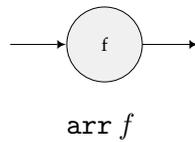
The second operation of the `Arrow` class, ⋙, constructs a new computation from two existing computations, using the result of the first as the input of the second.

$$f \ggg g$$
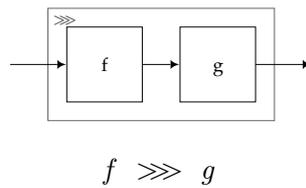
The final operation of the `Arrow` class makes it possible to pass values from one computation to another, besides those values which are used as input to the computation.

$$\text{first } f$$
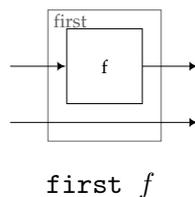
It is instructive to compare the ⋙ operator for arrows with the ⋙= operator for monads. In both cases the result of the left operand is passed as input to the right, but whereas the right operand of ⋙= *constructs* a computation, the right operand of ⋙ *is* a computation. Thus ⋙ is (for the user of the `Arrow` class) less flexible than ⋙=, because the result of the left operand cannot play a role in constructing the computation passed as the right operand.

The `first` operator for arrows fulfils a need that for monads is also met by ⋙=: passing through values that are not used by intermediate computations. For example, the computation m in Example 1 (sequencing) returns a value that is not used by the next computation, n, but is used in the final computation. The value is bound to the variable x by the λ-abstraction used as the continuation to m, and so the normal rules of lexical scoping bring x into scope in the remainder of the computation. In contrast, inputs to arrow computation are not λ-bound, so an additional operator is needed to explicitly pass through values that are needed in subsequent computations. (Example 5 on page 25 shows how to create an analogue of Example 1 (sequencing) using the arrow combinators.)

Several additional arrow combinators will be useful in what follows. These functions are not part of the `Arrow` class since it is possible to express them in terms of the three class

operations. The first is `second`, a cognate to `first`, but with the difference that the additional value is passed using the first element of the pairs in the input and output, not the second.

second f $\qquad\qquad$ arr swap $\ggg$ first f $\ggg$ arr swap
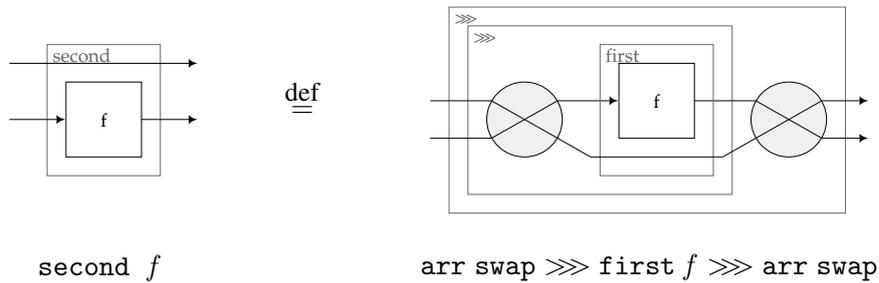
Here `swap` is defined as the function that exchanges the elements of a pair:

```
swap (x,y) = (y,x)
```

The second combinator, $\lll$, builds a computation that transforms pairs from two computations, passing the first element of the pair to the first computation and the second element of the pair to the second computation.

f $\lll$ g $\qquad\qquad$ first f $\ggg$ second g
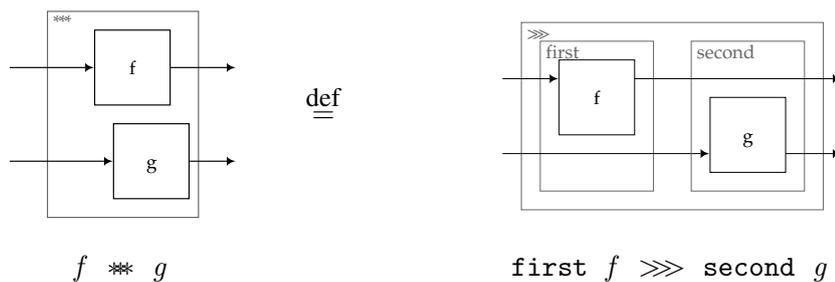
Note that the sequencing of effects places f before g. This ordering is reflected in the relative horizontal placement of the boxes in the diagram; we use the same convention throughout this chapter.

The final combinator, $\lll\!\!\&$, is similar to the $\lll$ combinator, but passes the same value as input to both computations:

f $\lll\!\!\&$ g $\qquad\qquad$ arr dup $\ggg$ (f $\lll$ g)
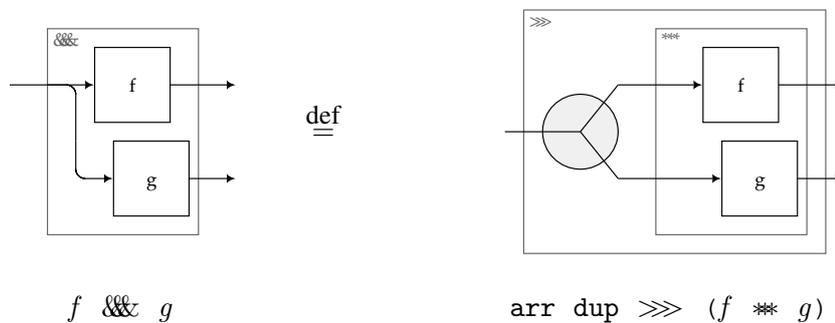
Here `dup` is defined as the function that takes a single value into the two elements of a pair:

```
dup x = (x,x)
```

2.2.2.2   Arrow examples

The `Arrow` interface has been used for a wide range of applications, including parsers and printers (Jansson and Jeuring, 1999), web interaction (Hughes, 2000), circuits (Paterson, 2001), graphic user interfaces (Courtney and Elliott, 2001), and robotics (Hudak, Courtney, Nilsson, and Peterson, 2003). We restrict our attention here to three simple instances. The first instance is the "identity" arrow of pure functions, where `arr` is the identity function, $\ggg$ is reverse function composition, and `first` applies its argument to the first element of a pair.

```
instance Arrow → where
  arr     = id
  f ⋙ g = g · f
  first f = f × id
```

(We write `f × g` for the function $\lambda(x,y) \to (f\ x, g\ y)$.)

The second instance generalises Swierstra and Duponcheel's two-phase parsers to arbitrary two-phase computations. We introduce a new type with four parameters: monads m and n for the "static" and "dynamic" parts of a computation, and input and result types α and β.

```
newtype TwoPhase (m :: * → *) (n : * → *) α β
       = TwoPhase { runTwoPhase :: m (α → n β) }
```

We can then write an `Arrow` instance for `TwoPhase`. A computation of type `TwoPhase m n` α β accepts an input of type α and returns a result of type β. All of the effects in the "static" monad m are performed before any of the effects in the "dynamic" monad n, and only computations in n, not computations in m, may depend on the arrow input.

```
instance (Monad m, Monad n) ⇒ Arrow (TwoPhase m n) where
  arr f = TwoPhase (return (return · f))
  TwoPhase f ⋙ TwoPhase g = TwoPhase (f ⋙= λh →
                                      g ⋙= λk →
                                      return (λa → h a ⋙= k))
  first (TwoPhase f) = TwoPhase (f ⋙= λh →
                                 return (λ(a,c) →
                                          h a ⋙= λb →
                                          return (b,c)))
```

The implementation of the `arr` operator is a straightforward combination of the `return` operators of the monads m and n. A computation of the form `f` $\ggg$ `g` uses the $\ggg\!=$ operator of m to extract the dynamic portions h and k from f and g, then sequences those dynamic portions using the $\ggg\!=$ operator of n, passing the arrow input a as input to h and the result of h a as input to k. Similarly, a computation of the form `first f` uses the $\ggg\!=$ operator of m to extract the dynamic portion h of f, then applies h to the first element of the arrow input `(a,c)`, passing

the second element through using the strength of $\mathfrak{n}$[5].

The third instance is a specialisation of `TwoPhase`, where $\mathfrak{m}$ is the identity monad. For each instance $\mathfrak{n}$ of `Monad`, this specialisation gives us the type of Kleisli arrows of the monad — that is, functions of the form $\alpha \to \mathfrak{n}\ \beta$. We will use this instance in translating the example computations, so it is helpful to write out the specialisation explicitly. We introduce a new type to represent Kleisli arrows, writing `Kleisli` and `runKleisli` for the constructor and destructor:

```
newtype Kleisli (n :: * → *) α β
        = Kleisli { runKleisli :: α → n β }
```

To obtain the `Kleisli` instance of the `Arrow` class we start with the `TwoPhase` instance and replace the $\ggg$ and `return` operations of the $\mathfrak{m}$ monad with the identity operations:

```
instance Monad n ⇒ Arrow (Kleisli n) where
  arr f = Kleisli (return · f)
  Kleisli f ⋙ Kleisli g = Kleisli (λa → f a ≫= g)
  first (Kleisli f) = Kleisli (λ(a,c) → f a ≫= λb → return (b,c))
```

Now we can write `Arrow` computations using any instance of `Monad`. We must also convert any primitive computations of the monad to arrow computations. For the monad from Section 2.2.1.2 we must convert `get` and `put`:

```
get⇝ :: Kleisli (State σ) () σ
get⇝ = Kleisli get

put⇝ :: Kleisli (State σ) σ ()
put⇝ = Kleisli put
```

We can now write Examples 1 (sequencing) and 2 (dataflow) as arrow computations.

Example 5 (sequencing with arrows).   Sequencing computations and collecting results is the purpose of the $\lll$ operator. Given arrow computations m of type $() \to \sigma$ and n of type $() \to \tau$, the analogue of Example 1 (sequencing) may be written $m \lll n$.

Example 6 (dataflow with arrows).   The arrow analogue of Example 2 (dataflow) may be written as a composition of three arrow computations as follows.

---

[5]The "strength" refers to an operator of type $\alpha \times \mathfrak{m}\ \beta \to \mathfrak{m}\ (\alpha \times \beta)$, i.e. a function that combines a value of type α with a computation returning a result of type β to form a computation that returns a pair of α and β. Every Haskell monad offers this functionality, but there is generally no need for an explicit operator: the combination of lexical scoping and the $\ggg$ operator make it straightforward to collect values that are in scope into the result of a monadic computation.

$$\mathtt{get}_{\leadsto} \ggg \mathtt{arr \ not} \ggg \mathtt{put}_{\leadsto}$$
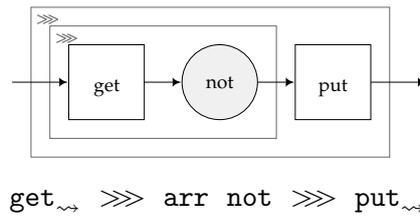
However, it is *not* possible to write Examples 3 (control flow) or 4 (higher-order) using the arrow combinators. Using the arrow combinators we can use the output of one computation to determine *what* the next computation should write, but not *whether* it should perform a write at all. The particular set of computations that should be performed is determined before any of the outputs is available. We might characterise this property as follows: using the arrow combinators, *dataflow is dynamic, but control flow is static*. This limited expressive power is, of course, a burden for users of arrows, but it is a boon for implementors; for example, it is precisely what is needed to implement the efficient parser combinators described at the beginning of Section 2.2.2.

In Section 2.2.2.4 we will extend the arrow operations with operations for choice and higher-order programming, (making it possible to write arrow versions of Examples 3 and 4) and show that not all arrow instances can support these new operations.

2.2.2.3   Arrow laws

Instances of `Arrow`, like instances of `Monad`, are subject to certain laws. Hughes (2000) gives nine laws which all instances of the `Arrow` class must satisfy.

The first, second and third laws say that a lifted identity function is a left and right unit for $\ggg$ and that composition is associative:

$$\mathtt{arr \ id} \ggg f \qquad\qquad\qquad f$$

$$f \ggg \mathtt{arr \ id} \qquad\qquad\qquad f$$

$$(f \ggg g) \ggg h \qquad f \ggg (g \ggg h)$$
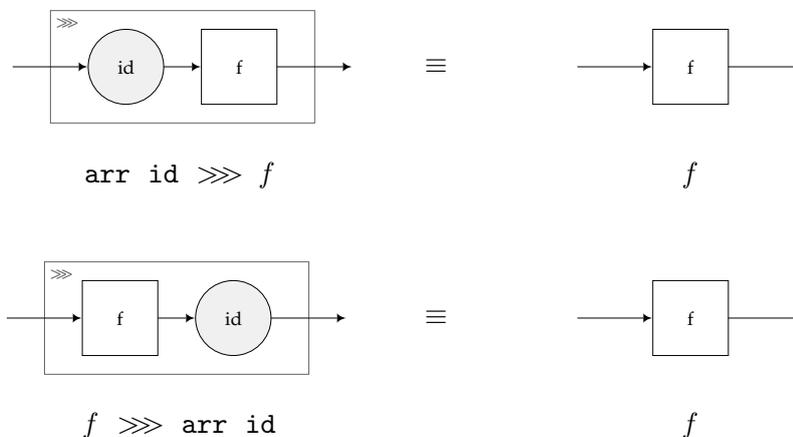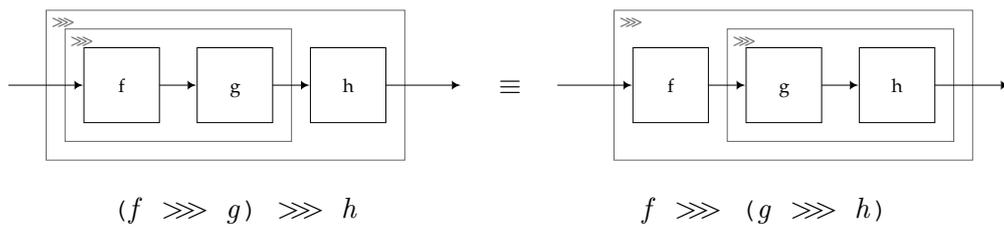
The fourth, fifth and sixth laws say that `arr` is a homomorphism for composition and `first` and that `first` is a homomorphism for composition:

$$\texttt{arr} \ (g \cdot f) \qquad \texttt{arr} \ f \ggg \texttt{arr} \ g$$

$$\texttt{arr} \ (f \times \texttt{id}) \qquad \texttt{first} \ (\texttt{arr} \ f)$$

$$\texttt{first} \ (f \ggg g) \qquad \texttt{first} \ f \ggg \texttt{first} \ g$$
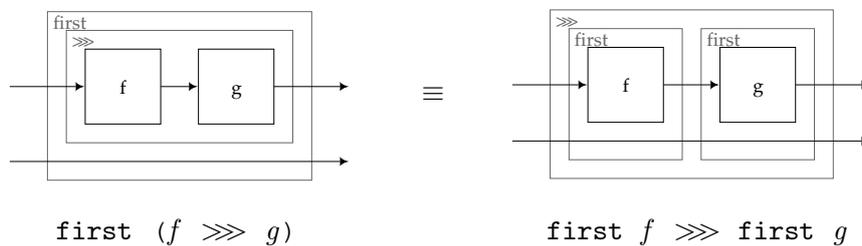
The last three laws allow pure computations to be moved to the left or right in the computational pipeline. The seventh law says that two computations which act independently on the elements of a pair may be interchanged if one of them is pure:

$$\texttt{second} \ (\texttt{arr} \ g) \ggg \texttt{first} \ f \qquad \texttt{first} \ f \ggg \texttt{second} \ (\texttt{arr} \ g)$$
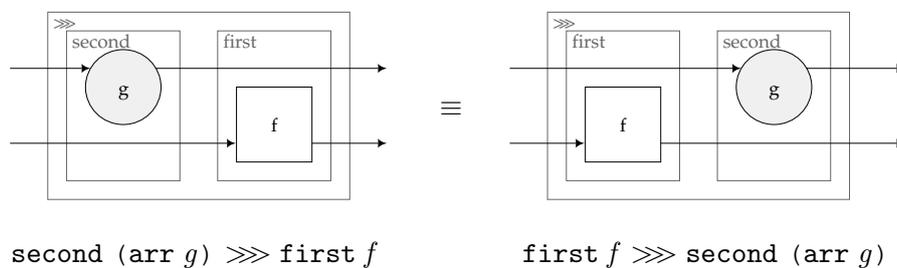
The eighth law says that projection of the first element of a pair may be equivalently performed either before or after a computation which acts only on that element:

$$\texttt{arr fst} \ggg f \qquad\qquad \texttt{first } f \ggg \texttt{arr fst}$$
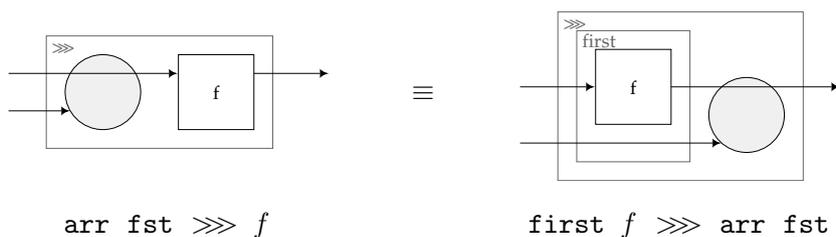
The ninth law says that values threaded alongside a computation may be grouped either before or after threading:

$$\texttt{arr assoc} \ggg \texttt{first } f \qquad\qquad \texttt{first (first } f) \ggg \texttt{arr assoc}$$
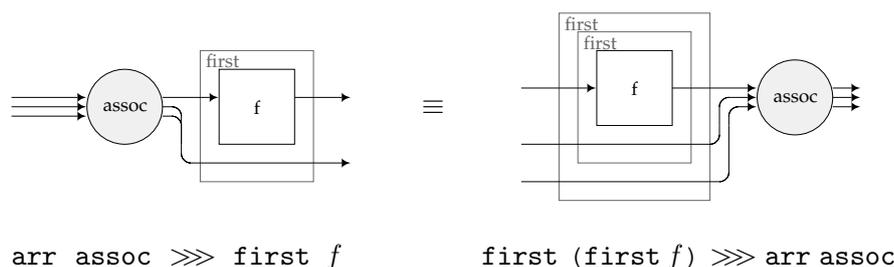
Here `assoc` is defined as the function that rearranges the three elements of a nested pair:

```
assoc ((x,y),z)  =  (x,(y,z))
```

It is easy to verify that these nine laws hold for pure functions and that they hold for `Kleisli` m whenever the monad laws hold for m.

2.2.2.4 Arrow variants

In Section 2.2.2.2 we stated that the arrow combinators are not sufficient to encode dynamic control flow — i.e., using the result of a computation to choose which of two computations to run next. In using an instance of `Monad` through the `Arrow` interface we lose the ability to express certain computations. If the underlying structure is sufficiently powerful then we can define additional combinators that provide the lost functionality. The operator `left`, a dual to `first`, is a general mechanism for selecting computations based on dynamic input. The input to the arrow `left a` has the type `Either` $\alpha\,\gamma$, which is the type of binary sums in Haskell. If the input is a left injection, `Left v`, then `a` is run with input `v`; otherwise, the input is passed through unchanged. Since there are useful instances of `Arrow` (such as Swierstra and Duponcheel's parsers) that do not support runtime branching, `left` is placed in a new class, `ArrowChoice` (Hughes, 2000).

```
class Arrow (⤳) ⇒ ArrowChoice (⤳) where
    left :: (α ⤳ β) → (Either α γ ⤳ Either β γ)
```

Now `Kleisli` is an instance of `ArrowChoice`.

```
instance Monad m ⇒ ArrowChoice (Kleisli m) where
   left (Kleisli m) = Kleisli branch
       where branch (Left l)  = m l ≫= λv → return (Left v)
             branch (Right r) = return (Right r)
```

Example 7 (control-flow with arrows). With `ArrowChoice`, Example 3 (control flow) can be written as a composition of three arrow values. The first is get$_⇝$, the arrow analogue of the monadic `get`; the second encodes a boolean as a value of `Either () ()`; the third optionally runs the computation `m` using `left`.

```
get⇝ ⋙ arr (λv → if v then Left () else Right ()) ⋙ left m
```

We can bring the expressive power of arrows closer to the expressive power of monads by lifting further monadic operations into the world of arrows. The final extension we describe closes the gap entirely. The `app` operator enables higher-order programming with arrows. As before, `app` is placed in a new class, `ArrowApply`, for arrow instances that support the full power of monadic programming.

```
class Arrow (⇝) ⇒ ArrowApply (⇝) where
   app :: (α ⇝ β, α) ⇝ β
```

The `app` operator is an arrow computation that accepts as input an arrow computation a and a value v, and returns the result of running a with v as input. Hughes (2000) gives the three laws which `app` must satisfy:

```
first (arr (λx → arr (λy → (x,y)))) ⋙ app  ≡  arr id
               first (arr (g ⋙)) ⋙ app  ≡  second g ⋙ app
               first (arr (⋙ h)) ⋙ app  ≡  app ⋙ h
```

Now we can make `Kleisli` an instance of `ArrowApply`:

```
instance Monad m ⇒ ArrowApply (Kleisli m) where
    app = Kleisli (λ(Kleisli k, v) → k v)
```

As with the other instances we have seen, it can be shown that the nine arrow laws and the laws for `app` hold for `Kleisli m` whenever the monad laws hold for `m`.

Example 8 (higher-order with arrows).

Using `ArrowApply` we can write an analogue of Example 4 (higher-order). From any computation m of type `() ⇝ (() ⇝ τ)` we can form a computation of type `() ⇝ τ` which performs the result of m.

```
(m &&& arr id) ⋙ app
```

In fact, given an instance of `ArrowApply` we can write an instance of `Monad`.

```
newtype Monadic (⤳) α = Monadic{ unMonadic :: () → α }

instance ArrowApply (⤳) ⇒ Monad (Monadic (⤳)) where
  return v = Monadic (arr (λ_ → v))
  Monadic m ⋙= k =
    Monadic (((m ⋙ arr (unMonadic · k)) &&& arr id) ⋙ app)
```

As before, it can be shown that the three monad laws hold for `Monadic` (⤳) whenever the laws for `Arrow` and `ArrowApply` hold for ⤳. Further, we can combine the `Monadic` instance of `Monad` and the `Kleisli` instance of `ArrowChoice` to obtain an instance of `ArrowChoice` from any instance of `ArrowApply`. We will investigate the connection between arrows and monads more formally in Section 2.4.

We claimed in Section 2.2.2.2 that it is not possible to write Example 4 (higher-order) using the arrow combinators. Example 8 (higher-order with arrows) shows that, given an additional operator, `app`, we can write an arrow version of Example 4. We now support our earlier claim by showing that not all instances of `Arrow` can support `app`. `Accy` is a type constructor whose second and third arguments are *phantom*, i.e. not used in the definition of the type.

```
newtype Accy μ α β = Acc {acc :: μ }
```

Then, for any monoid μ, we can make `Accy` μ an instance of `Arrow` by defining `arr` and (⋙) of the arrow to be the unit *e* and multiplication ⊛ of the monoid, respectively, and making `first` a no-op.

```
instance Monoid μ ⇒ Arrow (Accy μ) where
  arr _ = Acc e
  Acc m ⋙ Acc n = Acc (m ⊗ n)
  first (Acc m) = Acc m
```

Since a value of type `Accy` μ α β does not contain a value whose type involves α or β, a computation in the `Accy` arrow returns no result: it is executed purely for its effect, accumulation.

From the monoid laws for μ it is easy to show both that the arrow laws hold for `Accy` μ, and that `Accy` μ cannot support an `app` operation. To prove this second fact, suppose that we did have an `app` operation for `Accy` μ. Substituting the definitions of `arr`, `first`, and (⋙) into the second app law, then applying the monoid laws, gives the following equation for `app`:

```
acc app ≡ acc g ⊗ acc app
```

This is clearly unsatisfiable in general. For example, let μ be the monoid of integers under addition: there is no integer app such that `app = g + app` for every integer g.

2.2.3 Idioms

We now turn to an examination of *idioms*[6], the third and final computational interface in our taxonomy.

2.2.3.1 Idiom operators

The `Idiom` type class, like `Arrow` and `Monad`, provides an interface to effectful computation. Each instance of `Idiom` consists of a unary type constructor i, for representing computations, and two computation-forming operations, `pure` and $\circledast$ (pronounced "apply"). The concrete definition of the `Idiom` class is as follows:

```
class Idiom (i :: ⋆ → ⋆) where
    pure :: α → i α
    (⊛) :: i (α → β) → i α → i β
```

A value of type $i\,\tau$, where i is an idiom, represents a computation that performs some effects and then returns a result of type τ. The first operation of the `Idiom` class, `pure`, lifts a value to a computation that returns the value; it is analogous to the monadic `return`. The second operation, $\circledast$, constructs a new computation from two existing computations, applying the result of the first to the result of the second. (We present the laws associated with these operators in Section 2.2.3.3.)

It is worthwhile comparing the idiomatic operation for composing computations, $\circledast$, to the corresponding operations in the `Monad` and `Arrow` classes. The `Monad` interface offers $\ggg\!=$, which passes the result of a computation as input to a computation-creating function (Section 2.2.1.1). The `Arrow` interface offers the less-powerful $\ggg$, which passes the result of a computation as input to another computation (Section 2.2.2.1). The $\circledast$ operator of the `Idiom` interface is less powerful still: it does not pass the result of the first computation as input either to a computation-creating function or a computation. Instead, it combines the results of the two computations passed as operands; that is, the computations are first executed, then their results combined. There is thus no facility for one idiomatic computation to make use of the result of another.

---

[6]The interface described here was introduced by McBride (2004) under the name *Idiom* to capture a common pattern in functional programming. Subsequently McBride and Paterson (2008) changed the name to *applicative functor* to emphasise the view of idioms as an "abstract characterisation of an applicative style of effectful programming". We prefer the original name for a number of reasons, not least (as Gibbons and d. S. Oliveira (2009) point out) that it comes equipped with the convenient adjectival cognate "idiomatic".

2.2.3.2 Idiom examples

The `Idiom` interface has been used for a range of applications. The interface used by Swierstra and Duponcheel (1996) for parsers, some years before idioms were proposed as a general interface, is an extension of the `Idiom` class. Elliott (2008) has proposed the use of idioms for functional reactive programming, which is more typically implemented using arrows. Gibbons and d. S. Oliveira (2009) have shown that a traversal operator parameterised by an idiom is the essence of the *iterator* "design pattern" for object-oriented programs. Bringert and Ranta (2006) use a variant of the `Idiom` interface to define a class of "almost compositional functions" on syntax trees.

McBride and Paterson (2008) give a number of instances of `Idiom` that are of particular relevance to this chapter. The first connects arrows and idioms: if $\leadsto$ is an instance of `Arrow` then $(\tau \leadsto)$ (the partial application of the $\leadsto$ type constructor) is an idiom for any type τ. As usual, we must use a **newtype** declaration, `WrappedArrow`[7], to distinguish the particular set of arrow values that we wish to use with the `Idiom` interface.

```
newtype WrappedArrow ((⤳):: * → * → *) α β
        = WrapArrow {unwrapArrow :: α ⤳ β}
```

Then we can make `WrappedArrow` $(\leadsto)$ τ an instance of the `Idiom` class for any `Arrow` instance $\leadsto$ and any type τ. The `pure` function is built from a pure, constant arrow. The arrow created by $f \circledast v$ passes its input to the arrows f and v; the result is obtained by applying the result of f to the result of v.

```
instance Arrow (⤳) ⇒ Idiom (WrappedArrow (⤳) τ) where
  pure x = WrapArrow (arr (const x))
  WrapArrow f ⊛ WrapArrow v = WrapArrow ((f &&& v) ⋙
                                          arr (λ(g,w) → g w))
```

The second instance of interest connects the classes `Idiom` and `Monad`. From every instance of `Monad` we can obtain an instance of `Idiom`. The following **newtype** declaration wraps monadic computations.

```
newtype WrappedMonad (m :: * → *) α
        = WrapMonad { unwrapMonad :: m α }
```

Then every instance of `Monad` gives an instance of `Idiom` for which `pure` is the monadic `return` and $\circledast$ is a function that sequences its operands and applies the result of the first to the result of the second.

---

[7] McBride and Paterson (2008) call this type `EnvArrow` since, for each arrow type $\leadsto$ and type η, `WrappedArrow` $(\leadsto)$ η α is the type of arrow computations that read from an "environment" of type η and return a value of type α. The names `WrappedArrow` and `WrappedMonad` come from Paterson's proposed addition to the Haskell library (Paterson, 2008).

```
instance Monad m ⇒ Idiom (WrappedMonad m) where
   pure v = WrapMonad (return v)
   WrapMonad f ⊛ WrapMonad v = WrapMonad (f ≫= λg →
                                          v ≫= λw →
                                          return (g w))
```

The third instance of interest makes it possible to build new instances of `Idiom` from existing instances by straightforward functor composition. The type constructor `Compose` composes two functors.

```
newtype Compose (i :: * → *) (j :: * → *) α
      = Compose { deCompose :: i (j α) }
```

The definition of the `Idiom` instance for `Compose` simply lifts the `pure` and ⊛ of the inner idiom to the outer idiom.

```
instance (Idiom i, Idiom j) ⇒ Idiom (Compose i j) where
    pure v = Compose (pure (pure v))
    Compose f ⊛ Compose v = Compose (pure (⊛) ⊛ f ⊛ v)
```

A further instance of `Idiom` will be useful in exploring the connections with `Arrow` and `Monad`. The environment idiom captures the effect of reading from (constant) input. We can define the environment idiom in Haskell using the pure function arrow, partially applied to any input type τ.

```
instance Idiom ((→) τ) where
   pure v = λe → v
   f ⊛ v = λe → f e (v e)
```

This instance is isomorphic to the instance obtained by instantiating the arrow parameter of `WrappedArrow` to →, the type of pure functions, and to the instance obtained by instantiating the monad parameter of `WrappedMonad` to the environment monad (which is called `Reader` in the Haskell standard library).

We now have two ways to obtain an instance of `Idiom` from an instance m of `Monad`. The first way is direct: simply apply `WrappedMonad`. The second way is indirect: apply `Kleisli` to obtain an instance of `Arrow` and then apply `WrappedArrow` to obtain an instance of `Idiom`. Ignoring the isomorphisms introduced by the various **newtype** declarations, this latter method results in the type $\tau \to m\ \alpha$ (for the result type α), which is not equivalent to the type $m\ \alpha$ obtained using the direct method. We can therefore obtain at least two distinct instances of `Idiom` from any instance of `Monad`. However, it is easy to convert either instance into the other. The instance obtained with `WrappedMonad` may be converted into the instance obtained with `WrappedArrow` by composition with the environment idiom. The instance obtained with `WrappedArrow` may be converted into an instance isomorphic to the instance obtained with

`WrappedMonad` by instantiating τ to the unit type.

McBride and Paterson (2008) give one final instance that connects `Arrow` and `Idiom`. For any `Idiom` instance i, `StaticArrow` i is an arrow transformer that uses an idiomatic computation to construct an arrow computation.

```
newtype StaticArrow (i :: * → *) ((⤳) :: * → * → *) α β
    = Static { exStatic :: i (α ⤳ β) }
```

The arrangement of the type constructors dictates that all of the effects of the idiom occur before any of the effects of the arrow. The definitions of the `Arrow` methods for `StaticArrow` i (⤳) are then the simply operations of the arrow ⤳ lifted into the idiom i: `arr` creates a pure idiomatic computation that returns a pure arrow computation; $f \ggg g$ runs the idiomatic computations f and g, passing the results to the $\ggg$ operator of the underlying arrow; `first` f runs the idiomatic computation f and applies the `first` operator of the underlying arrow to the result.

```
instance (Idiom i, Arrow (⤳)) ⇒ Arrow (StaticArrow i (⤳)) where
    arr f                  = Static (pure (arr f))
    Static f ⤜≫ Static g = Static (pure (⤜≫) ⊛ f ⊛ g)
    first (Static f)       = Static (pure first ⊛ f)
```

We now have two ways to obtain an instance of `Arrow` from an instance m of `Monad`. The first way is direct: simply apply `Kleisli`. The second way is indirect: apply `WrappedMonad` to obtain an instance of `Idiom` and then apply `StaticArrow` (with the arrow parameter instantiated to the pure function arrow, →) to obtain an instance of `Arrow`. Ignoring the isomorphisms introduced by the various **newtype** declarations, this latter method results in the type m (α → β) (for input type α and result type β), which is generally not equivalent to the type α → m β obtained using the direct method. We can therefore obtain at least two distinct instances of `Arrow` from any instance of `Monad`. This time there is no general conversion procedure between the two. However, two approaches do yield equivalent results for certain monads. For the environment monad (where m α ≃ τ → α), the two approaches result in arrows that are isomorphic.

Using the idiom operators we can write a program analogous to Example 1 (sequencing) for sequencing two computations.

Example 9 (sequencing with idioms).

Suppose m and n are idiomatic computations with the types i σ and i τ respectively. Then the following computation of type i (σ, τ) executes m and n in order and returns the results as a pair.

```
pure (λx y → (x,y)) ⊛ m ⊛ n
```

However, there is no way to write idiomatic analogues of any of the programs in Examples 2 (dataflow), 3 (control flow) or 4 (higher-order).

In Section 2.2.2.2 we characterised arrow computations as having dynamic dataflow, but static control flow. This is manifested in our examples of state-passing as the ability to decide *what* to write based on the result of a prior computation, but not *whether* to write. The expressive power offered by the `Idiom` interface is even more limited: there is no way at all to pass the result of one computation as input to another computation. With idioms, therefore, there is no means to construct a state-passing computation that uses a prior result either to determine *whether* to perform a write or to determine *what* to write. We might characterise this property as follows: using the idiom combinators, *both dataflow and control flow are static*.

In Section 2.2.2.4 we used the `Accy` phantom monoid accumulator instance of `Arrow` to show that not all arrows supported `app`. We will use a similar instance for Idioms (first introduced by McBride and Paterson) to show that the `Idiom` operations alone cannot support the operation needed to implement Example 4.

The `AccI` type constructor takes two arguments, using only the first in its definition.

```
newtype AccI μ α = AccI {acci :: μ }
```

The `Idiom` instance for `AccI μ` is constructed by defining `pure` and `⊛` to be the unit *e* and multiplication $\otimes$ of the monoid μ, respectively. (We could obtain an equivalent instance by combining the `WrappedArrow` instance given earlier with the `Accy` instance of `Arrow`.)

```
instance Monoid μ ⇒ Idiom (AccI μ) where
  pure _           = AccI e
  AccI m ⊛ AccI n = AccI (m ⊗ n)
```

Now, if `AccI` supported the monad multiplication operator (commonly called `join`), then it would necessarily satisfy the following law, which holds for all monads:

```
join (pure x) ≡ x
```

However, since the `AccI` definition of `pure` discards its argument, this law cannot be satisfied.

2.2.3.3 Idiom laws

The `Idiom` interface, like `Monad` and `Arrow`, comes with a number of laws which must be satisfied by every instance.

The first law says that pure is a homomorphism for application.

$$\texttt{pure} \ (f \ \ v) \quad \equiv \quad \texttt{pure} \ f \ \circledast \ \texttt{pure} \ \ v$$

The second law says that a lifted identity function is a left unit for idiomatic application.

$$u \quad \equiv \quad \texttt{pure id} \ \circledast \ u$$

The third law says that nested applications can be flattened using a lifted composition operation.

$$u \ \circledast \ (v \ \circledast \ w) \quad \equiv \quad \texttt{pure} \ (\cdot) \ \circledast \ u \ \circledast \ v \ \circledast \ w$$

The fourth law says that pure computations can be moved to the left or right of other computations.

$$v \ \circledast \ \texttt{pure} \ x \quad \equiv \quad \texttt{pure} \ (\lambda \texttt{f} \rightarrow \texttt{f} \ x) \ \circledast \ v$$

(We observe that the latter three laws correspond to three combinators — I, B, and C — that form a basis for linear combinatory logic. The fourth law does not contain the C combinator directly, but is equivalent to a (less elegant) law which does:

$$f \ \circledast \ u \ \circledast \ \texttt{pure} \ x \quad \equiv \quad \texttt{pure flip} \ \circledast \ f \ \circledast \ \texttt{pure} \ x \ \circledast \ u$$

(Here `flip` is the Haskell name for the C combinator: `flip f x y = f y x`.) This law follows from the four idiom laws; conversely, the fourth idiom law follows from the first three idiom laws and this law.)

It is straightforward to show that for each of the instances in Section 2.2.3.2 these four laws follow from the laws on the underlying structures — for example, that the idiom laws for `WrappedMonad` m follow from the monad laws hold for m and that the idiom laws for `WrappedArrow` ($\rightsquigarrow$) follow from the arrow laws for $\rightsquigarrow$.

2.2.3.4   Sundries

We have seen that an idiom computation is composed of independent sub-computations whose results are combined to give the final output (Section 2.2.5.3). In particular, no sub-computation may make use of the result of another sub-computation. This independence leads to considerable freedom in writing idiom instances, exemplified in the following idiom transformer. The `Mirror` transformer converts any idiom i to an idiom `Mirror` i which reverses the sequence of effects of i. At the type level, `Mirror` is the identity:

```
newtype Mirror (i :: * → *) α = Mirror (i α)
```

The definition of `pure` is similarly trivial. The $\circledast$ operator for `Mirror` exchanges the order of the operands, and applies the result of the second to the result of the first.

```
instance Idiom i ⇒ Idiom (Mirror i) where
  pure f                = Mirror (pure f)
  Mirror f ⊛ Mirror p = Mirror (pure (flip ($)) ⊛ p ⊛ f)
```

(Here $ denotes the identity function at function type.)

It is straightforward to check that the idiom laws for `Mirror i` follow from the idiom laws for `i`.

2.2.4 Monoids

The reader might also wonder whether there are any further computational interfaces in the progression. Monadic computations may be dynamic both in dataflow and control flow. Arrows fix control flow statically, but retain dynamic dataflow; idioms introduce the restriction that dataflow must also be statically fixed. A natural question to ask is whether the fourth point in this matrix has any interesting occupants: that is, whether we might usefully define a class for capturing computations where control flow is dynamic, but dataflow is static. We will not give a definitive answer to this question, but note that it appears to be difficult to introduce operators for dynamic control flow without making dataflow dynamic also. For example we can define a new class `DynamicIdiom` which adds a facility to the `Idiom` interface for branching on the result of a computation:

```
class Idiom i ⇒ DynamicIdiom i where
    branch :: i Bool → i α → i α → i α
```

Now the `branch` operation makes it possible to write both Example 3 (control flow):

```
branch (WrapMonad (get ())) m (pure ())
```

and Example 2 (dataflow):

```
branch (WrapMonad (get ()))
       (WrapMonad (put False))
       (WrapMonad (put True))
```

Rather than pursue this question further we will consider what further restrictions we could introduce. One possibility is to remove the capacity for a computation to yield a result; this takes us from idioms to monoids. A monoidal effectful computation is simply a sequence of effectful subcomputations with fixed control flow, fixed dataflow, and no result. We saw in Section 2.2.3.2 that fixing the input parameter of an arrow yields an idiom. Similarly, fixing the output parameter of an idiom yields a monoid. In Haskell the `Monoid` type class is defined as follows:

```
class Monoid μ where
    e   :: μ
    (⊗) :: μ → μ → μ
```

We can construct a monoid from an idiom `i` by applying `i` to the unit type.

```
newtype Muffled :: (i :: * → *) = Muffle (i ())
```

The e value is obtained by lifting the unit value, `()`. The implementation of $\otimes$ combines its operands using $\circledast$ and uses a lifted constant function to discard the result.

```
instance Idiom i ⇒ Monoid (Muffled i) where
  e                      = Muffle (pure ())
  Muffle f ⊗ Muffle p = Muffle (pure (λ_ _ → ()) ⊛ f ⊛ p)
```

It is straightforward to check that the monoid laws for `Muffled i` (i.e. that $\otimes$ is associative with e as left and right unit) follow from the idiom laws for `i`.

An auxiliary function, `muffle`, replaces the result of any idiomatic computation with the unit value, making the computation available for use in the monoid:

```
muffle :: Idiom i ⇒ i α → Muffled i
muffle i = pure (λ_ → ()) ⊛ i
```

Section 2.2.3.2 showed how to obtain an instance `AccI μ` of `Idiom` for any `Monoid` instance μ. Using `Muffled` we can recover the original `Monoid` instance from `AccI`. That is, the `Monoid` instance for `Muffled (AccI μ)` is equivalent to the `Monoid` instance for μ.

Example 10 (sequencing with monoids).

If m and n are computations of type $i\,\sigma$ and $i\,\tau$ respectively (for some `Idiom` instance `i`) then the following monoidal computation executes m and n in order, discarding their results.

$$\texttt{muffle}\ m\ \otimes\ \texttt{muffle}\ n$$

2.2.5   Normal forms

The laws which accompany the idiom, arrow and monad interfaces group computations into equivalence classes. Any particular computation can be expressed equivalently using any member of its class. For example, if we wish to nest occurrences of the $\ggg$ operator in an arrow computation, the third arrow law gives us the choice whether to group to the left (writing $(f \ggg g) \ggg h$) or to the right (writing $f \ggg (g \ggg h)$). On occasion we may wish to check whether two computations belong to the same equivalence class. One way to answer this question is to attempt to find some way to rewrite one of the computations into the other by repeatedly applying laws.

Determining equivalence is much more convenient if we choose a canonical member of each equivalence class and define a normalisation procedure for obtaining the canonical member from any computation. Determining whether two computations are equivalent then be-

comes a matter of comparing their canonical forms.

2.2.5.1   Arrow normal forms

We begin by considering normal forms for arrows. Under the arrow laws, every arrow computation is equivalent to a term of the form shown in Figure 2.2, for some terms $f_1, \ldots f_n$, g and free variables of computation type $c_1 \ldots c_n$. The normal form is characterised by a lack of nesting (there is a single pipeline of computations), a maximisation of dataflow (the result of every computation is made available to every subsequent computation), and the alternation of pure computations and constants.

Example 11 (dataflow normal form).   The normal form for Example 6 (dataflow with arrows) is as follows:


```
((arr id ⋙ get↝)
   ⫴ arr id)       ⋙
                ((arr (not · fst) ⋙ put↝)
                    ⫴ arr id)         ⋙
                                    arr fst
```
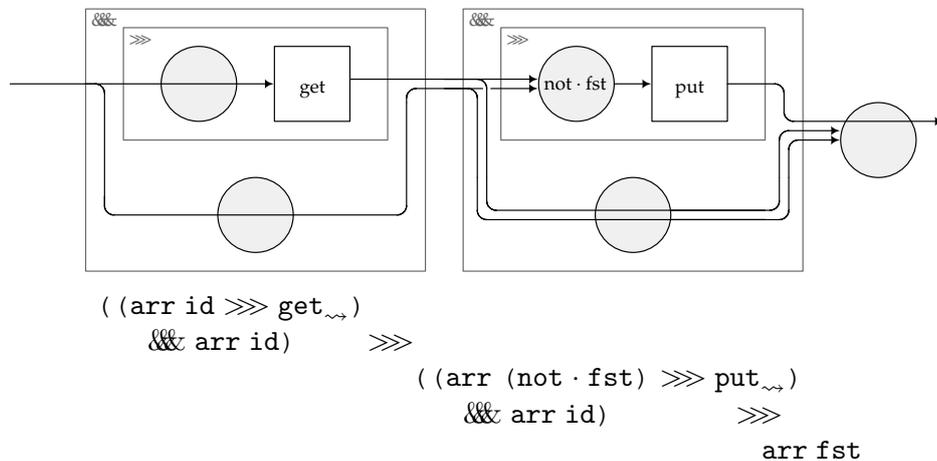
We can capture the normal form for `Arrow` in a Haskell datatype. Monad transformers — type constructors of kind $(\star \to \star) \to (\star \to \star)$ that construct monads from monads — are a mechanism used to modularise effectful functional programs (Liang, Hudak, and Jones, 1995). There is an analogous notion, *arrow transformers*, for constructing arrows from arrows. The particular arrow transformer in which we are interested does not introduce new effects, but rearranges arrow computations into normal form.

Our arrow transformer uses a technique introduced by Hughes (1995) and used to derive monad transformers by Hinze (2000): we represent operations as terms, and use the laws governing the operations to guide implementation. Hughes (2004) uses this technique to reduce occurrences of the composition operation for stream arrows, where composition is expensive, and notes that

> One can go on to build optimising arrow transformers that implement more and more algebraic laws[.]
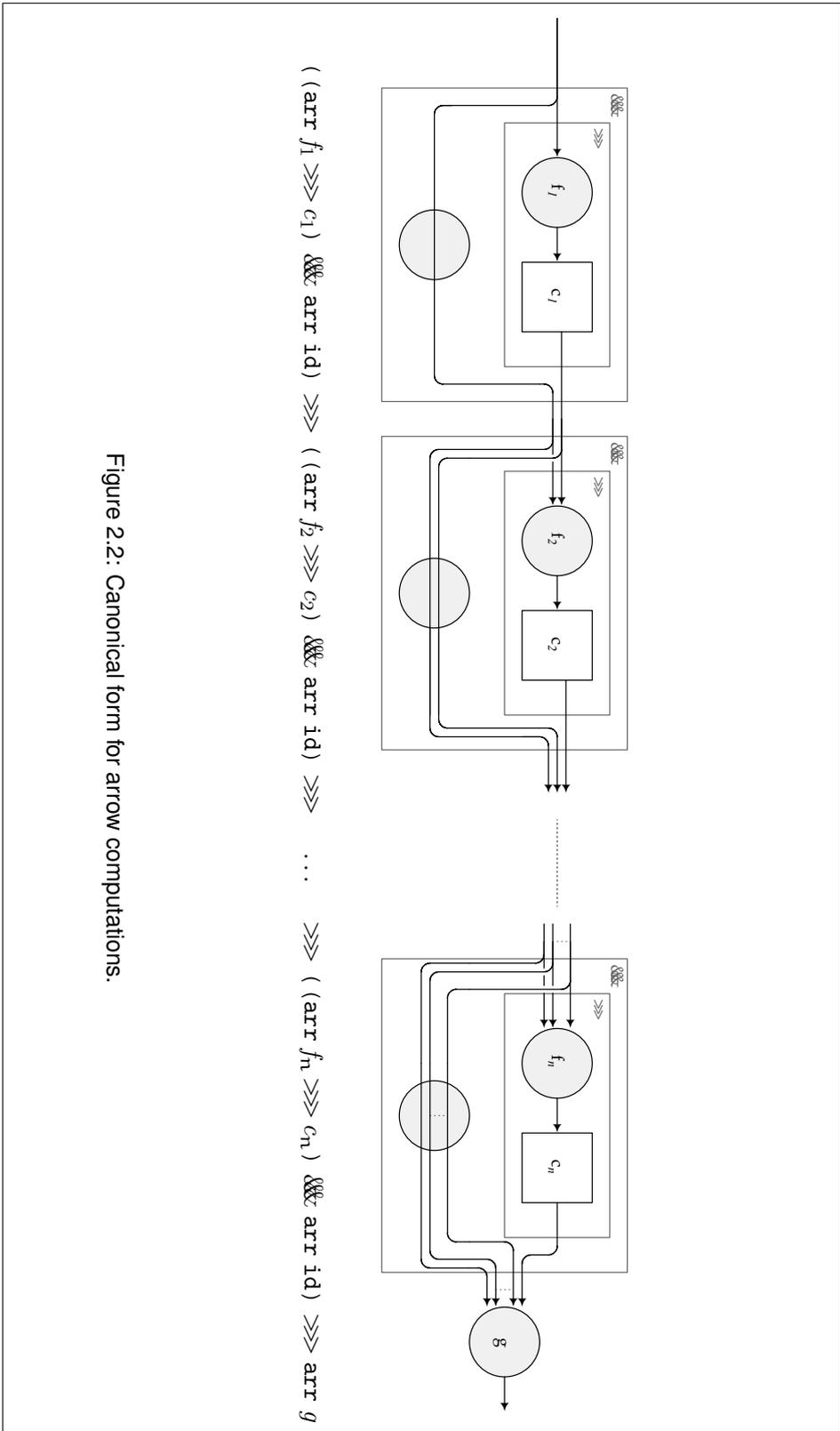
Figure 2.2: Canonical form for arrow computations.

The normalising arrow transformer given below is a kind of culmination of this idea (although we do not claim that the normal form given here is especially efficient).

Hughes (2004) further notes that the generalised algebraic data types (GHC User's Guide, Section 7.4.6) supported by the Glasgow Haskell Compiler are "invaluable" for constructing optimising arrow transformers. In fact, we can make do with a simpler and much more widely supported extension, namely *existential datatypes*: datatypes whose specification may contain type variables other than the parameters to the type. We will, however, make use of GHC's *notation* for generalised algebraic data types (which is, in our opinion, significantly clearer than the standard notation) giving a kind signature for the type constructor and a type signature for each data constructor.

The `NormA` datatype defines an arrow transformer that captures the normal form for arrows. The type constructor `NormA` takes a binary type operator denoting arrow computations to a binary type operator denoting *normalised* arrow computations.

```
data NormA :: (∗ → ∗ → ∗) → (∗ → ∗ → ∗) where
  Arr :: (α → β) → NormA (⤳) α β
  Seq :: (a → δ) → (δ ⤳ γ) → NormA (⤳) (γ, α) β → NormA (⤳) α β
```

A normalised arrow computation is either of the form `arr` g (for some function g), which we represent using the first constructor:

```
Arr g
```

or of the form `((arr` f `>>>` c`) &&& arr id) >>>` n, (for some function f, computation constant c and normalised computation n), which we represent using the second constructor

```
Seq f c n
```

We can then give a compositional translation from the three operators of the `Arrow` class into the `NormA` datatype — that is, give an instance of `Arrow` for `NormA` (⤳).

```
instance Arrow (⤳) ⇒ Arrow (NormA (⤳)) where
  arr                = Arr
  Arr f >>> Arr g    = Arr (g · f)
  Arr f >>> Seq g c h = Seq (g·f) c (Arr (id × f) >>> h)
  Seq g c h >>> s    = Seq g c (h >>> s)
  first (Arr f)      = Arr (f × id)
  first (Seq g c h)  = Seq (g·fst) c (Arr assoc⁻¹ >>> first h)
    where assoc⁻¹ (x,(y,z)) = ((x,y),z)
```

Each line in the definition of `>>>` corresponds to an equation which follows from the arrow laws. The first case for `>>>` corresponds to the fifth law (that `arr` is a homomorphism for composition). The second case for `>>>` corresponds to the following equation:

$$\text{arr f} \ggg ((\text{arr g} \ggg \text{c}) \; \&\&\& \; \text{arr id}) \ggg \text{h}$$

$$\equiv$$

$$((\text{arr } (\text{g} \cdot \text{f})) \ggg \text{c}) \; \&\&\& \; \text{arr id}) \ggg (\text{arr } (\text{id} \times \text{f}) \ggg \text{h})$$

which is an easy consequence of six laws: the left unit and associativity laws, the three homomorphism laws, and the seventh law. (See Appendix A.1 for the details.) The final case for $\ggg$ corresponds to an instantiation of the third law (that composition is associative).

The definition of `first` is similar. The first case corresponds to the fourth law (that `arr` is a homomorphism for `first`) and the second case corresponds to the following equation:

$$\text{first } (((\text{arr g} \ggg \text{c}) \; \&\&\& \; \text{arr id}) \ggg \text{h})$$

$$\equiv$$

$$((\text{arr } (\text{g} \cdot \text{fst}) \ggg \text{c}) \; \&\&\& \; \text{arr id}) \ggg \text{arr assoc}^{-1} \ggg \text{first h}$$

which is a straightforward consequence of six laws (Appendix A.1): the left unit and associativity laws, the three homomorphism laws, and the ninth law.

Before we can make practical use of the `NormA` transformer we require two further operations. The first, `promote`, lifts computations in the arrow $\rightsquigarrow$ to computations in the arrow `NormA` $(\rightsquigarrow)$. Computations lifted with `promote` behave as constants during normalisation. We will use `promote` to lift primitive computations such as $\text{get}_{\rightsquigarrow}$ and $\text{put}_{\rightsquigarrow}$.

```
promote :: Arrow (↝) ⇒ (α ↝ β) → NormA (↝) α β
promote f = Seq id f (Arr fst)
```

The second operation, `observe`, converts a value of type `NormA` $(\rightsquigarrow)$ α β back to a value of type $\alpha \rightsquigarrow \beta$:

```
observe :: Arrow (↝) ⇒ NormA (↝) α β → (α ↝ β)
observe (Arr a) = arr a
observe (Seq f c h) = (arr id &&& (arr f ⋙ c)) ⋙ observe h
```

It is easy to show that these definitions are correct: the definition of `promote` follows easily from six laws (Appendix A.1): the left identity and associativity laws, the three homomorphism laws, and the eighth law, while the definition of `observe` is simply the intended semantics for `Seq` that we gave earlier.

The types of the constructors of `NormA` enforce the constraint that only normalised computations can be represented. We have shown that the equations defining the translation to normal form are justified by the arrow laws. The final step in showing that the normalisation translation is correct is to check that it terminates on every finite input. This follows easily from an examination of the sizes of the values of type `NormA` passed to the `Arrow` operators on the left and right of each equation. We omit the details.

Example 12 (normalisation).   We can trace the normalisation of the state-negating compu-

tation of Example 6 (dataflow with arrows) by successively unfolding the definitions of the operations of `NormA`.

$$\mathtt{promote\ get}_{\leadsto} \ggg \mathtt{arr\ not} \ggg \mathtt{promote\ put}_{\leadsto}$$

=   (unfolding `promote`)

`Seq id get`$_{\leadsto}$ `(Arr fst)` $\ggg$ `arr not` $\ggg$ `Seq id put`$_{\leadsto}$ `(Arr fst)`

=   (unfolding `arr` for `NormA`)

`Seq id get`$_{\leadsto}$ `(Arr fst)` $\ggg$ `Arr not` $\ggg$ `Seq id put`$_{\leadsto}$ `(Arr fst)`

=   (unfolding $\ggg$ for `NormA`)

`Seq id get`$_{\leadsto}$
    `(Seq ((fst` $\ggg$ `not)` $\ggg$ `id) put`$_{\leadsto}$
       `(Arr (second (fst` $\ggg$ `not)` $\ggg$ `fst)))`

=   (unfolding $\ggg$ for pure functions)

`Seq id get`$_{\leadsto}$
    `(Seq (not · fst) put`$_{\leadsto}$
       `(Arr (second (fst` $\ggg$ `not)` $\ggg$ `fst)))`

=   (unfolding `second` for pure functions)

`Seq id get`$_{\leadsto}$ `(Seq (not · fst) put`$_{\leadsto}$ `(Arr fst))`

The application of `observe` to this final term reduces to the normalised state-negating computation given in Example 11:

`((arr id` $\ggg$ `get`$_{\leadsto}$`)` `&&&` `arr id)` $\ggg$
`((arr (not · fst)` $\ggg$ `put`$_{\leadsto}$`)` `&&&` `arr id)` $\ggg$
  `arr fst`

An unusually meticulous reader might notice that we have used only eight of the nine arrow laws in the definition of the normalising arrow transformer. The second law — that a lifted identity function is a right unit for composition — did not make an appearance in either the definition of the arrow combinators or the definition of `promote`. We shall have more to say about this curiosity in Section 2.3.5.

2.2.5.2   Monad normal forms

Under the monad laws every monadic computation is equivalent to a term of the form represented by the following existential datatype[8].

---

[8] Infix constructors, such as :$\ggg$=, are a GHC extension.

```
data NormM :: (⋆ → ⋆) → (⋆ → ⋆) where
  Return ::                    α              → NormM m α
  (:≫=) :: m β → (β → NormM m α) → NormM m α
```

That is, a normalised monadic computation is either of the form `return` v (for some value v), which we represent using the first constructor:

```
Return v
```

or of the form $c\ v \ggeq k$ (where c is a parameterised computation constant, v its argument, and k an expression denoting a function that returns a normalised computation), which we represent using the second constructor:

```
c  v :≫= k
```

Compared to the normal form for arrow computations in Section 2.2.5.1, the normal form for monads permits considerable flexibility. The normal form for arrows consists of a sequence of predetermined computational constants interspersed with pure functions; the result of each computation may determine the input to subsequent computations, but cannot affect the choice of constants. In contrast, each computation in the normal form for monads passes the result to a continuation which may use it (together with the results of any previous computations) in determining both the argument to the next computation and the next computational constant.

As we did with the arrow normaliser, we give an instance of `Monad` for our datatype; this serves as a translation from the operations of the monad class into `NormM`. The `Monad` instance for `NormM` replaces `Return` on the left of $\ggeq$ with application, using the left unit law, and replaces left nesting of `:≫=` with right nesting using the associativity law.

```
instance Monad m ⇒ Monad (NormM m) where
  return         = Return
  Return x    ≫= f = f x
  (m :≫= f) ≫= g = m :≫= (λx → f x ≫= g)
```

(As with the normaliser for `Arrow` in Section 2.2.5.1, it is easy to see that the translation terminates on finite input.) As before, we give functions for converting monadic computations to and from normal form. The `promoteM` function lifts a computation constant to normal form using the right unit law.

```
promoteM :: Monad m ⇒ m α → NormM m α
promoteM m = m :≫= Return
```

The inverse operation, `observeM`, simply maps the constructors of the `NormM` datatype back to their counterparts in the `Monad` class.

Figure 2.3: Canonical form for idiom computations.
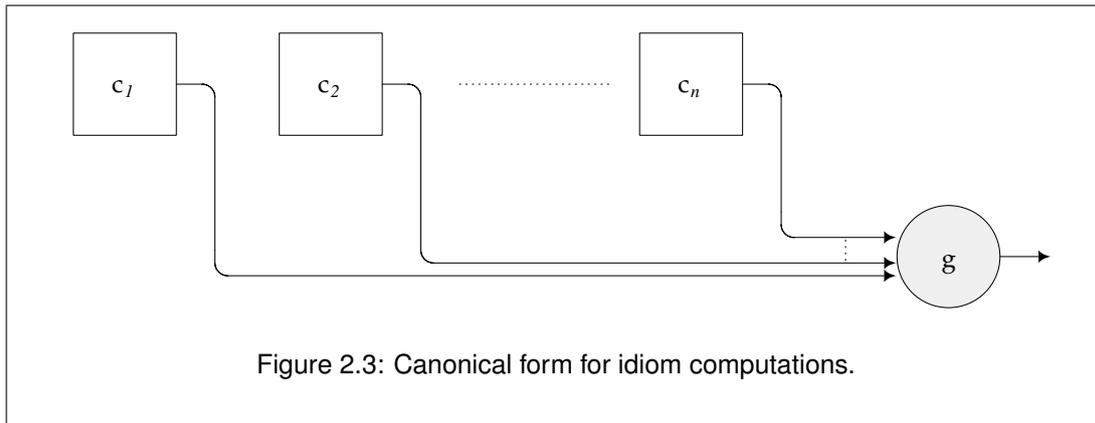
```
observeM :: Monad m ⇒ NormM m α → m α
observeM (Return x) = return x
observeM (m :≫ f) = m ≫ observeM · f
```

2.2.5.3 Idiom normal forms

Under the idiom laws every idiomatic computation is equivalent to a term of the form

$$\texttt{pure } g \circledast c_1 \circledast \ldots \circledast c_n$$

where $c_1 \ldots c_n$ are free variables of computation type and g is a pure term. Figure 2.3 illustrates the normal form using a graphical notation analogous to the arrow notation of Section 2.2.2.

As Figures 2.2 and 2.3 illustrate, idiom computations correspond approximately to arrow computations without input. This significantly simplifies things: the "plumbing" for piping results from one computation to another in the arrow diagram disappears when we move to idioms. Instead, the normal form for idioms represents a simple sequence of effects (indicated by the left-to-right ordering of $c_1 \ldots c_n$) and a final pure function g that combines the results.

We can capture the normal form for idioms using a Haskell datatype, as we did for arrows in Section 2.2.5.1 and monads in Section 2.2.5.2. The existential datatype NormI is an *idiom transformer* that represents normal forms of idiomatic computations.

```
data NormI :: (⋆ → ⋆) → (⋆ → ⋆) where
    Pure ::                     α → NormI i α
    (:⊛) :: NormI i (α → β) → i α → NormI i β
```

A normalised idiomatic computation is either of the form pure g (for some function g), which we represent using the first constructor:

```
Pure g
```

or of the form $n \circledast c$ (for some normalised computation n, and computation constant c),

which we represent using the second constructor:

$$n \; :\circledast \; c$$

We can then give an instance of `Idiom` for `NormI i`, which serves as a translation from the two operators of the `Idiom` class into the `NormI` datatype.

```
instance Idiom i ⇒ Idiom (NormI i) where
    pure = Pure
    Pure f ⊛ Pure x  = Pure (f x)
    u      ⊛ v :⊛ w = (Pure (·) ⊛ u ⊛ v) :⊛ w
    u      ⊛ Pure x  = Pure (λf → f x) ⊛ u
```

It is particularly easy to see that our definition is justified by the idiom laws. The three lines of the definition of ⊛ correspond directly to the first, third and fourth laws of Section 2.2.3.3.

As with `Arrow` and `Monad`, we need two further operations. The first, `promoteI`, lifts a constant to a normalised computation.

```
promoteI :: Idiom i ⇒ i α → NormI i α
promoteI i = Pure id :⊛ i
```

The definition of `promoteI` corresponds directly to the second law of Section 2.2.3.3. The second operation, `observeI`, directly encodes the desired semantics of the normal form.

```
observeI :: Idiom i ⇒ NormI i α → i α
observeI (Pure v)  = pure v
observeI (f :⊛ v) = observeI f ⊛ v
```

As with the normalising arrow transformer of Section 2.2.5.1, it remains only to check that the idiomatic normalisation translation given above terminates on all finite input. As before, this is easy to show by tracking the sizes of the values of type `NormI` passed to ⊛ on the left and right of each equation, and we omit the details.

2.2.5.4 Monoid normal forms

Under the monoid laws every monoidal computation is equivalent to a term of the form

$$c_1 \; \otimes \; (c_2 \; \otimes \; \ldots \; (c_n \; \otimes \; e))$$

where $c_1 \ldots c_n$ are free variables of computation type.

We can capture the normal form for monoids using a datatype, as we did for arrows, idioms and monads. The datatype `NormMon` is a *monoid transformer* that represents normal forms of monoidal computations. We do not need an existential type on this occasion, since monoid computations are not parameterised; nevertheless, we retain the GADT notation for consistency with the other examples.

```
data NormMon :: * → * where
    E     ::                    NormMon m
    (:⊗) :: m → NormMon m → NormMon m
```

The `Monoid` instance for `NormMon` translates the two monoid operators into the `NormMon` datatype, removing occurrences of `E` on the left using the left-unit law and replacing left nesting with right nesting using the associativity law.

```
instance Monoid m ⇒ Monoid (NormMon m) where
    e = E
    E         ⊗ m = m
    (m :⊗ n) ⊗ o = m :⊗ (n ⊗ o)
```

The `promoteMon` lifts a computation constant to normal form, using the right-unit law.

```
promoteMon :: Monoid m ⇒ m → NormMon m
promoteMon m = m ⊗ E
```

The inverse operation, `observeMon`, simply maps the constructors of `NormMon` back to their counterparts in the `Monoid` class.

```
observeMon :: Monoid m ⇒ NormMon m → m
observeMon E = e
observeMon (m :⊗ n) = m ⊗ observeMon n
```

2.3   Two views of arrows[9]

Section 2.2 presented three interfaces to computation in decreasing order of strength: monads embed higher-order effectful computation in a functional language; arrows are restricted to computation in which control flow is statically determined; idioms introduce the further restriction that one computation may not make use of the result of another.

The examples of Section 2.2 illustrate the relative strength of Haskell manifestations of monads, arrows and idioms. The normal forms of Section 2.2.5 reveal the types of computations expressible using those interfaces. Haskell is a convenient vehicle with which to explore the various interfaces: it is the language in which each was first introduced as a program structuring technique. Further, the type class system makes it convenient to give a common interface to several notions of computation within a single program. The Haskell presentation, however, has the unfortunate property that the interfaces differ not only semantically (which is crucial) but notationally (which is irrelevant). For instance, the `Arrow` class enforces a point-free style of programming based on composition rather than application (this is the notation), which is limited to expressing computations for which, to repeat the slogan of Section 2.2.2.2, "dataflow

---

[9]This section is a revision of Lindley et al. (2008a).

is dynamic, but control flow is static" (this is the semantics). These aspects are essentially independent: there is nothing inherently point-free about static control flow. Likewise for `Idiom`: the notation is applicative, but this is hardly essential. (Indeed, McBride and Paterson (2008) give an alternative presentation of idioms as *lax monoidal functors*.) A more semantic characterisation of idioms is that "both dataflow and control flow are static"; this has little or nothing to do with the applicative style.

In the following pages we will use an alternative formulation which avoids irrelevant differences of notation, making it easier to compare expressive power directly. Our approach is to show that idioms and monads correspond to static and higher-order variations on a metalanguage for arrow computations, the *arrow calculus*. We will use this metalanguage to reveal the formal relationships between the three interfaces, justifying the informal claims of Section 2.2.

2.3.1   Classic arrows

We refer to the presentation of arrows given in Section 2.2.2 as *classic arrows*.

We present both classic arrows and arrow calculus as *equational theories*, which are defined as follows.

Definition 1.   *A typed* equational theory T *consists of the following*

- *variables* $\mathsf{x}, \mathsf{y}, \mathsf{z}$

- *types* $\mathsf{A}, \mathsf{B}, \mathsf{C}$

- *terms* $\mathsf{L}, \mathsf{M}, \mathsf{N}$

- *type environments* $\Gamma ::= \cdot \mid \mathsf{x} : \mathsf{A}, \Gamma$

- *typing judgments* $\Gamma \vdash_{\mathsf{T}} \mathsf{M} : \mathsf{A}$

- *equational judgments* $\Gamma \vdash_{\mathsf{T}} \mathsf{M} = \mathsf{N} : \mathsf{A}$

Equational judgments must be well-formed: if $\Gamma \vdash_{\mathsf{T}} \mathsf{M} = \mathsf{N} : \mathsf{A}$ *then* $\Gamma \vdash_{\mathsf{T}} \mathsf{M} : \mathsf{A}$ *and* $\Gamma \vdash_{\mathsf{T}} \mathsf{N} : \mathsf{A}$. *We present the equational judgments via laws relating terms, writing* $\mathsf{M} = \mathsf{N}$ *as shorthand for* $\Gamma \vdash_{\mathsf{T}} \mathsf{M} = \mathsf{N} : \mathsf{A}$ *for all* Γ, A *in* T *such that* $\Gamma \vdash_{\mathsf{T}} \mathsf{M} : \mathsf{A}$ *and* $\Gamma \vdash_{\mathsf{T}} \mathsf{N} : \mathsf{A}$. *The equational theory is defined as the contextual and equivalence closure of the laws.*

Figure 2.4 (page 50) gives a standard definition of the theory of typed lambda calculus extended with pairs and unit, $\lambda^{\to \times 1}$. A type judgment $\Gamma \vdash \mathsf{M} : \mathsf{A}$ indicates that in environment Γ term M has type A. We use a Curry formulation, eliding types from terms. Products and functions satisfy beta and eta laws; the unit type satisfies an eta law. We use this definition as

a starting point for each of the theories which follow. For convenience we define a number of functions, such as `id`.

Figure 2.5 defines the theory of classic arrows, $\mathcal{C}$, together with various auxiliary functions. The theory is a straightforward recapitulation of the informal presentation of Section 2.2.2, without the conveniences such as pattern matching and type classes afforded by Haskell. The theory of classic arrows extends the core lambda calculus with a binary type constructor, $\leadsto$, and three constants (`arr`, `first`, $\ggg$) satisfying nine laws. These laws, together with the laws of the lambda calculus, define the equivalence relation of the theory.

2.3.2 Arrow calculus

Figure 2.6 (page 52) defines the theory of arrow calculus, $\mathcal{A}$. Arrow calculus extends the core lambda calculus with four constructs satisfying five laws. As before, the type $A \leadsto B$ denotes a computation that accepts a value of type A and returns a value of type B, possibly performing some side effects. We now have two syntactic categories. Terms, as before, are ranged over by L, M, N, and commands are ranged over by P, Q, R. In addition to the terms of the core lambda calculus, there is one new term form: arrow abstraction $\lambda^{\bullet} x. Q$. There are three command forms: arrow application $L \bullet M$, arrow unit $[M]$ (which resembles unit in a monad), and arrow bind **let** $x \Leftarrow P$ **in** Q (which resembles bind in a monad).

In addition to the term typing judgment

$$\Gamma \vdash M : A.$$

we now also have a command typing judgment

$$\Gamma; \Delta \vdash P \mathbin{!} A.$$

An important feature of the arrow calculus is that the command type judgment has two environments, Γ and Δ, where variables in Γ come from ordinary lambda abstractions $\lambda x. N$, while variables in Δ come from arrow abstractions $\lambda^{\bullet} x. Q$ and let bindings **let** $x \Leftarrow P$ **in** Q. There is no variable rule for the Δ environment: instead, the rules for arrow unit $[M]$ and arrow application $L \bullet M$ move variables from Δ into Γ.

We will give a translation of commands to classic arrows, such that a command P satisfying the judgment

$$\Gamma; \Delta \vdash P \mathbin{!} A$$

translates to a term $[\![P]\!]_{\Delta}$ satisfying the judgment

$$\Gamma \vdash [\![P]\!]_{\Delta} : \Delta \leadsto A.$$

Syntax:

$$\text{Types} \qquad A, B, C \quad ::= \quad \mathcal{B} \mid 1 \mid A{\times}B \mid A \to B$$

$$\text{Terms} \qquad L, M, N \quad ::= \quad x \mid \langle\rangle \mid \langle M, N\rangle \mid \mathbf{fst}\, L \mid \mathbf{snd}\, L \mid \lambda x.\, N \mid L\, M$$

$$\text{Environments} \quad \Gamma \qquad ::= \quad x_1 : A_1, \ldots, x_n : A_n$$

Types:

$$\frac{(x : A) \in \Gamma}{\Gamma \vdash x : A} \qquad\qquad\qquad\qquad \Gamma \vdash \langle\rangle : 1$$

$$\frac{\Gamma \vdash M : A \qquad \Gamma \vdash N : B}{\Gamma \vdash \langle M, N\rangle : A{\times}B} \qquad \frac{\Gamma \vdash L : A{\times}B}{\Gamma \vdash \mathbf{fst}\, L : A} \qquad \frac{\Gamma \vdash L : A{\times}B}{\Gamma \vdash \mathbf{snd}\, L : B}$$

$$\frac{\Gamma, x : A \vdash N : B}{\Gamma \vdash \lambda x.\, N : A \to B} \qquad\qquad \frac{\Gamma \vdash L : A \to B \qquad \Gamma \vdash M : A}{\Gamma \vdash L\, M : B}$$

Definitions:

$$
\begin{array}{llcl}
\texttt{id} & : & A \to A \\
\texttt{id} & = & \lambda x.\, x
\end{array}
\qquad
\begin{array}{lcl}
\texttt{swap} & : & A{\times}B \to B{\times}A \\
\texttt{swap} & = & \lambda z.\, \langle \mathbf{snd}\, z, \mathbf{fst}\, z\rangle
\end{array}
$$

$$
\begin{array}{llcl}
\texttt{dup} & : & A \to A{\times}A \\
\texttt{dup} & = & \lambda x.\, \langle x, x\rangle
\end{array}
\qquad
\begin{array}{lcl}
(\cdot) & : & (B \to C) \to (A \to B) \to (A \to C) \\
(\cdot) & = & \lambda f.\, \lambda g.\, \lambda x.\, f\, (g\, x)
\end{array}
$$

$$
\begin{array}{llcl}
\texttt{fst} & : & A{\times}B \to A \\
\texttt{fst} & = & \lambda z.\, \mathbf{fst}\, z
\end{array}
\qquad
\begin{array}{lcl}
\texttt{assoc} & : & (A{\times}B){\times}C \to A{\times}(B{\times}C) \\
\texttt{assoc} & = & \lambda z.\, \langle \mathbf{fst}\,(\mathbf{fst}\, z), \langle \mathbf{snd}\,(\mathbf{fst}\, z), \mathbf{snd}\, z\rangle\rangle
\end{array}
$$

$$
\begin{array}{llcl}
\texttt{snd} & : & A{\times}B \to B \\
\texttt{snd} & = & \lambda z.\, \mathbf{snd}\, z
\end{array}
\qquad
\begin{array}{lcl}
\texttt{apply} & : & (A \to B){\times}A \to B \\
\texttt{apply} & = & \lambda z.\, (\mathbf{fst}\, z\, (\mathbf{snd}\, z))
\end{array}
$$

$$
\begin{array}{lcl}
(\times) & : & (A \to C) \to (B \to D) \to (A{\times}B \to C{\times}D) \\
(\times) & = & \lambda f.\, \lambda g.\, \lambda z.\, \langle f\,(\mathbf{fst}\, z), g\,(\mathbf{snd}\, z)\rangle
\end{array}
$$

Laws:

$$
\begin{array}{rrcl}
(\beta_1^{\times}) & \mathbf{fst}\, \langle M, N\rangle & = & M \\
(\beta_2^{\times}) & \mathbf{snd}\, \langle M, N\rangle & = & N \\
(\eta^{\times}) & \langle \mathbf{fst}\, L, \mathbf{snd}\, L\rangle & = & L \\
(\beta^{\to}) & (\lambda x.\, N)\, M & = & N[x := M] \\
(\eta^{\to}) & \lambda x.\, (L\, x) & = & L \\
(\eta^1) & \langle\rangle & = & M
\end{array}
$$

Figure 2.4: Lambda calculus, $\lambda^{\to \times 1}$.

Syntax:

$$\text{Types} \quad A, B, C \quad ::= \quad \cdots \mid A \rightsquigarrow B$$

$$\text{Terms} \quad L, M, N \quad ::= \quad \cdots \mid \texttt{arr} \mid (\ggg) \mid \texttt{first}$$

Constants:

$$\texttt{arr} \ : \ (A \rightarrow B) \rightarrow (A \rightsquigarrow B)$$

$$(\ggg) \ : \ (A \rightsquigarrow B) \rightarrow (B \rightsquigarrow C) \rightarrow (A \rightsquigarrow C)$$

$$\texttt{first} \ : \ (A \rightsquigarrow B) \rightarrow (A{\times}C \rightsquigarrow B{\times}C)$$

Definitions:

$$\texttt{second} \ : \ (A \rightsquigarrow B) \rightarrow (C{\times}A \rightsquigarrow C{\times}B)$$

$$\texttt{second} \ = \ \lambda\texttt{f}.\,\texttt{arr swap} \ggg \texttt{first f} \ggg \texttt{arr swap}$$

$$(\&\!\&\!\&) \ : \ (C \rightsquigarrow A) \rightarrow (C \rightsquigarrow B) \rightarrow (C \rightsquigarrow A{\times}B)$$

$$(\&\!\&\!\&) \ = \ \lambda\texttt{f}.\,\lambda\texttt{g}.\,\texttt{arr dup} \ggg \texttt{first f} \ggg \texttt{second g}$$

Laws:

$(\rightsquigarrow_1)$	$\texttt{arr id} \ggg L$	$=$	L
$(\rightsquigarrow_2)$	$L \ggg \texttt{arr id}$	$=$	L
$(\rightsquigarrow_3)$	$(L \ggg M) \ggg N$	$=$	$L \ggg (M \ggg N)$
$(\rightsquigarrow_4)$	$\texttt{arr}\,(M \cdot L)$	$=$	$\texttt{arr}\,L \ggg \texttt{arr}\,M$
$(\rightsquigarrow_5)$	$\texttt{first}\,(\texttt{arr}\,L)$	$=$	$\texttt{arr}\,(L{\times}\texttt{id})$
$(\rightsquigarrow_6)$	$\texttt{first}\,(L \ggg M)$	$=$	$\texttt{first}\,L \ggg \texttt{first}\,M$
$(\rightsquigarrow_7)$	$\texttt{first}\,L \ggg \texttt{arr}\,(\texttt{id}{\times}M)$	$=$	$\texttt{arr}\,(\texttt{id}{\times}M) \ggg \texttt{first}\,L$
$(\rightsquigarrow_8)$	$\texttt{first}\,L \ggg \texttt{arr fst}$	$=$	$\texttt{arr fst} \ggg L$
$(\rightsquigarrow_9)$	$\texttt{first}\,(\texttt{first}\,L) \ggg \texttt{arr assoc}$	$=$	$\texttt{arr assoc} \ggg \texttt{first}\,L$

Figure 2.5: Classic arrows, $\mathcal{C}$ (extends $\lambda^{\rightarrow\times 1}$, Figure 2.4).

Syntax:

$$\text{Types} \qquad A, B, C \quad ::= \quad \cdots \mid A \rightsquigarrow B$$

$$\text{Terms} \qquad L, M, N \quad ::= \quad \cdots \mid \lambda^{\bullet}x. \, Q$$

$$\text{Commands} \quad P, Q, R \quad ::= \quad L \bullet M \mid [M] \mid \mathbf{let} \; x \Leftarrow P \; \mathbf{in} \; Q$$

Types:

$$\frac{\Gamma; \, x : A \vdash Q \, ! \, B}{\Gamma \vdash \lambda^{\bullet}x. \, Q : A \rightsquigarrow B} \qquad\qquad \frac{\Gamma \vdash L : A \rightsquigarrow B \qquad \Gamma, \Delta \vdash M : A}{\Gamma; \, \Delta \vdash L \bullet M \, ! \, B}$$

$$\frac{\Gamma, \Delta \vdash M : A}{\Gamma; \, \Delta \vdash [M] \, ! \, A} \qquad\qquad \frac{\Gamma; \, \Delta \vdash P \, ! \, A \qquad \Gamma; \, \Delta, \, x : A \vdash Q \, ! \, B}{\Gamma; \, \Delta \vdash \mathbf{let} \; x \Leftarrow P \; \mathbf{in} \; Q \, ! \, B}$$

Laws:

$$
\begin{array}{lrcl}
(\beta^{\rightsquigarrow}) & (\lambda^{\bullet}x. \, Q) \bullet M & = & Q[x := M] \\
(\eta^{\rightsquigarrow}) & \lambda^{\bullet}x. \, (L \bullet x) & = & L \\
(\text{left}) & \mathbf{let} \; x \Leftarrow [M] \; \mathbf{in} \; Q & = & Q[x := M] \\
(\text{right}) & \mathbf{let} \; x \Leftarrow P \; \mathbf{in} \; [x] & = & P \\
(\text{assoc}) & \mathbf{let} \; y \Leftarrow \mathbf{let} \; x{\Leftarrow}P \; \mathbf{in} \; Q \; \mathbf{in} \; R & = & \mathbf{let} \; x{\Leftarrow}P \; \mathbf{in} \; \mathbf{let} \; y{\Leftarrow}Q \; \mathbf{in} \; R
\end{array}
$$

Figure 2.6: The arrow calculus, $\mathcal{A}$ (extends $\lambda^{\rightarrow \times 1}$, Figure 2.4).

That is, the command P denotes an arrow, taking argument of type Δ and returning a result of type A. We explain this translation further in Section 2.3.3.

Here are the type rules for the four constructs. Arrow abstraction converts a command into a term.

$$\frac{\Gamma;\, x : A \vdash Q \mathbin{!} B}{\Gamma \vdash \lambda^{\bullet} x.\, Q : A \rightsquigarrow B}$$

Arrow abstraction closely resembles function abstraction, save that the body Q is a command (rather than a term) and the bound variable x goes into the second environment (separated from the first by a semicolon).

Conversely, arrow application builds a command from two terms.

$$\frac{\Gamma \vdash L : A \rightsquigarrow B \qquad \Gamma, \Delta \vdash M : A}{\Gamma;\, \Delta \vdash L \bullet M \mathbin{!} B}$$

Arrow application closely resembles function application. The argument term may contain variables from Δ, but the term denoting the arrow to be applied may not; this is because there is no way to apply an arrow that is itself yielded by another arrow. It is for this reason that we distinguish two environments, Γ and Δ: only variables in Γ, not variables in Δ, may be used to compute arrows that are applied to arguments.

Arrow unit promotes a term to a command.

$$\frac{\Gamma, \Delta \vdash M : A}{\Gamma;\, \Delta \vdash [M] \mathbin{!} A}$$

Note that in the hypothesis we have a term judgment with one environment (there is a comma between Γ and Δ), while in the conclusion we have a command judgment with two environments (there is a semicolon between Γ and Δ). This is the analogue of promotion of a function to an arrow in the classic formulation. It also resembles the unit of a monad.

Lastly, the value returned by a command may be bound.

$$\frac{\Gamma;\, \Delta \vdash P \mathbin{!} A \qquad \Gamma;\, \Delta,\, x : A \vdash Q \mathbin{!} B}{\Gamma;\, \Delta \vdash \mathtt{let}\ x \Leftarrow P\ \mathtt{in}\ Q \mathbin{!} B}$$

This resembles a traditional **let** term, save that the bound variable goes into the second environment, not the first. This is the analogue of arrow composition in the classic formulation; it also embodies the operation `first`. It also resembles the bind of a monad.

Arrow abstraction and application satisfy beta and eta laws, $(\beta^{\rightsquigarrow})$ and $(\eta^{\rightsquigarrow})$, while arrow unit and bind satisfy left unit, right unit, and associativity laws, (left), (right), and (assoc). Similar laws appear in the computational metalanguage of Moggi (1991).

Our notation is closely related to that of Paterson (2001). Here is a translation table, with our notation on the left and his on the right.

$\lambda^{\bullet}x.\, Q$	$\mathbf{proc}\, x \rightarrow Q$
$L \bullet M$	$L \prec M$
$[M]$	$\mathtt{arrowA} \prec M$
$\mathbf{let}\, x \Leftarrow P\, \mathbf{in}\, Q$	$\mathbf{do}\, \{x \leftarrow P; Q\}$

In essence, each is a minor syntactic variant of the other. The only difference of note is that we introduce arrow unit as an explicit construct, $[M]$, while Paterson uses the equivalent form $\mathtt{arrowA} \prec M$ where $\mathtt{arrowA}$ is $\mathtt{arr\ id}$. Our introduction of a separate construct for arrow unit is slightly neater, and avoids the need to introduce $\mathtt{arrowA}$ as a constant in the arrow calculus.

2.3.3 Translations

We now consider translations between the two formulations of arrows, and show they are equivalent. This goes further than Paterson (2001). Paterson provides a notation for arrows that is easier to read and to write and shows how to translate it into the classic formulation. Naturally, this requires that the five laws governing the notation follow from the nine laws governing classic arrows. We wish to use the arrow calculus not merely as a convenient notation, but as a complete replacement for the classic formulation. To justify replacing the classic formulation we show that it can be translated into the arrow calculus and that the nine laws governing classic arrows follow from the five laws governing the arrow calculus.

We begin by defining various notions of equivalence between theories. The strongest notion we shall need is Sabry and Felleisen's (1993) *equational correspondence*, which we present as a special case of a weaker notion, *equational equivalence*.

Definition 2. *Let* T *be an equational theory with typing judgments* $x : A \vdash_{\mathsf{T}} f : B$ *and* $x : B \vdash_{\mathsf{T}} f^{-1} : A$. *(These typing judgments can be viewed as translations on terms:* f *from* A *to* B *and* f^{-1} *from* B *to* A. *For convenience, we write* $f(M)$ *for* $f[x := M]$ *and* $f^{-1}(N)$ *for* $f^{-1}[x := N]$.) *We say that* A *is isomorphic to* B *and* f, f^{-1} *witness the isomorphism* $(f : A \simeq B)$ *if*

- *Translating from* A *to* B *and back is the identity,*

$$\Gamma \vdash_{\mathsf{T}} M : A \;\; implies \;\; \Gamma \vdash_{\mathsf{T}} f^{-1}(f(M)) = M : A$$

for all Γ, M, A *in* T.

- *Translating from* B *to* A *and back is the identity,*

$$\Gamma \vdash_\mathsf{T} N : B \;\; \textit{implies} \;\; \Gamma \vdash_\mathsf{T} f(f^{-1}(N)) = N : B$$

for all Γ, N, B *in* T.

As all of the theories we consider include function types and lambda abstractions, we choose to express the isomorphisms more concisely as pairs of closed terms $f : A \to B$ and $f^{-1} : B \to A$ rather than typing judgments $x : A \vdash_\mathsf{T} f : B$ and $x : B \vdash_\mathsf{T} f^{-1} : A$.

Definition 3. *Let* S, T *be equational theories, with a translation on types and terms* $[\![-]\!]$ *from* S *to* T. *We say that* $[\![-]\!]$ preserves typing *if*

$$\Gamma \vdash_\mathsf{S} M : A \;\; \textit{implies} \;\; [\![\Gamma]\!] \vdash_\mathsf{T} [\![M]\!] : [\![A]\!]$$

for all Γ, M, A *in* S.

Definition 4. *Let* S, T *be equational theories, with a translation on types and terms* $[\![-]\!]$ *from* S *to* T *that preserves typing. We say that* $[\![-]\!]$ *is* compositional *if*

$$\Gamma \vdash_\mathsf{S} N : A \;\; \textit{and} \;\; \Gamma, x : A \vdash_\mathsf{S} M : B$$

$$\textit{implies}$$

$$[\![\Gamma]\!] \vdash_\mathsf{T} [\![M[x := N]]\!] = [\![M]\!][x := [\![N]\!]] : [\![B]\!]$$

for all Γ, M, N, A, B *in* S.

Definition 5. *Let* S, T *be equational theories, with a compositional translation on terms and types* $[\![-]\!]$ *from* S *to* T *that preserves typing, and with a compositional inverse translation* $\langle\!\langle-\rangle\!\rangle$ *from* T *to* S *that also preserves typing. Further, translating a type from* S *to* T *and back yields a type isomorphic to the original type,*

$$f_A : A \simeq \langle\!\langle[\![A]\!]\rangle\!\rangle$$

for all A *in* S. *Similarly, translating a type from* T *to* S *and back yields a type isomorphic to the original type,*

$$g_A : A \simeq [\![\langle\!\langle A\rangle\!\rangle]\!]$$

for all A *in* T. *We say these translations form an* equational equivalence *(*$\mathsf{S} \sim \mathsf{T}$*) if*

- *The translation from S to T preserves equations,*

$$\Gamma \vdash_S M = N : A \quad \textit{implies} \quad [\![\Gamma]\!] \vdash_T [\![M]\!] = [\![N]\!] : [\![A]\!]$$

  for all Γ, M, N, A in S.

- *The translation from T to S preserves equations,*

$$\Gamma \vdash_T M = N : A \quad \textit{implies} \quad \langle\!|\Gamma|\!\rangle \vdash_S \langle\!|M|\!\rangle = \langle\!|N|\!\rangle : \langle\!|A|\!\rangle$$

  for all Γ, M, N, A in T.

- *Translating from S to T and back yields a term isomorphic to the original term,*

$$\Gamma \vdash_S M : A \quad \textit{implies} \quad \Gamma \vdash_S \langle\!|[\![M]\!]|\!\rangle[\Gamma := f(\Gamma)] = f_A(M) : \langle\!|[\![A]\!]|\!\rangle$$

  for all Γ, M, A in S (writing $N[\Gamma := f(\Gamma)]$ for $N[x_1 := f_{A_1}(x_1), \ldots, x_n := f_{A_n}(x_n)]$, given $\Gamma = x_1 : A_1, \ldots, x_n : A_n$).

- *Translating from T to S and back yields a term isomorphic to the original term,*

$$\Gamma \vdash_T M : A \quad \textit{implies} \quad \Gamma \vdash_T [\![\langle\!|M|\!\rangle]\!][\Gamma := g(\Gamma)] = g_A(M) : [\![\langle\!|A|\!\rangle]\!]$$

  for all Γ, M, A in T.

(This definition is analogous to saying that we have an equivalence of categories (Mac Lane, 1998), where $[\![-]\!]$ is left adjoint to $\langle\!|-|\!\rangle$ with unit f_A and counit g_A^{-1}.)

The special case of an equational equivalence where both isomorphisms are the identity is an *equational correspondence*.

Definition 6. *An* equational correspondence *between theories S and T ($S \cong T$) is an equational equivalence with translations $[\![-]\!] : S \to T, \langle\!|-|\!\rangle : T \to S$ where both f_A and g_A are the identity at each type A. (This is analogous to saying that we have an isomorphism of categories (Mac Lane, 1998) given by the translations $[\![-]\!] : S \to T$ and $\langle\!|-|\!\rangle : T \to S$.)*

We also introduce the notion of *equational embedding*, a map from a weaker into a stronger theory. An equational embedding of a theory S into a theory T may be defined as an equational equivalence between S and a *subtheory* of T. We instead use the following more direct definition, which is more convenient in practice.

Definition 7. *Let S, T be equational theories with a compositional translation on terms and types $[\![-]\!]$ from S to T that preserves typing, and with a compositional inverse translation $\langle\!|-|\!\rangle$ from the image of $[\![-]\!]$ (written $[\![S]\!]$) to S that also preserves typing,*

$$[\![\Gamma]\!] \vdash_T [\![M]\!] : [\![A]\!] \quad \textit{implies} \quad \langle\!|[\![\Gamma]\!]|\!\rangle \vdash_S \langle\!|[\![M]\!]|\!\rangle : \langle\!|[\![A]\!]|\!\rangle$$

for all Γ, M, A *in* S. *Further, translating a type from* S *to* T *and back yields a type isomorphic to the original type,*

$$f_A : A \simeq \langle\![[\![A]\!]]\!\rangle$$

for all A *in* S. *We say these translations form an* equational embedding *of* S *into* T *(*S $\hookrightarrow$ T*) if*

- *The translation from* S *to* T *preserves equations,*

$$\Gamma \vdash_S M = N : A \;\; implies \;\; [\![\Gamma]\!] \vdash_T [\![M]\!] = [\![N]\!] : [\![A]\!]$$

  for all Γ, M, N, A *in* S.

- *The translation from* $[\![S]\!]$ *to* S *preserves equations,*

$$[\![\Gamma]\!] \vdash_T [\![M]\!] = [\![N]\!] : [\![A]\!] \;\; implies \;\; \langle\![[\![\Gamma]\!]]\!\rangle \vdash_S \langle\![[\![M]\!]]\!\rangle = \langle\![[\![N]\!]]\!\rangle : \langle\![[\![A]\!]]\!\rangle$$

  for all Γ, M, N, A *in* S.

- *Translating from* S *to* $[\![S]\!]$ *and back yields a term isomorphic to the original term,*

$$\Gamma \vdash_S M : A \;\; implies \;\; \Gamma \vdash_S \langle\![[\![M]\!]]\!\rangle [\Gamma := f(\Gamma)] = f_A(M) : \langle\![[\![A]\!]]\!\rangle$$

  for all Γ, M, A *in* S.

Figure 2.7 shows the translation from the arrow calculus into classic arrows. An arrow calculus term judgment

$$\Gamma \vdash M : A$$

maps into a classic arrow judgment

$$\Gamma \vdash [\![M]\!] : A$$

while an arrow calculus command judgment

$$\Gamma; \Delta \vdash P \,!\, A$$

maps into a classic arrow judgment

$$\Gamma \vdash [\![P]\!]_\Delta : \Delta \rightsquigarrow A.$$

In $[\![P]\!]_\Delta$, we take Δ to stand for the sequence of variables in the environment, and in $\Delta \rightsquigarrow A$ we take Δ to stand for the product of the types in the environment. Hence, the denotation of a command is an arrow, with arguments corresponding to the environment Δ and result of type A.

The translation of the constructs of the core lambda calculus are straightforward homomorphisms. The translations of the remaining four constructs are shown twice, in the top half of the figure as equations on syntax, and in the bottom half in the context of type derivations; the latter are longer, but may be clearer to read. We comment briefly on each of the four:

Translation of terms:

$$\llbracket x \rrbracket \;=\; x$$
$$\llbracket (M, N) \rrbracket \;=\; (\llbracket M \rrbracket, \llbracket N \rrbracket)$$
$$\llbracket \mathbf{fst}\, L \rrbracket \;=\; \mathbf{fst}\, \llbracket L \rrbracket$$
$$\llbracket \mathbf{snd}\, L \rrbracket \;=\; \mathbf{snd}\, \llbracket L \rrbracket$$
$$\llbracket \lambda x.\, N \rrbracket \;=\; \lambda x.\, \llbracket N \rrbracket$$
$$\llbracket L\, M \rrbracket \;=\; \llbracket L \rrbracket\, \llbracket M \rrbracket$$
$$\llbracket \lambda^{\bullet} x.\, Q \rrbracket \;=\; \llbracket Q \rrbracket_x$$

Translation of commands:

$$\llbracket L \bullet M \rrbracket_\Delta \;=\; \mathtt{arr}\,(\lambda\Delta.\,\llbracket M \rrbracket) \ggg \llbracket L \rrbracket$$
$$\llbracket [M] \rrbracket_\Delta \;=\; \mathtt{arr}\,(\lambda\Delta.\,\llbracket M \rrbracket)$$
$$\llbracket \mathbf{let}\ x \Leftarrow P\ \mathbf{in}\ Q \rrbracket_\Delta \;=\; (\mathtt{arr\ id}\,\&\!\&\!\&\,\llbracket P \rrbracket_\Delta) \ggg \llbracket Q \rrbracket_{\Delta,x}$$

Translation preserves types:

$$\left\llbracket \dfrac{\Gamma;\, x : A \vdash Q\;!\;B}{\Gamma \vdash \lambda^{\bullet} x.\, Q : A \rightsquigarrow B} \right\rrbracket \;=\; \dfrac{\Gamma \vdash \llbracket Q \rrbracket_x : A \rightsquigarrow B}{\Gamma \vdash \llbracket Q \rrbracket_x : A \rightsquigarrow B}$$

$$\left\llbracket \begin{array}{c} \Gamma \vdash L : A \rightsquigarrow B \\ \Gamma, \Delta \vdash M : A \\ \hline \Gamma;\, \Delta \vdash L \bullet M\;!\;B \end{array} \right\rrbracket \;=\; \begin{array}{c} \Gamma \vdash \llbracket L \rrbracket : A \rightsquigarrow B \\ \Gamma, \Delta \vdash \llbracket M \rrbracket : A \\ \hline \Gamma \vdash \mathtt{arr}\,(\lambda\Delta.\,\llbracket M \rrbracket) \ggg \llbracket L \rrbracket : \Delta \rightsquigarrow B \end{array}$$

$$\left\llbracket \dfrac{\Gamma, \Delta \vdash M : A}{\Gamma;\, \Delta \vdash [M]\;!\;A} \right\rrbracket \;=\; \dfrac{\Gamma, \Delta \vdash \llbracket M \rrbracket : A}{\Gamma \vdash \mathtt{arr}\,(\lambda\Delta.\,\llbracket M \rrbracket) : \Delta \rightsquigarrow A}$$

$$\left\llbracket \begin{array}{c} \Gamma;\, \Delta \vdash P\;!\;A \\ \Gamma;\, \Delta,\, x : A \vdash Q\;!\;B \\ \hline \Gamma;\, \Delta \vdash \mathbf{let}\ x \Leftarrow P\ \mathbf{in}\ Q\;!\;B \end{array} \right\rrbracket \;=\; \begin{array}{c} \Gamma \vdash \llbracket P \rrbracket_\Delta : \Delta \rightsquigarrow A \\ \Gamma \vdash \llbracket Q \rrbracket_{\Delta,x} : \Delta{\times}A \rightsquigarrow B \\ \hline \Gamma \vdash (\mathtt{arr\ id}\,\&\!\&\!\&\,\llbracket P \rrbracket_\Delta) \ggg \llbracket Q \rrbracket_{\Delta,x} : \Delta \rightsquigarrow B \end{array}$$

Figure 2.7: Translating $\mathcal{A}$ into $\mathcal{C}$.

Translation:

$$
\begin{aligned}
\langle\!\langle \mathrm{x} \rangle\!\rangle &= \mathrm{x} \\
\langle\!\langle (M, N) \rangle\!\rangle &= (\langle\!\langle M \rangle\!\rangle, \langle\!\langle N \rangle\!\rangle) \\
\langle\!\langle \mathbf{fst}\, L \rangle\!\rangle &= \mathbf{fst}\, \langle\!\langle L \rangle\!\rangle \\
\langle\!\langle \mathbf{snd}\, L \rangle\!\rangle &= \mathbf{snd}\, \langle\!\langle L \rangle\!\rangle \\
\langle\!\langle \lambda\mathrm{x}.\, N \rangle\!\rangle &= \lambda\mathrm{x}.\, \langle\!\langle N \rangle\!\rangle \\
\langle\!\langle L\, M \rangle\!\rangle &= \langle\!\langle L \rangle\!\rangle \, \langle\!\langle M \rangle\!\rangle \\
\langle\!\langle \mathtt{arr} \rangle\!\rangle &= \lambda \mathrm{f}.\, \lambda^{\bullet}\mathrm{x}.\, [\mathrm{f}\ \mathrm{x}] \\
\langle\!\langle (\ggg) \rangle\!\rangle &= \lambda \mathrm{f}.\, \lambda \mathrm{g}.\, \lambda^{\bullet}\mathrm{x}.\, \mathbf{let}\ \mathrm{y} \Leftarrow (\mathrm{f} \bullet \mathrm{x})\ \mathbf{in}\ \mathrm{g} \bullet \mathrm{y} \\
\langle\!\langle \mathtt{first} \rangle\!\rangle &= \lambda \mathrm{f}.\, \lambda^{\bullet}\mathrm{z}.\, \mathbf{let}\ \mathrm{x} \Leftarrow \mathrm{f} \bullet \mathbf{fst}\, \mathrm{z}\ \mathbf{in}\ [\langle \mathrm{x}, \mathbf{snd}\, \mathrm{z} \rangle]
\end{aligned}
$$

Figure 2.8: Translating $\mathcal{C}$ into $\mathcal{A}$.
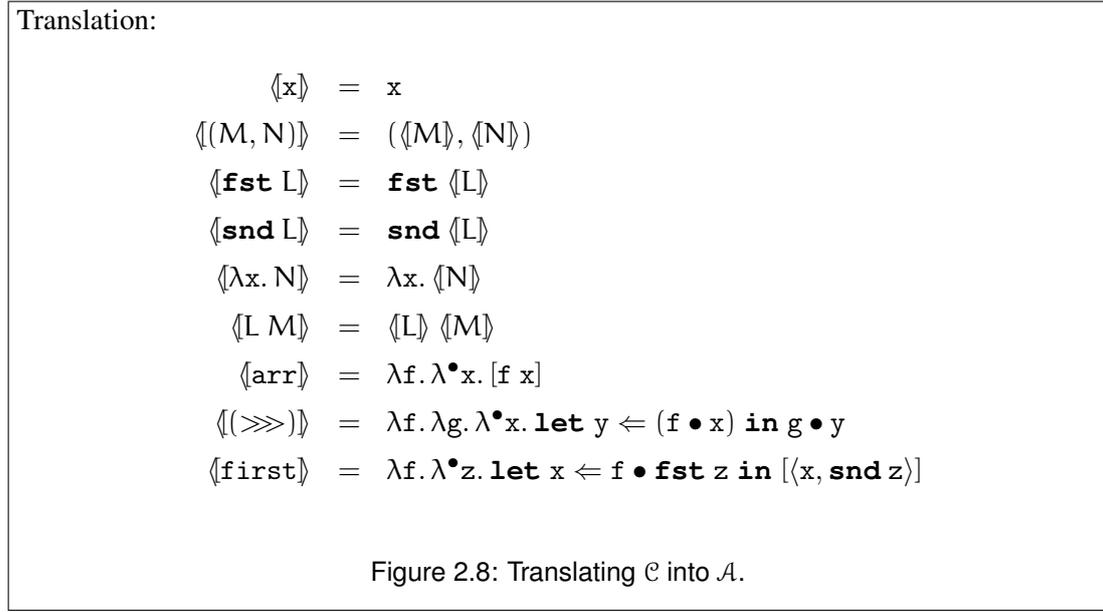
- $\lambda^{\bullet}\mathrm{x}.\, N$ translates straightforwardly; it is a no-op.

- $L \bullet M$ translates to $\ggg$.

- $[M]$ translates to $\mathtt{arr}$.

- $\mathbf{let}\ \mathrm{x} \Leftarrow P\ \mathbf{in}\ Q$ translates to pairing $\&\!\&\!\&$ (to extend the environment with P) and composition $\ggg$ (to then apply Q).

The translation uses the notation $\lambda\Delta.\, N$, which is given the obvious meaning: $\lambda\mathrm{x}.\, N$ stands for itself, and $\lambda\mathrm{x}_1, \mathrm{x}_2.\, N$ stands for $\lambda\mathrm{z}.\, N[\mathrm{x}_1 := \mathbf{fst}\, \mathrm{z}, \mathrm{x}_2 := \mathbf{snd}\, \mathrm{z}]$, and $\lambda\mathrm{x}_1, \mathrm{x}_2, \mathrm{x}_3.\, N$ stands for $\lambda\mathrm{z}.\, N[\mathrm{x}_1 := \mathbf{fst}\, (\mathbf{fst}\, \mathrm{z}), \mathrm{x}_2 := \mathbf{snd}\, (\mathbf{fst}\, \mathrm{z}), \mathrm{x}_3 := \mathbf{snd}\, \mathrm{z}]$, and so on. (This definition assumes a non-empty environment, and in fact Δ is always non-empty in this translation. However, it is not always non-empty in the related translations in Section 2.4. It is straightforward to adjust the definitions here to support the empty-environment case: for example, we might take the empty environment $\cdot$ to stand for the unit type when used as a type, and take $\lambda\mathrm{x}_1, \mathrm{x}_2.\, N$ to stand for $\lambda\mathrm{z}.\, N[\mathrm{x}_1 := \mathbf{snd}\, (\mathbf{fst}\, \mathrm{z}), \mathrm{x}_2 := \mathbf{snd}\, \mathrm{z}]$, and so on. A second possibility is to define the translation differently for empty and non-empty environments, retaining the translation we have given for the case where the environment is non-empty. A third possibility is to ensure via weakening (which we will define in Lemma 9) that the environment is never actually empty. Rather than obscure the presentation with these details we will retain our existing definitions, on the understanding that they can be easily adjusted to support the empty-environment case where necessary.)

The inverse translation, from classic arrows to the arrow calculus, is given in Figure 2.8. Again, the translation of the constructs of the core lambda calculus are straightforward homomorphisms. Each of the three constants translates to an appropriate term in the arrow calculus.

Demonstrating the equivalence of the two formulations of arrows requires that we translate the laws. In order to translate the $(\beta^{\leadsto})$ and (left) laws we must determine the meaning of translation on a substitution, which we do with the following lemma.

Lemma 8. *[Translating substitution from $\mathcal{A}$ to $\mathcal{C}$]*

The translations of substitution on terms and commands from $\mathcal{A}$ to $\mathcal{C}$ satisfy the following equations.

$$[\![M[x := N]]\!] = [\![M]\!][x := [\![N]\!]]$$

$$[\![P[x := N]]\!]_\Delta = \texttt{arr}\ (\lambda\Delta.(\Delta, [\![N]\!])) \ggg [\![P]\!]_{\Delta,x}$$

Proof. By mutual induction on the derivations of P and M. There is one case for each term form and each command form. Appendix A.2 gives the full proof. $\square$

Now consider the translation of the (assoc) law. The left side translates as follows:

$$[\![\textbf{let}\ y \Leftarrow (\textbf{let}\ x \Leftarrow P\ \textbf{in}\ Q)\ \textbf{in}\ R]\!]_\Delta$$
$$=\ (\texttt{arr id} \&\&\& ((\texttt{arr id} \&\&\& [\![P]\!]_\Delta) \ggg [\![Q]\!]_{\Delta,x})) \ggg [\![R]\!]_{\Delta,x}$$

and the right side translates as follows:

$$[\![\textbf{let}\ x \Leftarrow P\ \textbf{in}\ (\textbf{let}\ y \Leftarrow Q\ \textbf{in}\ R)]\!]_\Delta$$
$$=\ (\texttt{arr id} \&\&\& [\![P]\!]_\Delta) \ggg ((\texttt{arr id} \&\&\& [\![Q]\!]_{\Delta,x}) \ggg [\![R]\!]_{\Delta,x,y})$$

Notice that the left side contains the term $[\![R]\!]_{\Delta,x}$ — the translation of the command R in the environment Δ, x — while the right side contains the term $[\![R]\!]_{\Delta,x,y}$ — the translation of R in the environment Δ, x, y. The following two lemmas define *weakening* in the arrow calculus and its effects of the translation, allowing us to establish the equivalence of the two sides.

Lemma 9 (Weakening). *Suppose type environments $\Gamma, \Delta, \Gamma', \Delta'$, such that $\Gamma \subseteq \Gamma'$ and $\Gamma, \Delta \subseteq \Gamma', \Delta'$. Then if*

$$\Gamma \vdash M : A$$

is derivable for some type A then

$$\Gamma' \vdash M : A$$

is also derivable, and if

$$\Gamma; \Delta \vdash P\ !\ B$$

is derivable for some type B *then*

$$\Gamma'; \Delta' \vdash P \mathbin{!} B$$

is also derivable.

Proof. By a straightforward mutual induction on the derivations of P and M. □

Weakening allows us to move variables from Δ into Γ, and to increase both Γ and Δ, but not to move variables from Γ into Δ. The $\subseteq$ relation used to define weakening ignores variable order, so our definition also permits us to permute the environment.

Lemma 10. *[Translating weakening from $\mathcal{A}$ to $\mathcal{C}$]*

The translation of weakening from $\mathcal{A}$ to $\mathcal{C}$ for commands is as follows.

$$\left[\!\!\left[\dfrac{\Gamma; \Delta \vdash Q \mathbin{!} B}{\Gamma'; \Delta' \vdash Q \mathbin{!} B} \right]\!\!\right] \quad = \quad \dfrac{\Gamma \vdash [\![Q]\!]_\Delta : \Delta \rightsquigarrow B}{\Gamma' \vdash \mathtt{arr}\,(\lambda\Delta'.\Delta) \ggg [\![Q]\!]_\Delta : \Delta' \rightsquigarrow B}$$

Proof. By induction on the derivation of Q. There is one case for each command form. Appendix A.2 gives the full proof. □

Proposition 11. *The theory of classic arrows and the theory of the arrow calculus are in equational correspondence: $\mathcal{A} \cong \mathcal{C}$.*

Proof. In order to show the equational correspondence between $\mathcal{A}$ and $\mathcal{C}$ we must show that the translations between them satisfy the following four properties.

- The five laws of the arrow calculus follow from the nine laws of classic arrows. That is,

$$M = N \ \text{ implies } \ [\![M]\!] = [\![N]\!]$$

and

$$P = Q \ \text{ implies } \ [\![P]\!]_\Delta = [\![Q]\!]_\Delta$$

for all arrow calculus terms M, N and commands P, Q.

The proof requires five calculations, one for each law of the arrow calculus. Figure 2.9 shows one of these, the calculation to derive (right) from the classic arrow laws.

- The nine laws of classic arrows follow from the five laws of the arrow calculus. That is,

$$M = N \ \text{ implies } \ \langle\!\langle M \rangle\!\rangle = \langle\!\langle N \rangle\!\rangle$$

for all classic arrow terms M, N.

The proof requires nine calculations, one for each classic arrow law. Figure 2.10 shows one of these, the calculation to derive $(\rightsquigarrow_2)$ from the laws of the arrow calculus.

$\llbracket \textbf{let}\ x \Leftarrow M\ \textbf{in}\ [x] \rrbracket_\Delta$

$=$ (def $\llbracket - \rrbracket$)

$(\texttt{arr id}\ \&\&\&\ \llbracket M \rrbracket_\Delta) \ggg \texttt{arr snd}$

$=$ (def $\&\&\&$)

$\texttt{arr dup} \ggg \texttt{first (arr id)} \ggg \texttt{second}\ \llbracket M \rrbracket_\Delta \ggg \texttt{arr snd}$

$=$ $(\leadsto_5)$

$\texttt{arr dup} \ggg \texttt{arr (id}\times\texttt{id)} \ggg \texttt{second}\ \llbracket M \rrbracket_\Delta \ggg \texttt{arr snd}$

$=$ $(\texttt{id}\times\texttt{id} = \texttt{id})$

$\texttt{arr dup} \ggg \texttt{arr id} \ggg \texttt{second}\ \llbracket M \rrbracket_\Delta \ggg \texttt{arr snd}$

$=$ $(\leadsto_1)$

$\texttt{arr dup} \ggg \texttt{second}\ \llbracket M \rrbracket_\Delta \ggg \texttt{arr snd}$

$=$ def second

$\texttt{arr dup} \ggg \texttt{arr swap} \ggg \texttt{first}\ \llbracket M \rrbracket_\Delta \ggg \texttt{arr swap} \ggg \texttt{arr snd}$

$=$ $(\leadsto_4)$

$\texttt{arr (swap}\cdot\texttt{dup)} \ggg \texttt{first}\ \llbracket M \rrbracket_\Delta \ggg \texttt{arr (snd}\cdot\texttt{swap)}$

$=$ $(\texttt{swap}\cdot\texttt{dup} = \texttt{dup},\ \texttt{snd}\cdot\texttt{swap} = \texttt{fst})$

$\texttt{arr dup} \ggg \texttt{first}\ \llbracket M \rrbracket_\Delta \ggg \texttt{arr fst}$

$=$ $(\leadsto_8)$

$\texttt{arr dup} \ggg \texttt{arr fst} \ggg \llbracket M \rrbracket_\Delta$

$=$ $(\leadsto_4)$

$\texttt{arr (fst}\cdot\texttt{dup)} \ggg \llbracket M \rrbracket_\Delta$

$=$ $(\texttt{fst}\cdot\texttt{dup} = \texttt{id})$

$\texttt{arr id} \ggg \llbracket M \rrbracket_\Delta$

$=$ $(\leadsto_1)$

$\llbracket M \rrbracket_\Delta$

Figure 2.9: Proof of (right) in $\mathcal{C}$.

- Translating a term from the arrow calculus into classic arrows and back again is the identity (up to equivalence). That is,

$$\langle\!\langle\, \llbracket M \rrbracket \,\rangle\!\rangle = M$$

for all arrow calculus terms M. Translating a command of the arrow calculus into classic

$\langle\!\langle f \ggg \mathtt{arr\ id} \rangle\!\rangle$

$=$ $(\mathrm{def}\ \langle\!\langle - \rangle\!\rangle)$

$\lambda^\bullet x.\,(\mathbf{let}\ y \Leftarrow \langle\!\langle f \rangle\!\rangle \bullet x\ \mathbf{in}\ (\lambda^\bullet z.\,[\mathtt{id}\ z]) \bullet y)$

$=$ $(\beta^\rightarrow)$

$\lambda^\bullet x.\,(\mathbf{let}\ y \Leftarrow \langle\!\langle f \rangle\!\rangle \bullet x\ \mathbf{in}\ (\lambda^\bullet z.\,[z]) \bullet y)$

$=$ $(\beta^{\rightsquigarrow})$

$\lambda^\bullet x.\,(\mathbf{let}\ y \Leftarrow \langle\!\langle f \rangle\!\rangle \bullet x\ \mathbf{in}\ [y])$

$=$ (right)

$\lambda^\bullet x.\,(\langle\!\langle f \rangle\!\rangle \bullet x)$

$=$ $(\eta^{\rightsquigarrow})$

$\langle\!\langle f \rangle\!\rangle$

Figure 2.10: Proof of $(\rightsquigarrow_2)$ in $\mathcal{A}$.

arrows and back again is the identity (up to equivalence). That is,

$$\langle\!\langle\, [\![P]\!]_\Delta \,\rangle\!\rangle = \lambda^\bullet \Delta.\,P$$

for all arrow calculus commands P.

The proof requires four calculations, one for each construct of the arrow calculus. Figure 2.11 shows one of these, the calculation for arrow application.

- Translating from classic arrows into the arrow calculus and back again is the identity (up to equivalence). That is,

$$[\![\, \langle\!\langle M \rangle\!\rangle \,]\!] = M$$

for all classic arrow terms M.

The proof requires three calculations, one for each classic arrow constant. Figure 2.12 shows one of these, the calculation for $\ggg$.

The remaining cases of the proofs for each of these properties are given in Appendix A.2. The soundness of Paterson's notation depends only on the first of these properties. $\square$

2.3.4   Normal forms

Under the laws of the arrow calculus (including the lambda calculus laws), every arrow calculus command is equivalent to a command of the form

$$\lang\!\langle\, [\![L \bullet M]\!]_\Delta \,\rangle\!\rangle \bullet \Delta$$

$=\qquad$ (def $[\![-]\!]_\Delta$)

$$\langle\!\langle\, \mathtt{arr}\, (\lambda\Delta.\,[\![M]\!]) \ggg [\![L]\!] \,\rangle\!\rangle \bullet \Delta$$

$=\qquad$ (def $\langle\!\langle -\rangle\!\rangle$)

$$\lambda^\bullet\Delta.\,(\mathbf{let}\; y \Leftarrow (\lambda^\bullet z.\,[(\lambda\Delta.\,\langle\!\langle [\![M]\!]\rangle\!\rangle)\, z]) \bullet \Delta\; \mathbf{in}\; \langle\!\langle [\![L]\!]\rangle\!\rangle \bullet y) \bullet \Delta$$

$=\qquad$ (induction hypothesis)

$$\lambda^\bullet\Delta.\,(\mathbf{let}\; y \Leftarrow (\lambda^\bullet z.\,[(\lambda\Delta.\,M)\, z]) \bullet \Delta\; \mathbf{in}\; L \bullet y) \bullet \Delta$$

$=\qquad$ ($\beta^{\leadsto}$)

$$\mathbf{let}\; y \Leftarrow [(\lambda\Delta.\,M)\,\Delta]\; \mathbf{in}\; L \bullet y$$

$=\qquad$ ($\beta^{\to}$)

$$\mathbf{let}\; y \Leftarrow [M]\; \mathbf{in}\; L \bullet y$$

$=\qquad$ (left)

$$L \bullet M$$

Figure 2.11: Translating $L \bullet M$ to $\mathcal{C}$ and back.

$$\begin{aligned}
&\mathbf{let}\;\; x_1 \;\Leftarrow\; c_1 \;\bullet\; M_1 \;\; \mathbf{in}\\
&\dots\\
&\mathbf{let}\;\; x_n \;\Leftarrow\; c_n \;\bullet\; M_n \;\; \mathbf{in}\\
&\quad [N]
\end{aligned}$$

where $c_1,\dots,c_n$ are constants of arrow type and $M_1,\dots,M_n, N$ are in normal form. There is a connection to the normal form for classic arrows given in Section 2.2.5.1: the result of translating an arrow calculus command in normal form is a classic arrow term in normal form, except for the order of the operands to $\&\!\&\!\&$ (which is a matter of mere convenience).

The technical report on arrows (Lindley et al., 2008a) explains how to obtain a rewriting theory for arrow calculus from the equational theory by orienting the rules appropriately. Proofs of confluence and strong normalisation follow from a translation of arrow calculus into Moggi's metalanguage for monads, which is known to be strongly normalising (Lindley and Stark, 2005).

2.3.5   Redundancy of the second law

In Section 2.2.5.1 we saw that classic arrow terms can be normalised using eight of the nine laws: the right unit law ($\leadsto_2$) is not needed. The proof of the equational correspondence

$\llbracket \langle\!\langle (\ggg) \rangle\!\rangle \rrbracket$

$=$ (def $\langle\!\langle - \rangle\!\rangle$, $\beta^{\rightarrow}$)

$\llbracket \lambda\mathrm{f}.\,\lambda\mathrm{g}.\ \lambda^{\bullet}\mathrm{x}.\,(\textbf{let } \mathrm{y} \Leftarrow \mathrm{f} \bullet \mathrm{x} \textbf{ in } \mathrm{g} \bullet \mathrm{y}) \rrbracket$

$=$ (def $\llbracket - \rrbracket$, $(\leadsto_1)$)

$\lambda\mathrm{f}.\,\lambda\mathrm{g}.\,(\texttt{arr id \&\&\& f}) \ggg (\texttt{arr snd} \ggg \mathrm{g})$

$=$ (def $\&\&\&$)

$\lambda\mathrm{f}.\,\lambda\mathrm{g}.\ \texttt{arr dup} \ggg \texttt{first}\,(\texttt{arr id}) \ggg \texttt{arr swap} \ggg \texttt{first f} \ggg$
     $\texttt{arr swap} \ggg (\texttt{arr snd} \ggg \mathrm{g})$

$=$ $(\leadsto_5)$

$\lambda\mathrm{f}.\,\lambda\mathrm{g}.\ \texttt{arr dup} \ggg \texttt{arr}\,(\texttt{id}\times\texttt{id}) \ggg \texttt{arr swap} \ggg \texttt{first f} \ggg$
     $\texttt{arr swap} \ggg (\texttt{arr snd} \ggg \mathrm{g})$

$=$ $(\texttt{id}\times\texttt{id} = \texttt{id})$

$\lambda\mathrm{f}.\,\lambda\mathrm{g}.\ \texttt{arr dup} \ggg \texttt{arr id} \ggg \texttt{arr swap} \ggg \texttt{first f} \ggg$
     $\texttt{arr swap} \ggg (\texttt{arr snd} \ggg \mathrm{g})$

$=$ $(\leadsto_2)$

$\lambda\mathrm{f}.\,\lambda\mathrm{g}.\ \texttt{arr dup} \ggg \texttt{arr swap} \ggg \texttt{first f} \ggg$
     $\texttt{arr swap} \ggg (\texttt{arr snd} \ggg \mathrm{g})$

$=$ $(\leadsto_4, \texttt{swap} \cdot \texttt{dup} = \texttt{dup})$

$\lambda\mathrm{f}.\,\lambda\mathrm{g}.\ \texttt{arr dup} \ggg \texttt{first f} \ggg \texttt{arr swap} \ggg (\texttt{arr snd} \ggg \mathrm{g})$

$=$ $(\leadsto_3)$

$\lambda\mathrm{f}.\,\lambda\mathrm{g}.\ \texttt{arr dup} \ggg \texttt{first f} \ggg \texttt{arr swap} \ggg \texttt{arr snd} \ggg \mathrm{g}$

$=$ $(\leadsto_4, \texttt{snd} \cdot \texttt{swap} = \texttt{fst})$

$\lambda\mathrm{f}.\,\lambda\mathrm{g}.\ \texttt{arr dup} \ggg \texttt{first f} \ggg \texttt{arr fst} \ggg \mathrm{g}$

$=$ $(\leadsto_8)$

$\lambda\mathrm{f}.\,\lambda\mathrm{g}.\ \texttt{arr dup} \ggg \texttt{arr fst} \ggg \mathrm{f} \ggg \mathrm{g}$

$=$ $(\leadsto_4, \texttt{fst} \cdot \texttt{dup} = \texttt{id})$

$\lambda\mathrm{f}.\,\lambda\mathrm{g}.\ \texttt{arr id} \ggg \mathrm{f} \ggg \mathrm{g}$

$=$ $(\leadsto_1, \eta^{\rightarrow}(\times 2))$

$(\ggg)$

Figure 2.12: Translating $\ggg$ to $\mathcal{A}$ and back.

$$f \ggg \texttt{arr id}$$

$= \qquad (\leadsto_1)$

$$\texttt{arr id} \ggg f \ggg \texttt{arr id}$$

$= \qquad (\texttt{fst} \cdot \texttt{dup} = \texttt{id})$

$$\texttt{arr} \, (\texttt{fst} \cdot \texttt{dup}) \ggg f \ggg \texttt{arr id}$$

$= \qquad (\leadsto_4)$

$$\texttt{arr dup} \ggg \texttt{arr fst} \ggg f \ggg \texttt{arr id}$$

$= \qquad (\leadsto_8)$

$$\texttt{arr dup} \ggg \texttt{first} \, f \ggg \texttt{arr fst} \ggg \texttt{arr id}$$

$= \qquad (\leadsto_4)$

$$\texttt{arr dup} \ggg \texttt{first} \, f \ggg \texttt{arr} \, (\texttt{id} \cdot \texttt{fst})$$

$= \qquad (\texttt{id} \cdot \texttt{fst} = \texttt{fst})$

$$\texttt{arr dup} \ggg \texttt{first} \, f \ggg \texttt{arr fst}$$

$= \qquad (\leadsto_8)$

$$\texttt{arr dup} \ggg \texttt{arr fst} \ggg f$$

$= \qquad (\leadsto_4)$

$$\texttt{arr} \, (\texttt{fst} \cdot \texttt{dup}) \ggg f$$

$= \qquad (\texttt{fst} \cdot \texttt{dup} = \texttt{id})$

$$\texttt{arr id} \ggg f$$

$= \qquad (\leadsto_1)$

$$f$$

Figure 2.13: The $(\leadsto_2)$ law is redundant.

Syntax:

$$\text{Types} \quad A, B, C \quad ::= \quad \cdots \mid I\,A$$

Constants:

$$\texttt{pure} \ : \ A \to I\,A$$
$$(\circledast) \ : \ I\,(A \to B) \to I\,A \to I\,B$$

Laws:

$$(I_1) \qquad\qquad M \ = \ \texttt{pure id} \circledast M$$
$$(I_2) \quad \texttt{pure}\,M \circledast \texttt{pure}\,N \ = \ \texttt{pure}\,(M\,N)$$
$$(I_3) \qquad L \circledast (M \circledast N) \ = \ \texttt{pure}\,(\cdot) \circledast L \circledast M \circledast N$$
$$(I_4) \qquad M \circledast \texttt{pure}\,N \ = \ \texttt{pure}\,(\lambda\texttt{f}.\,\texttt{f}\,N) \circledast M$$

Figure 2.14: Idioms, $\mathfrak{I}$ (extends $\lambda^{\to\times 1}$, Figure 2.4).
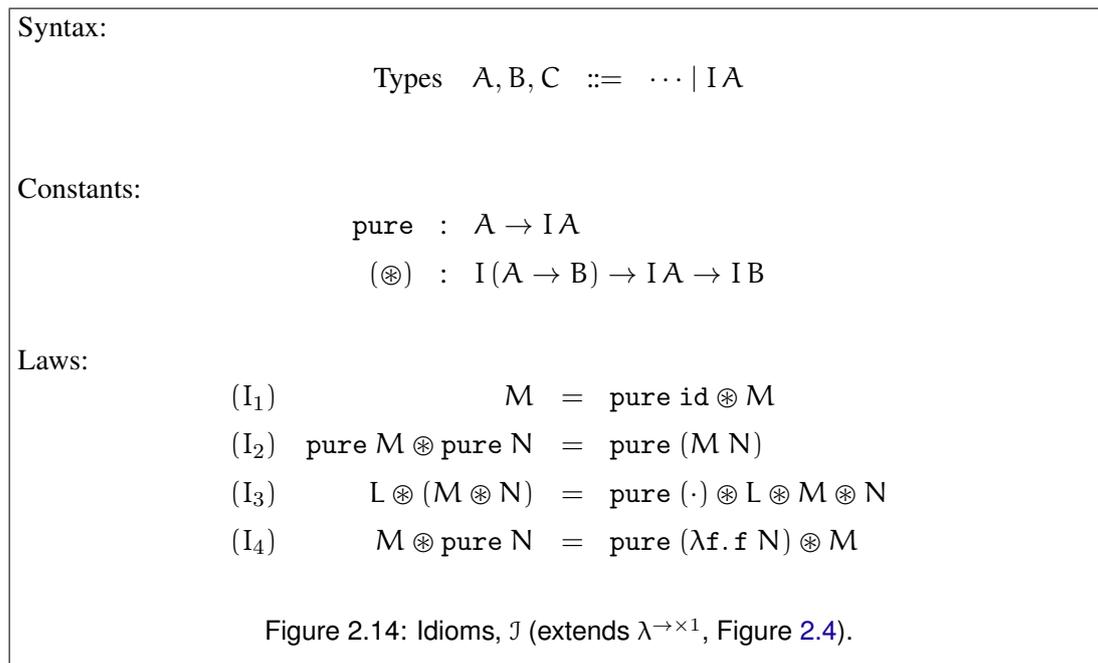
between arrow calculus and classic arrows contains a similar curiosity. From the five laws of the arrow calculus we can prove the nine classic laws, and from *eight* of the nine classic laws we can prove the five laws of the arrow calculus. The missing law is again ($\leadsto_2$), the right unit law for classic arrows. We now have two indirect proofs that this law is redundant. Still, a direct proof is more satisfactory; we give one in Figure 2.13.

2.4 Formal comparison of strength[10]

Having introduced the arrow calculus and shown that it is strictly equivalent to the classic presentation of arrows, we are ready to embark upon the formal comparison with idioms and monads. As we saw in Section 2.3, McBride and Paterson claimed that monads are the strongest of the three interfaces and arrows the weakest. In fact, as illustrated by the normal forms and examples in Sections 2.2.2 and 2.2.3, this is not the case: idioms are weaker than both arrows and monads. In this section we present variations of the arrow calculus that correspond to idioms and to monads and formally establish the correct ordering via a pair of embeddings.

2.4.1 Idioms and arrows

Figure 2.14 defines the theory of *idioms*, $\mathfrak{I}$, a straightforward recapitulation of the Haskell interface presented in Section 2.2.3. Idioms extend $\lambda^{\to\times 1}$ with a unary type constructor I

---

[10]This section is a revision of Lindley et al. (2008b)

Constants:

$$\text{delay} \; : \; (A \rightsquigarrow B) \rightarrow (1 \rightsquigarrow (A \rightarrow B))$$

Definitions:

$$\text{force} : (1 \rightsquigarrow (A \rightarrow B)) \rightarrow (A \rightsquigarrow B)$$

$$\text{force} = \lambda f.\, \text{arr}\, (\lambda x.\, \langle \langle \rangle, x \rangle) \ggg \text{first} \, f \ggg \text{arr}\, (\text{apply})$$

Laws:

$$(\rightsquigarrow_{S1}) \quad \text{force}\, (\text{delay}\, (M)) = M$$

$$(\rightsquigarrow_{S2}) \quad \text{delay}\, (\text{force}\, (M)) = M$$

Figure 2.15: Classic arrows with delay, $\mathcal{C}_S$ (extends $\mathcal{C}$, Figure 2.5).
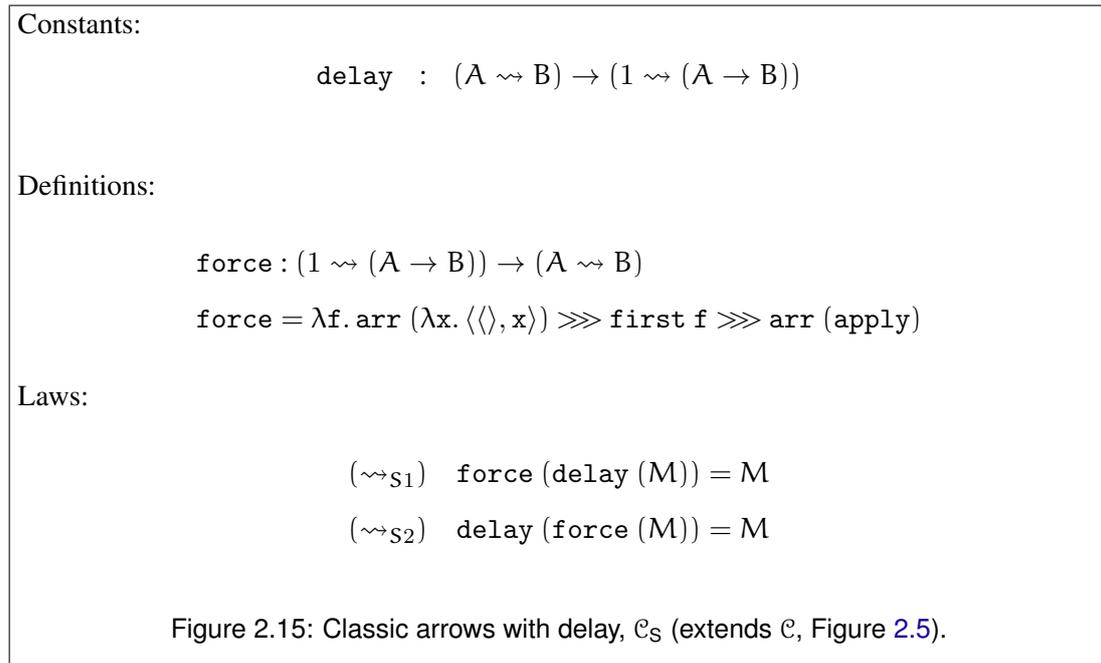
for computations of type $I\,A$ which return a value of type A and two constants, pure and ($\circledast$). There are four laws which, together with the laws of the lambda calculus, define the equivalence relation of the theory. For idioms to serve as a useful programming language we would additionally need constants for constructing basic computations. These play no significant role in the theory, so we omit them here.

In order to compare idioms and arrows we formalise static arrow computations, an analogue of the StaticArrow class in Section 2.2.3.2.

Figure 2.15 defines the theory of *classic arrows with delay* $\mathcal{C}_S$, a variant of classic arrows that supports static computation $\mathcal{C}_S$ by adding an extra constant, delay, which allows the input to a computation be delayed until after the side-effects have taken place. The inverse to delay, force, can be defined within the calculus.

Figure 2.16 defines the theory of *static arrows* $\mathcal{S}$, a variant of the arrow calculus in which the semantics of a computation is independent of input. The new command form **run** L is a restricted form of arrow application which runs an arrow computation without supplying it with an input, producing a value of function type. Note that variables from the Δ environment cannot occur in L. There are three new laws. The first law, run_1, states that arrow applications can be eliminated via decomposition into two parts. The first part runs the computation without using the input, returning a pure function. The second part applies this function to the input. The second law, run_2, states that running a pure computation is equivalent to lifting a pure function. The third law, run_3, allows **run** to be pushed into the body of a **let** binding. By

Syntax:

$$\text{Commands} \quad P, Q, R \quad ::= \quad \cdots \mid \textbf{run } L$$

Types:

$$\frac{\Gamma \vdash L : A \rightsquigarrow B}{\Gamma; \Delta \vdash \textbf{run } L \mathbin{!} A \to B}$$

Laws:

(run_1) $\qquad\qquad\qquad L \bullet M = \textbf{let } f \Leftarrow \textbf{run } L \textbf{ in } [f\ M]$

(run_2) $\qquad\qquad \textbf{run } (\lambda^\bullet x. [M]) = [\lambda x. M]$

(run_3) $\quad \textbf{run } (\lambda^\bullet x. \textbf{let } y \Leftarrow P \textbf{ in } Q) = \textbf{let } y \Leftarrow P \textbf{ in}$

$\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad \textbf{let } f \Leftarrow \textbf{run } (\lambda^\bullet \langle x, y \rangle. Q) \textbf{ in}$

$\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad [\lambda x. f \langle x, y \rangle]$

Figure 2.16: Static arrows, $\mathcal{S}$ (extends $\mathcal{A}$, Figure 2.6).

Static arrows to classic arrows with delay:

$$[\![\textbf{run } L]\!]_\Delta = \texttt{arr } (\lambda \Delta. \langle \rangle) \ggg \texttt{delay } [\![L]\!]$$

Classic arrows with delay to static arrows:

$$\langle\!\langle \texttt{delay} \rangle\!\rangle = \lambda x. \lambda^\bullet \langle \rangle. \textbf{run } x$$

Figure 2.17: Translating between $\mathcal{S}$ and $\mathcal{C}_S$.

---
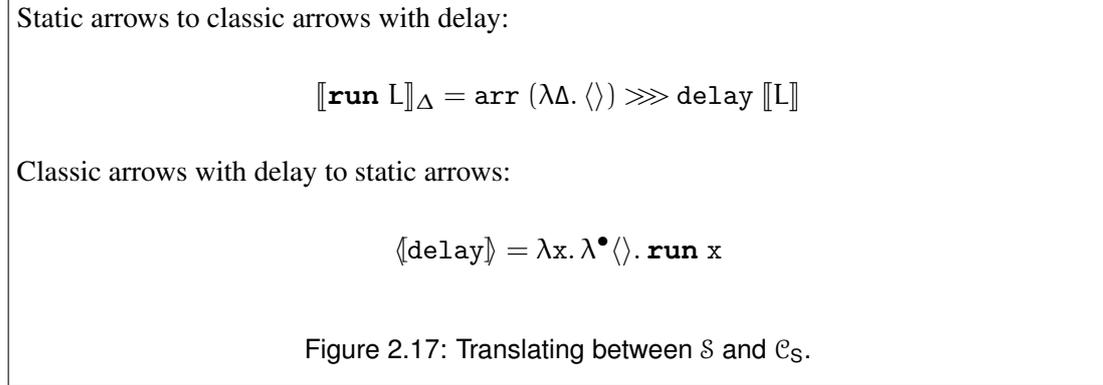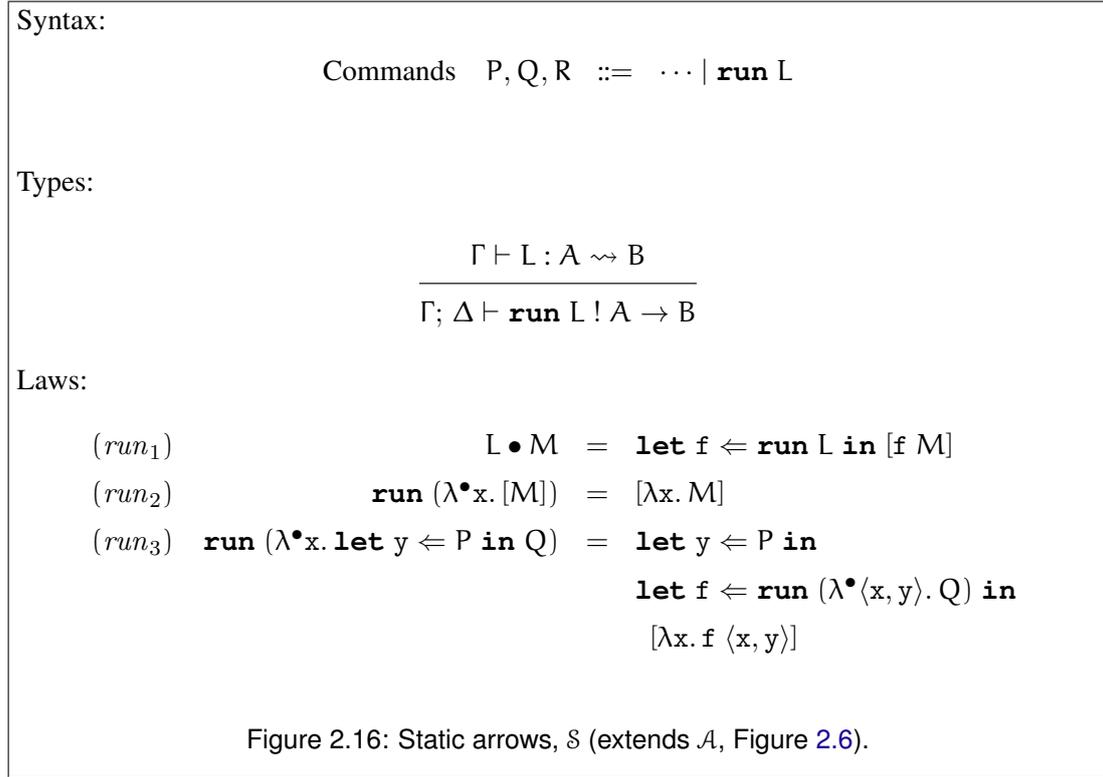
repeated application of these laws in conjunction with the other arrow calculus laws we can reduce all run commands in a term to the canonical form $\textbf{run } (f\ x_1\ \ldots\ x_n)$.

Figure 2.17 gives the translations between static arrows $\mathcal{S}$ and classic arrows with delay $\mathcal{C}_S$, which extend the translations of Figures 2.7 and 2.8. We write $\lambda\langle\rangle. M$ and $\lambda^\bullet\langle\rangle. Q$ for function and arrow abstractions with input type 1.

Proposition 12. *The theories of static arrows and classic arrows with delay are in equational correspondence:* $\mathcal{S} \cong \mathcal{C}_S$.

Proof. The proof requires an extension of Lemmas 8 and 10 to cover the construct $\textbf{run } L$ (Appendix A.3.1).

The equational correspondence extends the correspondence of Proposition 11. We must show that the three new laws of $\mathcal{S}$ follow from the nine laws of $\mathcal{C}$ plus the two additional laws of $\mathcal{C}_S$, and that the two laws of $\mathcal{C}_S$ follow from the five laws of $\mathcal{A}$ plus the three additional laws of $\mathcal{S}$. We must also show that translating a term from each theory to the other and back is the identity. There is one case for the command form **run** L of $\mathcal{S}$ and one for the term delay of $\mathcal{C}_S$.

Appendix A.3 gives the full proof. $\qquad\square$

Figure 2.18 gives the translations between idioms $\mathcal{I}$ and static arrows $\mathcal{S}$. (Here and throughout we elide the definition of the homomorphic translations on types and the corresponding type isomorphisms.)

As with the translation from $\mathcal{A}$ to $\mathcal{C}$, we must determine the meaning of translation from $\mathcal{S}$ to $\mathcal{I}$ on a substitution and on weakening, which we do with the following two lemmas.

Lemma 13. *[Translating substitution from $\mathcal{S}$ to $\mathcal{I}$]*

The translations of substitution on terms and commands from $\mathcal{S}$ to $\mathcal{I}$ satisfy the following equations.

$$\langle\!\langle M[x := N]\rangle\!\rangle = \langle\!\langle M\rangle\!\rangle[x := \langle\!\langle N\rangle\!\rangle]$$
$$\langle\!\langle Q[x := N]\rangle\!\rangle_\Delta = \texttt{pure}\,(\lambda g.\lambda\Delta.g\,(\Delta, \langle\!\langle N\rangle\!\rangle)) \circledast \langle\!\langle Q\rangle\!\rangle_{\Delta,x}$$

Proof. By mutual induction on the derivations of P and M. There is one case for each term form and each command form. Appendix A.4.1 gives the full proof. $\qquad\square$

Lemma 14. *[Translating weakening from $\mathcal{S}$ to $\mathcal{I}$]*

The translation of weakening from $\mathcal{S}$ to $\mathcal{I}$ for commands is as follows.

$$\left\langle\!\!\left\langle \left[\frac{\Gamma;\,\Delta \vdash Q\,!\,B}{\Gamma';\,\Delta' \vdash Q\,!\,B}\right]\right\rangle\!\!\right\rangle = \frac{\langle\!\langle\Gamma\rangle\!\rangle \vdash \langle\!\langle Q\rangle\!\rangle_\Delta : I(\langle\!\langle\Delta\rangle\!\rangle \to \langle\!\langle B\rangle\!\rangle)}{\langle\!\langle\Gamma'\rangle\!\rangle \vdash \texttt{pure}\,(\lambda g.\,\lambda\Delta'.g\,\Delta) \circledast \langle\!\langle Q\rangle\!\rangle_\Delta : I(\langle\!\langle\Delta'\rangle\!\rangle \to \langle\!\langle B\rangle\!\rangle)}$$

Proof. By induction on the derivation of Q. There is one case for each command form. Appendix A.4.1 gives the full proof. $\qquad\square$

Proposition 15. *The theories of idioms and static arrows are equationally equivalent: $\mathcal{I} \sim \mathcal{S}$.*

Proof. The equational equivalence has four components:

- The four laws of $\mathcal{I}$ follow from the eight laws of $\mathcal{S}$ (i.e. the five laws of $\mathcal{A}$ plus the three additional laws of $\mathcal{S}$). The proof requires four calculations, one for each law of $\mathcal{I}$.

Idioms to static arrows:

$$\llbracket I\,A \rrbracket = 1 \rightsquigarrow \llbracket A \rrbracket$$

$$\llbracket \mathtt{pure} \rrbracket = \lambda x.\, \lambda^\bullet \langle\rangle.\, [x]$$

$$\llbracket (\circledast) \rrbracket = \lambda h.\, \lambda a.\, \lambda^\bullet \langle\rangle.\, \mathbf{let}\ k \Leftarrow h \bullet \langle\rangle\ \mathbf{in}\ \mathbf{let}\ x \Leftarrow a \bullet \langle\rangle\ \mathbf{in}\ [k\ x]$$

Static arrows to idioms:

$$\langle\!\langle A \rightsquigarrow B \rangle\!\rangle = I\,(\langle\!\langle A \rangle\!\rangle \to \langle\!\langle B \rangle\!\rangle)$$

$$\langle\!\langle \lambda^\bullet x.\, P \rangle\!\rangle = \langle\!\langle P \rangle\!\rangle_x$$

where

$$\langle\!\langle \Gamma;\Delta \vdash P\,!\,A \rangle\!\rangle = \langle\!\langle \Gamma \rangle\!\rangle \vdash \langle\!\langle P \rangle\!\rangle_\Delta : I\,(\langle\!\langle \Delta \rangle\!\rangle \to \langle\!\langle A \rangle\!\rangle)$$

$$\langle\!\langle L \bullet M \rangle\!\rangle_\Delta = \mathtt{pure}\,(\lambda l.\, \lambda \Delta.\, l\ \langle\!\langle M \rangle\!\rangle) \circledast \langle\!\langle L \rangle\!\rangle$$

$$\langle\!\langle \mathbf{run}\ L \rangle\!\rangle_\Delta = \mathtt{pure}\,(\lambda l.\, \lambda \Delta.\, l) \circledast \langle\!\langle L \rangle\!\rangle$$

$$\langle\!\langle [M] \rangle\!\rangle_\Delta = \mathtt{pure}\,(\lambda \Delta.\, \langle\!\langle M \rangle\!\rangle)$$

$$\langle\!\langle \mathbf{let}\ x \Leftarrow P\ \mathbf{in}\ Q \rangle\!\rangle_\Delta = \mathtt{pure}\,(\lambda p.\, \lambda q.\, \lambda \Delta.\, q\ \langle \Delta, p\ \Delta \rangle) \circledast \langle\!\langle P \rangle\!\rangle_\Delta \circledast \langle\!\langle Q \rangle\!\rangle_{\Delta,x}$$

Type isomorphism on idioms:

$$f_{I(A)} : I\,A \simeq I\,(1 \to \langle\!\langle \llbracket A \rrbracket \rangle\!\rangle)$$

$$f_{I(A)} = \lambda a.\, \mathtt{pure}\,(\lambda x.\, \lambda\langle\rangle.\, f_A(x)) \circledast a$$

$$f_{I(A)}^{-1} = \lambda a.\, \mathtt{pure}\,(\lambda x.\, f_A^{-1}\,(x\ \langle\rangle)) \circledast a$$

Type isomorphism on static arrows:

$$g_{A \rightsquigarrow B} : A \rightsquigarrow B \simeq 1 \rightsquigarrow (\llbracket \langle\!\langle A \rangle\!\rangle \rrbracket \to \llbracket \langle\!\langle B \rangle\!\rangle \rrbracket)$$

$$g_{A \rightsquigarrow B} = \lambda a.\, \lambda^\bullet \langle\rangle.\, \mathbf{let}\ f \Leftarrow \mathbf{run}\ a\ \mathbf{in}\ [\lambda x.\, g_B(f\,(g_A^{-1}(x)))]$$

$$g_{A \rightsquigarrow B}^{-1} = \lambda a.\, \lambda^\bullet x.\, \mathbf{let}\ h \Leftarrow a \bullet \langle\rangle\ \mathbf{in}\ [g_B^{-1}(h\,(g_A(x)))]$$

Figure 2.18: Translating between $\mathcal{I}$ and $\mathcal{S}$.

- The eight laws of $\mathcal{S}$ follow from the four laws of $\mathcal{J}$. The proof requires eight calculations, one for each law of $\mathcal{S}$. The cases for $(\beta^{\rightsquigarrow})$ and (left) require Lemma 13 and the cases for $(\beta^{\rightsquigarrow})$, (assoc) and (run_3) require Lemma 14.

- Translating a term M from $\mathcal{J}$ into $\mathcal{S}$ and back back gives a term isomorphic to M:

$$\langle\!\langle [\![M]\!] \rangle\!\rangle = f_A(M)$$

  There are two cases, one for `pure` and one for $\circledast$. Figure 2.18 defines the isomorphism we need at the type $I(A)$; it is straightforward to construct isomorphisms at the types of `pure` and $\circledast$ from this. We need the definition of $f_{A \rightarrow B}$, which is the same for each isomorphism:

$$f_{A \rightarrow B} : A \rightarrow B \simeq \langle\!\langle [\![A]\!] \rangle\!\rangle \rightarrow \langle\!\langle [\![B]\!] \rangle\!\rangle$$
$$f_{A \rightarrow B} = \lambda h.\, \lambda x.\, f_B\ (h\ (f_A^{-1}(x)))$$
$$f_{A \rightarrow B}^{-1} = \lambda h.\, \lambda x.\, f_B^{-1}\ (h\ (f_A(x)))$$

  Combining this with $f_{I(A)}$ gives the following isomorphism at $A \rightarrow I(A)$, the type of `pure`:

$$f_{A \rightarrow I(A)} = \lambda h.\, \lambda x.\, (\text{pure}\ (\lambda x.\, \lambda\langle\rangle.\, f_A(x)) \circledast (h\ (f_A^{-1}(x))))$$

  The other isomorphisms we need, both here and in the propositions that follow, are similarly straightforward to construct.

- Translating a term M from $\mathcal{S}$ into $\mathcal{J}$ and back gives a term isomorphic to M:

$$[\![\langle\!\langle M \rangle\!\rangle]\!] = g_A(M)$$

  There is one case, for $\lambda^{\bullet}x.\, Q$. The proof involves showing a corresponding property for commands. Translating a command P from $\mathcal{S}$ into $\mathcal{J}$ and back gives a term isomorphic to $\lambda^{\bullet}\Delta.\, P$:

$$[\![\langle\!\langle P \rangle\!\rangle_\Delta]\!] = g_{\Delta \rightsquigarrow A}(\lambda^{\bullet}\Delta.\, P)$$

  There are four cases, one for each command form of $\mathcal{S}$.

Appendix A.4 gives the proof in full.   □

Remark The type isomorphism $g_{A \rightsquigarrow B} : A \rightsquigarrow B \simeq 1 \rightsquigarrow ([\![\langle\!\langle A \rangle\!\rangle]\!] \rightarrow [\![\langle\!\langle B \rangle\!\rangle]\!])$ gives an alternative characterisation of static arrows. One half of the isomorphism can be defined in the arrow

calculus, $\mathcal{A}$:

$$g^{-1}_{A\rightsquigarrow B} : (1 \rightsquigarrow (\llbracket \langle\!\langle A \rangle\!\rangle \rrbracket \rightarrow \llbracket \langle\!\langle B \rangle\!\rangle \rrbracket)) \rightarrow (A \rightsquigarrow B)$$

$$g^{-1}_{A\rightsquigarrow B} = \lambda a.\, \lambda^\bullet x.\, \textbf{let } h \Leftarrow a \bullet \langle\rangle \textbf{ in } [g^{-1}_B(h\,(g_A(x)))]$$

The inverse of $g^{-1}_{A\rightsquigarrow B}$ can be defined in static arrows, $\mathcal{S}$, using **run** :

$$g_{A\rightsquigarrow B} : (A \rightsquigarrow B) \rightarrow (1 \rightsquigarrow (\llbracket \langle\!\langle A \rangle\!\rangle \rrbracket \rightarrow \llbracket \langle\!\langle B \rangle\!\rangle \rrbracket))$$

$$g_{A\rightsquigarrow B} = \lambda a.\, \lambda^\bullet \langle\rangle.\, \textbf{let } f \Leftarrow \textbf{run } a \textbf{ in } [\lambda x.\, g_B(f\,(g^{-1}_A(x)))]$$

The function $g_{A\rightsquigarrow B}$ and **run** L are inter-definable. The definition of **run** L in terms of $g_{A\rightsquigarrow B}$ is as follows:

$$\textbf{run } L \equiv (g_{A\rightsquigarrow B}(L)) \bullet \langle\rangle$$

Figure 2.19 gives a translation $\llbracket - \rrbracket$ from $\mathcal{S}$ into $\mathcal{A}$ and a translation $\langle\!\langle - \rangle\!\rangle$ from the image of $\llbracket - \rrbracket$ into $\mathcal{S}$.

Lemma 16. *[Translating substitution from $\mathcal{S}$ to $\mathcal{A}$]*

The translations of substitution on terms and commands from $\mathcal{S}$ to $\mathcal{A}$ satisfy the following equations.

$$\llbracket M[x := N] \rrbracket = \llbracket M \rrbracket [x := \llbracket N \rrbracket]$$

$$\llbracket Q[x := N] \rrbracket = \textbf{let } q \Leftarrow \llbracket Q \rrbracket_{\Delta,x} \textbf{ in } [\lambda\Delta.q\,(\Delta, \llbracket N \rrbracket)]$$

Proof. By mutual induction on the derivations of P and M. There is one case for each term form and each command form. Appendix A.5.1 gives the proof in full. $\square$

Lemma 17. *[Translating substitution from the image of $\llbracket - \rrbracket$ to $\mathcal{S}$]*

The translations of substitution on terms and commands from the image of $\llbracket - \rrbracket$ to $\mathcal{S}$ satisfy the following equations.

$$\langle\!\langle M[x := N] \rangle\!\rangle = \langle\!\langle M \rangle\!\rangle [x := \langle\!\langle N \rangle\!\rangle]$$

$$\langle\!\langle Q[x := N] \rangle\!\rangle = \langle\!\langle Q \rangle\!\rangle [x := \langle\!\langle N \rangle\!\rangle]$$

Lemma 18. *[Translating weakening from $\mathcal{S}$ to $\mathcal{A}$]*

The translation of weakening from $\mathcal{S}$ to $\mathcal{A}$ for commands is as follows.

$$\left\llbracket \frac{\Gamma;\, \Delta \vdash Q \,!\, B}{\Gamma';\, \Delta' \vdash Q \,!\, B} \right\rrbracket \;=\; \frac{\llbracket \Gamma \rrbracket;\, \cdot \vdash \llbracket Q \rrbracket_\Delta \,!\, \llbracket \Delta \rrbracket \rightarrow \llbracket B \rrbracket}{\llbracket \Gamma' \rrbracket;\, \cdot \vdash \textbf{let } q \Leftarrow \llbracket Q \rrbracket_\Delta \textbf{ in } [\lambda\Delta'.q\,\Delta] \,!\, \llbracket \Delta' \rrbracket \rightarrow \llbracket B \rrbracket}$$

Static arrows to arrows:

$$[\![A \rightsquigarrow B]\!] = 1 \rightsquigarrow ([\![A]\!] \to [\![B]\!])$$

$$[\![\lambda^\bullet x. P]\!] = \lambda^\bullet \langle \rangle. [\![P]\!]_x$$

where

$$[\![\Gamma; \Delta \vdash P \,!\, A]\!] = [\![\Gamma]\!]; \cdot \vdash [\![P]\!]_\Delta \,!\, [\![\Delta]\!] \to [\![A]\!]$$

$$[\![L \bullet M]\!]_\Delta = \mathbf{let}\ l \Leftarrow [\![L]\!] \bullet \langle \rangle\ \mathbf{in}\ [\lambda\Delta. l\ [\![M]\!]]$$

$$[\![\mathbf{run}\ L]\!]_\Delta = \mathbf{let}\ h \Leftarrow [\![L]\!] \bullet \langle \rangle\ \mathbf{in}\ [\lambda\Delta. h]$$

$$[\![[M]]\!]_\Delta = [\lambda\Delta. [\![M]\!]]$$

$$[\![\mathbf{let}\ x \Leftarrow P\ \mathbf{in}\ Q]\!]_\Delta = \ \mathbf{let}\ p \Leftarrow [\![P]\!]_\Delta\ \mathbf{in}$$
$$\mathbf{let}\ q \Leftarrow [\![Q]\!]_{\Delta,x}\ \mathbf{in}\ [\lambda\Delta. q\ \langle \Delta, p\ \Delta \rangle]$$

$[\![\mathbb{S}]\!]$ to static arrows:

$$\langle\!\langle 1 \rightsquigarrow (A \to B) \rangle\!\rangle = \langle\!\langle A \rangle\!\rangle \rightsquigarrow \langle\!\langle B \rangle\!\rangle$$

$$\langle\!\langle \lambda^\bullet \langle \rangle. P \rangle\!\rangle = \lambda^\bullet x. \mathbf{let}\ h \Leftarrow \langle\!\langle P \rangle\!\rangle\ \mathbf{in}\ [h\ x]$$

where

$$\langle\!\langle \Gamma; \Delta \vdash P \,!\, A \rangle\!\rangle = \langle\!\langle \Gamma \rangle\!\rangle; \langle\!\langle \Delta \rangle\!\rangle \vdash \langle\!\langle P \rangle\!\rangle \,!\, \langle\!\langle A \rangle\!\rangle$$

$$\langle\!\langle L \bullet M \rangle\!\rangle = \mathbf{run}\ \langle\!\langle L \rangle\!\rangle$$

$$\langle\!\langle [M] \rangle\!\rangle = [\langle\!\langle M \rangle\!\rangle]$$

$$\langle\!\langle \mathbf{let}\ x \Leftarrow P\ \mathbf{in}\ Q \rangle\!\rangle = \mathbf{let}\ x \Leftarrow \langle\!\langle P \rangle\!\rangle\ \mathbf{in}\ \langle\!\langle Q \rangle\!\rangle$$

Type isomorphism on static arrows:

$$f : A \simeq A \text{ is the identity isomorphism.}$$

Figure 2.19: Embedding $\mathbb{S}$ into $\mathcal{A}$.

Proof. By induction on the derivation of Q. There is one case for each command form. Appendix A.5.1 gives the proof in full. □

Proposition 19. *The translations* $[\![-]\!]$ *and* $\langle\!\langle-\rangle\!\rangle$ *define an equational embedding of static arrows into arrow calculus:* $\mathcal{S} \hookrightarrow \mathcal{A}$.

Proof. The equational embedding has three components:

- The eight laws of $\mathcal{S}$ follow from the five laws of $\mathcal{A}$. The proof requires eight calculations, one for each law of $\mathcal{S}$. The cases for $(\beta^{\rightsquigarrow})$ and (left) depend on Lemma 16, and the cases for (assoc) and (run_3) depend on Lemma 18.

- For terms in the image of the translation $[\![-]\!]$, the five laws of $\mathcal{A}$ follow from the eight laws of $\mathcal{S}$. The proof requires five calculations, one for each law of $\mathcal{A}$. The cases for $(\beta^{\rightsquigarrow})$ and (left) depend on Lemma 17

- Translating a term M from $\mathcal{S}$ into $\mathcal{A}$ and back is the identity:

$$\langle\!\langle[\![M]\!]\rangle\!\rangle = M$$

  There is one case, for $\lambda^{\bullet}x. Q$. The proof involves showing a corresponding property for commands. Translating a command P from $\mathcal{S}$ into $\mathcal{A}$ and back gives a term isomorphic to $\lambda^{\bullet}\Delta. P$:

$$\langle\!\langle[\![P]\!]_{\Delta}\rangle\!\rangle = \mathbf{run}\ (\lambda^{\bullet}\Delta. P)$$

  or, equivalently,

$$\mathbf{let}\ d \Leftarrow \langle\!\langle[\![P]\!]_{\Delta}\rangle\!\rangle\ \mathbf{in}\ [d\ \Delta] = P$$

  There are four cases, one for each command form of $\mathcal{S}$.

  Appendix A.5 gives the proof in full. □

In summary: idioms are equationally equivalent to static arrows, which embed into arrow calculus (Figure 2.20).

2.4.2 Arrows and monads

Figure 2.21 defines the theory of *monads*, $\mathcal{M}$, a straightforward recapitulation of the Haskell interface presented in Section 2.2.1. Like $\mathcal{I}$, $\mathcal{M}$ extends $\lambda^{\rightarrow\times 1}$ with a unary type constructor M for computations of type $M\ A$ which return a value of type A and two constants, $\mathbf{return}$ and $(\ggg)$. There are three laws which, together with the laws of the lambda calculus, define

Figure 2.20: Relating idioms to arrows.

Syntax:

$$\text{Types}\quad A, B, C \quad ::= \quad \cdots \mid M\,A$$

Constants:

$$\texttt{return}\ :\ A \to M\,A$$

$$(\ggg)\ :\ M\,A \to (A \to M\,B) \to M\,B$$

Laws:

$$(M_1)\quad \texttt{return}\,N \ggg M\ =\ M\,N$$

$$(M_2)\qquad M \ggg \texttt{return}\ =\ M$$

$$(M_3)\quad (L \ggg M) \ggg N\ =\ L \ggg (\lambda x.\, M\,x \ggg N)$$

Figure 2.21: Monads, $\mathcal{M}$ (extends $\lambda^{\to \times 1}$, Figure 2.4).
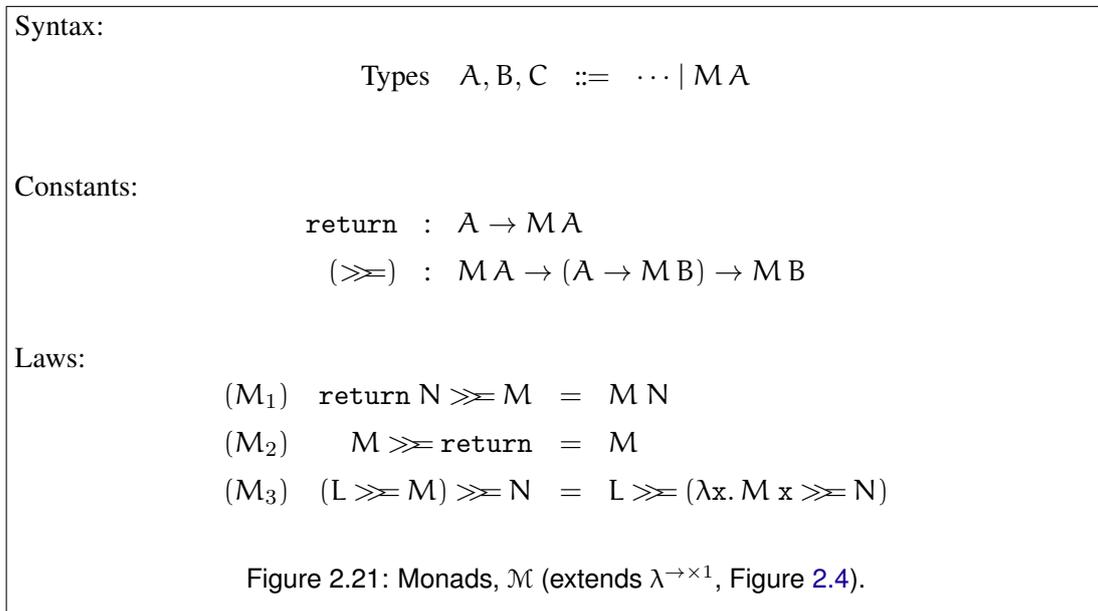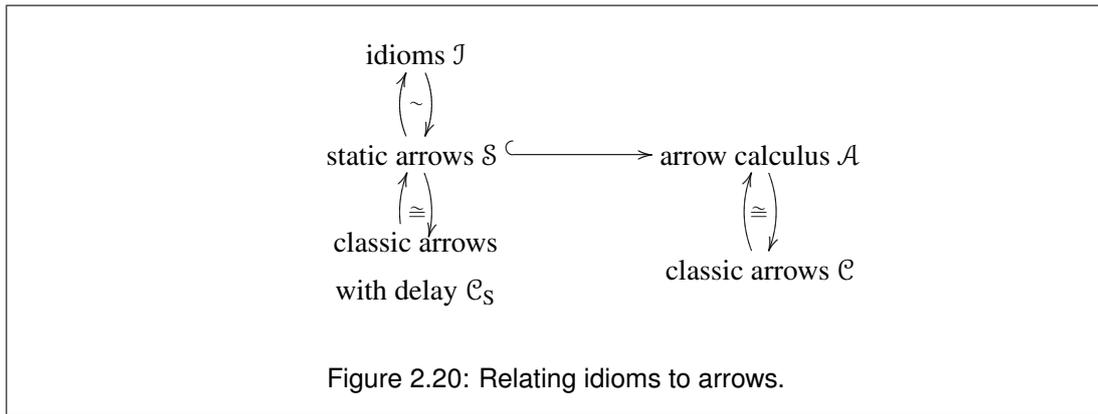
the equivalence relation of the theory. As before, we do not consider the additional constants needed for constructing basic computations.

In order to compare arrows and monads we consider arrows extended with application. We briefly recall the theory of classic arrows with apply and then introduce a higher-order extension of the arrow calculus that is in equational correspondence with $\mathcal{C}_{\text{app}}$.

Figure 2.22 defines the theory of *classic arrows with apply* $\mathcal{C}_{\text{app}}$, a recapitulation of the `ArrowApply` class described in Section 2.2.2.4 (and in more detail by Hughes (2000)).

Figure 2.23 defines the theory of *higher-order arrows* $\mathcal{H}$, an extension of the arrow calculus to support the application of an arrow that is itself yielded by another arrow. The new command form $L \star M$ lifts the central restriction on arrow application. Now the arrow to apply may be the result of a command, and the command denoting the arrow may contain free variables in both Γ and Δ. The additional laws are simply the beta and eta laws for $\star$.

Constants:

$$\texttt{app} \quad : \quad (A \rightsquigarrow B) {\times} A \rightsquigarrow B$$

Laws:

$(\rightsquigarrow_{H1}) \quad \texttt{first}\,(\texttt{arr}\,(\lambda x.\,\texttt{arr}\,(\lambda y.\,\langle x, y\rangle))) \ggg \texttt{app} \quad = \quad \texttt{arr id}$

$(\rightsquigarrow_{H2}) \qquad\qquad\quad \texttt{first}\,(\texttt{arr}\,(L{\ggg})) \ggg \texttt{app} \quad = \quad \texttt{second}\,L \ggg \texttt{app}$

$(\rightsquigarrow_{H3}) \qquad\qquad\quad \texttt{first}\,(\texttt{arr}\,({\ggg}L)) \ggg \texttt{app} \quad = \quad \texttt{app} \ggg L$

Figure 2.22: Classic arrows with apply, $\mathcal{C}_{\mathsf{app}}$ (extends $\mathcal{C}$, Figure 2.5).

Syntax:

$$\text{Commands} \quad P, Q, R \quad ::= \quad \cdots \mid L \star M$$

Types:

$$\frac{\Gamma, \Delta \vdash L : A \rightsquigarrow B \qquad \Gamma, \Delta \vdash M : A}{\Gamma; \Delta \vdash L \star M \,!\, B}$$

Laws:

$(\beta^{\mathsf{app}}) \quad (\lambda^{\bullet} x.\, Q) \star M \quad = \quad Q[x := M]$

$(\eta^{\mathsf{app}}) \qquad \lambda^{\bullet} x.\, (L \star x) \quad = \quad L$

Figure 2.23: Higher-order arrows, $\mathcal{H}$ (extends $\mathcal{A}$, Figure 2.6).
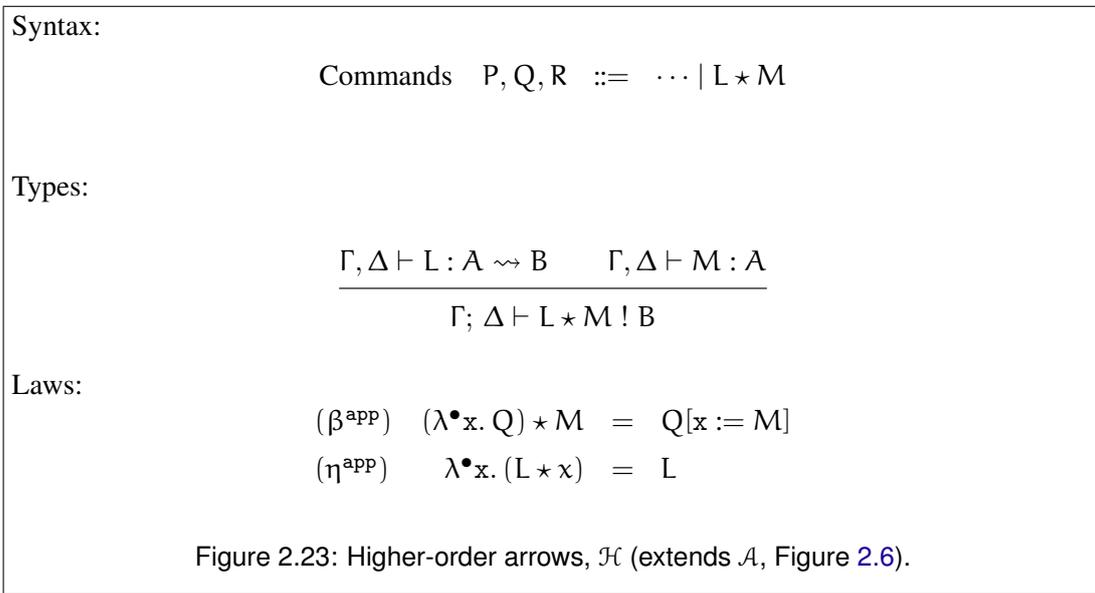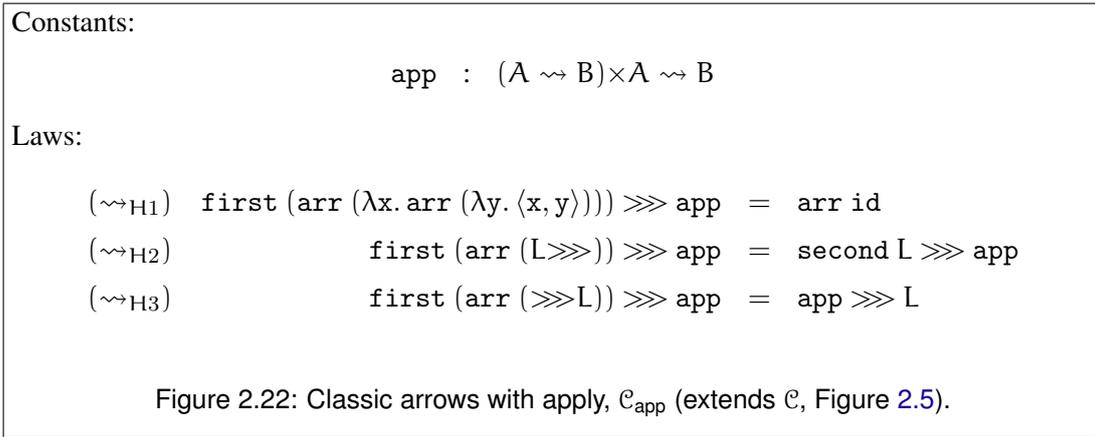
In fact, the η^{app} law is redundant:

$$\lambda^{\bullet} x.\, L \star x$$
$$= \qquad (\eta^{\rightsquigarrow})$$
$$\lambda^{\bullet} x.\, (\lambda^{\bullet} y.\, L \bullet y) \star x$$
$$= \qquad (\beta^{\mathsf{app}})$$
$$\lambda^{\bullet} x.\, L \bullet x$$
$$= \qquad (\eta^{\rightsquigarrow})$$
$$L$$

We retain it for convenience.

Figure 2.24 gives the translations between the higher-order arrows $\mathcal{H}$ and classic arrows with apply $\mathcal{C}_{\mathsf{app}}$, which extend the translations of Figures 2.7 and 2.8.

Higher-order arrows to classic arrows with apply:

$$\llbracket L \star M \rrbracket_\Delta = \mathtt{arr}\ (\lambda\Delta.\ (\llbracket L \rrbracket, \llbracket M \rrbracket)) \ggg \mathtt{app}$$

Classic arrows with apply to higher-order arrows:

$$\langle\!\langle \mathtt{app} \rangle\!\rangle = \lambda^\bullet p.\ (\mathbf{fst}\ p) \star (\mathbf{snd}\ p)$$

Figure 2.24: Translating between $\mathcal{H}$ and $\mathcal{C}_{app}$.
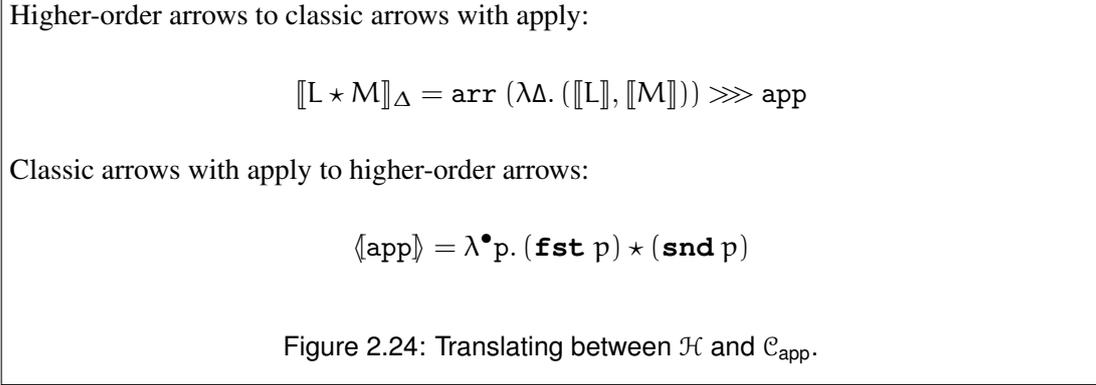
Proposition 20. *The theories of higher-order arrows and classic arrows with apply are in equational correspondence:* $\mathcal{H} \cong \mathcal{C}_{app}$.

Proof. The proof requires an extension of Lemmas 8 and 10, to cover the $L \star M$ construct (Appendix A.6.1).

The proof of equational correspondence extends the proof of Proposition 11. We must show that the laws of each theory follow from the laws of the other. There is one calculation to show that the $(\beta^{\mathtt{app}})$ law follows from the laws of $\mathcal{C}_{app}$ and there are three calculations to show that laws $(\leadsto_{H1})$, $(\leadsto_{Hs})$ and $(\leadsto_{H3})$ follow from the laws of $\mathcal{H}$. We must also show that translating terms from each theory to the other and back is the identity. There is one calculation to show that translating the command $L \star M$ to $\mathcal{C}_{app}$ and back gives $\lambda^\bullet\Delta.\ (L \star M)$ and one calculation to show that translating $\mathtt{app}$ to $\mathcal{H}$ and back is the identity.

The full proof is given in Appendix A.6.   □

Figure 2.25 gives the translations between higher-order arrows $\mathcal{H}$ and monads $\mathcal{M}$.

Lemma 21. *[Translating substitution from $\mathcal{H}$ to $\mathcal{M}$]*

The translations of substitution on terms and commands from $\mathcal{H}$ to $\mathcal{M}$ satisfy the following equations.

$$\langle\!\langle M[x := N] \rangle\!\rangle = \langle\!\langle M \rangle\!\rangle [x := \langle\!\langle N \rangle\!\rangle]$$
$$\langle\!\langle Q[x := N] \rangle\!\rangle = \langle\!\langle Q \rangle\!\rangle [x := \langle\!\langle N \rangle\!\rangle]$$

Proposition 22. *The theories of monads and higher-order arrows are equationally equivalent:* $\mathcal{M} \sim \mathcal{H}$.

Proof. In order to demonstrate the equational equivalence, we must show that the three laws of $\mathcal{M}$ follow from the seven laws of $\mathcal{H}$, and vice versa. We must also show that translating

Monads to higher-order arrows:

$$[\![M\,A]\!] = 1 \rightsquigarrow [\![A]\!]$$

$$[\![\texttt{return}]\!] = \lambda x.\, \lambda^\bullet \langle\rangle.\, [x]$$

$$[\![(\ggg)]\!] = \lambda a.\, \lambda h.\, \lambda^\bullet \langle\rangle.\, \textbf{let }x \Leftarrow a \star \langle\rangle \textbf{ in } (h\,x) \star \langle\rangle$$

Higher-order arrows to monads:

$$\langle\!\langle A \rightsquigarrow B \rangle\!\rangle = \langle\!\langle A \rangle\!\rangle \rightarrow M\, \langle\!\langle B \rangle\!\rangle$$

$$\langle\!\langle \lambda^\bullet x.\, P \rangle\!\rangle = \lambda x.\, \langle\!\langle P \rangle\!\rangle$$

where

$$\langle\!\langle \Gamma;\, \Delta \vdash P\,!\,A \rangle\!\rangle = \langle\!\langle \Gamma, \Delta \rangle\!\rangle \vdash \langle\!\langle P \rangle\!\rangle : M\, \langle\!\langle A \rangle\!\rangle$$

$$\langle\!\langle L \bullet M \rangle\!\rangle = \langle\!\langle L \rangle\!\rangle\, \langle\!\langle M \rangle\!\rangle$$

$$\langle\!\langle L \star M \rangle\!\rangle = \langle\!\langle L \rangle\!\rangle\, \langle\!\langle M \rangle\!\rangle$$

$$\langle\!\langle [M] \rangle\!\rangle = \texttt{return}\, \langle\!\langle M \rangle\!\rangle$$

$$\langle\!\langle \textbf{let }x \Leftarrow P \textbf{ in } Q \rangle\!\rangle = \langle\!\langle P \rangle\!\rangle \ggg \lambda x.\, \langle\!\langle Q \rangle\!\rangle$$

Type isomorphism on monads:

$$f_{M(A)} : M\,A \simeq 1 \rightarrow M\, \langle\!\langle [\![A]\!] \rangle\!\rangle$$

$$f_{M(A)} = \lambda a.\, \lambda u.\, a \ggg (\lambda x.\, \texttt{return}\, (f_A(x)))$$

$$f^{-1}_{M(A)} = \lambda h.\, h\, \langle\rangle \ggg (\lambda x.\, \texttt{return}\, (f^{-1}_A(x)))$$

Type isomorphism on higher-order arrows:

$$g_{A \rightsquigarrow B} : A \rightsquigarrow B \simeq [\![\langle\!\langle A \rangle\!\rangle]\!] \rightarrow (1 \rightsquigarrow [\![\langle\!\langle B \rangle\!\rangle]\!])$$

$$g_{A \rightsquigarrow B} = \lambda a.\, \lambda x.\, \lambda^\bullet \langle\rangle.\, \textbf{let }v \Leftarrow a \bullet (g^{-1}_A(x)) \textbf{ in } [g_B(v)]$$

$$g^{-1}_{A \rightsquigarrow B} = \lambda h.\, \lambda^\bullet x.\, \textbf{let }v \Leftarrow (h\,(g_A(x))) \star \langle\rangle \textbf{ in } [g^{-1}_B(v)]$$

Figure 2.25: Translating between $\mathcal{M}$ and $\mathcal{H}$.

a term from each theory to the other and back gives a term that is isomorphic to the original. This involves showing a corresponding property for commands: for each command P of $\mathcal{H}$ we show that the following holds:

$$\llbracket \langle\!\langle P \rangle\!\rangle \rrbracket = \lambda^\bullet \langle\rangle. \textbf{let } z \Leftarrow P \textbf{ in } [g_A(z)]$$

We highlight one additional point of interest. Both $L \bullet M$ and $L \star M$ translate to regular function application in $\mathcal{M}$. How can translating the results back to $\mathcal{H}$ give terms that are isomorphic to both $L \bullet M$ and $L \star M$? Once again, we must pay careful attention to the type environments. When L does not use variables from Δ, the two forms of arrow application are equivalent, as the following shows:

$$L \bullet M$$
$$=\qquad (\eta^{\text{app}})$$
$$(\lambda^\bullet x. L \star x) \bullet M$$
$$=\qquad (\beta^{\rightsquigarrow})$$
$$L \star M$$

Translating from $\mathcal{H}$ to $\mathcal{M}$ and back changes all Δ-variables into Γ-variables, so we are free to make use of this equivalence in the case for $L \star M$.

Appendix A.7 gives the proof of equational equivalence in full. $\square$

Proposition 23. *There is an equational embedding of arrow calculus into higher-order arrows:*
$\mathcal{A} \hookrightarrow \mathcal{H}$.

The translation $\llbracket - \rrbracket$ is the inclusion map from $\mathcal{A}$ to $\mathcal{H}$, the translation $\langle\!\langle - \rangle\!\rangle$ is the identity on $\mathcal{A}$, and f is the identity isomorphism.

Remark The type isomorphism $g_{A \rightsquigarrow B} : A \rightsquigarrow B \simeq \llbracket \langle\!\langle A \rangle\!\rangle \rrbracket \to (1 \rightsquigarrow \llbracket \langle\!\langle B \rangle\!\rangle \rrbracket)$ gives an alternative characterisation of higher-order arrows. One half of the isomorphism can be defined in the arrow calculus, $\mathcal{A}$:

$$g_{A \rightsquigarrow B} : (A \rightsquigarrow B) \to (\llbracket \langle\!\langle A \rangle\!\rangle \rrbracket \to (1 \rightsquigarrow \llbracket \langle\!\langle B \rangle\!\rangle \rrbracket))$$
$$g_{A \rightsquigarrow B} = \lambda a. \lambda x. \lambda^\bullet \langle\rangle. \textbf{let } v \Leftarrow a \bullet (g_A^{-1}(x)) \textbf{ in } [g_B(v)]$$

The inverse of $g_{A \rightsquigarrow B}$ can be defined in higher-order arrows, $\mathcal{H}$, using $\star$:

$$g_{A \rightsquigarrow B}^{-1} : (\llbracket \langle\!\langle A \rangle\!\rangle \rrbracket \to (1 \rightsquigarrow \llbracket \langle\!\langle B \rangle\!\rangle \rrbracket)) \to (A \rightsquigarrow B)$$
$$g_{A \rightsquigarrow B}^{-1} = \lambda h. \lambda^\bullet x. \textbf{let } v \Leftarrow (h \; (g_A(x))) \star \langle\rangle \textbf{ in } [g_B^{-1}(v)]$$

Figure 2.26: Relating arrows to monads.
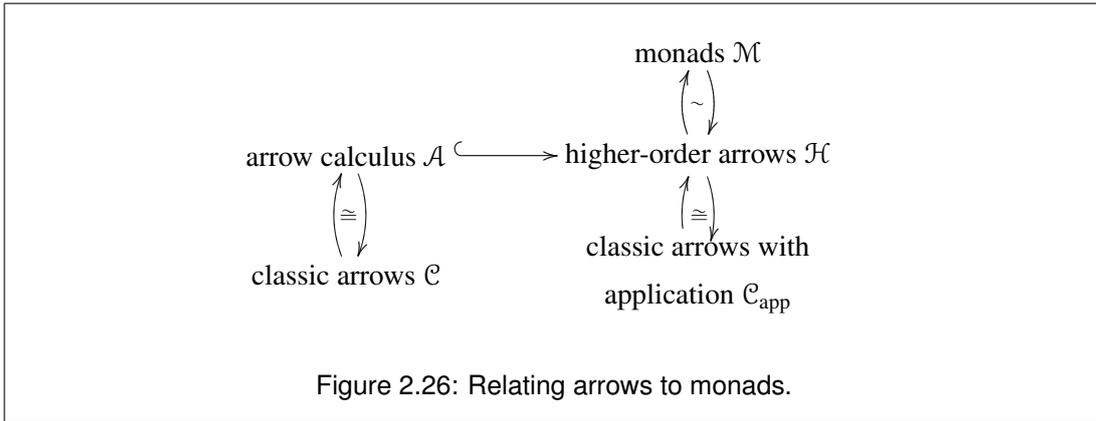
The function $g^{-1}_{A \leadsto B}$ and $L \star M$ are inter-definable. The definition of $L \star M$ in terms of $g^{-1}_{A \leadsto B}$ is as follows:

$$L \star M \equiv g^{-1}_{(A \leadsto B) \times A \leadsto B}(\lambda p. \lambda^\bullet \langle \rangle. (\texttt{fst}\ p)(\texttt{snd}\ p)) \bullet \langle L, M \rangle$$

In summary: monads are equationally equivalent to higher-order arrows, and there is an equational embedding of arrow calculus into higher-order arrows (Figure 2.26).

2.4.2.1 Redundancy of the second higher-order arrows law

A look at the proof of (β^{app}) (Appendix A.6.2) reveals another redundancy: $(\leadsto_{H2})$ is not required to prove (β^{app}). From the classic laws—with apply but excluding $(\leadsto_{H2})$—we can prove the laws of higher-order arrows, and from these we can in turn prove the classic laws—including $(\leadsto_{H2})$. It follows that $(\leadsto_{H2})$ must be redundant. Appendix A.8 gives a direct proof of the redundancy, deriving $(\leadsto_{H2})$ from $(\leadsto_{H1})$, $(\leadsto_{H3})$, and the classic arrow laws.

2.4.3 Closing remarks on expressive power

Our work in Section 2.4 establishes an order of expressive power between three calculi using equational embeddings. In one respect the ordering that emerges appears surprising: both the syntax of the least expressive language, $\mathcal{S}$, and the syntax of the most expressive language, $\mathcal{H}$, are extensions of the language that is intermediate in expressiveness, $\mathcal{A}$. The syntax of $\mathcal{H}$ is the syntax of $\mathcal{A}$ with the addition of $\star$, so it is not surprising that there is an embedding of $\mathcal{A}$ into $\mathcal{H}$. However, the syntax of $\mathcal{S}$ is the syntax of $\mathcal{A}$ with the addition of $\texttt{run}\ L$, and yet there the embedding is the other way around: $\mathcal{A}$ is embedded into $\mathcal{S}$! How can a adding new constructs to a language make it less powerful?

In order to answer to this question we must first note that we are not here dealing with concrete programming languages, but with a more abstract flavour of calculus. Our calculi are

not furnished with concrete value domains, or evaluation relations: each calculus admits many possible interpretations, just as the various type classes in Section 2.2 have many instances. Just as the laws associated with the type classes restrict the possible instances of the class, so the equational theory of each calculus restricts which interpretations we can give to the calculus. $\mathcal{S}$ extends not only the syntax, but also the equational theory of $\mathcal{A}$, and it thereby introduces further restrictions on the set of possible interpretations, so rendering the language less powerful.

We can illustrate the principle using a more familiar calculus, and a rather extreme example. In Figure 2.4 we gave the equational theory of a simply-typed lambda calculus $\lambda^{\rightarrow \times 1}$. Suppose that we extend this calculus to obtain $\lambda_{\perp}^{\rightarrow \times 1}$, which has an additional constant $\perp_A$, and a law ($\perp$) which equates every other term with this constant:

$$L, M, N \quad ::= \quad \cdots \mid \perp_A$$

$$(\perp) \quad \perp_A = L$$

Clearly only one-point value domains will be suitable for interpreting $\lambda_{\perp}^{\rightarrow \times 1}$: we have extended the syntax of the language, but severely curtailed its expressive power.

Turning back to the relationship between arrow calculus, $\mathcal{A}$, and static arrows, $\mathcal{S}$, we can show that there is no equational embedding corresponding to the syntax inclusion of $\mathcal{A}$ in $\mathcal{S}$ by exhibiting terms that are equal in $\mathcal{S}$ but not in $\mathcal{A}$, a circumstance which is forbidden by the second criterion in our definition of equational embedding (Definition 7).

Lemma 24. *Suppose* b_1 *and* b_2 *are distinct constants of base type* $\mathcal{B}$, *and* c *is a constant of type* $\mathcal{B} \rightsquigarrow 1$. *Let* U *and* V *be the following terms:*

$$U = \textbf{\textit{let}}\ x \Leftarrow c \bullet b_1\ \textbf{\textit{in}}\ [\langle \rangle]$$
$$V = \textbf{\textit{let}}\ x \Leftarrow c \bullet b_2\ \textbf{\textit{in}}\ [\langle \rangle]$$

It can be shown that $U = V$ *in* $\mathcal{S}$.

Proof.

$$\text{\textbf{let} } x \Leftarrow c \bullet b_1 \textbf{ in } [\langle\rangle]$$

$= \qquad (run_1)$

$$\textbf{let } x \Leftarrow \textbf{let } f \Leftarrow \textbf{run } c \textbf{ in } [f\ b_1] \textbf{ in } [\langle\rangle]$$

$= \qquad (\text{assoc})$

$$\textbf{let } f \Leftarrow \textbf{run } c \textbf{ in let } x \Leftarrow [f\ b_1] \textbf{ in } [\langle\rangle]$$

$= \qquad (\text{left})$

$$\textbf{let } f \Leftarrow \textbf{run } c \textbf{ in } [\langle\rangle]$$

$= \qquad (\text{left})$

$$\textbf{let } f \Leftarrow \textbf{run } c \textbf{ in let } x \Leftarrow [f\ b_2] \textbf{ in } [\langle\rangle]$$

$= \qquad (\text{assoc})$

$$\textbf{let } x \Leftarrow \textbf{let } f \Leftarrow \textbf{run } c \textbf{ in } [f\ b_2] \textbf{ in } [\langle\rangle]$$

$= \qquad (run_1)$

$$\textbf{let } x \Leftarrow c \bullet b_2 \textbf{ in } [\langle\rangle]$$

<div align="right">□</div>

Lemma 25. *Let* U *and* V *be defined as in Lemma 24. It cannot be shown that* $\mathsf{U} = \mathsf{V}$ *in* $\mathcal{A}$.

Proof. We will show that U and V cannot be equated in $\mathcal{A}$ by interpreting $\mathcal{A}$ in the more familiar setting of lambda calculus and showing that the interpretations of U and V are not equal. Using the boolean state arrow to interpret $\leadsto$, we set:

$$A \leadsto B = \texttt{Bool} \times A \rightarrow \texttt{Bool} \times B$$

$$\mathcal{B} = \texttt{Bool}$$

$$b_1 = \texttt{True}$$

$$b_2 = \texttt{False}$$

$$c_3 = \texttt{put}_{\leadsto} \qquad (\text{i.e. } \lambda p.\ \langle \textbf{snd } p, \langle\rangle\rangle)$$

and interpret the constructs of the arrow calculus using the translations of $\mathcal{A}$ into $\mathcal{C}$ (Figure 2.7) and the standard definition of the state arrow (Section 2.2.2.2). Then U is interpreted as follows:

$$[\![\textbf{let } x \Leftarrow \texttt{put}_{\leadsto} \bullet \texttt{True in } [\langle\rangle]]\!]_\Delta$$

$= \qquad (\text{def } [\![-]\!])$

$$(\texttt{arr id \&\&\& (arr } (\lambda \Delta.\texttt{True}) \ggg \texttt{put}_{\leadsto})) \ggg \texttt{arr } (\lambda \Delta, x.\ \langle\rangle)$$

$= \qquad (\text{def arr}, \ggg, \&\&\&, \text{etc.})$

$$\lambda\_.\ \langle \texttt{True}, \langle\rangle \rangle$$

and by an identical process V is interpreted as the term $\lambda\_.\ \langle \texttt{False}, \langle\rangle \rangle$, which can evidently not be equated with the interpretation of U. <div align="right">□</div>

Proposition 26. *The syntax inclusion of arrow calculus $\mathcal{A}$ into static arrows $\mathbb{S}$ does not form an equational embedding.*

Proof. If the syntax inclusion of $\mathcal{A}$ into $\mathbb{S}$ were an equational embedding then the equality relations of $\mathcal{A}$ and $\mathbb{S}$ would coincide on terms that do not contain **run** L. However, Lemmas 24 and 25 show that the equality relation of $\mathbb{S}$ is strictly larger.   □

2.5   Future work

We have characterised idioms, monads and arrows as variations on a single calculus, establishing the relative order of strength as *idiom, arrow, monad* in contrast to the putative order of *arrow, idiom, monad*. The variations that bring the arrow calculus into correspondence with idioms and with monads may be characterised either by type isomorphisms or by extensions to the equational theory.

While the beta and eta laws of arrow calculus are more straightforward to work with than the somewhat ad-hoc laws of classic arrows, relating the three interfaces by bringing each into correspondence with a variant of arrow calculus is a somewhat circuitous route. It appears to be possible to establish the relationships more directly, by presenting idioms and monads as variants of classic arrows, in which the operators and laws of classic arrows are augmented with the isomorphisms given above.

However, the arrow calculus has already proved useful independently of our investigations here. Atkey (2008) draws inspiration from the arrow calculus to give a categorical semantics for arrows; Vizzotto, Bois, and Sabry (2009) use the arrow calculus as the basis for a quantum programming language.

Atkey has suggested an alternative formulation of static arrows in which the **run** construct is replaced with a command-level lambda abstraction:

$$\frac{\Gamma; \Delta, \mathrm{x} : A \vdash P \mathbin{!} B}{\Gamma; \Delta \vdash \lambda^{\star}\mathrm{x}.\, P \mathbin{!} A \to B}$$

together with beta and eta laws.

$$
\begin{aligned}
(\beta^{run}) \quad &\mathbf{let}\ \mathrm{f} \Leftarrow \lambda^{\star}\mathrm{x}.\, P\ \mathbf{in}\ [\mathrm{f}\ M] \;\; = \;\; P[\mathrm{x} := M] \\
(\eta^{run}) \quad &\lambda^{\star}\mathrm{x}.\, \mathbf{let}\ \mathrm{f} \Leftarrow P\ \mathbf{in}\ [\mathrm{f}\ \mathrm{x}] \;\; = \;\; P
\end{aligned}
$$

This new form of abstraction and **run** are inter-definable.

$$\lambda^{\star}\mathrm{x}.\, P = \mathbf{run}\ (\lambda^{\bullet}\mathrm{x}.\, P)$$

$$\mathbf{run}\ L = \lambda^{\star}\mathrm{x}.\, L \bullet \mathrm{x}$$

The duality of this approach is particularly appealing: static arrows add a new form of abstraction, while higher-order arrows add a new form of application. The laws of this new form of abstraction also fit the standard pattern. However, there are apparent disadvantages to the proposed laws: they appear less convenient to use, and it is not clear how to obtain rewrite rules from them.

The arrow calculus is the analogue for arrows of Moggi's computational metalanguage (Moggi, 1991). A further direction for possible future investigation is an analogue for arrows of Moggi's (1989) computational lambda calculus.

Another unresolved question, raised in Section 2.2.3.2, is whether there are monads m other than the environment monad for which $\alpha \to m\ \beta \simeq m\ (\alpha \to \beta)$. For such instances the idiom, arrow and monad interfaces are equivalently powerful (Proposition 15, Proposition 22).

Chapter 3

Abstracting controls [1]

3.1 Introduction

Suppose we want to present users with an HTML form for entering a pair of dates (such as an arrival and departure date for booking a hotel). In our initial design, we represent a date using a single text field. Later, we choose to replace each date by a pair of pulldown menus, one to select a month and one to select a day.

In typical web frameworks, such a change will require widespread modifications to the code. Under the first design, the HTML form will contain *two text fields*, and the code that handles the response will need to extract and parse the text entered in each field to yield a pair of values of appropriate type, perhaps an abstract date type. Under the second design, however, the HTML will contain *four menus*, and the code that handles the response will need to extract the choices for each menu and combine them in pairs to yield each date.

How can we structure a program so that it is isolated from this choice? We want to capture the notion of *a part of a form*, specifically a part for collecting values of a given type or purpose; we call such an abstraction a *formlet*. The designer of the formlet should choose the HTML presentation, and decide how to process the input into a date value. Clients of the formlet should be insulated from the choice of HTML presentation, and also from the calculation that yields the abstract value. And, of course, we should be able to compose formlets to build larger formlets.

Once described, this sort of abstraction seems obvious and necessary. But remarkably few web frameworks support it. Three existing web programming frameworks that do support some degree of abstraction over form components are WASH (Thiemann, 2005), iData (Plasmeijer and Achten, 2006) and WUI (Hanus, 2006, 2007), each having distinctive features and

---

[1]This chapter is a revision of Cooper et al. (2008a).

limitations. (We discuss these further in Section 3.8.)

Our contribution is to reduce form abstraction to its essence. We show that a semantics of formlets can be obtained by the composition of the three standard *idioms* that capture the effects needed for form abstraction. We will further argue that while idioms are a good fit for capturing form abstraction, monads and arrows are not. Furthermore, we illustrate how the semantics can be extended to support additional features (such as checking form input for validity), either by composing with additional standard idioms or by generalising to *indexed* and *parameterised* idioms.

We will begin with an example, showing how formlets might appear to the programmer (Section 3.2). Subsequent sections give a semantics in terms of idioms (Section 3.3), define the syntactic sugar used throughout the chapter (Section 3.4), show how formlets fit into a web programming environment (Section 3.5), and show how to extend the basic abstraction with static XHTML validation and user-input validation (Section 3.6). We conclude by describing a number of implementations of formlets in various languages (Section 3.7) and discussing related work (Section 3.8).

Formlets can be implemented in any functional programming language; we present them here in OCaml. OCaml is a particularly good fit for our exposition: we can give the semantics in terms of idiom composition using functors and define the new syntactic forms using Camlp4, the standard tool for extending OCaml syntax.

3.2 Formlets by example

Now we illustrate formlets, as they might appear to the programmer, with an example (Figure 3.1). We assume familiarity with HTML and OCaml. This section covers our OCaml implementation, and so has features that may vary in another implementation of formlets. We use a special syntax for programming with formlets, which is defined formally in Section 3.4; this syntax is part of the implementation, and makes formlets easier to use, but is not an essential part of the abstraction.

The formlet `date_formlet` has two text input fields, labelled "Month" and "Day." Upon submission, this formlet will yield a `date` value representing the date entered. The user-defined `make_date` function translates the day and month into a suitable representation.

A formlet expression consists of a *body* and a *yields clause*. The formlet expression `date_formlet` has the body

```
let date_formlet : date formlet = formlet
  <div>
    Month: {input_int ⇒ month}
    Day:   {input_int ⇒ day}
  </div>
  yields make_date month day

let travel_formlet : (string × date × date) formlet =
formlet
  <#>  Name: {input ⇒ name}
    <div>
      Arrive: {date_formlet ⇒ arrive}
      Depart: {date_formlet ⇒ depart}
    </div>
    {submit "Submit"}
  </#>
  yields (name, arrive, depart)

let display_itinerary : string × date × date → xml =
  fun (name, arrive, depart) →
  <html>
    <head><title>Itinerary</title></head>
    <body>
      Itinerary for: {xml_text name}
      Arriving: {xml_of_date arrive}
      Departing: {xml_of_date depart}
    </body>
  </html>

handle travel_formlet display_itinerary
```

Figure 3.1: Date example

```
let date_formlet : date formlet =
  pure (fun (month,day) → make_date month day)
  ⊛ (tag "div" []
    (pure (fun () month () day () → (month,day))
      ⊛ xml (xml_text "Month: ")  ⊛ input_int
      ⊛ xml (xml_text "Day:   ")  ⊛ input_int
      ⊛ xml (xml_text "\n     ")))

let travel_formlet : (string × date × date) formlet =
  pure (fun (name, (arrive,depart)) →
       (name, arrive, depart))
  ⊛ (pure (fun () name (arrive,depart) () →
             (name, (arrive,depart)))
    ⊛ xml (xml_text "Name: ") ⊛ input
    ⊛ (tag "div" []
       (pure (fun () arrive () depart → (arrive,depart))
         ⊛ xml (xml_text "Arrive: ") ⊛ date_formlet
         ⊛ xml (xml_text "Depart: ") ⊛ date_formlet))
    ⊛ xml (submit "Submit"))

let display_itinerary : string × date × date → xml =
  fun (name, arrive, depart) →
  xml_tag "html" []
    (xml_tag "head" []
      (xml_tag "title" [] (xml_text "Itinerary")) @
    (xml_tag "body" []
      (xml_text "Itinerary for: " @ xml_text name @
       xml_text "Arriving: " @ xml_of_date arrive @
       xml_text "Departing: " @ xml_of_date depart)))

handle travel_formlet display_itinerary
```

Figure 3.2: Date example (desugared and simplified)

```
<div>
 Month: {input_int ⇒ month}
 Day:   {input_int ⇒ day}
</div>
```

and its yields clause is

```
make_date month day
```

The body of a formlet expression is a *formlet quasiquote*. This is like an XML literal expression but with embedded *formlet bindings*. A formlet binding $\{f \Rightarrow p\}$ binds the result yielded by f to the pattern p within the scope of the yields clause. Here f is an expression that evaluates to a formlet and the type yielded by the formlet must be the same as the type accepted by the pattern. Thus the variables `month` and `day` will be bound to the values yielded by the two instances of the `input_int` formlet. The bound formlet f will render some HTML which will take the place of the formlet binding when the outer formlet is rendered.

The value `input_int` is a formlet of type `int formlet` that renders as an HTML text input element and parses the submission as type `int`. It is built from the primitive formlet `input` which presents an input element and yields the entered string. Although `input_int` is used here twice, the system prevents any field name clashes.

It is important to realise that any given formlet defines behavior at two distinct points in the program's execution: first when the form structure is built up, and much later (if at all) when the form is submitted by the user, when the outcome is processed. The behaviour at the first point is determined by the body of the formlet and the behaviour at the second point by the yields clause.

Next we illustrate how user-defined formlets can be usefully combined to create larger formlets. Continuing Figure 3.1, `travel_formlet` asks for a name, an arrival date, and a departure date. The library function `submit` returns the HTML for a submit button; its string argument provides the label for the button.

(The syntax $<\#> \cdots </\#>$ enters the XML parsing mode without introducing a root XML node; its result is an XML forest, with the same type as XML values introduced by a proper XML tag. The syntax $<\#/>$ denotes an empty XML forest. We borrow this notation from WASH.)

Having created a formlet, how do we use it? For a formlet to become a form, we need to connect it with a *handler*, which will consume the form input and perform the rest of the user interaction. The function `handle` attaches a handler to a formlet; we describe it in more detail in Section 3.5.

Continuing the above example, we render `travel_formlet` onto a full web page, and

```
type xml = xml_item list
 and tag = string
 and attrs = (string × string) list
 and xml_item   (* abstract *)

val xml_tag : tag → attrs → xml → xml
val xml_text : string → xml
```

Figure 3.3: The `xml` abstract type.

attach a handler, `display_itinerary`, that displays the chosen itinerary back to the user. (The abstract type `xml` is given in Figure 3.3; we construct XML using special syntax, which is defined in terms of the `xml_tag` and `xml_text` functions, as shown formally in Section 3.4.)

This is a simple example; a more interesting application might render another form on the `display_itinerary` page, one which allows the user to confirm the itinerary and purchase tickets; it might then take actions such as recording the purchase in a database, and so on.

This example demonstrates the key characteristics of the formlet abstraction: static binding (we cannot fetch the value of a form field that is not in scope), structured results (the month and day fields are packaged into an abstract `date` type, which is all the formlet consumer sees), and composition (we reuse the date formlet twice in `travel_formlet`, with no danger of field-name clashes).

3.2.1 Syntactic sugar

Figure 3.2 shows the desugared version of the date example.

XML values are constructed using the `xml_tag` and `xml_text` functions and the standard list concatenation operator, `@`. Formlet values are only slightly more complicated. The `xml` function lifts an XML value into the formlet; the `tag` function is a counterpart to `xml_tag` for formlets (Figure 3.5); composition of formlets makes use of the standard idiom operations `pure` and ⊗ (Figure 3.4). Section 3.3 covers the formlet primitives in detail.

The sugar makes it easier to freely mix static XML with formlets. Without the sugar, dummy bindings are needed to bind formlets consisting just of XML (see the unit arguments to the `pure` functions in Figure 3.2), and formlets nested inside XML have to be rebound (see the second call to `pure` in the body of `travel_formlet` in Figure 3.2). Section 3.4 gives a desugaring algorithm and shows how to simplify the generated code.

3.2.2   Life without formlets

Now consider implementing the above example using the standard HTML/CGI interface. We
would face the following difficulties with the standard interface:

- There is no static association between a form definition and the code that handles it,
  so the interface is fragile. This means the form and the handling code need to be kept
  manually in sync.

- Field values are always received individually and always as strings: the interface pro-
  vides no facility for processing data or giving it structure.

- Given two forms, there is generally no easy way to combine them into a new form with-
  out danger of name clashes amongst the fields—thus it is not easy to write a form that
  abstractly uses subcomponents. In particular, using a form twice within a larger form is
  difficult.

Conventional web programming frameworks such as PHP (PHP) and Ruby on Rails (Hans-
son, 2008) enable abstraction only through templating or textual substitution, hence there is
no automatic way to generate fresh field names, and any form "abstraction" (such as a tem-
plate) still exposes the programmer to the concrete field names used in the form. Even ad-
vanced systems such as PLT Scheme (Graunke et al., 2001b), JWIG (Christensen et al., 2003),
scriptlets (Elsman and Larsen, 2004), Ocsigen (Balat, 2006), Lift (lif, 2008) and the original
design for Links (Cooper et al., 2006) all fall short in the same way.

Formlets address all of the above problems: they provide a static association between a
form and its handler (ensuring that fields referenced actually exist and are of the right type),
they allow processing raw form data into structured values, and they allow composition, in part
by generating fresh field names at runtime.

3.3   Semantics

We wish to give a semantics of formlets using a well-understood formalism. We shall show
that formlets conform to the *idiom* interface described in Section 2.2.3. We begin with a con-
crete implementation in OCaml, which we then factor using standard idioms to give a formal
semantics.

Figure 3.4 gives the idiom interface in OCaml.

```
module type IDIOM =
sig
  type α t
  val pure : α → α t
  val (⊛) : (α → β) t → α t → β t
end
```

Figure 3.4: The idiom interface

```
type env = (string × string) list

module type FORMLET =
sig
  include IDIOM
  val xml   : xml → unit t
  val tag   : tag → attrs → α t → α t
  val input : string t
  val run   : α t → xml × (env → α)
end
```

Figure 3.5: The formlet interface

3.3.1 A concrete implementation

Figures 3.5 and 3.6 give a concrete implementation of formlets in OCaml.

The type $\alpha\,t$ is the type of formlets that return values of type α (the library exposes this type at the top-level as α `formlet`). Concretely $\alpha\,t$ is defined as a function that takes a *name source* (integer) and returns a triple of an XML *rendering*, a *collector* function of type `env` $\to\alpha$ and an updated name source. The formlet operations ensure that the names generated in the rendering are the names expected (in the environment) by the collector.

The `pure` operation is used to create constant formlets whose renderings are empty and whose collector always returns the same value irrespective of the environment. The ⊛ operation applies an $(\alpha \to \beta)$ `formlet` to an α `formlet`. The name source is threaded through each formlet in turn. The resulting renderings are concatenated and the collectors composed. Together `pure` and ⊛ constitute the fundamental idiom operations. (To constitute an idiom they must, of course, satisfy the four laws given in Section 2.2.3.3. It is straightforward to verify these laws for `Formlet`.)

The `xml` operation creates a formlet whose rendering is the argument; the `tag` operation wraps the given formlet's rendering in a new element with the specified tag name and attributes.

The primitive formlet `input` generates HTML input elements. A single name is generated

```
  module Formlet : FORMLET = struct
    type α t = int → (xml × (env → α) × int)

    let pure x i₀ = ([], const x, i₀)
    let (⊛) f p i₀ = let (x, g, i₁) = f i₀ in
                     let (y, q, i₂) = p i₁ in
                       (x @ y, (fun env → g env (q env)), i₂)

    let xml x i₀ = (x, const (), i₀)
    let tag t attrs fmlt i₀ = let (x, f, i₁) = fmlt i₀ in
                                (xml_tag t attrs x, f, i₁)

    let next_name i₀ = ("input_" ^ string_of_int i, i₀ + 1)
    let input i₀ = let (w, i₁) = next_name i₀ in
                     (xml_tag "input" [("name", w)] [],
                      List.assoc w, i₁)

    let run c = let (x, f, _) = c 0 in (x, f)
  end
```

Figure 3.6: The formlet idiom

---

from the name source, and this name is used both in the rendering and the collector. The full implementation includes a range of other primitive formlets for generating the other HTML form elements such as `<textarea>` and `<option>`.

The `run` operation "runs" a formlet by supplying it with an initial name source (we use 0); this produces a rendering and a collector function.

3.3.2 Idioms

We saw in Chapter 2 that idioms capture the idea of *oblivious* computation. Using idioms, the result of each subcomputation is not available to other subcomputations; rather, all computations are executed before their results are collated as a final step. Formlets fit this pattern: the sub-formlets cannot depend on one another's results, and the final result yielded by a formlet is a pure function of the results of the sub-formlets.

3.3.3 Factoring formlets

Now we introduce the three basic idioms into which the formlet idiom factors. (Figures 3.7, 3.8, and 3.9). Besides the standard idiom operations in the interface, each idiom comes with operations corresponding to primitive effects and a *run* operation for executing the effects

```
  module Namer :
  sig
    include IDIOM
    val next_name : string t
    val run : α t → α
  end =
  struct
    type α t = int → α × int
    let pure v i₀ = (v, i₀)
    let (⊛) f p i₀ = let (g, i₁) = f i₀ in
                        let (q, i₂) = p i₁ in
                          (g q, i₂)
    let next_name i₀ = ("input_" ^ string_of_int i₀, i₀+1)
    let run v = fst (v 0)
  end
```

Figure 3.7: The name generation idiom

```
  module Environment :
  sig
    include IDIOM
    val lookup : string → string t
    val run : α t → env → α
  end =
  struct
    type α t = env → α
    and env = (string × string) list
    let pure v e = v
    let (⊛) f p e = f e (p e)
    let lookup = List.assoc
    let run v = v
  end
```

Figure 3.8: The environment idiom

and extracting the final result. A computation in the `Namer` idiom (Figure 3.7) has type $int \to \alpha \times int$; it is a function from a counter to a result and a possibly-updated counter. The `next_name` operation uses this counter to construct a fresh name, updating the counter. The `Namer` idiom may be obtained from the standard state monad (Section 2.2.1.2); however, whereas in the monad the `next_name` can be defined in terms of `get` and `put`, in the idiom obliviousness requires that we provide it as a primitive. A computation in the `Environment` idiom (which we defined in Haskell in Section 2.2.3.2) has type $env \to \alpha$; it receives an environment and returns a result. The `lookup` operation retrieves values from the environment by

```
module XmlWriter :
sig
  include IDIOM
  val xml : xml → unit t
  val tag : tag → attrs → α t → α t
  val run : α t → xml × α
end =
struct
  type α t = xml × α
  let pure v = ([], v)
  let (⊛) (x, f) (y, p) = (x @ y, f p)
  let xml x = (x,())
  let tag t a (x,v) = (xml_tag t a x, v)
  let run v = v
end
```

Figure 3.9: The XML accumulation idiom

```
module Compose (F : Idiom) (G : Idiom) :
sig
  include IDIOM with type α t = (α G.t) F.t
  val refine : α F.t → (α G.t) F.t
end =
struct
  type α t = (α G.t) F.t
  let pure x = F.pure (G.pure x)
  let (⊛) f x = F.pure (⊛_G) ⊛_F f ⊛_F x
  let refine v = F.pure G.pure ⊛_F v
end
```

Figure 3.10: Idiom composition

name. As we saw in Section 2.2.3.2, the environment idiom may be obtained directly from the
standard environment monad. A computation in the XmlWriter idiom has type $xml \times \alpha$ and
so produces both XML and a result; the XML is generated by the primitive xml and tag oper-
ations and concatenated using ⊛. As with Namer and Environment, XmlWriter corresponds
directly to a standard monad, the monoid accumulator. The monoid is the free monoid on the
set of XML nodes, and pure and ⊛ correspond to its unit and multiplication.

Idioms can be combined in a variety of ways to form new idioms: for example, the product
of two idioms is also an idiom. We have chosen to obtain XmlWriter from the accumulator
monad, but it can also be obtained as the product of the AccI phantom monoid idiom (Sec-
tion 2.2.3.2) and the identity idiom.

```
module Formlet : FORMLET = struct
  module N = Namer
  module A = XmlWriter
  module E = Environment
  module AE = Compose (A) (E)
  include Compose (N) (AE)
  let xml x = N.pure (AE.refine (A.xml x))
  let tag t ats f = N.pure (A.tag t ats) ⊛N f
  let input = N.pure (fun n → A.tag "input" [("name",n)]
                              (A.pure (E.lookup n)))
            ⊛N N.next_name
  let run v = let (xml, collector) = A.run (N.run v) in
              (xml, E.run collector)
end
```

Figure 3.11: The formlet idiom (factored)

We define idiom composition in OCaml using a functor, Compose (Figure 3.10), the OCaml analogue of the Idiom instance for the Compose type in Section 2.2.3.2. The formlet idiom is just the composition of the three primitive idioms Namer, XmlWriter, and Environment (Figure 3.11).

To work with a composed idiom, we need to be able to lift the primitive operations from the component idioms into the composed idiom. Given idioms F and G, we can lift any idiomatic computation of type α G.t to an idiomatic computation of type (α G.t) F.t using F.pure, and lift one of type α F.t to one of type (α G.t) F.t using Compose(F)(G).refine.

In defining the composed formlet idiom, a combination of N.pure and AE.refine is used to lift the result of the A.xml operation. The tag operation is lifted differently as its third argument is a formlet: here we apply the A.tag t ats operation to it. The run operation simply runs each of the primitive run operations in turn. The input operation is the most interesting. It generates a fresh name and uses it both to name an input element and, in the collector, for lookup in the environment.

3.3.4 A note on monads and arrows

We saw in Chapter 2 that the idiom interface is the least powerful of three alternatives. Could we use the more powerful (and more widely used) *monad* interface instead to offer additional power to users of formlets? In fact, it is not difficult to see that there is no monad corresponding to the formlet type.

Intuitively, the problem is that a ≫= operation for formlets

$$(\ggeq) \; :: \; \alpha \; \texttt{formlet} \; \rightarrow \; (\alpha \rightarrow \beta \; \texttt{formlet}) \; \rightarrow \; \beta \; \texttt{formlet}$$

would involve passing the result of the left operand to the right operand, a function which constructs a formlet. However, none of the results of the various formlets on a page can be made available until all of the formlets have been fully constructed and presented to the user, since the results are only obtained after the form has been submitted. Thus, although each of the primitive idioms used to construct formlets is also a monad, their composition is not a monad.

Our focus in this chapter is on the composition of formlets within a single page. However, a typical web interaction involves a sequence of such pages, each containing forms. It may be that monads are more suited than idioms to structuring interactions, where the result of each page is typically used in constructing the next page in the sequence, and indeed WASH uses monads in this way (Thiemann, 2005). (Hughes (2000) suggests that arrows, which also favour sequential composition, are even more suitable for this task.)

As we saw in Chapter 2, every idiom yields an arrow, so we could also use the arrow interface for formlets. However, since the arrow thus obtained is *static*, we would not gain any additional power by doing so; since arrows are also more cumbersome to use in practice (at least without the special notation described in Section 2.3.2), we choose not to follow this route.

3.4   Syntax

The syntax presented in Section 3.2 can be defined as syntactic sugar, which desugars into uses of the basic formlet operations. Here we formally define the syntax and its translation. We add two new kinds of expression: XML quasiquotes, (or XML literals with embedded evaluable expressions), and formlet expressions, denoting formlet values. Figure 3.12 gives the grammar for these expressions. Using Camlp4 it is easy to extend OCaml with new syntactic forms in this way and our implementation closely follows the formal description given in this section.

The desugaring transformations are shown in Figure 3.13. The operation $[\![\cdot]\!]$ desugars the formlet expressions in a program; it is a homomorphism on all syntactic forms except XML quasiquotes and formlet expressions. The operation $(\cdot)^*$ desugars XML quasiquotes and nodes. The operation $z^{\dagger}$ denotes a pattern aggregating the sub-patterns of z where z ranges over formlet quasiquotes and nodes. In an abuse of notation, we also let $z^{\dagger}$ denote the expression that reconstructs the value matched by the pattern. (Of course, we need to be somewhat careful in the OCaml implementation to properly reconstruct the value from the matched pattern.) Finally, z° is a formlet that tuples the outcomes of sub-formlets of z.

Expressions

$$e \quad ::= \quad \cdots \mid r \qquad\qquad\qquad\qquad \text{(XML)}$$

$$\mid \textbf{formlet } q \textbf{ yields } e \quad \text{(formlet)}$$

XML quasiquotes

$$m \quad ::= \quad s \mid \{\, e \,\} \mid \texttt{<}t \; ats\texttt{>}m_1 \ldots m_k\texttt{</}t\texttt{>} \qquad\qquad \text{node}$$

$$r \quad ::= \quad \texttt{<}t \; ats\texttt{>}m_1 \ldots m_k\texttt{<}t\texttt{>} \mid \texttt{<#>}m_1 \ldots m_k\texttt{</#>} \quad \text{quasiquote}$$

Formlet quasiquotes

$$n \quad ::= \quad s \mid \{\, e \,\} \mid \{\, f \Rightarrow p \,\} \mid \texttt{<}t \; ats\texttt{>}n_1 \ldots n_k\texttt{</}t\texttt{>} \quad \text{node}$$

$$q \quad ::= \quad \texttt{<}t \; ats\texttt{>}n_1 \ldots n_k\texttt{</}t\texttt{>} \mid \texttt{<#>}n_1 \ldots n_k\texttt{</#>} \quad \text{quasiquote}$$

Meta variables

e expression	f formlet-type expression	t	tag
p pattern	s string	ats	attribute list

Figure 3.12: Quasiquote syntax.
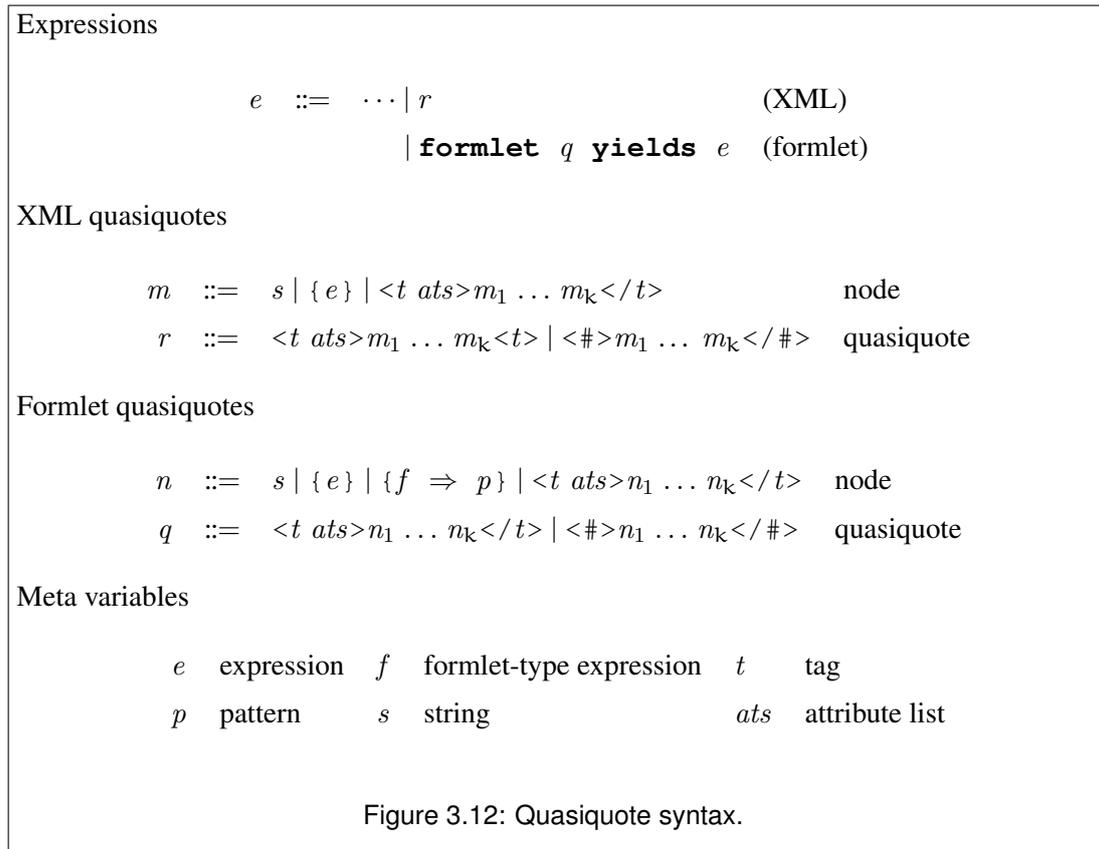
As a simple example of desugaring, consider the `input_int` formlet used earlier. We can define it as follows using the formlet syntax:

```
let input_int : int formlet =
  formlet <#>{input ⇒ i}</#>
  yields int_of_string i
```

Under the translation given in Figure 3.13, the body becomes

```
pure (fun i → int_of_string i) ⊛ (pure (fun i → i) ⊛ input)
```

We can use the idiom laws (and η-reduction) to simplify the output a little, giving the following semantically-equivalent code:

```
pure int_of_string ⊛ input
```

As a richer example, recall `date_formlet` from Figure 3.1, reproduced below:

```
let date_formlet : date formlet = formlet
  <div>
   Month: {input_int ⇒ month}
   Day:   {input_int ⇒ day}
  </div>
  yields make_date month day
```

$$\llbracket r \rrbracket \;=\; r^*$$

$$\llbracket \textbf{formlet } q \textbf{ yields } e \rrbracket \;=\; \texttt{pure } (\textbf{fun } q^\dagger \to \llbracket e \rrbracket) \;\circledast\; q^\circ$$

$$s^* \;=\; \texttt{xml\_text } s$$

$$\{e\}^* \;=\; \llbracket e \rrbracket$$

$$(\texttt{<}t\ ats\texttt{>}m_1 \ \ldots\ m_\mathrm{k}\texttt{</}t\texttt{>})^* \;=\; \texttt{xml\_tag } t\ ats\ (\texttt{<\#>}m_1 \ \ldots\ m_\mathrm{k}\texttt{</\#>})^*$$

$$(\texttt{<\#>}m_1 \ \ldots\ m_\mathrm{k}\texttt{</\#>})^* \;=\; m_1^* \ @ \ \cdots \ @\ m_\mathrm{k}^*$$

$$s^\circ \;=\; \texttt{xml (xml\_text } s)$$

$$\{e\}^\circ \;=\; \texttt{xml } \llbracket e \rrbracket$$

$$\{f \Rightarrow p\}^\circ \;=\; \llbracket f \rrbracket$$

$$(\texttt{<}t\ ats\texttt{>}n_1 \ \ldots\ n_\mathrm{k}\texttt{</}t\texttt{>})^\circ \;=\; \texttt{tag } t\ ats\ (\texttt{<\#>}n_1 \ \ldots\ n_\mathrm{k}\texttt{</\#>})^\circ$$

$$(\texttt{<\#>}n_1 \ \ldots\ n_\mathrm{k}\texttt{</\#>})^\circ \;=\; \texttt{pure } (\textbf{fun } n_1^\dagger \ \ldots\ n_\mathrm{k}^\dagger \to (n_1^\dagger, \ \ldots, n_\mathrm{k}^\dagger))$$

$$\circledast\, n_1^\circ \ \ldots\ \circledast\ n_\mathrm{k}^\circ$$

$$s^\dagger \;=\; ()$$

$$\{e\}^\dagger \;=\; ()$$

$$\{f \Rightarrow p\}^\dagger \;=\; p$$

$$(\texttt{<}t\ ats\texttt{>}n_1 \ \ldots\ n_\mathrm{k}\texttt{</}t\texttt{>})^\dagger \;=\; (n_1^\dagger, \ \ldots, n_\mathrm{k}^\dagger)$$

$$(\texttt{<\#>}n_1 \ \ldots\ n_\mathrm{k}\texttt{</\#>})^\dagger \;=\; (n_1^\dagger, \ \ldots, n_\mathrm{k}^\dagger)$$

Figure 3.13: Desugaring XML and formlets.

In this case the desugaring algorithm produces the following output:

```
let date_formlet : date formlet =
  pure (fun ((),month,(),day,()) → make_date month day)
  ⊛ (tag "div" []
      (pure (fun () month () day () → ((),month,(),day,()))
         ⊛ xml (xml_text "Month: ") ⊛ input_int
         ⊛ xml (xml_text "Day:   ") ⊛ input_int
         ⊛ xml (xml_text "\n     "))))
```

It is easy to optimise this code by removing the extra units from the body of the inner `pure` and from the arguments to the function in the outer `pure`, and we have applied this minor optimisation in the simplified version given in Figure 3.2. However, the code is still rather inefficient: in particular, the variables `month` and `day` are rebound in each pure function. Similarly, in the `travel_formlet` of Figure 3.2 the variables `arrive` and `depart` are bound three times.

Section 3.6.3 outlines an alternate desugaring that obviates this rebinding.

3.4.1 Completeness

The formlet syntax is complete: everything expressible with the formlet operations can be expressed directly in the syntax. For example, the ⊛ operator of the formlet idiom may be written as a function ap using syntactic sugar:

```
let ap : (α → β) formlet → α formlet → β formlet =
  fun f p → formlet <#>{f ⇒ g}{p ⇒ q}</#> yields g q
```

Under the desugaring transformation, the body becomes

```
(pure (fun (g,q) → g q)) ⊛ (pure (fun g q → (g,q)) ⊛ f ⊛ p)
```

which, under the idiom laws, is equivalent to f ⊛ p. And pure, too, can be defined in the sugar:

```
let pure : α → α formlet =
  fun x → formlet <#/> yields x
```

In this case desugaring produces the following code for the body:

```
pure (fun () → x) ⊛ pure ()
```

which is equal to pure x under the homomorphism law for idioms.

3.5 Pragmatics

We now consider the practicalities of using formlets in a typical web programming environment. We assume that the runtime system provides the following functions:

```
val pickle_cont : (α → β) → string
val unpickle_cont : string → (α → β)
val cgi_args : unit → env
val send_document : xml → unit
```

The first two, pickle_cont, and unpickle_cont, convert a function to and from a textual representation. We leave the details of the operation of these functions unspecified: for example, pickle_cont might serialise the code underlying the function to a string, or store the function in a table and return a string that acts as a key into the table. (We describe the approach taken in Links in the next chapter.) We will use pickle_cont and unpickle_cont to persist continuation functions for formlets across requests. The types are inherently unsafe: there is nothing in the types of the functions to prevent the caller from pickling a function at one type and unpickling it at another.

The third function, `cgi_args`, retrieves the CGI arguments submitted with the current request: these correspond to the values entered into a form by the user. The fourth, `send_document`, sends an XML document to the client, completing the response.

Rather than use these functions directly in programming with formlets, we provide the following interface, which is both safer and more convenient.

```
val handle   : α formlet → (α → xml) → xml
val interact : (unit → xml) → unit
```

We have already used the `handle` function in the opening example (Figure 3.1): it attaches a continuation function to a formlet. It may be implemented as follows:

```
let handle formlet handler =
  let xml, collector = Formlet.run formlet in
  let continuation env = handler (collector env) in
  <form>
    <input type="hidden" name="_k"
           value="{pickle_cont continuation}"/>
    {xml}
  </form>
```

The `handle` function extracts rendering and collector from the formlet, and inserts the rendering it into a form together with the serialised continuation, which it stores as a hidden field. The continuation uses the value returned by the collector as the argument to the handler, which constructs the next page.

The `interact` function is intended for use at top-level; it accepts a function which constructs the initial page. Recall that a CGI program is invoked either to generate a page or to process a form submission. We assume here that the same program is run on each occasion to perform one of these tasks. The first time that the program is invoked in a sequence of interactions, `interact` calls this function to construct the initial page. On subsequent invocations of the program, `interact` extracts the CGI arguments, and rebuilds and runs the continuation of the last request. The `interact` function may be implemented as follows:

```
let interact make_page =
    send_document
      (match cgi_args with
         | [] → make_page ()
         | args → unpickle_cont (List.assoc "_k" args) args)
```

Note that the only time that there are no CGI arguments is when the program is initially run; on subsequent invocations there is always at least one CGI argument, namely the continuation, `"_k"`.

```
module type XIDIOM =
sig
  type (ψ, α) t
  val pure : α → (ψ, α) t
  val (⊛) : (ψ, α → β) t → (ψ, α) t → (ψ, β) t
end
```

Figure 3.14: The indexed idiom interface

3.6 Extensions

The formlet abstraction is robust, as we proceed to demonstrate by extending it in a number of ways.

3.6.1 XHTML validation

The problem of statically enforcing validity of HTML (and indeed, XML) is well-studied (Brabrand, Møller, and Schwartzbach, 2001, Thiemann, 2002, Hosoya and Pierce, 2003, Møller and Schwartzbach, 2005). Such schemes are essentially orthogonal to the work presented here: we can incorporate a type system for XML with little disturbance to the core formlet abstraction.

Of course, building static validity into the type system requires that we have a whole family of types for HTML rather than just one. For instance, we might have separate types for `block` and `inline` entities (as in Elsman and Larsen's (2004) SMLserver), or even a different type for every tag (as in Hosoya and Pierce's (2003) XDuce).

Fortunately, it is easy to push the extra type parameters through our formlet construction. The key component that needs to change is the `XmlWriter` idiom, to which we must add an extra parameter denoting the XML type. The construction we need is what we call an *indexed idiom*; it is roughly analogous to Wadler and Thiemann's (2003) effect-indexed monad. Figure 3.14 gives the definition of an indexed idiom, `XIDIOM`, in OCaml. Like idioms, indexed idioms satisfy the four laws given in Section 3.3. They can be pre- and post-composed with other idioms to form new indexed idioms. Pre-composing the name generation idiom with the indexed XML writer idiom pre-composed with the environment idiom gives us an indexed formlet idiom.

As a proof of concept, we have implemented a prototype of formlets with XML typing in OCaml using a minor variant of Elsman and Larsen's (2004) encoding of MiniXHTML, a fragment of XHTML 1.0.

Figure 3.15 gives a partial definition of MiniXHTML in OCaml. XML values are repre-

```
    type (+β, +ι) flw
    type blk
    type inl
    type no
    type inpre
     ...
    module MiniXHTML :
    sig
     type +ψ t
     val text : string → ψ t
     val p : ((no,inl) flw × γ) t → ((blk,ι) flw × γ) t
     val pre : ((no,inl) flw × inpre) t → ((blk,ι) flw × γ) t
     ...
     val empty : ψ t
     val concat : ψ t → ψ t → ψ t
    end =
    struct
     type +ψ t = xml
     ...
    end
```

Figure 3.15: MiniXHTML fragment

sented using the type +ψ MiniXHTML.t, with the parameter ψ encoding information about the contexts in which those values are permitted to appear[2]. The type parameter ψ is a *phantom type*: it does not appear in the definition of t (which is simply the xml type used throughout this chapter), but is used only to enforce constraints on the contexts in which values of t may be used. In MiniXHTML, ψ is always instantiated with a pair of types, the first of which represents the "entity type" of the element and the second whether the element contains pre elements as descendents. (Elsman and Larsen use a pair of type parameters instead of a single parameter instantiated with a pair.) The entity type is constructed from the uninhabited types flw, blk, inl and no, which indicate whether an element is a flow entity, whether it contains block entities, and so on. For example, the type of p specifies that the entity type of its argument is (no,inl) flw, an inline entity which does not contain block entities, and that the entity type of the result is (blk,ι) flw, a block entity. (Further details of the typing scheme may be found in Elsman and Larsen (2004).)

Figure 3.16 gives a definition of the XML-indexed formlet interface, which extends the formlet interface of Figure 3.5, to support static XHTML validation. There are now two type

---

[2]The covariance annotation + on the parameterised MiniXHTML.t and flw types increases the opportunities for polymorphism under OCaml's relaxed value restriction (Garrigue, 2002).

```
module type XFORMLET =
sig
 include XIDIOM
 val text : string → (ψ,unit) t
 val p : ((no,inl) flw × γ,α) t → ((blk,ι) flw × γ,α) t
 val pre : ((no,inl) flw × inpre,α) t → ((blk,ι) flw × γ,α) t
 ...
 val input : ((no,inl) flw × γ,unit) t
           → ((β,inl) flw × γ,string) t
 val run : (ψ,α) t → ψ MiniXHTML.t × (env → α)
end
```

Figure 3.16: The XML-indexed formlet interface

```
type α perhaps = Value of α | Error of string

module type VFORMLET = sig
  include IDIOM
  val xml   : xml → unit t
  val tag   : tag → attrs → α t → α t
  val input : string t
  val run   : α t → (xml × (env → (xml, α) either))
  val extract : (α → β perhaps) → α formlet → β formlet
end
```

Figure 3.17: The input-validating formlet interface

parameters, which denote the type of XML emitted when the formlet is rendered and the type of the result yielded when the resulting form is submitted. In place of a single operation for lifting XML into the formlet we now have one operation for each element: p, pre, input, and so on. The desugaring given in Figure 3.13 must be altered accordingly, so that the rule for desugaring XML literals is replaced with a family of rules, one for each tag.

It is straightforward to extend the definitions here to include the full set of entities given by Elsman and Larsen and to handle XML attributes.

3.6.2 Input validation

Validation of user input is a common need in form processing: on submission, the program should check whether the data is ill-formed, and if so, repeatedly re-display the form to the user (with error messages) as long as ill-formed data is submitted.

Formlets extend to this need if we make two changes to the interface. The first is the

```
let int (s : string) : int perhaps =
  try Value (int_of_string s)
  with Failure → Error (s ^ " isn't an integer")

let checked_int : int formlet =
  formlet <#>{extract int input ⇒ v}</#>
  yields v
```

Figure 3.18: The input-validating formlet `checked_int`.

```
let positive_int : int formlet =
  extract
    (fun i →
        if i > 0 then Value i
        else Error (string_of_int s ^ " isn't positive"))
  checked_int
```

Figure 3.19: The input-checking formlet `positive_int`.

addition of a new function, `extract`, for attempting to extract a value from a submitted formlet (Figure 3.17). The `extract` function takes two arguments: a "partial" function which attempts to convert a value of type α to a value of type β, and an input-validating formlet which attempts to return a result of type α; it returns a new input-validating formlet that attempts to return a result of type β. The second change to the interface is an adjustment to the type of the collector function returned by `run`. This now returns either a value (if the input is valid) or an error page (if it is not).

The `input_int` formlet presented earlier inherits its error checking behaviour from the `int_of_string` function: it simply aborts execution with a `Failure` exception if parsing fails. Using `extract` we can write a new formlet, `checked_int`, which intercepts the exception and adds a suitable error message to the page, which is then redisplayed to the user (Figure 3.18).

The `extract` function can be used to add validation to any formlet, regardless of whether the formlet already includes validation. For example, we can write a formlet, `positive_int`, based on `checked_int`, that accepts only positive numbers (Figure 3.19). Similarly, building on `positive_int`, we could write a new version of the date formlet that checked that the numbers denoted a valid date, and a new version of the travel formlet that checked for the correct ordering of arrival and departure.

In order to support the new behaviour we compose the formlet implementation from Figure 3.11 with two further idioms. The first is simply the `XmlWriter` idiom presented in Fig-

```
module Failure :
sig
  include IDIOM
  val fail : α t
  val run : α t → α option
end = struct
  type α t = α option
  let pure v = Some v
  let (⊛) f a = match f, a with
    | Some f, Some a → Some (f a)
    | _              → None
  let fail = None
  let run = id
end
```

Figure 3.20: The failure idiom

ure 3.9, which is used to construct the error page to be presented when the submitted input is invalid. The second idiom, `Failure`, captures partial computations, i.e. computations which either return a value or fail. There is a single primitive effect, `fail`, which causes the whole computation to fail. As before, `Failure` may be obtained from a standard monad.

The implementation of the input-validating formlet is shown in Figure 3.21. We base the implementation on the factored `Formlet` of Figure 3.11, but we must expose the idioms from which `Formlet` is constructed in order to write the updated `input` operation.

The idiom operations `pure` and ⊛ are obtained by idiom composition; `xml` and `tag` are obtained by simply lifting the original implementations into the new idiom. The `input` function is a little more involved than the original: as before, we allocate a name to be used by both the rendering and the collector, but now we must insert it into the `<input>` element of the error page as well. The `extract` function passes the value returned by the collector to the supplied extractor function. There are then three possibilities: either extraction succeeds with a value; or it fails with an error message, which is incorporated into the error page; or the collector function fails before the extractor can be run, in which case the error is propagated. The `run` function simply runs the component idioms and puts the result into a slightly more convenient form. No change is needed to the desugaring algorithm.

Input-validating formlets may be constructed and composed in precisely the same way as the standard formlets presented previously. However, the *behaviour* differs significantly. The control flow of a program using standard formlets is simple: the program constructs a page to be sent to the user; the page is submitted; a handler function receives the result and constructs a new page, and so on. The control flow of a program using formlets with input validation

```
module Formlet : FORMLET
  with type α t = α Environment.t XmlWriter.t Namer.t =
struct
  (* as Figure 3.11 *)
end

module VFormlet : VFORMLET =
struct
 module A = XmlWriter
 module O = Failure
 module F = Formlet
 module AO = Compose(A)(O)
 include Compose(F)(AO)
 let xml x = F.pure (const (AO.refine (A.xml x))) ⊛_F F.xml x
 let tag t ats f = F.tag t ats (F.pure (A.tag t ats) ⊛_F f)
 let input =
     N.pure (fun n →
               (A.tag "input" ["name",n]
                (A.pure
                   (E.pure (fun s → A.tag "input" ["name",n]
                                       (A.pure (O.pure s)))
                    ⊛_E E.lookup n))))
      ⊛_N N.next_name)
 let extract extractor f =
  (F.pure
    (fun v → let (x, v') = A.run v in
     match opt_map extractor (O.run v') with
     | Some (Value v) → A.pure (const (O.pure v)) ⊛_A A.xml x
     | Some (Error e) → A.pure (const O.fail)
                          ⊛_A A.xml (x_text e @ x)
     | None          → A.pure (const O.fail) ⊛_A A.xml x))
    ⊛_F formlet
 let run f =
   let (x, c) = F.run f in
     (x, (fun env →
            let v = c env in
            let (error_xml, failed) = A.run v in
             match O.run failed with
               | None → Left error_xml
               | Some a → Right a))
end
```

Figure 3.21: The input-validating formlet implementation

```
  let date_formlet : (_, date) NFormlet.t =
   plug (tag "div" []
             (text "Month: " @ hole @ text "Day: " @ hole))
      (pure (fun month day → make_date month day)
         ⊛ input_int ⊛ input_int)
```

Figure 3.22: Date example, desugared using multi-holed contexts

is more involved: the program constructs a page to be sent to the user; this page is displayed repeatedly (with error messages, as appropriate) until the submitted input is valid; only then is the result passed to a handler, which constructs a new page. In particular, submission of invalid input causes the *whole page* to be redisplayed; it is therefore necessary for the input-validating formlet to capture the page context in which the form appears. To support this behaviour the Links implementation of formlets introduces a `page` construct for associating formlets with page contexts. We give the details in Appendix B.1.

3.6.3 Multi-holed contexts

The presentation of formlets we have given in this chapter relies on lifting the `tag` constructor from the `XmlWriter` idiom into the `Formlet` idiom. As illustrated by the desugaring of the date example in Section 3.4 this makes it difficult to separate the raw XML from the semantic content of formlets and requires nested formlet values to be rebound.

Besides obfuscating the code, this rebinding is inefficient. We can obtain a more efficient formlet implementation that does separate raw XML from semantic content by adapting the formlet datatype to accumulate a list of XML values rather than a single XML value, and replacing `tag` with a general operation, `plug`, for plugging the accumulated list into a multi-holed context. Further, this leads to a much more direct desugaring transformation. For example, the desugared version of the date example is shown in Figure 3.22.

Statically typing `plug` in OCaml requires some ingenuity. Using phantom types, we encode the number of holes in a context, or the number of elements in a list, as the difference between two type-level Peano numbers (Lindley, 2008). As with XHTML typing the key component that needs to change is the `XmlWriter` idiom. This now needs to be parameterised over the number of XML values in the list it accumulates. The construction we need is what we call a *parameterised idiom*, the idiom analogue of a parameterised monad (Atkey, 2009). Recall that the indexed idioms introduced in Section 3.6.1 have an additional type parameter, leading to a family of computation types in place of the single type of computations that is available in a standard idiom. Parameterised idioms allow even more flexibility, using *two* type parameters

```
  module type PIDIOM = sig
    type (μ,ν,α) t
    val pure :  α → (μ,μ,α) t
    val (⊛) : (μ,ν,α → β) t → (σ,μ,α) t → (σ,ν,β) t
  end
```

Figure 3.23: The parameterised idiom interface

for the index; this allows the type of the index to change as a computation progresses.

The OCaml definition of a parameterised idiom as shown in Figure 3.23. Besides the type parameter for the result type there are two additional type parameters μ and ν. For the parameterised XML writer idiom these encode the length of the list of XML values as $\nu - \mu$.

Like idioms, and indexed idioms, parameterised idioms satisfy the four laws given in Section 3.3. They can be pre- and post-composed with other idioms to form new parameterised idioms. Pre-composing the name generation idiom with the parameterised XML writer idiom pre-composed with the environment idiom gives a parameterised formlet idiom.

We have implemented a prototype of formlets with a multi-holed plugging operation in OCaml based on parameterised idioms (Appendix B.2).

Statically-typed multi-holed contexts can be combined with statically-typed XHTML (Lindley, 2008). Lifting the result to idioms gives either an *indexed parameterised idiom*—that is, an idiom with an extra type parameter for the XML type and two extra type parameters for the number of XML values in the accumulated list—or, by attaching the XML type to both of the other type parameters, a parameterised idiom.

3.6.4   Other extensions

These are by no means the only useful extensions to the basic formlet abstraction. For example, we might wish to translate validation code to JavaScript to run on the client (Hanus, 2007), or enforce separation between those portions of the program that deal with presentation and those that treat application-specific computation, a common requirement in large web projects. Either of these may be combined with the formlet abstraction without injury to the core design presented here.

3.7   Implementations

Besides the OCaml implementation of formlets used in this chapter there are currently implementations of formlets for three other languages.

Formlets were initially implemented in Links (Cooper et al., 2008b), which provides the syntax presented here. The Links implementation includes many features, such as a full suite of HTML controls (textareas, pop-up menus, radio buttons, etc.), which are not described here. Strugnell has ported a commercial web-based project-management application originally implemented in PHP to the Links version of formlets (Strugnell, 2008). He gives an in-depth comparison between Links formlets and forms implemented in PHP.

Eidhof has released a Haskell implementation of formlets (Eidhof, 2008), based on the description in Cooper et al. (2008b). The current revision contains a number of useful extensions. For example, validation functions include support for monadic actions, making it possible to perform IO (such as reading from a database) while checking the validity of user input.

McCarthy has added formlets to PLT Scheme based on the description in Cooper et al. (2008a); they are available from version 4.1.1 onwards. Support for a variant of the formlet syntax is provided using macros.

3.8 Related work

The WASH, iData and WUI frameworks all support aspects of the form abstraction we have presented. Underlying all these systems is the essential mode of form abstraction we describe, although they vary significantly in their feature sets and limitations.

WASH The WASH/CGI Haskell framework (Thiemann, 2005) supports a variety of web application needs, including forms with some abstraction. WASH supports user-defined types as the result of an individual form field, through defining a `Read` instance, which parses the type from a string. It also supports aggregating data from multiple fields using a suite of tupling constructors, but it does not allow arbitrary calculations from these multiple fields into other data types, such as our abstract `date` type. In particular, the tupling constructors still expose the structure of the form fields, preventing true abstraction. For example, given a one-field component, a programmer cannot modify it to consist of two fields without also changing all the uses of the component.

iData The iData framework (Plasmeijer and Achten, 2006) supports a high degree of form abstraction, calling its abstractions *iData*. Underlying iData is an abstraction much like formlets. Unlike formlets, where form abstraction is separated from control flow (the function `handle` attaches a handler to a formlet), iData integrate control flow and form composition into a single abstraction. An iData program defines a single web page consisting of a collection of interdependent iData. Whenever a form element is edited by the user, the form is submitted and

then re-displayed to the user with any dependencies resolved. The iTasks library (Plasmeijer, Achten, and Koopman, 2007) builds on top of iData by enabling or disabling iData according to the state of the program.

WUI   The WUI (*Web User Interface*) library (Hanus, 2006, 2007) implements form abstractions for the functional logic programming language Curry. The basic units are called WUIs, defined by the type `WuiSpec`. Like iData, WUIs are designed around the idea of editing; the WUI library provides particular support for presenting application data to the user for modification. Consequently, each WUI of type `WuiSpec` α both consumes and generates a value of type α; the consumed value models the default or current value for the component. WUIs are similar in spirit to formlets: there is a library of basic WUIs for primitive types and a set of combinators for constructing WUIs for user-defined types, such as `transformWSpec`, which is an analogue of the `mapFormlet` function definable in the formlet idiom:

```
transformWSpec :: (α → β, β → α) → WuiSpec α → WuiSpec β
```

```
val mapFormlet : (α → β) → α formlet → β formlet
let mapFormlet g f = formlet <#>{f ⇒ x}</#> yields g x
```

WUIs also support user-input validation through an interface that is similar to our extension of formlets in Section 3.6.2 .

WUIs cannot be characterised as idioms, due to the negative occurrence of the type parameter in the definition of `WuiSpec`; for the same reason, it appears that WUIs cannot strictly implement formlets. However, formlets can implement WUIs: a WUI is equivalent to a value of type $\alpha \rightarrow \alpha$ `formlet`.

The implementation of WUIs is rather different from the approach presented here. Whereas formlets are based on functional abstraction (and so can be adapted to any functional programming language), the WUI library takes advantage of the distinctive features of Curry; for example, it uses logic variables to generate the equivalent of fresh names for input fields, avoiding the need to use a monad or idiom to generate fresh names, as with formlets.

3.9   Conclusions

We have described *formlets*, a construct for composable HTML form fragments that capture the essence of form abstraction. We have argued that formlets are best modelled as idioms (rather than monads or arrows), and shown how to extend the core abstraction in several orthogonal directions, such as static validation of generated XHTML and dynamic validation of user-input, while remaining within the idiom framework. Formlets can be implemented within any

functional language, and there are currently formlet libraries for Haskell, OCaml, Links and PLT Scheme. The latter three implementations also support the syntactic sugar described here, which makes formlets more pleasant to use in practice.

Chapter 4

Serialising continuations[1]

4.1 Introduction

The `Formlet` library described in Chapter 3 depends on a small number of primitive functions. One of these primitives, `pickle_cont`, serialises a function into an externalisable representation which can be sent to a client. Links functions are represented as values of OCaml, the host language for the Links prototype implementation; serialising a Links function involves converting between its OCaml representation and a string encoding the same structure. This chapter describes the approach to serialisation used in the Links implementation.

4.1.1 Requirements

We apply several criteria in the search for a suitable approach to serialisation for Links.

1. *Efficiency*. Space efficiency is of primary importance to keep network traffic low and avoid imposing too great a burden on clients.

2. *Type-safety*. Applying a deserialisation function to invalid data should never cause a program crash.

3. *Extensibility*. Most of the time an out-of-the-box serialiser is sufficient, but there is sometimes a need for a user to supply a specialised implementation of serialisation at a particular type.

4. *Maintainability*. A solution which requires the user to write and maintain large amounts of code to perform basic serialisation is not acceptable.

---

[1]This chapter is a revision of Yallop (2007)

We must emphasise that these are the requirements for the Links prototype, not for serialisation in general. Other systems may well require further properties, such as adherence to an externally-specified format (such as ASN.1), forwards and backward compatibility, and so on. We also choose not to address security considerations (notably maintaining the secrecy and integrity of serialised data) in what follows, taking the view that these may be addressed externally, for example by encrypting serialised functions before passing them to the client (Baltopoulos and Gordon, 2009).

4.1.2 Existing approaches

We will first consider two existing approaches to serialisation: the standard OCaml module `Marshal`, and *pickler combinators*.

4.1.2.1 Marshal

The OCaml standard library includes a module, `Marshal`, with functions

```
val to_string   : ∀α.α → extern_flags list → string
val from_string : ∀α.string → int → α
```

that allow almost any value to be serialised and reconstructed. While `Marshal` is certainly easy to use, its design is problematic when judged for flexibility and safety. The encoding of values is unspecified and fixed, leaving no way to specialise the encoding at a particular type. The type of the `to_string` function places no restrictions on its input, delaying detection of attempts to serialise unserialisable values until runtime.

```
# Marshal.to_string (lazy 0) [];;
Exception: Invalid_argument
        "output_value: abstract value (outside heap)".
```

Most seriously of all, it is easy to use `Marshal` to write programs that crash by interpreting a reconstructed value at a wrong type.

```
# (Marshal.from_string (Marshal.to_string 0 []) 0 : float);;
Segmentation fault
```

These flaws make `Marshal` most suitable for use as a basis for a safe implementation that includes some independent means of verifying the integrity and suitability of marshalled data.

4.1.2.2 Pickler combinators

Compositionality is perhaps the greatest benefit of functional programming (Hughes, 1989), so it is natural to seek a combinator approach to the serialisation problem. There is a natural way to

structure a combinator library for serialisation, as evinced by the similarity of proposed designs by Kennedy (2004) for Haskell and Elsman (2005) for SML. Starting with a parameterised type of serialisers

```
type α pu
```

together with serialisers for primitive types

```
val int : int pu
val unit : unit pu
val bool : bool pu
```

and combinators for parameterised types that take serialisers to serialisers

```
val list : ∀α.α pu → α list pu
val pair : ∀α, β.(α pu) × (β pu) → (α × β) pu
```

and a function which takes conversions between types to conversions between serialisers

```
val wrap : ∀α, β.(α → β) → (β → α) → (α pu → β pu)
```

we can encode a wide variety of datatypes. To serialise user-defined algebraic types we need some way of discriminating between the constructors. The typical solution is to provide a combinator (called `data` in Elsman (2005), `alt` in Kennedy (2004)) whose argument maps constructors to integers:

```
val alt : ∀α.(α → int) → α pu list → α pu
```

We might then write a `pu` for OCaml's `option` type as follows:

```
let option : ∀α.α pu → α option pu =
  fun a →
    alt
      (function None → 0 | (Some _) → 1)
      [wrap (fun () → None)   (fun None → ()) unit;
       wrap (fun v  → Some v) (fun (Some v) → v) a];
```

The combinator approach has none of the problems seen with `Marshal`: the user can choose the encoding at each type, and there is no lack of type-safety. There are, however, serious drawbacks particular to this approach. First, it requires recapitulating the structure of each user-defined type to obtain a serialiser: this is a prime example of the "boilerplate" which much recent research has sought ways to "scrap" (Lämmel and Peyton Jones, 2003). Secondly, the requirement for the user to supply a mapping from constructors to integers can lead to errors that are hard to track down. Finally, there are difficulties in handling both cyclic values, which can arise in ML through the use of references, and mutually recursive datatypes. The `refCyc` combinator described by Elsman (2005) supports cyclic values, but requires the user to supply a "dummy" value to start off the cycle:

```
    val refCyc : ∀α.α → α pu → α ref pu
```

Not only is this a rather unpleasant imposition, but it is not apparent whether it can be readily generalised to the OCaml situation, where mutability is introduced not by the `ref` type, but by records with arbitrary numbers of mutable fields. Karvonen (2007) notes that the need for a dummy value can be eliminated by using an additional generic function that constructs witnesses for types.

4.2 Generic functions

The operations provided by pickler combinators are examples of *generic functions*. Generic functions are based on a form of polymorphism which lies between the extremes of *parametric* and *ad-hoc*. The behaviour of a parametrically polymorphic function, such as `length`, is the same at every type. The behaviour of an ad-hoc polymorphic function, such as `sum`, is different at each type at which it is defined. The behaviour of a generic function, such as `pickle` or `eq`, also varies at each type, but in a way that is related to the type structure.

Put another way, parametric polymorphic functions are parameterised by *type*, ad-hoc polymorphic functions by *instance*, and generic functions by the *shape of types*. For example, `eq` is defined in the same way for every (immutable) record type: two values of a record type are equal if the values of corresponding fields are equal. The definition of `eq` varies with the shape of its argument type: in the case of records it is a function of the set of fields in the type.

The definition of a generic function at a particular type typically follows the definition of the type exactly. There is therefore no real reason to require a programmer to write out the definitions of such functions for each new type he defines. The Haskell definition specifies a means by which a programmer can request the implementation to derive suitable definitions automatically (for a fixed set of standard type classes) (Peyton Jones and Hughes, 1999). In the following pages we describe an analogous facility for OCaml, which relieves the programmer of the need to write out serialisation functions when defining new types. Like Haskell's facility, our system can be applied to any suitable type. However, the Links prototype only needs to apply it at a fixed set of types, namely those types which are used to represent Links syntax and Links values within the OCaml code that forms the Links implementation.

4.2.1 Example: generic equality

We begin with a simple example. Figure 4.1 gives the definitions of two datatypes which support equality. The code is all standard OCaml, except for the clause

```
    deriving (Eq)
```

```
type point = { mutable x : int; mutable y : int }
    deriving (Eq)

type α seq = Nil | Cons of α × (α seq)
  deriving (Eq)
```

Figure 4.1: Deriving equality functions for types

```
let v = Eq.eq<point seq>
          (Cons ({x = 10; y = 20 }, Nil))
          (Cons ({x = 10; y = 20 }, Nil))
```

Figure 4.2: Using generic equality

at the end of each definition, which requests the generation of equality functions at the types
`point` and `seq`.

Figure 4.2 shows how we might use the generated functions. The notation

Eq.eq<t>

calls the generic equality function at the type `t`. Here `t` must be either a type for which equality
has been derived, such as `point`, or some combination of derived types and builtin types such
as `point seq` or `point × int`.

Like the new keyword, the program which generates definitions from types is called *deriv-
ing*. The syntax extensions supported by *deriving* are implemented using the Camlp4 frame-
work which we used to implement the formlets syntax of Section 3.4; the output is standard
OCaml code. For Figure 4.1, the output is shown in Figure 4.3[2]. Each identifier in the
deriving clause in the original program generates a module which matches a particular sig-
nature of the same name; in this example the signature is EQ (Figure 4.5), which contains a type
component `a` and a value component `eq` specifying a function of type `a → a → bool`. The
structure of the types involved is, naturally, reflected in the structure of the modules generated:
the `seq` type is recursive, so a recursive module, `Eq_seq`, is generated; it is parameterised, so
the output includes a parameterised module (or *functor*), `Eqs`. We will describe the scheme in
detail in the sections that follow.

Figure 4.4 shows the output of *deriving* for Figure 4.2. Figure 4.2 involved the use of
the equality function at type `point seq`, so the generated code involves the application of the
module generated for `seq` to the module generated for `point`.

---

[2]Here and elsewhere we have taken the liberty of increasing legibility a little by shortening generated names and
removing trivial bindings. The code is operationally identical to the actual code generated by *deriving*.

```
type point = { mutable x : int; mutable y : int }

module rec Eq_point : EQ with type a = point =
struct
  type a = point
  let eq = (==)
end

type α seq = Nil | Cons of α × (α seq)

module Eqs (Eq_α : EQ) =
struct
  module rec Eq_seq : EQ with type a = Eq_α.a seq =
  struct
    type a = Eq_α.a seq
    let eq l r = match l, r with
    | Nil, Nil → true
    | Cons (x1, x2), Cons (y1, y2) →
        Eq_α.eq x1 y1 && Eq_seq.eq x2 y2 && true
    | _  → false
  end
end

module Eq_seq (Eq_α : EQ) =
struct
    module P = Eqs(Eq_α)
    include P.Eq_seq
end
```

Figure 4.3: Output of *deriving* for Figure 4.1

```
let v = let module Eq
              : EQ with type a = point seq
              = Eq_seq(Eq_point)
        in Eq.eq
           (Cons ({x = 10; y = 20}, Nil))
           (Cons ({x = 10; y = 20}, Nil))
```

Figure 4.4: Output of *deriving* for Figure 4.2

Finally, *deriving* extends the syntax of signatures. Figure 4.6 shows how to export derived
functions in module signatures. Adding the phrase **deriving** (Eq) to a type in the signa-
ture specifies that the generated equality function for that type should be exported along with

```
module type EQ =
sig
  type a
  val eq : a → a → bool
end
```

Figure 4.5: Signature of the Eq class

```
module type Points = sig
    type point deriving (Eq)
    type α seq deriving (Eq)
end
```

Figure 4.6: Deriving signatures for equality

```
module type Points = sig
    type point
    module Eq_point : EQ with type a = point
    type α seq
    module Eq_seq (Eq_α : EQ) : EQ with type a = Eq_α.a seq
end
```

Figure 4.7: Output of *deriving* for Figure 4.6

the type. The derived function may then be invoked in the usual way.

Figure 4.7 shows the output of *deriving* for Figure 4.6. Each deriving clause adds a module binding to the signature, giving the name and type of the corresponding generated module in the implementation.

Generic equality may appear a relatively useless addition to OCaml, since OCaml already boasts two polymorphic equality predicates, = and ==, which test for structural and physical equality respectively. However, there are situations in which neither of the built-in functions is suitable. The structure-sharing serialiser described in Section 4.3 requires a predicate that equates structurally-equal immutable values (unlike ==) and distinguishes physically equal mutable values (unlike =). Without the first it will share too little; without the second it would share too much, since conflating distinct mutable values changes the semantics of programs. (In OCaml sharing of immutable values is detectable as well, using ==, but the precise behaviour is not specified; we are therefore unconcerned about changing the semantics of programs that depend on its use.) In short, we need a predicate corresponding to the equality operator of Standard ML, which behaves like OCaml's = on immutable values, and like OCaml's == on

(a) Classes

```
class Eq a                        module type EQ = sig
  where                             type a
    eq :: a → a → Bool              val eq : a → a → bool
                                  end
```

(b) Superclasses

```
class (Show a, Eq a) ⇒ Num a   module type NUM = sig
  where                          type a
    (+) :: a → a → a             val (+) : a → a → a
    (-) :: a → a → a             val (-) : a → a → a
                                 module Show : SHOW
                                   with type a = a
                                 module Eq : EQ
                                   with type a = a
                                 end
```

(c) Instances

```
instance Eq Int                  module EqInt : EQ
  where                            with type a = int =
    eq = eqInt                   struct
                                   type a = int
                                   let eq = eqInt
                                 end
```

(d) Instances for parameterised types

```
instance Eq a ⇒ Eq [a]           module EqList (A : EQ) : EQ
  where                           with type a = A.a list =
    eq l r =                     struct
       and (zipWith eq l r)       type a = A.a list
                                  let eq l r =
                                     and (zipWith A.eq l r)
                                 end
```

Figure 4.8: Correspondence between type classes and modules

mutable values.

4.2.2 Modules and type classes

As we have said, the design of *deriving* is inspired by the construct of the same name in Haskell. Haskell's *deriving* is tied to type classes, which have no direct parallel in OCaml. However,

there is a well-known correspondence between ML's modules and Haskell's type classes (Kiselyov, 2004, Dreyer, Harper, Chakravarti, and Keller, 2007, Wehr and Chakravarty, 2008), which we use to guide our design. The relevant facets of the correspondence are illustrated in Figure 4.8:

(a) A class in Haskell maps to a signature in OCaml with a type component to specify the overloaded type and a set of value components to specify the methods.

(b) A superclass maps to a signature which contains the superclass module type as a component.

(c) An instance of the class maps to a structure implementing the signature, with sharing constraints to expose the representation of the overloaded type.

(d) An instance for a parameterised type maps to a functor which takes for each type parameter a structure satisfying the class signature and yields another such structure.

We will use the language of type classes to describe our design in this chapter, referring to the signatures used by *deriving* as "classes", to generated structures as "instances", and so on.

4.2.3 Types in OCaml

Figure 4.9 shows a subset of the OCaml grammar for type definitions that is accepted by *deriving*. Some of the constructs not listed here are accepted by *deriving* in certain circumstances: for instance, private types (which expose constructors that can appear in patterns, but not expressions) are permitted for classes such as Eq which do not construct values. For simplicity we assume that all polymorphic variant tags are unary; the full implementation supports nullary tags as well.

Figure 4.10 shows a simplified form for types. This is the form used internally by *deriving*. The principal feature of the simplified form is that every non-trivial type expression is named at top-level; only "atomic" type expressions can occur as subexpressions. The **as** binder, which can be used to name subexpressions in the full type language, is not permitted. For reasons that will appear shortly we also distinguish type names bound locally (i.e. in the same declaration group as the type expression in which they occur) from type names bound elsewhere. Only local type names, rather than arbitrary type constructors, are permitted in nested positions; for example, in the following standard OCaml definitions the body of sum3 is not valid in the normalised type language.

```
type (α, β) sum2 = [ `One of α | `Two of β ]
type (α, β, γ) sum3 = [ Three of γ | (α, β) sum2 ]
```

Type declarations

$\quad d$ ::= **type** $(\alpha_1, \ldots, \alpha_n)$ c = r ... **and** $(\alpha_1, \ldots, \alpha_n)$ c = r

Type representations

$\quad r$ ::= { $\langle$**mutable**$\rangle$ l : e; ...; $\langle$**mutable**$\rangle$ l : e }   *record*
$\qquad\quad C$ $\langle$**of** $e, \ldots, e\rangle$ | ... | C $\langle$**of** $e, \ldots, e\rangle$ *variant*
$\qquad\quad e$ *type expression*

Type expressions

$\quad e$ ::= α *type variable*
$\qquad\quad e \times \ldots \times e$ *tuple*
$\qquad\quad [$ `$T$ **of** $e$ | ... | `T **of** e | e | ... | e]   *polymorphic variant*
$\qquad\quad (e,\ \ldots,\ e)$ c *application*
$\qquad\quad e$ **as** α *named expression*

Type names c Type variables α

Data constructors C Polymorphic variant tags   `T Record labels l

Figure 4.9: OCaml type language
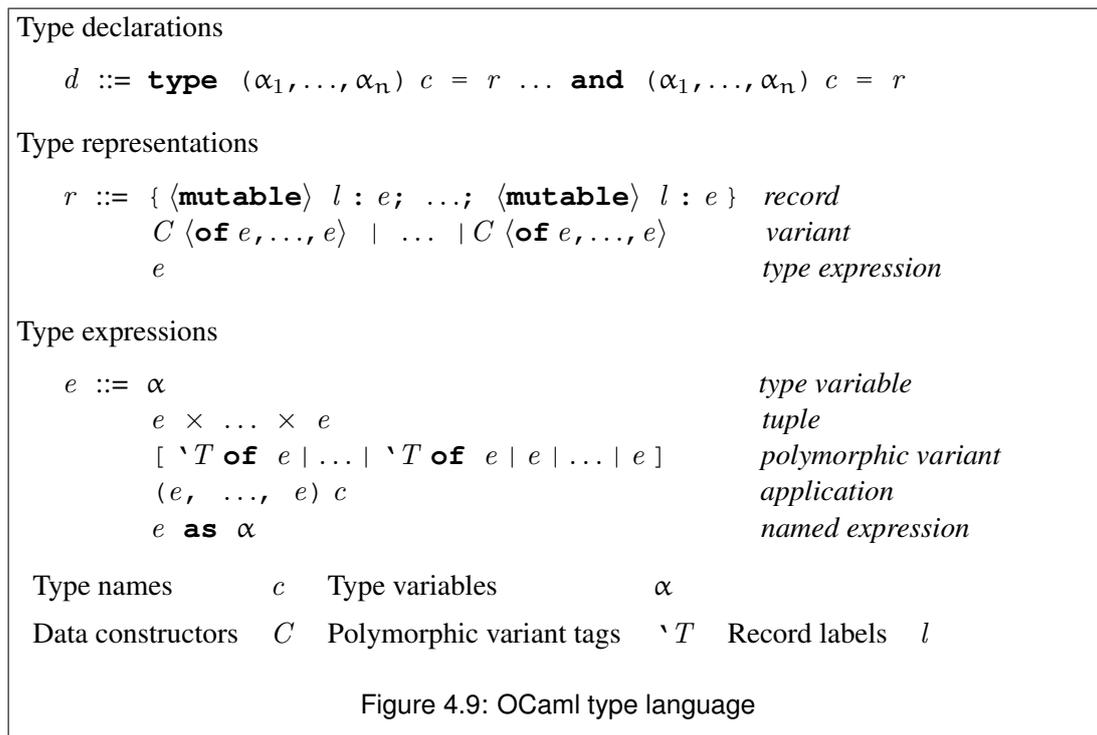
To normalise the definition we must bind the application of `sum2` to a top-level name within the same group.

```
type (α, β) sum2 = [ `One of α | `Two of β ]
type (α, β, γ) s = (α, β) sum2
 and (α, β, γ) sum3 = [ Three of γ | s ]
```

Note also that, in contrast to the type language of Figure 4.9, the normalised type language does not allow type constructors to be applied to type arguments within their own definitions. For example, in the code above, `s` is not applied to type arguments in the definition of `sum3`. To obtain standard OCaml code from this definition we must replace this occurrence of `s` by `(α, β, γ) s`. The intention is to disallow non-regular recursion. We explain the reasons for this restriction in the following section.

4.2.4   Recursive functors

Mutually-recursive types often present an obstacle to generic programming schemes (Rodriguez, Löh, and Jeuring, 2009). In our case it is the use of modules to encode generic functions that introduces a difficulty. Under the correspondence described in Section 4.2.2, the natural encoding of a parameterised mutually-recursive type

Type declarations

$$d \; ::= \; \textbf{type} \; (\alpha_1, \ldots, \alpha_n) \; t \; = \; r \; \ldots \; \textbf{and} \; (\alpha_1, \ldots, \alpha_n) \; t \; = \; r$$

Type representations

$$
\begin{array}{llll}
r \; ::= & \{ \; \langle \textbf{mutable} \rangle \; l : a; \; \ldots; \; \langle \textbf{mutable} \rangle \; l : a \} & \textit{record} \\
& a \; \times \; \ldots \; \times \; a & \textit{tuple} \\
& C \; \langle \textbf{of} \; a, \ldots, a \rangle \; | \; \ldots \; | \; C \; \langle \textbf{of} \; a, \ldots, a \rangle & \textit{variant} \\
& [\; \text{`}T \; \textbf{of} \; a \; | \ldots | \; \text{`}T \; \textbf{of} \; a \; | \; t \; | \ldots | \; t \;] & \textit{polymorphic variant} \\
& (a, \; \ldots, \; a) \; c & \textit{application} \\
& a & \textit{atomic type}
\end{array}
$$

Atomic type expressions

$$
\begin{array}{lll}
a \; ::= & \alpha & \textit{type variable} \\
& t & \textit{locally bound name}
\end{array}
$$

| Local type names | t | Non-local type names | c | Type variables | α |
| Data constructors | C | Polymorphic variant tags | $\text{`}T$ | Record labels | l |

Figure 4.10: OCaml type language, normalised

```
type α tree =
    EmptyT
  | Branch of α × (α forest)
and α forest =
    EmptyF
  | Trees of (α tree) × (α forest)
```

is a group of mutually-recursive functors

```
module rec Eq_tree(Eq_α : EQ)
  : EQ with type a = Eq_α.a tree =
struct
  ...
end
and Eq_forest(Eq_α : EQ)
  : EQ with type a = Eq_α.a forest =
struct
  ...
end
```

However, OCaml does not permit recursive functors, so an alternative encoding is needed. While recursive functors are prohibited, there is no problem with functors whose bodies contain recursive modules as components; we can therefore encode the recursive group as a single functor whose body is a group of recursive modules, then project out the modules separately.

```
module Eqs(Eq_α : EQ) =
struct
  module rec Eq_tree : EQ with type a = Eq_α.a tree =
  struct
    ...
  end
  and Eq_forest : EQ with type a = Eq_α.a forest =
  struct
    ...
  end
end
module Eq_tree(Eq_α : EQ) = struct
  module P = Eqs(Eq_α)
  include P.Eq_tree
end
module Eq_forest(Eq_α : EQ) = struct
  module P = Eqs(Eq_α)
  include P.Eq_forest
end
```

This solution requires that all types in the group have the same parameters and that every occurrence of the type constructors on the right hand side of the definition is applied to exactly those parameters in the same order. We therefore adopt this restriction for types to be processed by *deriving*. (Elsman (1998), investigating an encoding of generic equality in Standard ML, is obliged to adopt a similar restriction due to the lack of support for polymorphic recursion.)

Using this encoding of recursive functors, applications of type names bound elsewhere generate applications of functors to module arguments, while applications of type names bound locally generate unapplied module names: references to α forest generate references to the module Eq_forest, for example, not to Eq_forest Eq_α. It is for this reason that we distinguish local from non-local names in the internal type representation used by *deriving* (Figure 4.10).

4.2.5   Implementation of generic equality

Figures 4.11 and 4.12 give an implementation of Eq. The $\ll \ldots \gg$ notation is a quotation for code; antiquotations are written between dollar signs $ \ldots $ or, for antiquoted names, as italicised variables such as t. To improve readability, we use an ellipsis notation in the figures to indicate sequences of syntactic elements, such as identifiers; these represent standard list-processing techniques, such as folds, in the actual OCaml implementation.

The notation Eq.eq<e>, for invoking the generic equality function at type e, is defined in terms of the **deriving** notation: first, the type expression e is bound to a type constructor, t;

an instance of Eq for t, Eq_t, is derived ; the result is bound to a local module identifier, Eq, from which the eq field is projected.

The **deriving** notation applies the meta-function def to the preceding type definition. A type definition with parameters $\alpha_1 \ldots \alpha_n$ generates a functor, Eqs, which abstracts n instances of EQ named Eq_α_1 ... Eq_α_n. For the m type constructors in the definition the body of the functor contains a set of recursive modules, Eq_r_1 ...Eq_r_m, each of which is also an instance of EQ. Both the functor arguments and the recursive module names are in scope throughout the body of the functor; it is therefore possible to refer to any atomic type definition by name (either Eq_α_i or Eq_r_i) at any point in the generated code. The body of each recursive module is generated by the meta-function rep, which we describe next. For convenience each type constructor t_i also generates a top-level functor Eq_r_i which simply applies Eqs to its arguments and projects the corresponding component.

The rep meta-function constructs a structure matching the signature EQ from a type representation r, ensuring that the type component a matches r. An application of a type constructor generates structures matching EQ for each of the arguments and a functor that is applied to those arguments for the type constructor in function position. A type variable generates a reference to the corresponding functor argument (of Eqs); a locally-bound name generates a reference to the corresponding member of the recursive module group generated by def. Other types generate a fresh structure whose eq member is generated by the meta-function fun.

The fun meta-function constructs an equality function of type t $\rightarrow$ t $\rightarrow$ bool from a type expression t. For mutable record types the function is simply (==), the physical equality predicate. Two values of product type (immutable records or tuples) are considered equal if corresponding components in each value are equal. Two values of variant or polymorphic variant type are considered equal if they have matching constructors or tags and matching arguments. The notation #t in the generated equality function for polymorphic variants matches a value of type t, which must be a polymorphic variant type.

We note in passing that the normal form used for types (Figure 4.10), which names every subexpression, results in code that does not construct "dictionaries" (i.e. evaluate module-level terms) during traversal of a value. This contrasts with the earlier version of *deriving* (Yallop, 2007), whose output constructed a fresh dictionary for each subvalue encountered during traversal.

4.2.6 Specializing generic equality

One of our complaints about OCaml's Marshal was the lack of a facility for *specialisation*: providing behaviour at a particular type that differs from the default. This is accomplished us-

Translation of <> notation

```
Eq.eq<e> = let module Eq =
             struct
               type t = e deriving (Eq)
               include Eq_t
             end in Eq.eq
```

Module bindings from type definitions

def $\ll$ **type** $(\alpha_1, \ldots \alpha_n)$ $t_1 = r_1$ ... **and** $(\alpha_1, \ldots \alpha_n)$ $t_m = r_m$ $\gg$ =

    $\ll$ **module** Eqs (Eq_α_1 : EQ) ... (Eq_α_n : EQ) = **struct**

      module rec Eq_r$_1$: EQ **with type** a = r_1 [Eq_α_i.a/α_i]

        = \$$rep$ r_1\$

      ...

      and Eq_r$_m$: EQ **with type** a = r_m [Eq_α_i.a/α_i]

        = \$$rep$ r_m\$

      end

      module Eq_r$_1$ (Eq_α_1 : EQ)...(Eq_α_n : EQ) = **struct**

        module P = Eqs(Eq_α_1)...(Eq_α_n)

        include P.Eq_r$_1$

      end

      module Eq_r$_m$ (Eq_α_1 : EQ)...(Eq_α_n : EQ) = **struct**

        module P = Eqs(Eq_α_1)...(Eq_α_n)

        include P.Eq_r$_m$

      end

Module implementations from type representations

Constructor applications

   rep $\ll (a_1, \ldots, a_n)$ $c \gg$ = $\ll$ Eq_c (Eq_a_1) ... (Eq_a_n) $\gg$

Type variables

   rep $\ll \alpha \gg$ = $\ll$ Eq_$\alpha \gg$

Locally bound names

   rep $\ll$ t $\gg$ = $\ll$ Eq_t $\gg$

Other types

   rep r = $\ll$ **struct**

         type a = r [Eq_α_i.a/α_i]

         let eq = \$$fun$ r\$

       end $\gg$

Figure 4.11: Generation of the `eq` function (1)

Equality functions from type representations

Records with mutable fields

fun $\ll$ { ...; **mutable** $l_j : t$; ... } $\gg$ =
    $\ll$ (==) $\gg$

Records without mutable fields

fun $\ll$ { $l_1 : a_1$; ... ; $l_n : a_n$ } $\gg$ =
    $\ll$ **fun** { l_1 = x$_1$; ...; l_n = x$_n$ } { l_1 = y$_1$; ...; l_n = y$_n$ } $\rightarrow$
        Eq_a_1 x$_1$ y$_1$ && ... && Eq_a_n x$_n$ y$_n$ && **true** $\gg$

Tuples

fun $\ll$ $a_1 \times ... \times a_n$ $\gg$ =
    $\ll$ **fun** (x$_1$, ... x$_n$) (y$_1$, ... y$_n$) $\rightarrow$
        Eq_a_1 x$_1$ y$_1$ && ... && Eq_a_n x$_n$ y$_n$ && **true** $\gg$

Variants

fun $\ll$ C^1 $\langle$**of** a_1^1, ..., $a_m^1\rangle$ | ... | C^n $\langle$**of** a_1^n, ... , $a_m^n\rangle$ $\gg$ =
    $\ll$ **fun** l r $\rightarrow$ **match** l, r **with**
        | C^1 (x$_1$, ..., x$_m$), C^1 (y$_1$, ..., y$_m$) $\rightarrow$
            Eq_a_1^1 x$_1$ y$_1$ && ... && Eq_a_m^1 x$_m$ y$_m$ && **true**
        ...
        | C^n (x$_1$, ..., x$_m$), C^n (y$_1$, ..., y$_m$) $\rightarrow$
            Eq_a_1^n x$_1$ y$_1$ && ... && Eq_a_m^n x$_m$ y$_m$ && **true**
        | _ $\rightarrow$ **false** $\gg$

Polymorphic variants

fun $\ll$ [\`$T_1$ **of** $a_1$ |...| \`T_n **of** a_n | t_1 | ... | t_m] $\gg$ =
    $\ll$ **fun** l r $\rightarrow$ **match** l, r **with**
        | \`$T_1$ x, \`T_1 y $\rightarrow$ Eq_a_1 x y
        ...
        | \`$T_n$ x, \`T_n y $\rightarrow$ Eq_a_n x y
        | (#t_1 **as** x), (#t_1 **as** y) $\rightarrow$ Eq_t_1 x y
        ...
        | (#t_m **as** x), (#t_m **as** y) $\rightarrow$ Eq.t_m x y
        | _ $\rightarrow$ **false** $\gg$

Figure 4.12: Generation of the eq function (2)

```
module IntSet : sig
  type t deriving (Eq)
  val empty : t
  val add : int → t → t
  val mem : int → t → bool
end =
struct
  type t = int list
  let empty = []
  let add i l = i :: l
  let mem = List.mem
  module Eq_t : EQ with type a = t =
  struct
    type a = t
    let eq l r = Eq.eq<int list> (sort l) (sort r)
  end
end
```

Figure 4.13: Integer sets using integer lists

ing *deriving* by writing a module definition with a particular name and type instead of adding
the **deriving** (...) annotation to the type. For example, consider the implementation of
integer sets given in Figure 4.13, in which unordered lists of integers are used as the repre-
sentation type. Using *deriving* we can define equality on IntSet to consider two sets equal if
the results of sorting their representing lists are equal. The builtin equality operator regards
such sets as unequal, exposing the fact that there are various representations for values that the
abstraction considers equivalent:

```
let onetwo = IntSet.add 1 (IntSet.add 2 (IntSet.empty))
let twoone = IntSet.add 2 (IntSet.add 1 (IntSet.empty))

Eq.eq<IntSet.t> onetwo twoone  ⟹ true
onetwo = twoone ⟹ false
```

(Precisely the same problem arises with the set implementation provided in the OCaml standard
library: sets considered equal under the set equality predicate do not always satisfy =.) Using
deriving's Eq, our abstraction-respecting definition of equality will be used wherever Eq is used
to compare values of IntSet.t; in particular, it will be used where such values occur within a
larger data structure:

```
Eq.eq<IntSet.t list> [onetwo; twoone] [twoone; onetwo]
⟹ true
```

4.2.7 Related work

There is a significant body of literature on generic programming. A common approach (Cheney and Hinze, 2002, Hinze, Löh, and d. S. Oliveira, 2006b, Weirich, 2006) is to reflect types as values. Generic functions can then be written by case analysis on the structure of these values. An alternative view (Jansson and Jeuring, 1997, Gibbons, 2006) treats datatypes as fixed points of regular functors, exposing the recursive structure in a way that allows a variety of traversals to be expressed generically. A third approach (Lämmel and Peyton Jones, 2003), particularly suited to generic transformations, uses a type-safe cast operator to locate nodes of particular types within larger values. Hinze, Jeuring, and Löh (2006a) give a useful comparison of designs based on these and other approaches.

The preprocessor-based approach to deriving instances of generic functions is used in the Haskell tools DrIFT (Winstanley, 1997) and Derive (Mitchell and O'Rear, 2006); several Camlp4 extensions involve the generation of functions from type declarations, including Martin Jambon's *json-static* (Jambon, 2007), Martin Sandin's *Tywith* (Sandin, 2004), Daniel de Rauglaudre's IoXML (de Rauglaudre, 2002), and Markus Mottl's *bin-prot* and *sexplib* (Mottl, 2007, 2008).

The pickler combinators described in Section 4.1.2.2 are an example of *type-indexed values* (Yang, 1998). A type indexed value follows the shape of a type — for example, the pickler combinator value denoted by `list (pair bool bool)` corresponds to the ML type `(bool × bool) list` — and encodes a function over that type. A similar approach can be used to encode a wide variety of generic functions in ML. Fernandez, Fisher, Foster, Greenberg, and Mandelbaum (2008) describe a framework for generic programming based on type-indexed values in the PADS/ML language, using serialisation to and from XML as the main example.

The main problem in using type-indexed values for generic programming in ML is the difficulty of combining generic functions. Ideally, we would like to use each type indexed value to denote not only a pickler at the corresponding type, but also an equality predicate, a printing function, and any other generic function we might think of. We can, of course, use a distinct type-indexed value for each generic function, but this rapidly becomes clumsy. The root of the problem is the lack of higher-kinded polymorphism (i.e. abstraction over non-nullary type constructors) in ML; given such a facility we could separate the type structure encoded by a type-indexed value from the particular generic function encoded. Karvonen (2007) describes a workaround in which generic functions (i.e. type-indexed values) are coded open-recursively and later combined using the module system. Unfortunately, this only postpones the problem: before the values can be used there is a "knot-tying" step at which the recursion is closed, at which point it is no longer possible to incorporate additional generic functions.

For the moment, then, the approaches to generic programming that have been used successfully in Haskell cannot be ported to OCaml, since they all appear to require advanced type system features such as generalised algebraic datatypes or higher-kinded polymorphism that are currently unavailable there. This may change in future if the OCaml team's plans for supporting first-class modules come to fruition. First-class modules (Russo, 2000) allow the encoding of certain types of higher-kinded polymorphism (albeit with a relatively high degree of syntactic overhead), which should make it possible to support fully-combinable generic functions. At that point it will be time to revisit the design of *deriving*.

4.3   Generic serialisation

We now return to the subject of serialisation introduced in Section 4.1.

As we indicated earlier, our aim is to construct a serialiser that is suitable as a basis for implementing the primitive `Formlet` function `pickle_cont`. Before we move on to the design of our serialiser, it may be helpful to note the various "levels" involved in the implementation. At the outermost level, `pickle_cont` is a primitive Links function that is used to implement the Links library for constructing HTML forms. At the next level down, the serialisation functions on which `pickle_cont` is built are written in OCaml; they consequently operate on OCaml values (in the case under consideration, on the first-order representation of Links functions that is used within the Links prototype). Finally, these serialisation functions are generated during the compilation of an OCaml program (in our case, during the compilation of the Links prototype) by the Camlp4 preprocessor. We should perhaps also note that, while implementing serialisation for Links was the motivation for *deriving*, there is nothing Links-specific in any of the generic functions described in this chapter, including the generic serialisers which we describe in this section.

There are two serialisation schemes provided as standard with *deriving*: `Dump`, a simple value-oriented serialiser, and `Pickle`, a more realistic serialiser based on `Dump`, which supports features such as compression, structure-sharing and serialisation of cyclic values. We restrict our attention here to the `Pickle` class.

The essential idea behind `Pickle` is to build a graph representing a value which is written out (using `Dump`) when complete. `Pickle` endeavours to maximise sharing, using a single representation for multiple values, by commencing the serialisation of each value with a comparison to values of the same type which have been serialised already. This strategy results in compact output, but a relatively slower serialiser; however, it is straightforward to obtain a faster serialisation algorithm which only preserves (rather than introduces) sharing from the

```
 module type PICKLE =
 sig
   type a
   module Hash : HASH with type a = a
   module Typeable : TYPEABLE with type a = a
   val pickle : a → pstate → id × pstate
   val unpickle : id → pstate → a × pstate
   val to_buffer : Buffer.t → a → unit
   val to_string : a → string
   val to_channel : out_channel → a → unit
   val from_stream : char Stream.t → a
   val from_string : string → a
   val from_channel : in_channel → a
 end
```

Figure 4.14: Signature of the `Pickle` class

algorithm given here.

Figure 4.14 gives the signature for `Pickle`. The `Pickle` class has two direct superclasses, `Hash`, and `Typeable`, and one further superclass, `Eq`, which is a direct superclass of `Hash`. The facilities offered by the `Hash` and `Typeable` classes — hashing for arbitrary values, and conversion to and from a universal type — will prove useful in the implementation of sharing-maximising serialisation (Section 4.3.1). We describe `Hash` and `Typeable` in detail in Sections 4.3.3 and 4.3.4. .

The `Pickle` class contains eight functions. The first two of these, `pickle` and `unpickle`, provide an interface that is suitable for constructing further instances of `Pickle`. We describe them in detail in Sections 4.3.1–4.3.2. The other six functions, whose names take the form `to_X` and `from_X`, provide a more user-friendly interface comparable to `Marshal`, and have straightforward definitions in terms of `pickle` and `unpickle`.

4.3.1   The pickling algorithm

The pickling algorithm requires three pieces of state, which we represent as a record and thread through the computation.

```
type id = int
type pstate = {
  nextid : id;
  ids    : id DynMap.t;
  graph  : node IdMap.t;
}
```

The `nextid` field is a source of fresh identifiers, `ids` maps values (converted to universal type) to identifiers, and `graph` stores the partially-constructed graph of a value, mapping identifiers to nodes. The state is persistent, not mutable: functions "update" the state by returning a new value.

The work is performed by two functions: `allocate`, which allocates identifiers for values, and `store_node`, which records the association between an identifier and the node in the graph which represents the corresponding value.

Underlying the `allocate` function is the `DynMap` module, which provides a type of finite maps with heterogeneously-typed keys.

```
module DynMap :
sig
 type α t
 val empty : α t
 module Ops (H : HASH) (T : TYPEABLE with type a = H.a) =
 struct
  val add  : H.a → α → α t → α t
  val find : H.a → α t → α option
 end
end
```

The type component α t is the type of finite maps whose value type is α. There are three value components, two of which depend on classes defined elsewhere. The `empty` value has no dependencies; it is bound to an empty map. The `add` and `find` functions depend on the `Hash` and `Typeable` classes. The `find` function first searches for all keys whose hash and type match the key supplied as argument, then compares any such keys found for equality with the argument key; if there is a match, the corresponding value is returned. The `add` function converts the key supplied as argument to universal type and inserts it into the map, using the hash and type representation as an index.

We can now define the `allocate` function at each type `a` which is an instance of `Hash` and `Typeable`.

```
module DynOps = DynMap.Ops(Hash_a)(Typeable_a)

let allocate : a → (id → pstate → pstate) → pstate → id × pstate
= fun v store ({nextid = id; ids = ids} as t) →
   match DynOps.find v ids with
     | Some id → (id, t)
     | None → (id, store id {t with
                                 ids = DynOps.add v id ids;
                                 nextid = id + 1})
```

The `allocate` function begins by searching the map `ids` to determine to determine whether

the value has been previously serialised. If it finds a match it returns the identifier already allocated; if there is no match then a fresh identifier is allocated, the association between identifier and value is recorded using `add` and the function `store` is called to serialise the value. This strategy of allocating identifiers before serialising subvalues is sufficient for serialisation of cyclic values: the next time a reference to this node is encountered there will be an identifier in the map, so serialisation will not loop.

The `store_node` function simply records the association between an identifier and a node.

```
let store_node : node → id → pstate → pstate
 = fun node id state →
     {state with graph = IdMap.add id node state.graph}
```

Nodes of the graph are represented as values of the `node` type. A node is either a sequence of bytes (representing a primitive value such as an integer or a string), or a value of algebraic type, which is represented as an optional index for the constructor and a sequence of identifiers for the subvalues.

```
type node = Bytes of string
          | Appl of int option × id list
   deriving (Dump)
```

Once the graph has been fully constructed we discard the other components of the state record. In order to reduce the size of the serialised data we first change the representation of the graph. As we have seen, during construction the graph takes the form of a finite map from identifiers to `node` values. Serialising this directly takes up an unnecessarily large amount of space, since we must serialise an identifier and a constructor (representing either `Bytes` or `Appl`) alongside every node. We eliminate the need to serialise constructors by changing the representation to use three maps, one for each of the three constructor configurations (`Bytes`, `Appl` with a constructor index, and `Appl` without a constructor index). For example, pickling might construct the following graph:

```
{1 ⇒ Appl (Some 2, [4;3]),
 2 ⇒ Appl (None, [1;4]),
 3 ⇒ Bytes "p",
 4 ⇒ Bytes "k"}
```

This is replaced with three maps, eliminating the need to store constructors:

```
{3 ⇒ "p",
 4 ⇒ "k"}
{1 ⇒ (2, [4;3])}
{2 ⇒ [1;4]}
```

Listing the pairs of keys and values sequentially gives the following:

Pickle at record types

$pickle$ ≪ { l_1 : a_1; ...; l_n : a_n } ≫ =
    ≪ **fun** ({ l_1 = x$_1$; ...; l_n = x$_n$ } **as** v) →
        allocate v
         (**fun** id s →
           let x$_1$, s = Pickle_a_1.pickle x$_1$ s **in**
           ...
           let x$_n$, s = Pickle_a_n.pickle x$_n$ s **in**
            store_node (Appl (None, [x$_1$; ...; x$_n$])) id s) ≫

Pickle at polymorphic variant types

$pickle$ ≪ [`$T_1$ **of** $a_1$ |...| `T_n **of** a_n | t_1 | ... | t_m] ≫ =
    ≪ **fun** v → **match** v **with**
       `T_1 x → tag (Pickle_a_1.pickle x) $taghash\ T_1$$ v
       ...
     | `T_n x → tag (Pickle_a_n.pickle x) $taghash\ T_n$$ v
     | #t_1 **as** x → tag (Pickle_t_1.pickle x) s_1 v
       ...
     | #t_m **as** x → tag (Pickle_t_m.pickle x) s_m v ≫

(Where $s_i \neq taghash\ T_j$ for all i in $1 \ldots n$ and j in $1 \ldots m$).

Figure 4.15: Generation of the `pickle` function

```
(3, "p"),   (4, "k"), (1, (2, [4;3])), (2, [1;4])
```

We then eliminate the need to serialise identifiers alongside each node by replacing the identifiers used during construction of the graph with the offsets of the corresponding nodes. In our example the values contain the identifier references 1, 3 and 4, which refer respectively to the third, first and second nodes in the sequence. Substituting these, we arrive at the following sequence of unlabelled nodes, which we serialise using `Dump`:

```
"p", "k", (2,[2;1]) [3;2]
```

Figure 4.15 gives the implementation of `pickle` for record and polymorphic variant types. Other cases are similar in the essential details and so omitted. Pickling a value of record type involves passing to `allocate` a function that pickles each field value and then stores a node in the graph which points to the nodes for the fields. Pickling a value of polymorphic variant type involves an auxiliary function, `tag`, which is defined in terms of `allocate` and `store_node`.

```
let tag : (pstate → id × pstate) → int → a → pstate → id × pstate
 = fun pickler t v state →
    let id, pstate = pickler state in
       allocate v (store_node (Appl (Some t, [id]))) state
```

The `tag` function takes as argument an function, `pickler`, which stores a value v; it returns a function which stores a tagged value ‘T v. The second parameter, t, denotes the tag T; the third parameter is the complete value to be stored, ‘T v.

A polymorphic variant type can be declared in any of a number of compatible ways. In particular, the order of tags does not affect the denotation of the type: [‘A | ‘B] is compatible with [‘B | ‘A], and both may be used to denote the same type within a single program (Section 4.3.3). In assigning numbers to tags, therefore, we cannot simply use the order in which the tags appear in the declaration. Instead we use a hash of each tag name, ensuring that the mapping from tags to integers is globally consistent. The hash function we use is the same function used by the OCaml compiler to represent polymorphic variant tags (Garrigue, 1998). The type checker rejects types containing tags, such as ‘squigglier and ‘premiums, whose hash numbers collide. Using the same hash function as OCaml therefore ensures that we will not be troubled by collisions. In the case where a polymorphic variant type t extends some other type s we assign a tag number to s that does not collide with the hashes of the other tags of t.

4.3.2 The unpickling algorithm

Detecting cycles is straightforward; reconstructing cycles is trickier. Values in OCaml are constructed from subvalues: the subvalues must exist first, but this breaks down if the value is among its own subvalues (either directly or as a more distant descendent). As we saw in Section 4.1.2.2, Elsman solves this problem by requiring the user to supply an initial value to start the recursion going. Our solution is to use low-level primitives to allocate an uninitialised value whose address can be used in constructing subvalues. In generated code the risk of a mistake in such memory manipulation is low, whilst the advantages — increased abstraction and ease of use — are great.

Like the pickling algorithm, the unpickling algorithm threads a piece of state throughout the computation. The state we need is a finite map whose keys are identifiers and whose values are either `node` representations of values or fully deserialised values, represented using the universal type, `dynamic`.

```
type ustate = ((node, dynamic) either) IdMap.t
```

At the start of unpickling, every value in the map is a `node`. The unpickling process replaces each of these with fully reconstructed values.

The unpickling algorithm is based on two combinators, `variant` and `record`, which perform unpickling for particular types and update the state.

```
let variant : (int × id × ustate → a × ustate) → id → ustate
          → a × ustate =
 fun load id s → match IdMap.find id s with
   | Left (Appl (Some c, [id])) →
      let v, s = load (c, id, s) in
      let s = IdMap.add id (Right (Typeable.mk<a> v)) s in
        (v, s)
   | Right v →
      (Typeable.cast v, s)
   | _ → raise UnpicklingError
```

The `variant` combinator handles unpickling for a variant type `a`. It takes three arguments: a function, `load`, which reconstructs a value from a `node`, an identifier, `id`, and the unpickling state, `s`. The behaviour depends on whether `id` resolves to a `node` or a reconstructed value. If it resolves to a `node`, then the integer representing the tag and the identifiers representing the subvalues are passed to `load`, which constructs and returns the value; this value is upcast to `dynamic` and stored in the map. If, instead, `id` resolves to a value then this value is downcast to the appropriate type and returned. The wildcard branch handles the case where the `node` found in the map has the wrong shape for a variant value.

The `record` function is similar, but the order of operations is adapted to handle recursive values, since recursion arises from mutability, and mutability is introduced in OCaml at record types.

```
let record : int → (a × id list × ustate → ustate) → id → ustate
          → a × ustate =
 fun size load id s → match IdMap.find id s with
   | Left (Appl (None, ids)) →
      let v = Obj.magic (Obj.new_block record_tag size) in
      let s = IdMap.add id (Right (Typeable.mk<a> v)) s in
      let s = load (v, ids, s) in
        (v, s)
   | Right v →
      (Typeable.cast v, s)
   | _ → raise UnpicklingError
```

The `record` combinator takes the same arguments as the variant type, plus an additional argument, `size`, indicating the number of fields in the record. If the state resolves the identifier to a `node` then low-level primitives are used to allocate memory for a record with `size` fields. This uninitialised memory is stored in the map (at type `dynamic`) and the function `load` is called to populate it with the values of the deserialised record fields. Storing the value in the map *before* calling `load` ensures that within `load` the identifier will resolve to this same value, allowing the construction of recursive objects. Of course, it is vital that `load` treat the uninitialised

---

Unpickle at record types

$unpickle \ll \{ \ l_1 \ : \ a_1; \ \ldots; \ l_n \ : \ a_n \ \} \gg =$

```
        ≪ let module Mutable =
              struct type t = { mutable l₁ : a₁;
                                    ... ;
                                    mutable lₙ : aₙ }
              end in
           record n
             (fun (self, ids, s) → match ids with
                | [ x₁; ...; xₙ ] →
                     let mself : Mutable.t = Obj.magic self in
                     let x₁, s = Pickle_a₁.unpickle x₁ s in
                        ...
                     let xₙ, s = Pickle_aₙ.unpickle xₙ s in
                     let () = mself.Mutable.l₁ <- x₁ in
                        ...
                     let () = mself.Mutable.lₙ <- xₙ in
                       s)
                | _  → raise UnpicklingError) ≫
```

Unpickle at polymorphic variant types

$unpickle \ll [\ `T_1 \ \textbf{of} \ a_1 \ |\ldots| \ `T_n \ \textbf{of} \ a_n \ | \ t_1 \ | \ \ldots \ | \ t_m] \gg =$

```
        ≪ variant
            (fun (tag, x, s → match tag with
             $taghash T₁$ →
               let x, s = Pickle_a₁.unpickle x s in `T₁ x, s,
              ...
            | $taghash Tₙ$ →
               let x, s = Pickle_aₙ.unpickle x s in `Tₙ x, s
            | s₁ →
               let x, s = Pickle_t₁.unpickle x s in (x :> a), s
              ...
            | sₘ →
               let x, s = Pickle_tₘ.unpickle x s in (x :> a), s
            | _ →
               raise UnpicklingError) ≫
```

(Where $s_i \neq taghash \ T_j$ for all i in $1 \ldots n$ and j in $1 \ldots m$).

Figure 4.16: Generation of the `unpickle` function

value as "write-only"; this is guaranteed by the unpickling algorithm, which does not inspect deserialised values.

Figure 4.16 gives the implementation of `unpickle` for record and polymorphic variant types. The generated function for a record type `r` includes the declaration of a type structurally similar to `r`, but with every field declared mutable. The function passed to `record` receives an uninitialised value of sufficient size, together with identifiers for the fields, and the state. The function casts the uninitialised value, `self`, to the mutable type, which makes it possible to *first* reconstruct subvalues which may contain references to `self`, *then* assign the subvalues to the fields of `self`.

The `unpickle` function for polymorphic variants is straightforward: first the argument of each tag is reconstructed, then the tag is applied to the reconstructed value. The tag numbers $s_1 \ldots s_m$ used for the types $t_1 \ldots t_m$ must, of course, match those generated for use in the `pickle` function (Figure 4.15).

4.3.2.1 Immutable cycles

The approach we describe here is sufficient to handle all cycles based on mutability; there is no attempt to handle the immutable cycles which arise from OCaml's value recursion extension. This extension permits non-function values on the right-hand-side of **let rec** bindings, so long as they meet certain syntactic restrictions which prevent reading partially constructed values. For example, the declaration

```
let rec x = 1 :: 2 :: x
```

binds `x` to a cyclic list of integers whose values alternate between `1` and `2`. Values of this form cause unpickling to loop.

However, as a side effect of the implementation, unpickling will successfully reconstruct immutable cyclic objects where the cycles pass through records. For example, given the type declaration

```
type α cons = { car : α ; cdr : α seq }
and α seq = Cons of α cons
    deriving (Eq, Hash, Typeable, Pickle)
```

the following declaration for `y` creates a cyclic value that can be pickled and unpickled without problems.

```
let rec y = Cons {car = 1; cdr = Cons {car = 2; cdr = y}}
```

```
  module type TYPEABLE =
  sig
    type a
    val mk : a → dynamic
    val cast : dynamic → a
    val has_type : dynamic → bool
    val type_rep : typerep
  end
```

Figure 4.17: Signature of the `Typeable` class

```
  module TagMap = Map.Make(Interned)
  type t = [`Generative of Interned.t × typerep list
            |`Variant of (typerep option TagMap.t)]
          × int
  and typerep = t lazy_t
```

Figure 4.18: The `typerep` type

4.3.3 Dynamic typing

The `ids` map used in the algorithm in Section 4.3.1 is used to store values of arbitrary type. To achieve this, *deriving* includes support for a form of dynamic typing. The `Typeable` class (Figure 4.17) provides an upcast, `mk`, to a universal type, `dynamic`, and a safe downcast operation, `cast`, from `dynamic` to each instance type, which raises an exception on failure. The `has_type` predicate tests whether a value of universal type can be cast to the instance type.

```
Typeable.cast<int>
    (List.hd [Typeable.mk<int> 3; Typeable.mk<unit> ()])
⟹ Some 3
Typeable.cast<int>
    (List.hd [Typeable.mk<unit> (); Typeable.mk<int> 3])
⟹ CastFailure
```

There are well-known techniques for implementing dynamic typing in pure SML (see p105-106 of Filinski (1996), for example), but these deal only with generative types, not the structural types found in OCaml. Following Lämmel and Peyton Jones (2003), we use instead an implementation based on pairs of values and representations of their types together with an unsafe cast that is performed only if type representations match, which allows us to test types for structural equality where appropriate. (The `type_rep` operation of the `Typeable` class retrieves this type representation for a particular instance.) Our implementation divides types into nominal (or "generative") types — i.e. variants and records — which are represented as

a string for the type constructor and a sequence of argument types, and structural types —
i.e. polymorphic variants— which are represented as lazy infinite trees. Tuples are treated as
nominal, using the arity as the type constructor. The concrete definition of `typerep` is given
in Figure 4.18. We use interned strings, with constant-time equality testing, to represent gener-
ative types and polymorphic variant tags. A final point to note is that type representations are
lazy. Laziness allows us to construct recursive type representations for recursive types; without
laziness our attempts to construct such representations are rejected as unsafe, since they do
not meet the syntactic restrictions mentioned in Section 4.3.2.1. Syme (2006) gives a general
account of using laziness to introduce recursion in this way.

The definition of equality on type representations corresponds closely to the OCaml defini-
tion of equality on types. Nominal types are considered equal if they have the same constructors
and equal arguments. Polymorphic variant types are considered equal if they have the same set
of labels and if the arguments to corresponding labels are equal. Recursive polymorphic variant
types have cyclic representations; termination of the equality test is assured by associating an
identifier with each node — the `int` component of the type representation — which is used to
record which nodes have been visited, i.e. to detect cycles. Since polymorphic variant types
are always regular all recursion eventually passes through an already visited node.

Polymorphic variant types which are considered equivalent in the OCaml type system are
treated as equivalent by *deriving* regardless of how they are declared, so the following cast will
succeed:

```
type α seq = [`Nil | `Cons of α × α seq]
 deriving (Typeable)

type nil = [`Nil]
  deriving (Typeable)
type intlist = [ nil | `Cons of int × α ] as α
  deriving (Typeable)

Typeable.cast<intlist>
  (Typeable.mk<int seq>
    (`Cons (1, `Cons (2, `Cons (3, `Nil)))))
⟹ Some (`Cons (1, `Cons (2, `Cons (3, `Nil))))
```

Similarly, the type representation used by Typeable does not respect module abstraction bound-
aries, so the following cast also succeeds:

```
module M : sig
    type t deriving (Typeable)
    val v : t
    val make : int → t
end =
struct
    type t = int deriving (Typeable)
    let v = 0
    let make = id
end

Typeable.cast<int> (Typeable.mk<t> M.v)
⟹ Some 0
```

This is a deliberate design decision: `Typeable`'s raison-d'être is to maximise sharing, which is best achieved by identifying types which can easily be determined to have the same representation. For example, the following code only serialises a single copy of the integer s, since `Typeable` (and hence `Pickle`) treats the types `int` and `M.t` as the same.

```
Pickle.to_string<int × M.t> (s, M.make s)
```

However, casts between distinct nominal types never succeed, even if they are likely to have the same memory layout; the following cast fails:

```
type cartesian = { x : float ; y : float }
type polar     = { r : float ; t : float }
Typeable.cast<polar>
     (Typeable.mk<cartesian> { x = 1.0; y = 3.0 })
⟹ CastFailure
```

This difference reflects the fact that a programmer can cause a single value to have different types using module abstraction, but there is no way for a value to receive more than one variant or record type in a program that does not use unsafe operations.

Extension of equality on the type representation to a total ordering makes `typerep` values suitable for use as keys in finite maps.

```
module TypeRep :
sig
  type t = typerep
  val compare : t → t → int
  val eq : t → t → bool
  ...
end
```

With the addition of the `compare` function, the `TypeRep` module matches the signature required by the various functors that construct collection types. The `compare` function returns

a negative, zero or positive number to indicate that the first argument is respectively less than, equal to or greater than the second.

To distinguish nominal types at runtime, *deriving* constructs a unique string for each type constructor:

```
type α r = R of α deriving (Typeable)
⟹
module Typeable_r (Typeable_α : TYPEABLE)
    : TYPEABLE with type a = Typeable_α.a r =
 Typeable.Defaults
   (struct
      type a = Typeable_α.a r
      let type_rep =
       let t =
          (`Generative (intern "t.ml_2_1381210804.8705_r",
                        [V_a.type_rep]),
           fresh_id ()) in
         lazy t
    end)
  end
```

The string is constructed from the source file name, the type name and a timestamp. A significant shortcoming in the current implementation is the lack of any guarantee that the same string will be emitted by different runs of the program (in fact, it is guaranteed that this will *not* happen!). The inclusion of the timestamp is intended to compensate for the fact that the file name and type name are not alone sufficient to guarantee global uniqueness; there is only limited information about the context available to *deriving* when the definition is seen. A solution to this problem is given in Billings, Sewell, Shinwell, and Strniša (2006) in the context of a modification to the OCaml compiler, but a solution based on local syntactic transformations remains elusive.

4.3.4 Sharing

One component of the state used by the pickling algorithm is a map from values to identifiers. The pickling of each value v commences (in the `allocate` function) by examining the map to discover whether we have already assigned an identifier to some value equal to v. The map makes use of two classes. The `Typeable` class, described in Section 4.3.3, allows us to convert each value to a universal type, and so use a single map to store elements of many types. The `Hash` class, which we shall describe now, computes a hash of each value; this enables us to give a fast implementation of the search for duplicates.

Figure 4.19 gives the interface to the `Hash` class. There are two functions: `hash`, which

```
module type HASH =
sig
  type a
  module Eq : EQ with type a = a
  val merge : a → hstate →  hstate
  val hash : ?depth:int → a → int
end
```

Figure 4.19: Signature of the `Hash` class

```
type hstate = {
  code : int ;
  nodes : int
}
exception Done of int

let hash ?(depth=5) v =
  try (merge v { code = 0; nodes = depth }).code
  with Done code → code

let merge_int : int → hstate → hstate =
  fun i s →
    if s.nodes > 0
    then { code = s.code × 65599 + i; nodes = s.nodes − 1 }
    else raise (Done s.code)
```

Figure 4.20: Implementation of `hash`

computes a hash of a value, examining a number of nodes bounded above by the optional `depth` argument, and `merge`, a lower-level function for combining intermediate results obtained from hashing subvalues. There is one superclass, `Eq`: we require each hashable type to support equality, and impose the constraint that the hashes of two values must be the same whenever the values are equal:

        Hash.hash<a> l = Hash.hash<a> r   *whenever*   Eq.eq<a> l r

Figure 4.20 gives the types and functions used in the implementation of `Hash`. The `hstate` record tracks the hash code and the number of nodes examined in its computation, so that traversal can be halted (by raising the exception `Done`) when sufficiently many nodes have been examined. The `hash` function in the interface has a common implementation at every type, given here: `merge` is called with an initial state, and computation ends with a result either when `merge` returns, or when `Done` is raised with the hash code as argument. The `merge_int`

Merge at record types

$merge$ ≪ { l_1 : a_1; ...; l_n : a_n } ≫ =
　　≪ **fun** { l_1 = x$_1$; ...; l_n = x$_n$ } s →
　　　　Hash_a$_1$.merge x$_1$ (... (Hash_a$_n$.merge x$_n$ s))≫

Merge at polymorphic variant types

$merge$ ≪ [`$T_1$ **of** $a_1$ |...| `T_n **of** a_n | t_1 | ... | t_m] ≫ =
　　≪ **fun** v s → **match** v **with**
　　　`T_1 x → Hash_a$_1$.merge x (merge_int $taghash$ $T_1$$ s)
　　　 ...
　　| `T_n x → Hash_a$_n$.merge x (merge_int $taghash$ $T_n$$ s)
　　| #t_1 **as** x → Hash_t$_1$.merge x s
　　　 ...
　　| #t_m **as** x → Hash_t$_n$.merge x s ≫

Figure 4.21: Generation of the `Hash.merge` function

---

function gives the implementation of `merge` at type `int`; this forms the basis of the hashing function at every other type. The hashing function, based on the magic number 65599, is a common one, and matches the function used in the OCaml standard library.

Figure 4.21 gives the implementation of `Hash.merge` at record and variant types. There are no surprises: at record types the hash is computed by merging the hash of each field; at variant types the hash is computed by merging the hash of the tag and the hash of the argument.

4.3.5 Specialisation example: alpha-equivalence

The equality predicate specified in Section 4.2 enables the `Pickle` serialiser to introduce sharing between immutable values that have identical structure, potentially shrinking the serialised representation. Since equality can, like all operations provided by *deriving*, be customised at a particular type, there is room for even greater compression if we can specialise the definition of equality to a more inclusive predicate. For example, if we have a representation of lambda terms we might wish to consider all α-equivalent terms equal (although this will not always be a sensible choice, since there are many programs that can distinguish between α-equivalent terms!). If we introduce an instance of `Eq` which is visible to the `Pickle` instance for lambda terms then when terms are serialised only one member of each α-equivalence class will be written out. (We may also need to specialise `Hash` to maintain the correct relationship between `Hash` and `Eq`.)

Here we encounter a small technical difficulty with the current design of *deriving*. The preprocessor inserts generated instance definitions immediately following the definition of the

associated type. Further, since Eq is a superclass of Pickle, the Eq instance for a type must be visible within its Pickle instance. Our definition of Eq must follow the type definition for the constructors to be visible, but the preprocessor will insert the Pickle definition between the definition of the type and our definition of Eq (which the preprocessor ignores). We can dodge the difficulty using a recursive module, which allows us to bring a definition of Eq into scope within the preceding Pickle definition, but this is a rather distasteful necessity. We plan to address the difficulty in future versions of *deriving*.

Figure 4.22 defines a datatype, exp, for representing lambda terms, together with specialised instances for Eq and Hash that are visible to the generated instance for Pickle. The Eq instance uses α-equivalence, rather than structural equality; the Hash instance correspondingly ignores variable names, computing a hash based on constructors only. Parts (a) and (b) of Figure 4.23 show the sharing introduced by Pickle for a particular lambda term (a call-by-value fixpoint combinator) represented as a value of exp.

There is one further opportunity to increase the potential for sharing. OCaml strings are mutable, so the default definition of equality compares their physical addresses rather than their contents. A value-oriented instance of Eq for strings (taking OCaml's = for eq) leads to further sharing under Pickle, as seen in Figure 4.23 (c). In order to change the predicate used to compare strings we can define a type alias for string, istring, and a module Eq_istring, and replace the references to string in the definition of exp with references to istring.

```
type istring = string
module Eq_istring = struct
  type a = istring
  let eq = (=)
end
...
type exp = Var of istring
        | App of exp × exp
        | Lam of istring × exp
          deriving (Eq)
```

This definition allows Eq.eq<exp> to use structural equality for string comparisons, but has no effect on the rest of the program. Our notation for overloaded operations is generally more cumbersome than in a language such as Haskell where there is no need to specify the type at each call. In this case the burden becomes an advantage: *deriving* selects instances according to the declared name of a type, so we can use different instances for a given type at various points in the program by introducing a type alias for each instance. (A similar point is made in Dreyer et al. (2007), which couples Haskell-style overloading with an operation for explicitly making a particular instance "canonical" in a certain scope.)

```
  module Env = Env.Make(String)
  module rec Exp :
sig
   type exp = Var of string
            | App of exp × exp
            | Lam of string × exp
              deriving (Eq, Hash, Typeable, Pickle)
end =
struct
  module Eq_exp = struct
    open Exp
    type a = exp
    let eq l r
    = let rec alpha_eq (lenv, renv as envs) n = function
       | Var l, Var r →
         (match Env.mem l lenv, Env.mem r renv with
          | true, true → Env.find l lenv = Env.find r renv
          | false, false → l = r
          | _ → false)
       | App (fl,pl), App (fr,pr) →
         alpha_eq envs n (fl, fr) && alpha_eq envs n (pl, pr)
       | Lam (vl,bl), Lam (vr,br) →
         alpha_eq
           (Env.add vl n lenv, Env.add vr n renv)
           (n+1)
           (bl, br)
       | _  → false
     in alpha_eq (Env.empty, Env.empty) 0 (l,r)
  end
  module Hash_exp = Hash.Defaults(struct
     module Eq = Eq_exp
     type a = exp
     let rec merge v s = match v with
     | Var _ → merge_int 1 s
     | App (f, p) → merge f (merge p (merge_int 2 s))
     | Abs (_, body) → merge body (merge_int 3 s)
  end)
  type exp = Var of string
           | App of exp × exp
           | Lam of string × exp
             deriving (Typeable, Pickle)
end
```

Figure 4.22: Using α-equivalence as equality to increase sharing

Figure 4.23: Sharing lambda terms
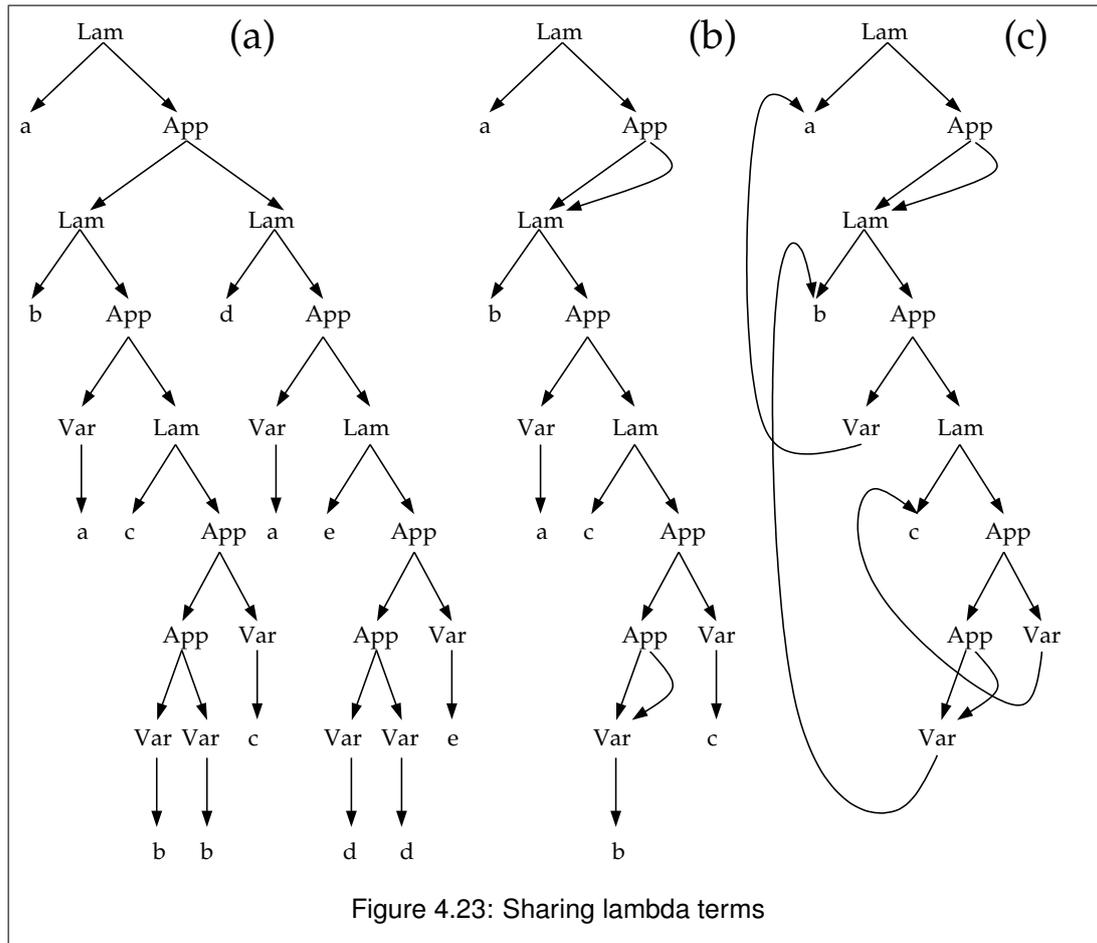
4.3.6 Compactness of serialised data

We have measured the spacewise performance of the `Pickle` serialiser described in Sections 4.3.1–4.3.2 in two simple applications. First, we used Camlp4 to parse a selection of variously-sized OCaml files (`path.ml` (1.8k), `predef.ml` (7.6k) and `includemod.ml` (15k)) from the OCaml distribution, passed the resulting syntax trees to the `Marshal` and `Pickle` serialisers, and compared the size of the output. Next, we compared the sizes of the representations generated for lambda terms by each serialiser on lambda terms, using the α-equivalence-aware serialiser developed in Section 4.3.5 and three collections of randomly-generated open lambda terms published by Liang and Nadathur (2002)[3]. The results of both tests are shown in Figure 4.24.

While `Pickle` can save space by introducing sharing we have made only modest attempts to make the format in which values are stored compact. It is therefore little surprise that, in the "OCaml" test, the default output of `Pickle` is shown to be significantly (although not

---

[3] These terms are available at http://cs.hofstra.edu/~cscccl/lambda-examples/.

	Marshal	**Pickle** (× Marshal)	**Pickle (specialised)** (× Marshal)
OCaml (path.ml)	5752	9178 (**1.59**)	-
OCaml (predef.ml)	27890	39596 (**1.42**)	-
OCaml (includemod.ml)	53292	75839 (**1.42**)	-
Lambda (reg.)	8732	21948 (**2.51**)	5233 (**0.60**)
Lambda (med.)	34993	89990 (**2.57**)	10045 (**0.29**)
Lambda (big.)	63513	176811 (**2.78**)	15259 (**0.24**)

Figure 4.24: Comparative size (bytes) of the output of Pickle and Marshal serialisers

disastrously) less compact than the default output of `Marshal`. In the results of the `Lambda` test, where the `Pickle` serialiser is enhanced using domain knowledge, the performance of `Pickle` is much more impressive, yielding output between two and four times smaller than `Marshal` (and up to eleven times smaller than the uncustomised `Pickle` serialiser). These results suggest that it is worthwhile exploring domain-specific solutions to serialisation problems, and that extensibility to incorporate domain knowledge is a valuable facet of the design of a serialiser.

4.3.7 Safety

We saw in Section 4.1.2.1 the disastrous effects of passing invalid data to the `Marshal` functions. Since `Pickle` is type-aware, no such problems arise. If the string that we pass to `Pickle.from_string` is a valid representation of the declared return type then it will be interpreted at that type; if it is not, an exception will be raised.

```
Pickle.from_string<float> (Pickle.to_string<int> 0)
⟹ 0.0
Pickle.from_string<Exp.Exp.exp>
      (Pickle.to_string<int> 0)
⟹ Exception: Pickle.Pickle.UnpicklingError
      "Error unpickling constructor".
```

Although this behaviour is "safe" in the sense that it will not cause a program crash, it may be that it is not quite what is desired. For some applications it might be more appropriate to ignore representation compatibility, and raise an exception whenever there is a mismatch between serialisation and deserialisation types. It would be straightforward to add this behaviour by serialising a representation of the type along with the value, but until the difficulty with

mapping type constructors to unique names described in Section 4.3.3 is resolved it would only be possible to check equality of serialised types within the same compiled instance of the program which produced them.

4.3.8 Related work

The serialisation problem has received considerable attention in the ML community. The Hash-Caml (Billings et al., 2006) project extends the OCaml bytecode compiler with type-passing to make the standard `Marshal` operation type-safe. Cohen and Herrmann (Cohen and Herrmann, 2005) discuss the implementation of a staged serialiser in MetaOCaml which bears some similarity to our preprocessor-based approach, with even more reliance on unsafe low-level operations, used by the serialiser to break through abstraction barriers. Tack, Kornstaedt, and Smolka (2005) describe the addition of serialisation as a primitive service to Alice ML, giving an elegant account of their serialisation algorithm in terms of a domain-specific instruction set.

A number of existing Camlp4 extensions construct serialisation functions from types. Markus Mottl's *sexplib* (Mottl, 2008) and *bin-prot* (Mottl, 2007) support textual and binary serialisation protocols, but do not currently handle sharing or cyclic data. Daniel de Rauglaudre's IoXML (de Rauglaudre, 2002) supports conversions between OCaml values and XML. Martin Jambon's *json-static* (Jambon, 2007) supports serialisation to and from JavaScript Object Notation.

We discussed two libraries of serialisation combinators (Elsman, 2005, Kennedy, 2004) in Section 4.1.2.2. Karvonen (2007) discusses ways to address some of the deficiencies in Elsman's library. Finally, serialisation has been a common example program in the generic programming community: see, for example, Hinze (2004).

One disadvantage suffered by our approach in comparison to almost all alternatives is a certain lack of type-safety: it is easy to use Camlp4 to generate code that is ill-typed. The fact that all Camlp4-generated code is subsequently type-checked by the OCaml compiler makes this less serious a problem than it might otherwise be: it is not possible to actually execute an ill-typed program. Nevertheless, it would certainly be preferable to eliminate the possibility of generating ill-typed code altogether.

4.4 Conclusions and future work

We have described the design and implementation of *deriving*, the system used in the prototype implementation of Links for constructing generic functions from type definitions, and its particular application to the serialisation problem. The *deriving* encoding of serialisation places

a much lower burden on the user and much greater coverage than the combinator approach; it
offers greater safety and flexibility than standard OCaml marshalling. Perhaps most enticingly,
the specialisation property, which allows a user to supply an alternative implementation for a
derived function at a particular type, makes it possible to improve serialisation using domain
knowledge without writing any serialisation code, leading to a considerable reduction in the
size of serialised data. Generated instances produce acceptably compact output compared to
`Marshal`, even without specialisation.

We have also shown how an understanding of the correspondence between modules and
type-classes can guide software design and described techniques to make it viable in the ab-
sence of a full system of recursive modules.

While the correspondence between type classes and modules is a useful guiding principle in
the design of our *deriving* framework, using modules to represent the "dictionaries" containing
overloaded function implementations has a number of drawbacks related to the distinction in
OCaml between the module and expression languages. Any functions that are parameterised
over instances (such as a `print` function which displays values of any type that is a member
of the `Show` class) must be wrapped in functors. For similar reasons, overloaded functions are
not first-class citizens in our current design: they are tied to the module language, and cannot
be passed around freely as values. Representing dictionaries using value-level products such
as objects or records would address these limitations, but would limit the flexibility of *deriving*
in other ways, most notably in the encoding of "constructor classes" (Jones, 1993). While
functors permit abstraction over type constructors, OCaml does not support the higher-kinded
type variables that are necessary to encode dictionaries for constructor classes at the value level.
However, objects offer additional advantages: they are easier to compose than modules, since
there are no problems with name clashes between type components, and overriding of virtual
functions offers a convenient analogue to the default method facility of Haskell type classes. It
is not yet clear which encoding of dictionaries — modules, records or objects — provides the
set of features that is most convenient in practice.

Finally, *deriving* currently provides more support for *using* the generic functions supplied
with the implementation than for *writing* new generic functions; adding generic functions to
deriving currently requires a certain degree of Camlp4 expertise. It is obviously desirable
to eliminate this bias. The *deriving* implementation provides a fairly convenient framework
for expressing generic functions in a Camlp4 extension, but it would be preferable to write
generic functions directly in OCaml. One possible approach is to use *deriving* to generate
general value-level representations of datatypes which can then be used by generic functions to
traverse values of those types. As we said in Section 4.2.7, the planned addition of first-class

modules to OCaml will make it easier to separate type representations from generic function implementations in this way.

Chapter 5

Signed and sealed

5.1 Introduction

An abstract type definition divides a program into two parts. The first part *defines* the type, and
acts on its representation. The second part *uses* the type, and acts on its interface. Language
features for creating this abstraction boundary may be classified into two styles. The first uses
a type signature to distinguish values of the abstract type from values of the type with which
it is implemented. For example, in Standard ML we might choose to define a type of complex
numbers as follows.

```
structure Complex =
struct
  type complex = real × real
  fun make (x,y) = (x,y)
  fun real (x,y) = x
  fun imag (x,y) = y
  fun conj (x,y) = (x, ~y)
  fun plus ((u,v), (x,y)) = (u+x, v+y)
end :>
sig
  type complex
  val make : real × real → complex
  val real : complex → real
  val imag : complex → real
  val conj : complex → complex
  val plus : complex × complex → complex
end
```

The type system enforces the distinction between the abstract type `complex` and its representa-
tion type `real × real`, rejecting attempts to conflate the one with the other outside the region
of code between **struct** and **end** where the `complex` type is defined. We dub this style of

abstraction *signing*.

The second style of abstraction involves tagging values of the type as they cross from the section where the abstract type is defined to the section where it is used. Again, in Standard ML we might define a type of complex numbers in terms of a pair of `reals` as follows.

```
abstype
   complex = MkComplex of real × real
with
   fun make (x,y) = MkComplex (x,y)
   fun real (MkComplex (x,y)) = x
   fun imag (MkComplex (x,y)) = y
   fun conj (MkComplex (x,y)) = MkComplex (x, ~y)
   fun plus (MkComplex (u,v), MkComplex (x,y))
            = MkComplex (u+x, v+y)
end
```

The `MkComplex` data constructor can be used to construct and deconstruct values of the new type, but the **abstype** keyword delimits the scope of `MkComplex` to the section of the program between **with** and **end**; in other parts of the program such values cannot be deconstructed. We dub this style of abstraction *sealing*[1].

Both signing and sealing appear in modern functional languages as the preferred means of defining abstract types. In Standard ML both mechanisms are available, as illustrated above. Since signing involves drawing the abstraction boundary in the types and sealing involves drawing the abstraction boundary in the terms, it is no surprise that abstract types in Scheme are typically based on sealing (Matthews and Ahmed, 2008). While languages related to Haskell — Gofer/Hugs (Jones and Peterson, 1999) and Miranda (Turner, 1985) — use signing, abstract types in Haskell itself use sealing, albeit a variant in which the abstraction is enforced statically. Figure 5.1 summarises.

We have said that we might choose either of two *notations* — type-based or term-based — for defining abstract types. A second consideration is how the values of an abstract type are *represented*: is a value of type `complex` represented identically, or merely isomorphically, to a value of type `real × real`? In some circumstances a distinct representation for each abstract type might be preferable. For example, overloading may, for some languages, be implemented by runtime inspection of the representation of a value in order to pick the appropriate implementation of an overloaded function; it is obviously undesirable for complex numbers to be displayed identically to pairs of reals, even if we use the same overloaded function name for printing both. Conversely, if there is no need to distinguish values at runtime then it may be

---

[1]The "signing" and "sealing" terminology, while evocative, is not entirely standard. "Sealing" is sometimes used in the ML literature for the operation of creating abstract types through signature ascription. However, the use of "sealing" for creating abstract types through data constructors can be traced at least as far back as Morris (1973).

	SML `sig`	SML `abstype`	Scheme	Haskell `newtype`
Notation	Types	Terms	Terms	Terms
Checking	Static	Static/Dynamic	Dynamic	Static

Figure 5.1: Abstraction schemes

more efficient to use identical representations, eliminating a layer of indirection and enabling space-saving optimisations such as flattening nested pairs. Thus we find, for example, that the data constructor introduced by Haskell's **newtype** has no runtime representation (Peyton Jones and Hughes, 1999). (Overloading in Haskell is resolved via static generation of implicit "dictionary" arguments to overloaded functions, not by runtime inspection of values.) Similarly, constructors of unary sums in Standard ML may be erased, since there is no way for a program to detect the erasure (although such issues are not considered in the Definition).

Comparing the two alternatives for representing values of abstract type with the two styles of abstract type definition, we find that there is apparently a close correspondence. It is not difficult to see that a sealing constructor can be used to give a distinct representation to values of an abstract type. It is likewise clear that the signing style need not introduce any representational overhead. It is easy to reach the conclusion that the signing style fits particularly well with a statically-typed language in which values of abstract type are represented identically to values of the representation type, whereas sealing fits well with a dynamic language in which each abstract type has a distinct representation.

For the language designer, then, there may be ostensible reasons to prefer either the signing or the sealing style of abstraction. In this vein, the designers of Haskell write (of an example program written in the sealing style)

> "The Show instance for Stack can be different from the Show instance for lists, and there is no ambiguity about whether a given subexpression is a Stack or a list. It was unclear to us how to achieve this effect with abstype." (Hudak et al., 2007)

Likewise, the designer of Miranda claimed (of the signing style provided in that language):

> "Note that the mechanism for data type abstraction which is presented here is inextricably bound up with strong [...] typing. There would seem to be no equivalent mechanism available in a language which delays its type checking until run time. By contrast, the traditional account of data type abstraction as an act of encapsulation would appear to be equally applicable to both strongly and weakly typed languages." (Turner, 1985)

In fact, as we demonstrate in this chapter, the styles are interconvertible: there is an automatic translation between them preserving operational equivalence with respect to a standard

semantics. That is, we may define abstract types using either the signing or the sealing style without constraining the runtime representation of values. For example, as we show in Section 5.5, it is possible to extend Haskell with a construct for signing by translation into the built-in constructs, thus avoiding the need for the user to write tags; we could equally well add such a mechanism to Scheme. While it appeared to the Haskell designers that `newtype` was necessary to avoid ambiguity in overloaded functions, it would, in fact, have been possible to use a construct based on signing without any such ambiguity.

Early plans for this dissertation connected the discussion above more closely with the work in Chapter 4. Generic functions, as discussed in that chapter, are closely connected to overloading; functions such as `eq` and `pickle` (Sections 4.2.1 and 4.3) are overloaded for all algebraic types (although the behaviour at each type is determined by the structure of the type rather than specified in an ad-hoc fashion). As discussed above, overloading is, in turn, connected to abstract types: an approach to overloading that depends on runtime inspection of values can only offer different behaviour at an abstract type and its representation if the two are represented differently at runtime. In this chapter we show that the representation of a value of abstract type is not tied to the notation used for the definition of the type; we had hoped to build upon this foundation to show that similar freedoms exist when designing systems of overloading and generic programming, but time constraints ultimately prevented such an investigation.

Our evidence that the two styles of abstract type are interconvertible is based on relational parametricity (Reynolds, 1983, Wadler, 1989). Pitts (2000) gives an appealing presentation of parametricity in the presence of polymorphism and partial functions, in which the usual denotational characterisation of admissible relations is replaced by a purely syntactic approach. Our evidence is an application of the central result in a minor extension to Pitts' system.

This chapter makes the following contributions:

1. A **characterisation** of the two essential styles of abstract type, *signing* and *sealing* (Sections 5.1 and 5.2.1).

2. **Evidence** that the two styles are interconvertible (Section 5.4) via a type-indexed function (Section 5.3) in a higher-order language with polymorphism and recursion, based on an extension of Pitts' PolyPCF (Section 5.2).

3. An **application** of the result: a robust implementation of abstract types using *signing* to Haskell (Section 5.5), by translation into Haskell's *sealing*-style construct, `newtype`.

5.2 PolyPCF with tags

In order to demonstrate the equivalence of the two styles of defining abstract types we begin by defining a programming language in which both styles can be expressed. Our language of choice is an extension of Pitts' *PolyPCF* (Pitts, 2000), which was designed to investigate the operational behaviour of programs built from partial polymorphic functions. The signing style of abstract type definition can be reduced to a use case of polymorphic types, already present in PolyPCF. In order to capture the sealing style we extend PolyPCF with *tags*, a sort of unary variant type with a constructor $\mathbf{in}_\mathsf{T}$, destructor $\mathbf{out}_\mathsf{T}$ and type constructor $\mathsf{T}(-)$, for each T of an infinite set of tag names *Tag*.

It is convenient to establish the necessary results in terms of polymorphic types and tag types. The language constructs which we will ultimately use to compare signing and sealing — **signtype** and **sealtype** — may be reduced to particular uses of these more primitive constructs. Starting with polymorphism and tags simplifies both the initial presentation of PolyPCF and the proofs. (One less desirable feature of this approach is that our simpler language does not capture certain aspects of the signing and sealing styles, most notably the scoping of tags that is essential to the sealing style. For the moment we emulate this scoping by side conditions on the programs in which our abstract types are used, leaving a closer emulation of the two styles to future work.)

We will first illustrate by example how abstract types may be defined in PolyPCF (Section 5.2.1), then proceed to the formal development of the core language (Sections 5.2.2–5.2.10). Section 5.4 gives the formal definition of the derived forms **signtype** and **sealtype**.

5.2.1 Example

The four programs in Figure 5.2 illustrate the encoding of the two styles of type abstraction. Our examples make use of types for pairs and numbers which are not directly supported by PolyPCF, but which can be Church-encoded in the usual way (Girard, Taylor, and Lafont, 1989). We will use pairs and numbers as though PolyPCF supported them directly.

The program on the upper left gives a signing-style definition of a complex number type using **signtype**. The program on the lower left gives the same definition written in plain PolyPCF without **signtype**.

The program on the upper right gives a sealing-style definition of a complex number type using **sealtype**. Since PolyPCF is explicitly typed — unlike, say, Standard ML — we must give a type signature for each exported value. The program on the lower right gives the same

Signing

```
signtype α = num × num
with make : num × num → α
         = λp:num × num (p)
     real : α → num
         = λp:num × num (fst p)
     imag : α → num
         = λp:num × num (snd p)
     conj : α → α
         = λp:num × num (⟨fst p, −snd p⟩)
     plus : α → α → α
         = λp:num × num (λq:num × num (⟨fst p + fst q,
                                         snd p + snd q⟩))
in M
```

Signing, desugared

```
(Λα (λmake:num × num → α
    (λreal:α → num
    (λimag:α → num
    (λconj:α → α
    (λplus:α → α → α (M)))))))
(num × num)
(λp:num × num (p))
(λp:num × num (fst p))
(λp:num × num (snd p))
(λp:num × num (⟨fst p, −snd p⟩))
(λp:num × num (λq:num × num (⟨fst p + fst q,
                              snd p + snd q⟩)))
```

Sealing

```
sealtype α = C(num × num)
with make : num × num → α
```
= λp:num × num (in_C p)
```
     real : α → num
```
= λp:C(num × num) (fst (out_C p))
```
     imag : α → num
```
= λp:C(num × num) (snd (out_C p))
```
     conj : α → α
```
= λp:C(num × num) (in_C ⟨fst (out_C p), −snd (out_C p)⟩)
```
     plus : α → α → α
```
= λp:C(num × num)
  (λq:C(num × num)
    (in_C ⟨fst (out_C p) + fst (out_C q),
          snd (out_C p) + snd (out_C q)⟩))
```
in M
```

Sealing, desugared

```
(Λα (λmake:num × num → α
    (λreal:α → num
    (λimag:α → num
    (λconj:α → α
    (λplus:α → α → α (M)))))))
(C(num × num))
```
(λp:num × num (in_C p))
(λp:C(num × num) (fst (out_C p)))
(λp:C(num × num) (snd (out_C p)))
(λp:C(num × num) (in_C ⟨fst (out_C p), −snd (out_C p)⟩))
(λp:C(num × num) (λq:C(num × num) (in_C ⟨fst (out_C p) + fst (out_C q),
                                       snd (out_C p) + snd (out_C q)⟩)))

Figure 5.2: An abstract type of complex numbers in extended PolyPCF

definition written in plain PolyPCF without **sealtype**.

The remainder of this section is a straightforward recapitulation of Pitts (2000). Most figures and definitions are adapted from this work, the key difference being the addition of tags. Adding tags is straightforward: no new insight or style of proof is required. The bracketed numbers accompanying each definition and figure refer to the corresponding definition or figure in Pitts' work.

5.2.2 Syntax

Fig. 5.3 defines the syntax of our extension of PolyPCF. We follow standard conventions: for example, application is left associative and we omit parentheses where possible.

PolyPCF augments the familiar polymorphic lambda calculus with fixpoint recursion and polymorphic lists, which serve the role of a ground type. We add a type constructor, $T(-)$, for tagged types, and corresponding term constructors, **in**$_T$ - for tagging and **out**$_T$ - for untagging.

5.2.3 Typing

Figure 5.4 gives the typing relation $\Gamma \vdash M : \tau$ between typing environments Γ, terms M and types τ. As the rules are entirely standard we refrain from commenting further.

5.2.4 Evaluation

Definition 27. *(Values). The set V of values is drawn from the set of closed terms of closed type generated by the following grammar:*

$$
\begin{aligned}
V \quad ::= \quad & \lambda x : \tau(M) \\
& \Lambda \alpha(M) \\
& \textit{\textbf{nil}}_\tau \\
& M :: M \\
& \textit{\textbf{in}}_T\, M
\end{aligned}
$$

Figure 5.5 gives the evaluation relation for extended PolyPCF. The rules are mostly determined by the definition of values. Function application is call-by-name.

5.2.5 Equivalence

In order to show that the two styles of creating abstract types are equivalent we must first make clear what we mean by "equivalence" of terms. A common approach is to base equivalence on the notion of *context*: two terms are considered equivalent if they may be interchanged in any program context without effect on the observable behaviour of the program. This raises the

τ	::=		(**Types**)
		α	type variable
		τ → τ	function type
		∀α(τ)	∀ − type
		τ list	list type
		T(τ)	tagged type
M	::=		(**Terms**)
		x	variable
		λx : τ(M)	function abstraction
		M M	function application
		Λα(M)	type generalisation
		M τ	type specialisation
		fix(M)	fixpoint recursion
		nil$_τ$	empty list
		M :: M	non-empty list
		case M **of** {**nil** ⇒ M | x :: x ⇒ M}	case expression
		in$_T$ M	tagging expression
		out$_T$ M	untagging expression

Notes

(i)   α and x range over disjoint countably infinite sets *TyVar* and *Var* of *type variables* and *variables* respectively.

(ii) The constructions $∀α(−)$, $λx : τ(−)$, $Λα(−)$, and **case** M **of**{**nil** ⇒ M′ | x :: x′ ⇒ −} are binders. We will identify types and terms up to renaming of bound variables and bound type variables.

(iii) We write $ftv(e)$ for the finite set of free type variables of an expression e (be it a type or a term) and $fv(M)$ for the finite set of free variables of a term M.

(iv) We write $e[τ/α]$ for the capture-avoiding substitution of a type τ for all free occurrences of a type variable α in a type or term e and $M[M′/x]$ for the capture-avoiding substitution of a term M′ for all free occurrences of a variable x in M.

Figure 5.3: Syntax of the PolyPCF language extended with tags [Pitts' Figure 1]

$$\Gamma, x : \tau \vdash x : \tau$$

$$\frac{\Gamma, x : \tau_1 \vdash M : \tau_2}{\Gamma \vdash \lambda x : \tau_1(M) : \tau_1 \to \tau_2} \qquad \frac{\Gamma \vdash F : \tau_1 \to \tau_2 \qquad \Gamma \vdash A : \tau_1}{\Gamma \vdash F\, A : \tau_2}$$

$$\frac{\Gamma, \alpha \vdash M : \tau}{\Gamma \vdash \Lambda\alpha(M) : \forall\alpha(\tau)} \qquad \frac{\Gamma \vdash G : \forall\alpha(\tau_1)}{\Gamma \vdash G\, \tau_2 : \tau_1[\tau_2/\alpha]}$$

$$\frac{\Gamma \vdash F : \tau \to \tau}{\Gamma \vdash \mathbf{fix}(F) : \tau}$$

$$\Gamma \vdash \mathbf{nil}_\tau : \tau\, \mathtt{list} \qquad \frac{\Gamma \vdash H : \tau \qquad \Gamma \vdash T : \tau\, \mathtt{list}}{\Gamma \vdash H :: T : \tau\, \mathtt{list}}$$

$$\frac{\Gamma \vdash L : \tau_1\, \mathtt{list} \qquad \Gamma \vdash M_1 : \tau_2 \qquad \Gamma, h : \tau_1, t : \tau_1\, \mathtt{list} \vdash M_2 : \tau_2}{\Gamma \vdash \mathbf{case}\, L\, \mathbf{of}\, \{\mathbf{nil} \Rightarrow M_1 \mid h :: t \Rightarrow M_2\} : \tau_2}$$

$$\frac{\Gamma \vdash M : \tau}{\Gamma \vdash \mathbf{in}_T\, M : T(\tau)} \qquad \frac{\Gamma \vdash M : T(\tau)}{\Gamma \vdash \mathbf{out}_T\, M : \tau}$$

Notes

(i) Typing judgments take the form $\Gamma \vdash M : \tau$ where

   (a) the *typing environment* Γ is a pair A, Δ with A a finite subset of *TyVar* and Δ a function defined on a finite subset $dom(\Delta)$ of *Var* and mapping each $x \in dom(\Delta)$ to a type with free type variables in A;

   (b) M is a term with $ftv(M) \subseteq A$ and $fv(M) \subseteq dom(\Delta)$;

   (c) τ is a type with $ftv(\tau) \subseteq A$.

(ii) The notation $\Gamma, x : \tau$ indicates the typing environment obtained from the typing environment $\Gamma = A, \Delta$ by properly extending the function Δ by mapping $x \notin dom(\Delta)$ to τ. Similarly, Γ, α is the typing environment obtained by properly extending A with an $\alpha \notin A$.

(iii) The explicit type information included in the syntax of function abstractions and empty lists ensures that, given Γ and M, there is at most one τ for which $\Gamma \vdash M : \tau$ holds.

Figure 5.4: Typing assignment relation for PolyPCF with tags [Pitts' Figure 2]

$$V \Downarrow V$$

$$\frac{F \Downarrow \lambda x : \tau(M) \qquad M[A/x] \Downarrow V}{F\,A \Downarrow V} \qquad\qquad \frac{G \Downarrow \Lambda\alpha(M) \qquad M[\tau/\alpha] \Downarrow V}{G\,\tau \Downarrow V}$$

$$\frac{F\,\mathbf{fix}(F) \Downarrow V}{\mathbf{fix}(F) \Downarrow V}$$

$$\frac{L \Downarrow \mathbf{nil}_\tau \qquad M_1 \Downarrow V}{\mathbf{case}\,L\,\mathbf{of}\,\{\mathbf{nil} \Rightarrow M_1 \mid h :: t \Rightarrow M_2\} \Downarrow V}$$

$$\frac{L \Downarrow H :: T \qquad M_2[H/h, T/t] \Downarrow V}{\mathbf{case}\,L\,\mathbf{of}\,\{\mathbf{nil} \Rightarrow M_1 \mid h :: t \Rightarrow M_2\} \Downarrow V}$$

$$\frac{M' \Downarrow \mathbf{in}_T\,M \qquad M \Downarrow V}{\mathbf{out}_T\,M' \Downarrow V}$$

Figure 5.5: PolyPCF evaluation relation [Pitts' Figure 3]

question of what is meant by "program" and "observable behaviour"; the standard answers are that a program is a closed term of ground type (such as *boolean*) and its observable behaviour the constant (if any) to which it evaluates. PolyPCF uses lists in place of ground type, so it is reasonable to take "program" to mean a closed term of list type and "observable behaviour" to mean whether the program evaluates to **nil**.

While this notion of contextual equivalence is pleasingly simple, it is in certain respects too concrete for our purposes. For example, it is not possible to identify program contexts up to change of bound variables, since the result of plugging an open term into a context depends on whether the names of the binders in the context match the names of the free variables in the term. It is more convenient to work with a more abstract notion of *observational congruence* (Theorem 29), defined as the largest relation satisfying certain desirable properties (Section 5.2.6). Pitts (2000) shows that for PolyPCF this notion coincides with the more intuitive concept of contextual equivalence.

Our plan, then, is to reason in terms of observational congruence instead of the equivalent, but lower-level notion of contextual equivalence. We will first define (in Section 5.2.9) a *logical relation*, Δ, parameterised by a tuple of binary term relations, which, under certain conditions, coincides with observational congruence (Sections 5.2.7–5.2.10). By instantiating Δ with the

relation between `signtype`-bound terms and `sealtype`-bound terms we will be able to show that the construct used to define an abstract type makes no difference to the outcome of a program (Section 5.4.5).

5.2.6 Relations

Definition 28. *(**Properties of relations**). [Pitts' Definition 2.2] Suppose $\mathcal{E}$ is a set of 4-tuples* (Γ, M, M', τ) *satisfying*

$$\Gamma \vdash M \; \mathcal{E} \; M' : \tau \Rightarrow (\Gamma \vdash M : \tau \; \& \; \Gamma \vdash M' : \tau)$$

where we write $\Gamma \vdash M \; \mathcal{E} \; M' : \tau$ *instead of* $(\Gamma, M, M', \tau) \in \mathcal{E}$.

(i) $\mathcal{E}$ *is* compatible *if it is closed under the axioms and rules in Fig. 5.6. It is* substitutive *if it is closed under the rules in Fig. 5.7.*

(ii) *Compatible relations are automatically* reflexive. *A PolyPCF* precongruence *is a compatible, substitutive relation which is also transitive. A PolyPCF* congruence *is a precongruence which is also symmetric. (Symmetry and transitivity here refer to the term components of the 4-tuple).*

(iii) *We write* Typ *for the set of closed types* τ, *i.e. those for which* $ftv(\tau) = \emptyset$. *Given* $\tau \in Typ$, *we write* $Term(\tau)$ *for the set of closed terms* M, *i.e. those for which* $\emptyset \vdash M : \tau$. *The relation* $\mathcal{E}$ *is* adequate *if for all types* $\tau \in Typ$ *and closed terms* $M, M' \in Term(\tau \, \mathtt{list})$

$$\emptyset \vdash M \; \mathcal{E} \; M' : \tau \, \mathtt{list} \Rightarrow (M \Downarrow \boldsymbol{nil}_\tau \Leftrightarrow M' \Downarrow \boldsymbol{nil}_\tau)$$

Theorem 29. *(**PolyPCF observational congruence**). [Pitts' Theorem 2.3] There is a largest adequate, compatible and substitutive relation. It is an equivalence relation and hence is the largest adequate PolyPCF congruence relation. We call it* PolyPCF observational congruence *and write it as* $=_{\mathtt{obs}}$.

5.2.7 Frame stacks

Definition 30. *(**Frame Stacks**). [Pitts' Definition 3.2] Frame stacks provide a way to denote evaluation contexts. The grammar for PolyPCF frame stacks is*

$$S \quad ::= \quad \mathrm{Id} \mid S \circ F$$

where F *ranges over* frames*:*

$$\Gamma, x : \tau \vdash x \ \mathcal{E} \ x : \tau$$

$$\frac{\Gamma, x : \tau_1 \vdash M \ \mathcal{E} \ M' : \tau_2}{\Gamma \vdash \lambda x : \tau_1(M) \ \mathcal{E} \ \lambda x : \tau_1(M') : \tau_1 \to \tau_2}$$

$$\frac{\Gamma \vdash F \ \mathcal{E} \ F' : \tau_1 \to \tau_2 \qquad \Gamma \vdash A \ \mathcal{E} \ A' : \tau_1}{\Gamma \vdash (F\,A) \ \mathcal{E} \ (F'\,A') : \tau_2}$$

$$\frac{\alpha, \Gamma \vdash M \ \mathcal{E} \ M' : \tau}{\Gamma \vdash \Lambda\alpha(M) \ \mathcal{E} \ \Lambda\alpha(M') : \forall\alpha(\tau)} \qquad \frac{\Gamma \vdash G \ \mathcal{E} \ G : \forall\alpha(\tau_1)}{\Gamma \vdash (G\,\tau_2) \ \mathcal{E} \ (G'\,\tau_2) : \tau_1[\tau_2/\alpha]}$$

$$\frac{\Gamma \vdash F \ \mathcal{E} \ F' : \tau \to \tau}{\Gamma \vdash \mathbf{fix}(F) \ \mathcal{E} \ \mathbf{fix}(F') : \tau}$$

$$\Gamma \vdash \mathbf{nil}_\tau \ \mathcal{E} \ \mathbf{nil}_\tau : \tau \, \mathtt{list}$$

$$\frac{\Gamma \vdash H \ \mathcal{E} \ H' : \tau \qquad \Gamma \vdash T \ \mathcal{E} \ T' : \tau \, \mathtt{list}}{\Gamma \vdash (H :: T) \ \mathcal{E} \ (H' :: T') : \tau \, \mathtt{list}}$$

$$\frac{\Gamma \vdash L \ \mathcal{E} \ L' : \tau_1 \, \mathtt{list} \qquad \Gamma \vdash M_1 \ \mathcal{E} \ M_1' : \tau_2}{\Gamma, h : \tau_1, t : \tau_1 \, \mathtt{list} \vdash M_2 \ \mathcal{E} \ M_2' \vdash \tau_2}$$
$$\Gamma \vdash (\mathbf{case}\,L\,\mathbf{of}\,\{\mathbf{nil} \Rightarrow M_1 \mid h :: t \Rightarrow M_2\})$$
$$\mathcal{E} \ (\mathbf{case}\,L'\,\mathbf{of}\,\{\mathbf{nil} \Rightarrow M_1' \mid h :: t \Rightarrow M_2'\}) : \tau_2$$

$$\frac{\Gamma \vdash M \ \mathcal{E} \ M' : \tau}{\Gamma \vdash \mathbf{in}_T\,M \ \mathcal{E} \ \mathbf{in}_T\,M' : T(\tau)} \qquad \frac{\Gamma \vdash M \ \mathcal{E} \ M' : T(\tau)}{\Gamma \vdash \mathbf{out}_T\,M \ \mathcal{E} \ \mathbf{out}_T\,M' : \tau}$$

Figure 5.6: Compatibility properties [Pitts' Figure 4]

$$\frac{\alpha, \Gamma \vdash M \ \mathcal{E} \ M' : \tau_1}{\Gamma[\tau_2/\alpha] \vdash M[\tau_2/\alpha] \ \mathcal{E} \ M'[\tau_2/\alpha] : \tau_1[\tau_2/\alpha]}$$

$$\frac{\Gamma, x : \tau_1 \vdash M \ \mathcal{E} \ M' : \tau_2 \qquad \Gamma \vdash N \ \mathcal{E} \ N' : \tau_1}{\Gamma \vdash M[N/x] \ \mathcal{E} \ M'[N'/x] : \tau_2}$$

Figure 5.7: Substitutivity properties [Pitts' Figure 5]

$$
\mathsf{F} \quad ::= \quad (-\, M)
$$
$$
(-\, \tau)
$$
$$
(\textbf{\textit{out}}_\mathsf{T}\, -)
$$
$$
(\textbf{\textit{case}} - \textbf{\textit{of}}\{\textbf{\textit{nil}} \Rightarrow M \mid x :: x \Rightarrow M\})
$$

Figure 5.8 gives the typing rules for frame stacks. The judgment $\Gamma \vdash S : \tau \multimap \tau'$ holds for a frame stack S which, under the assumptions in the typing environment Γ, has argument type τ and result type τ'. Given $\tau, \tau' \in \mathit{Typ}$, we write $\mathit{Stack}(\tau, \tau')$ for the set of frame stacks S for which $\emptyset \vdash S : \tau \multimap \tau'$. We are particularly interested in the case in which τ' is a list type, and so define the following shorthand:

$$
\mathit{Stack}(\tau) \overset{\mathrm{def}}{=} \bigcup_{\tau' \in \mathit{Typ}} \mathit{Stack}(\tau, \tau'\, \texttt{list})
$$

Definition 31. *(Applying a frame stack to a term). [Pitts' Definition 3.4] The analogue for frame stacks of the operation of filling the hole of an evaluation context with a term is given by the operation $S, M \mapsto S\,M$, of* applying a frame stack to a term. *It is defined by induction on the length of the stack:*

$$
\begin{cases}
\mathrm{Id}\,M & \overset{\mathit{def}}{=} M \\
(S \circ F)\,M & \overset{\mathit{def}}{=} S(F[M])
\end{cases}
$$

where $F[M]$ is the term which results from replacing '$-$' by M in the frame F. Note that if $S \in \mathit{Stack}(\tau, \tau')$ and $M \in \mathit{Term}(\tau)$, then $S\,M \in \mathit{Term}(\tau')$.

Theorem 32. *(A structural induction principle for PolyPCF termination). [Pitts' Theorem 3.6]*

For all closed types $\tau, \tau' \in \mathit{Typ}$, for all frame stacks $S \in \mathit{Stack}(\tau, \tau'\, \texttt{list})$, and for all closed terms $M \in \mathit{Term}(\tau)$, we have

$$
S\,M \Downarrow \textbf{\textit{nil}}_{\tau'} \Leftrightarrow S \top M
$$

where the relation $(-) \top (-)$ is inductively defined by the rules in Figure 5.9.

The new rules for tagged types straightforwardly follow the pattern for other type constructors: there is one rule that pairs the introduction form $\textbf{in}_\mathsf{T}\, -$ with a stack whose top frame is the corresponding elimination form $\textbf{\textit{out}}_\mathsf{T}\, -$, and one rule that moves the elimination form $\textbf{\textit{out}}_\mathsf{T}\, -$ from the top of the stack to the term.

$$\Gamma \vdash \mathrm{Id} : \tau \rightarrowtail \tau$$

$$\frac{\Gamma \vdash S : \tau' \rightarrowtail \tau'' \qquad \Gamma \vdash A : \tau}{\Gamma \vdash S \circ (-A) : (\tau \to \tau') \rightarrowtail \tau''}$$

$$\frac{\Gamma \vdash S : \tau'[\tau/\alpha] \rightarrowtail \tau'' \qquad \alpha \text{ not free in } \Gamma}{\Gamma \vdash S \circ (-\tau) : \forall \alpha(\tau') \rightarrowtail \tau''}$$

$$\frac{\Gamma \vdash S : \tau' \rightarrowtail \tau'' \qquad \Gamma \vdash M_1 : \tau' \qquad \Gamma, h : \tau, t : \tau \, \mathtt{list} \vdash M_2 : \tau'}{\Gamma \vdash S \circ (\mathbf{case} - \mathbf{of} \, \{\mathbf{nil} \Rightarrow M_1 \mid h :: t \Rightarrow M_2\}) : \tau \, \mathtt{list} \rightarrowtail \tau''}$$

$$\frac{\Gamma \vdash S : \tau \rightarrowtail \tau'}{\Gamma \vdash S \circ \mathbf{out}_\mathrm{T} - : \mathrm{T}(\tau) \rightarrowtail \tau'}$$

Figure 5.8: Typing frame stacks [Pitts' Figure 6]

5.2.8   Term and stack relations

Definition 33. *(Term- and stack-relations). [Pitts' Definition 3.8] A PolyPCF term-relation is a binary relation between (typeable) closed terms. Given closed PolyPCF types $\tau, \tau' \in Typ$, we write*

$$Rel(\tau, \tau')$$

for the set of term-relations that are subsets of $Term(\tau) \times Term(\tau')$. A PolyPCF stack-relation is a binary relation between (typeable) frame stacks whose result types are list types. We write

$$StRel(\tau, \tau')$$

for the set of stack-relations that are subsets of $Stack(\tau) \times Stack(\tau')$.

Using the $(-) \; \top \; (-)$ relation (Figure 5.9) we can manufacture a stack-relation from a term-relation and vice versa, as follows:

Definition 34. *(The $(-)^\top$ operation on relations). [Pitts' Definition 3.9] Given any $\tau, \tau' \in Typ$ and $\mathrm{r} \in Rel(\tau, \tau')$, define $\mathrm{r}^\top \in StRel(\tau, \tau')$ by*

$$(S, S') \in \mathrm{r}^\top \;\overset{def}{\Leftrightarrow}\; \forall (M, M') \in \mathrm{r} \; (S \top M \Leftrightarrow S' \top M');$$

and given any $\mathrm{s} \in StRel(\tau, \tau')$ define $\mathrm{s}^\top \in Rel(\tau, \tau')$ by

$$(M, M') \in \mathrm{s}^\top \;\overset{def}{\Leftrightarrow}\; \forall (S, S') \in \mathrm{s}(S \top M \Leftrightarrow S' \top M')$$

$$\frac{S = S' \circ (- A) \qquad S' \top M[A/x]}{S \top \lambda x : \tau(M)} \qquad \frac{S \circ (- A) \top F}{S \top F\, A}$$

$$\frac{S = S' \circ (- \tau) \qquad S' \top M[\tau/\alpha]}{S \top \Lambda\alpha(M)} \qquad \frac{S \circ (- \tau) \top G}{S \top G\, \tau}$$

$$\frac{S \circ (- \mathbf{fix}(F)) \top F}{S \top \mathbf{fix}(F)}$$

$$\frac{S = \mathrm{Id}}{S \top \mathbf{nil}_\tau}$$

$$\frac{S = S' \circ (\mathbf{case} - \mathbf{of}\, \{\mathbf{nil} \Rightarrow M_1 \mid h :: t \Rightarrow M_2\})}{S' \top M_1}{S \top \mathbf{nil}_\tau}$$

$$\frac{S = S' \circ (\mathbf{case} - \mathbf{of}\, \{\mathbf{nil} \Rightarrow M_1 \mid h :: t \Rightarrow M_2\})}{S' \top M_2[H/h, T/t]\})}{S \top H :: T}$$

$$\frac{S \circ (\mathbf{case} - \mathbf{of}\, \{\mathbf{nil} \Rightarrow M_1 \mid h :: t \Rightarrow M_2\}) \top M}{S \top \mathbf{case}\, M\, \mathbf{of}\, \{\mathbf{nil} \Rightarrow M_1 \mid h :: t \Rightarrow M_2\}}$$

$$\frac{S = S' \circ (\mathbf{out}_T -) \qquad S' \top M}{S \top \mathbf{in}_T M} \qquad \frac{S \circ (\mathbf{out}_T -) \top M}{S \top \mathbf{out}_T M}$$

Figure 5.9: Structural termination relation [Pitts' Figure 7]

Definition 35. (⊤⊤-*Closed term-relations*). *[Pitts' Definition 3.10] A term-relation* r *is* ⊤⊤-*closed if* $r = r^{\top\top}$, *or equivalently if* $r^{\top\top} \subseteq r$, *or equivalently if* $r = s^\top$ *for some stack-relation* s, *or equivalently if* $r = (r')^{\top\top}$ *for some term-relation* r'.

5.2.9   Action of type constructors on term relations

We are now ready to define the central logical relation Δ. Each type constructor in PolyPCF has an associated "action" on term relations. The combination of these actions defines a logical relation, parameterised by a tuple of term-relations.

Definition 36. *(Action of $\rightarrow$ on term-relations). [Pitts' Definition 4.1] Given $r_1 \in Rel(\tau_1, \tau_1')$ and $r_2 \in Rel(\tau_2, \tau_2')$, we define $r_1 \rightarrow r_2 \in \mathsf{Rel}(\tau_1 \rightarrow \tau_2, \tau_1' \rightarrow \tau_2')$ by:*

$$(\mathsf{F}, \mathsf{F}') \in r_1 \rightarrow r_2 \overset{def}{\Leftrightarrow} \forall (\mathsf{A}, \mathsf{A}') \in r_1((\mathsf{F}\,\mathsf{A}, \mathsf{F}'\,\mathsf{A}') \in r_2).$$

Definition 37. *(Action of $\forall$ on term-relations). [Pitts' Definition 4.2] Let τ_1 and τ_1' be PolyPCF types with at most a single free type variable, α say. Suppose R is a function mapping term-relations $r \in Rel(\tau_2, \tau_2')$ (any $\tau_2, \tau_2' \in Typ$) to term-relations $R(r) \in Rel(\tau_1[\tau_2/\alpha], \tau_1'[\tau_2'/\alpha])$. Then we can form a term-relation $\forall r(R(r)) \in Rel(\forall \alpha(\tau_1), \forall \alpha(\tau_1'))$ as follows:*

$$(\mathsf{G}, \mathsf{G}') \in \forall r(R(r)) \overset{def}{\Leftrightarrow} \forall \tau_2, \tau_2' \in Typ(\forall r \in Rel(\tau_2, \tau_2')((\mathsf{G}\,\tau_2, \mathsf{G}'\,\tau_2') \in R(r))).$$

Definition 38. *(Action of (list$-$) on term-relations). [Pitts' Definition 4.3] Given $\tau, \tau' \in Typ$, $r_1 \in Rel(\tau, \tau')$ and $r_2 \in Rel(\tau\ \mathtt{list}, \tau'\ \mathtt{list})$, define $1 + (r_1 \times r_2) \in Rel(\tau\ \mathtt{list}, \tau'\ \mathtt{list})$ by:*

$$1 + (r_1 \times r_2) \overset{def}{=} \{(\boldsymbol{nil}_\tau, \boldsymbol{nil}_{\tau'})\} \cup \{(\mathsf{H} :: \mathsf{T}, \mathsf{H}' :: \mathsf{T}') \mid (\mathsf{H}, \mathsf{H}') \in r_1 \,\&\, (\mathsf{T}, \mathsf{T}') \in r_2\}$$

Note that the subset relation makes $Rel(\tau\ \mathtt{list}, \tau'\ \mathtt{list})$ into a complete lattice and that, for each r_1 the function $r_2 \mapsto (1 + (r_1 \times r_2))^{\top\top}$ is monotone. Therefore we can form its greatest (post-)fixed point:

$$(r_1)\ \mathtt{list} \overset{def}{=} \nu r_2 (1 + (r_1 \times r_2))^{\top\top}.$$

Thus $(r_1)\ \mathtt{list}$ is the unique term-relation satisfying

$$(r_1)\ \mathtt{list} = (1 + (r_1 \times (r_1)\ \mathtt{list}))^{\top\top}$$

$$\forall r_2 (r_2 \subseteq (1 + (r_1 \times r_2))^{\top\top} \Rightarrow r_2 \subseteq (r_1)\ \mathtt{list}$$

Definition 39. *(Action of $T(-)$ on term relations). Given $r \in \mathsf{Rel}(\tau, \tau')$ define $\mathtt{T}(r) \in Rel(\mathtt{T}(\tau), \mathtt{T}(\tau'))$ by*

$$\mathtt{T}(r) \overset{def}{=} \{(\mathsf{M}, \mathsf{M}') \mid (\boldsymbol{out}_\mathsf{T}\,\mathsf{M}, \boldsymbol{out}_\mathsf{T}\,\mathsf{M}') \in r\}$$

Definition 40. *(The logical relation Δ). [Pitts' Figure 8] For each PolyPCF type τ and each list $\vec{\alpha} = \alpha_1, \ldots, \alpha_n$ of distinct type variables containing the free type variables of τ, we define a function from tuples of term-relations to term-relations*

$$r_1 \in Rel(\tau_1, \tau_1'), \ldots, r_n \in Rel(\tau_n, \tau_n') \mapsto \Delta_\tau(\vec{r}/\vec{\alpha}) \in Rel(\tau[\vec{\tau}/\vec{\alpha}], \tau[\vec{\tau}'/\vec{\alpha}']).$$

where Δ is defined as in Figure 5.10.

$$\Delta_{\alpha_i}(\vec{r}/\vec{\alpha}) \quad \overset{\text{def}}{=} \quad r_i$$

$$\Delta_{\tau \to \tau'}(\vec{r}/\vec{\alpha}) \quad \overset{\text{def}}{=} \quad \Delta_{\tau}(\vec{r}/\vec{\alpha}) \to \Delta_{\tau'}(\vec{r}/\vec{\alpha})$$

$$\Delta_{\forall \alpha(\tau)'}(\vec{r}/\vec{\alpha}) \quad \overset{\text{def}}{=} \quad \forall r(\Delta_{\tau}(r^{\top\top}/\alpha, \vec{r}/\vec{\alpha}))$$

$$\Delta_{\tau \, \texttt{list}}(\vec{r}/\vec{\alpha}) \quad \overset{\text{def}}{=} \quad (\Delta_{\tau}(\vec{r}/\vec{\alpha})) \, \texttt{list}$$

$$\Delta_{\texttt{T}(\tau)}(\vec{r}/\vec{\alpha}) \quad \overset{\text{def}}{=} \quad \texttt{T}(\Delta_{\tau}(\vec{r}/\vec{\alpha}))$$

Figure 5.10: Definition of the logical relation Δ [Pitts' Figure 8]

Definition 41. *(**Logical relation on open terms**). [Pitts' Definition 4.5] Suppose $\Gamma \vdash M : \tau$ and $\Gamma \vdash M' : \tau$ hold, with $\Gamma = \alpha_1, \ldots, \alpha_m, x_1 : \tau_1, \ldots, x_n : \tau_n$ say. Write*

$$\Gamma \vdash M \, \Delta \, M' : \tau \tag{5.1}$$

to mean: given any $\sigma_i, \sigma_i' \in \mathit{Typ}$ and $r_i \in \mathit{Rel}(\sigma_i, \sigma_i')$ (for $i = 1..m$) with each $r_i \, \top\top$-closed, then for any $(N_j, N_j') \in \Delta_{r_j}(\vec{r}/\vec{\alpha})$ (for $i = 1..n$) it is the case that

$$(M[\vec{\sigma}/\vec{\alpha}, \vec{N}\vec{x}], M'[\vec{\sigma}'/\vec{\alpha}, \vec{N}'\vec{x}]) \in \Delta_{\tau}(\vec{r}/\vec{\alpha})$$

5.2.10 Fundamental Property

Proposition 42. *(**'Fundamental Property' of the logical relation**). [Pitts' Proposition 4.6] The relation (5.1) between open PolyPCF terms is compatible and substitutive, in the sense of Definition 28.*

The proof of Proposition 42 depends on the following Lemma:

Lemma 43. *[Pitts' Lemma 4.11] For each open type τ, with free type variables in $\vec{\alpha}$ say, if the term-relations $\vec{r}$ are $\top\top$-closed, then so is the term-relation $\Delta_r(\vec{r}/\vec{\alpha})$ defined in Figure 5.10. In particular for each closed type τ, $\Delta_{\tau}() \in \mathit{Rel}(\tau, \tau)$ is $\top\top$-closed.*

Pitts' proof of this lemma proceeds by induction on the structure of types, showing that the action of each type constructor takes $\top\top$-closed relations to $\top\top$-closed relations. We have added a family of type constructors for tagged types $\texttt{T}(-)$ and therefore need the following lemma to extend this theorem to our augmented system.

Lemma 44 ($\texttt{T}(-)$ preserves $\top\top$-closure)**.** *Suppose $r \in \mathit{Rel}(\tau, \tau')$.*

(i) Suppose given values $\mathbf{in}_\mathsf{T}\, M$ and $\mathbf{in}_\mathsf{T}\, M'$ of types $\mathsf{T}(\tau)$ and $\mathsf{T}(\tau')$ respectively, satisfying $(M, M') \in r$.

If r is $\top\top$-closed then $(\mathbf{in}_\mathsf{T}\, M, \mathbf{in}_\mathsf{T}\, M') \in \mathsf{T}(r)$.

(ii) If $(S, S') \in r^\top$ then $(S \circ \mathbf{out}_\mathsf{T} -, S' \circ \mathbf{out}_\mathsf{T} -) \in \mathsf{T}(r)^\top$.

(iii) If r is $\top\top$-closed then so is $\mathsf{T}(r)$.

Proof.

(i) The statement in (i) follows from the equivalence

$$\mathbf{out}_\mathsf{T}\, \mathbf{in}_\mathsf{T}\, M \Downarrow V \iff M \Downarrow V$$

(ii) Suppose $(S, S') \in r^\top$. For any $(N, N') \in \mathsf{T}(r)$ we have

$$
\begin{aligned}
S \circ \mathbf{out}_\mathsf{T} - \top N \quad &\Leftrightarrow \quad S \top \mathbf{out}_\mathsf{T}\, N \\
&\qquad \text{(by definition of } (-) \top (-)) \\
&\Leftrightarrow \quad S' \top \mathbf{out}_\mathsf{T}\, N' \\
&\qquad \text{(since } (\mathbf{out}_\mathsf{T}\, N, \mathbf{out}_\mathsf{T}\, N') \in r \\
&\qquad\quad \text{and } (S, S') \in r^\top) \\
&\Leftrightarrow \quad S' \circ \mathbf{out}_\mathsf{T} - \top N' \\
&\qquad (\text{ by definition of } (-) \top (-))
\end{aligned}
$$

It follows that $(S \circ \mathbf{out}_\mathsf{T} -, S' \circ \mathbf{out}_\mathsf{T} -) \in \mathsf{T}(r)^\top$.

(iii) Suppose $(S, S') \in r^\top$, $(N, N') \in r$, $(M, M') \in \mathsf{T}(r)^{\top\top}$. We must show that $(M, M') \in \mathsf{T}(r)$.

By (ii) we have $(S \circ \mathbf{out}_\mathsf{T} -, S' \circ \mathbf{out}_\mathsf{T} -) \in \mathsf{T}(r)^\top$, and hence

$$S \circ \mathbf{out}_\mathsf{T} - \top M \quad \Leftrightarrow \quad S' \circ \mathbf{out}_\mathsf{T} - \top M'$$

Therefore

$$S \top \mathbf{out}_\mathsf{T}\, M \quad \Leftrightarrow \quad S' \top \mathbf{out}_\mathsf{T}\, M'$$

Thus, by the definition of $(-)^\top$, $(\mathbf{out}_\mathsf{T}\, M, \mathbf{out}_\mathsf{T}\, M') \in (r^\top)^\top = r$, and so $(M, M') \in \mathsf{T}(r)$ by the definition of $\mathsf{T}(-)$. It follows that $\mathsf{T}(r)$ is $\top\top$-closed.

$\square$

Theorem 45. *[Pitts' Theorem 4.15] The logical relation ([5.1](#)) coincides with PolyPCF observational congruence:*

$$\Gamma \vdash M \stackrel{obs}{=} M' : \tau \iff \Gamma \vdash M \Delta M' : \tau \tag{5.2}$$

Having established the necessary preliminaries to the equivalence proof, we now turn to the development of the conversion between the two styles of abstract type.

5.3 Tagging and untagging

5.3.1 Example

The conversion between signing-style and sealing-style definitions of abstract types is performed by a pair of type-indexed functions which insert and remove tags as necessary. For example, to convert the function `plus` in the signing-style implementation of the complex type given in Section [5.2.1](#) into a function that can be used in the sealing-style implementation we generate a function of type

$$(\alpha \to \alpha \to \alpha) \to (C(\alpha) \to C(\alpha) \to C(\alpha))$$

where α is the type variable denoting the abstract type of complex numbers and C is the tag used in the sealed representation. The conversion function generated for the type of `plus` in the sealed representation is

```
λh:α→α→α
  (λx:C(α)
    ((λj:α→α
      (λy:C(α)((λz:α(inC z))  (j  ((λz:C(α)(outC z))  y))))))
    (h  ((λz:C(α)(outC z))  x))))
```

which, after the "administrative" redexes introduced by the generation function are reduced, becomes

$$\lambda h : (\alpha \to \alpha \to \alpha)(\lambda x : C(\alpha)\,(\lambda y : C(\alpha)\,(\textbf{in}_C\,(h\,(\textbf{out}_C\,x)\,(\textbf{out}_C\,y)))))$$

Conversely, moving from the sealed to the signed representation requires a function of type

$$(C(\alpha) \to C(\alpha) \to C(\alpha)) \to (\alpha \to \alpha \to \alpha)$$

In this case the generated term, after administrative reductions, is

$$\lambda h : (C(\alpha) \to C(\alpha) \to C(\alpha))(\lambda x : \alpha\,(\lambda y : \alpha\,(\textbf{out}_C\,(h\,(\textbf{in}_C\,x)\,(\textbf{in}_C\,y)))))$$

5.3.2 Definition

We now give the definitions of the type-indexed functions that translate between tagged and untagged representations of abstract types.

For each type τ with a free variable α and no occurrences of the tag constructor T we will define functions $C^+_{T,\alpha}[\tau]$ and $C^-_{T,\alpha}[\tau]$ with types

$$C^+_{T,\alpha}[\tau] \;\; : \;\; \tau \to \tau[T(\alpha)/\alpha]$$
$$C^-_{T,\alpha}[\tau] \;\; : \;\; \tau[T(\alpha)/\alpha] \to \tau$$

and define operations $-^+_{T,\alpha}$ and $-^-_{T,\alpha}$ on types:

$$\tau^+_{T,\alpha} \;\; = \;\; \tau$$
$$\tau^-_{T,\alpha} \;\; = \;\; \tau[T(\alpha)/\alpha]$$

Let p range over $\{+, -\}$ and write $\bar{p}$ for the operation that flips the polarity, so that

$$\bar{+} \;\overset{\text{def}}{=}\; - \qquad\qquad \bar{-} \;\overset{\text{def}}{=}\; +$$

Then $C^+_{T,\alpha}[\tau]$ and $C^-_{T,\alpha}[\tau]$ are defined as follows:

$$C^p_{T,\alpha}[\forall\beta(\tau)] \;\; = \;\; \lambda x : \forall\beta(\tau)^p (\Lambda\beta(C^p_{T,\alpha}[\tau]\,(x\,\beta)))$$
$$C^p_{T,\alpha}[\tau_1 \to \tau_2] \;\; = \;\; \lambda h : (\tau_1 \to \tau_2)^p (\lambda x : \tau^{\bar{p}}_1 (C^p_{T,\alpha}[\tau_2]\,(h\,(C^{\bar{p}}_{T,\alpha}[\tau_1]\,x))))$$
$$C^p_{T,\alpha}[\tau\;\texttt{list}] \;\; = \;\; \texttt{mapList}\,(\tau^p)\,(\tau^{\bar{p}})\,(C^p_{T,\alpha}[\tau])$$
$$C^p_{T,\alpha}[T'(\tau)] \;\; = \;\; \lambda x : T'(\tau)^p (\textbf{in}_T\,(C^p_{T,\alpha}[\tau](\textbf{out}_T\,x))$$
$$C^p_{T,\alpha}[\beta] \;\; = \;\; \lambda x : \beta(x)$$
$$C^+_{T,\alpha}[\alpha] \;\; = \;\; \lambda x : \alpha(\textbf{in}_T\,x)$$
$$C^-_{T,\alpha}[\alpha] \;\; = \;\; \lambda x : T(\alpha)(\textbf{out}_T\,x)$$

where $\texttt{mapList} : \forall\alpha(\forall\beta((\alpha \to \beta) \to (\alpha\;\texttt{list} \to \beta\;\texttt{list})))$ is defined in the usual way:

```
Λα (Λβ (λf:α→β
   (fix (λm:α list→β list
        (λl:α list
         (case l of { nil ⇒ nil | h::t ⇒ f h :: m t)))))))
```

The implementation of $C^p_{T,\alpha}[-]$ at each type constructor is almost entirely determined by its type. An equivalent implementation may be obtained in the following manner. Let $\texttt{map}_\tau$ be the map of the bifunctor corresponding to τ, with type

$$\forall\beta(\forall\gamma((\beta \to \gamma) \to (\gamma \to \beta) \to \tau[\beta/\alpha] \to \tau[\gamma/\alpha]))$$

(In general, if τ has n free type variables, map_τ will take $2n$ functions, since each type variable may occur positively or negatively in τ). Let $\mathrm{in}_{T,\alpha}$ and $\mathrm{out}_{T,\alpha}$ be defined as follows.

$$
\mathrm{in}_{T,\alpha} \quad \overset{\mathrm{def}}{=} \quad \lambda x : \alpha(\mathbf{in}_T\, x)
$$
$$
\mathrm{out}_{T,\alpha} \quad \overset{\mathrm{def}}{=} \quad \lambda x : T(\alpha)(\mathbf{out}_T\, x)
$$

Then

$$
C^+_{T,\alpha}[\tau] \quad \overset{\mathrm{def}}{=} \quad \lambda x : \tau^+_{T,\alpha}(\mathrm{map}_\tau\, \alpha\, (T(\alpha))\, \mathrm{in}_{T,\alpha}\, \mathrm{out}_{T,\alpha})
$$
$$
C^-_{T,\alpha}[\tau] \quad \overset{\mathrm{def}}{=} \quad \lambda x : \tau^-_{T,\alpha}(\mathrm{map}_\tau\, (T(\alpha))\, \alpha\, \mathrm{out}_{T,\alpha}\, \mathrm{in}_{T,\alpha})
$$

Proposition 46. *The functions $C^-_{T,\alpha}[\tau]$ and $C^+_{T,\alpha}[\tau]$ are mutual inverses. That is, for all M of type τ and M' of type $T(\tau)$,*

$$
C^-_{T,\alpha}[\tau]\, (C^+_{T,\alpha}[\tau]\, M) \quad \overset{obs}{=} \quad M \tag{5.3}
$$
$$
C^+_{T,\alpha}[\tau]\, (C^-_{T,\alpha}[\tau]\, M') \quad \overset{obs}{=} \quad M' \tag{5.4}
$$

Proof. Either directly using the standard β- and η- equalities that follow from the evaluation rules, or indirectly via map fusion. $\square$

5.4 Signing and sealing

In Section 5.2.1 we introduced the constructs **sealtype** and **signtype** for creating abstract types. We now give formal definitions of each construct, both directly and via desugaring into the core language.

5.4.1 Syntax

$M ::= \ldots$
$| \ \mathbf{signtype}\ \alpha =\ \tau\quad \mathbf{with}\, x_1 : \tau_1 = M_1 \ldots x_n : \tau_n = M_n\ \mathbf{in}\, M$
$| \ \mathbf{sealtype}\ \alpha = T(\tau)\ \mathbf{with}\, x_1 : \tau_1 = M_1 \ldots x_n : \tau_n = M_n\ \mathbf{in}\, M$

As we noted in the introduction, the distinction between the two constructs is somewhat obscured in PolyPCF by the need to supply type signatures for each exported component in both the **signtype** and **sealtype** variants. In a language with type inference the signatures can be omitted in **sealtype** definitions.

5.4.2 Typing

The typing rules for **signtype** and **sealtype** are as follows.

$$\frac{\begin{array}{c} \Gamma \vdash M_i : \tau_i[\tau/\alpha] \quad (\forall i.1 \leqslant i \leqslant n) \\ \Gamma, \alpha, x_1 : \tau_1, \ldots, x_n : \tau_n \vdash M : \tau' \qquad \alpha \notin \mathrm{fv}(\tau') \end{array}}{\Gamma \vdash \textbf{signtype}\ \alpha = \tau\ \textbf{with}\ x_1 : \tau_1 = M_1 \ldots x_n : \tau_n : M_n\ \textbf{in}\ M : \tau'}$$

$$\frac{\begin{array}{c} \Gamma \vdash M_i : \tau_i[T(\tau)/\alpha] \quad (\forall i.1 \leqslant i \leqslant n) \\ \Gamma, \alpha, x_1 : \tau_1, \ldots, x_n : \tau_n \vdash M : \tau' \qquad \alpha \notin \mathrm{fv}(\tau') \end{array}}{\Gamma \vdash \textbf{sealtype}\ \alpha = T(\tau)\ \textbf{with}\ x_1 : \tau_1 = M_1 \ldots x_n : \tau_n : M_n\ \textbf{in}\ M : \tau'}$$

We associate two types with each of the fields x_i of the interface of an abstract type defined with **signtype** or **sealtype**. The first is the type τ_i ascribed to that field in the signature; this is the type given to x_i within the continuation M. The second is the concrete type, in which every occurrence of the abstract type variable α has been replaced by the representation type: that is, with τ for **signtype** and with $T(\tau)$ for **sealtype**. This concrete type must match the type of the term M_i that is bound to the field.

The hypotheses for each rule further specify that each of the fields of the interface is in scope only within the continuation, M, and not within the terms bound to the other fields. This is perhaps too restrictive for practical use, but it is a restriction that is easily overcome with a further layer of sugar. We can obtain mutually-recursive definitions using an n-ary fixpoint combinator for any signature of n operations. For example, a combinator for creating two mutually recursive functions may be defined, rather impenetrably, as follows:

```
fix₂ : ∀α(∀β(∀γ(∀δ((((α→β × γ→δ) → α→β)
              × ((α→β × γ→δ) → γ→δ))
            → (α→β × γ→δ))))))
fix₂ = Λα(Λβ(Λγ(Λδ
        (fix
          (λfix₂:((((α→β × γ→δ) → α→β)
                × ((α→β × γ→δ) → γ→δ))
              → (α→β × γ→δ))
              → ((((α→β × γ→δ) → α→β)
                × ((α→β × γ→δ) → γ→δ))
              → (α→β × γ→δ))
          (λfs:((α→β × γ→δ) → α→β)
              × ((α→β × γ→δ) → γ→δ)
            (⟨λx:α((fst fs) (fix₂ fs) x),
             λx:γ((snd fs) (fix₂ fs) x)⟩)))))))))
```

5.4.3 Evaluation

The evaluation rules for **signtype** and **sealtype** are as follows.

$$\frac{M[\tau/\alpha, M_1/x_1, \ldots M_n/x_n] \Downarrow V}{\textbf{signtype}\,\alpha = \tau\,\textbf{with}\,x_1 : \tau_1 = M_1 \ldots x_n : \tau_n : M_n\,\textbf{in}\,M \Downarrow V}$$

$$\frac{M[T(\tau)/\alpha, M_1/x_1, \ldots M_n/x_n] \Downarrow V}{\textbf{sealtype}\,\alpha = T(\tau)\,\textbf{with}\,x_1 : \tau_1 = M_1 \ldots x_n : \tau_n : M_n\,\textbf{in}\,M \Downarrow V}$$

Evaluation is straightforward: the continuation M is evaluated after substituting the abstract type $T(\tau)$ or τ for the abstract type variable α and the fields $M_1 \ldots M_n$ for the corresponding variables $x_i \ldots x_n$. Like PolyPCF function application, evaluation of the **signtype** and **sealtype** constructs is call-by-name.

5.4.4 Desugaring

Both **signtype** and **sealtype** are derived forms; there is a straightforward translation into core PolyPCF terms, as shown by the following lemma.

Lemma 47 (Desugaring). *Let* I, J, K *and* L *be defined as follows for some non-negative integer* n, *terms* $M, M_1, \ldots M_n$ *and types* $\tau, \tau_1, \ldots \tau_n$.

$$
\begin{aligned}
I &\stackrel{def}{=} \textbf{\textit{signtype}}\,\alpha = \tau\,\textbf{\textit{with}}\,x_1 : \tau_1 = M_1 \ldots x_n : \tau_n : M_n\,\textbf{\textit{in}}\,M \\
J &\stackrel{def}{=} \textbf{\textit{sealtype}}\,\alpha = T(\tau)\,\textbf{\textit{with}}\,x_1 : \tau_1 = M_1 \ldots x_n : \tau_n : M_n\,\textbf{\textit{in}}\,M \\
K &\stackrel{def}{=} (\Lambda\alpha(\lambda x_1 : \tau_1(\ldots \lambda x_n : \tau_n(M))\ldots))\,(\tau)\,M_1\,\ldots\,M_n \\
L &\stackrel{def}{=} (\Lambda\alpha(\lambda x_1 : \tau_1(\ldots \lambda x_n : \tau_n(M))\ldots))\,(T(\tau))\,M_1\,\ldots\,M_n
\end{aligned}
$$

Then the following hold:

$$\Gamma \vdash I : \tau' \iff \Gamma \vdash K : \tau' \tag{5.5}$$

$$\Gamma \vdash J : \tau' \iff \Gamma \vdash L : \tau' \tag{5.6}$$

$$I \Downarrow V \iff K \Downarrow V \tag{5.7}$$

$$J \Downarrow V \iff L \Downarrow V \tag{5.8}$$

where $\alpha \notin fv(\tau')$.

Proof. The statements 5.5 and 5.6 follow directly from the definition of the typing relation $\vdash\ :\ $ in Section 5.4.2 and Figure 5.4.

The statements 5.7 and 5.8 follow directly from the definition of the evaluation relation $\Downarrow$ in Section 5.4.3 and Figure 5.5. $\qquad\square$

It is more conventional to present abstract types in terms of existential types (Pierce, 2002, Chapter 24) than universal types, as we have done in Lemma 47. It would certainly be possible to give the desugaring of **signtype** and **sealtype** in terms of existentials. For example, using the extension of PolyPCF to existential types given by Pitts (2000), we might show that the following terms are equivalent:

$$\textbf{signtype } \alpha = \tau \textbf{ with } x : \tau' = M' \textbf{ in } M$$

$$\textbf{open } (\textbf{pack } \tau, M' \textbf{ as } \exists\alpha(\tau')) \textbf{ as } \alpha, x \textbf{ in } M$$

The equivalence of this encoding to the desugaring of Lemma 47 follows immediately from equation 54 of Pitts (2000).

5.4.5 Equivalence

Proposition 48. *Abstract type definitions with* **signtype** *may be converted to observationally-equivalent* **sealtype** *definitions and conversely as shown below.*

$$\textit{signtype}\ \alpha = \tau\ \textit{with}\ x_1 : \tau_1 = M_1 \ldots x_n : \tau_n = M_n\ \textit{in}\ M$$
$$\overset{obs}{=\!=}$$
$$\textit{sealtype}\ \alpha = T(\tau)\ \textit{with}\ x_1 : \tau_1 = C_{T,\alpha}^+\ [\tau_1]\ M_1 \ldots x_n : \tau_n = C_{T,\alpha}^+\ [\tau_n]\ M_n\ \textit{in}\ M$$

$$\textit{sealtype}\ \alpha = T(\tau)\ \textit{with}\ x_1 : \tau_1 = M_1 \ldots x_n : \tau_n = M_n\ \textit{in}\ M$$
$$\overset{obs}{=\!=}$$
$$\textit{signtype}\ \alpha = \tau\ \textit{with}\ x_1 : \tau_1 = C_{T,\alpha}^-\ [\tau_1]\ M_1 \ldots x_n : \tau_n = C_{T,\alpha}^-\ [\tau_n]\ M_n\ \textit{in}\ M$$

Proof. Let
$$\mathbb{R}_T[\tau] = \{\langle M,\ T(M)\rangle\ |\ \ \vdash M : \tau\}^{\top\top}$$
and
$$\alpha \vdash M_i : \tau_i$$
for each $1 \leqslant i \leqslant n$.

Comparing the definitions of Δ, $C_{T,\alpha}^+$ and $C_{T,\alpha}^-$ reveals that

$$\langle M_i,\ C_{T,\alpha}^+[\tau_i]\,M_i\rangle \in \Delta_{\tau_i}(\mathbb{R}_T[\tau]/\alpha) \tag{5.9}$$

$$\langle C_{T,\alpha}^-[\tau_i]\,M_i,\ M_i\rangle \in \Delta_{\tau_i}(\mathbb{R}_T[\tau]/\alpha) \tag{5.10}$$

By Theorem 42 ("Fundamental Property") Δ is compatible, and hence reflexive. We can instantiate the relation (5.1) to give

$$\alpha, x_1 : \tau_1, \ldots x_n : \tau_n \vdash M\ \Delta\ M : \tau'$$

From (5.9) and (5.10) it follows that

$$\langle M[\tau/\alpha, \vec{M_i}/\vec{x_i}],\ M[T(\tau)/\alpha, C_{T,\alpha}^+[\vec{\tau_i}]\,M_i/\vec{x_i}]\rangle \in \Delta_{\tau'}(\mathbb{R}_T[\tau]/\alpha)$$

$$\langle M[\tau/\alpha, C_{T,\alpha}^-[\vec{\tau_i}]\,M_i/\vec{x_i}],\ M[T(\tau)/\alpha, \vec{M_i}/\vec{x_i}]\rangle \in \Delta_{\tau'}(\mathbb{R}_T[\tau]/\alpha)$$

and so (since $\alpha \notin fv(\tau')$),

$$\langle M[\tau/\alpha, \vec{M_i}/\vec{x_i}],\ M[T(\tau)/\alpha, C_{T,\alpha}^+[\vec{\tau_i}]\,M_i/\vec{x_i}]\rangle \in \Delta_{\tau'}()$$

$$\langle M[\tau/\alpha, C_{T,\alpha}^-[\vec{\tau_i}]\,M_i/\vec{x_i}],\ M[T(\tau)/\alpha, \vec{M_i}/\vec{x_i}]\rangle \in \Delta_{\tau'}()$$

and hence (by Theorem 45),

$$M[\tau/\alpha, \vec{M_i}/\vec{x_i}] \stackrel{\text{obs}}{=} M[T(\tau)/\alpha, C_{T,\alpha}^+[\vec{\tau_1}]\,M_i/\vec{x_i}]$$

$$M[T(\tau)/\alpha, \vec{M_i}/\vec{x_i}] \stackrel{\text{obs}}{=} M[\tau/\alpha, C_{T,\alpha}^-[\vec{\tau_i}]\,M_i/\vec{x_i}]$$

and hence (by the evaluation rules for **sealtype** and **signtype**),

$$\mathbf{signtype}\ \alpha = \tau\ \mathbf{with}\ x_1 : \tau_1 = M_1 \ldots x_n : \tau_n = M_n\ \mathbf{in}\ M$$
$$\stackrel{\text{obs}}{=}$$
$$\mathbf{sealtype}\ \alpha = T(\tau)\ \mathbf{with}\ x_1 : \tau_1 = C_{T,\alpha}^+\,[\tau_1]\,M_1 \ldots x_n : \tau_n = C_{T,\alpha}^+\,[\tau_n]\,M_n\ \mathbf{in}\ M$$

$$\mathbf{sealtype}\ \alpha = T(\tau)\ \mathbf{with}\ x_1 : \tau_1 = M_1 \ldots x_n : \tau_n = M_n\ \mathbf{in}\ M$$
$$\stackrel{\text{obs}}{=}$$
$$\mathbf{signtype}\ \alpha = \tau\ \mathbf{with}\ x_1 : \tau_1 = C_{T,\alpha}^-\,[\tau_1]\,M_1 \ldots x_n : \tau_n = C_{T,\alpha}^-\,[\tau_n]\,M_n\ \mathbf{in}\ M$$

$\square$

Syntax

d	::=		declarations
		$p = e$ **where** $d_1; \ldots; d_n$	value
		data $T\,\alpha_1 \ldots \alpha_n = c_1 \mid \ldots \mid c_n$	datatype
		newtype $T\,\alpha_1 \ldots \alpha_n = C\,\{unC :: \tau\}$	newtype
		type $T\,\alpha_1 \ldots \alpha_n = \tau$	type synonym
		$x :: \tau$	signature
τ	::=		types
		α	type variable
		T	type constructor
		$\tau_1\,\tau_2$	type application
c	::=	$C\,\tau_1 \ldots \tau_n$	constructor spec.
e	::=		expressions
		x	variable
		C	constructor
		$e_1\,e_2$	application
		$\lambda p_1 \ldots p_n \to e$	abstraction
		case e **of** $p_1 \to e_1 \ldots p_n \to e_n$	case match
p	::=		patterns
		x	variable
		$C\,p_1 \ldots p_n$	constructor

x, x_i	variables	α, α_i	type variables
C, C_i	constructors	T, T_i	type constructors

Figure 5.11: (Subset of) Haskell syntax

5.5   Signed types in Haskell

5.5.1   Introduction

Having investigated the equivalence of the signing and sealing styles of defining abstract types, we will now develop an application of the result to Haskell.

A number of Haskell's features conspire to make it a suitable showcase for an implementation of signing in terms of sealing: it has the obvious prerequisite of sealing-style abstract types, and the Template Haskell extension (Sheard and Peyton Jones, 2002) provides the required metaprogramming facilities. Pureness and laziness provide additional benefits. In a

pure language there is no need for an SML-style "value restriction", so we can insert applications of transformation functions within bindings without losing polymorphism. In a lazy language the performance penalties of applying these transformation functions are amortized. Finally, mutable types (such as SML's `ref`) appear to pose an insurmountable problem for the technique described here in that they do not admit the `map` function on which our technique fundamentally depends: we cannot use a function of type $s \rightarrow t$ to construct an `t ref` from an `s ref` in a way that preserves equality.

To create an abstract type in Haskell the programmer defines a datatype in a module which does not export the data constructors. Haskell provides a special form of datatype definition, introduced with the **newtype** keyword, for creating type isomorphisms with a single, unary constructor. Unlike a constructor introduced with **data**, which forces evaluation when used in a pattern match, a **newtype** constructor has no effect on evaluation: its sole effect is to change the type of a value. We can use **newtype** within a module to define the abstract type of complex numbers as follows.

```haskell
module Complex (Complex, make, real, imag, conj, plus)
where
    newtype Complex = MkComplex (Float, Float)

    make (x,y) = MkComplex (x,y)

    real (MkComplex (x,y)) = x

    imag (MkComplex (x,y)) = y

    conj c = MkComplex (real c, -imag c)

    plus (MkComplex (u,v)) (MkComplex (x,y)) =
        MkComplex (u+x,v+y)
```

The `Complex` type is abstract because the constructor `MkComplex` is not included in the list of exported identifiers on the first line. All creation and inspection of `Complex` values outside this module must take place through the five functions in this interface.

The designers of Haskell, as we saw on page 157, chose to provide this style of definition rather than the signing style because of concerns about how types defined with signing would interact with type classes — in particular, about potential ambiguity between type-class instances given for the representation and abstract types (Hudak et al., 2007).

In this section we describe a Haskell extension written using Template Haskell that translates abstract type definitions written in the signing style into definitions in the sealing style, demonstrating the feasibility of adding the signing style of abstract type definition to Haskell.

5.5.2 Example: complex numbers

We begin with a familiar example. The abstract type of complex numbers may be written as
follows using the Template Haskell extension:

```
$ (signed
     [d| type Complex = (Float, Float)

         make :: (Float, Float) → Complex
         make (x,y) = (x,y)

         real :: Complex → Float
         real (x,y) = x

         imag :: Complex → Float
         imag (x,y) = y

         conj :: Complex → Complex
         conj c = (real c, −imag c)

         plus :: Complex → Complex → Complex
         plus (u,v) (x,y) = (u+x,v+y) |])
```

The Template Haskell quote operation `[d| ... |]` and unquote operation `$ ( ... )` convert
between actual code and the abstract syntax trees used to represent it. (Template Haskell de-
fines a number of quote operations corresponding to syntax classes in the Haskell grammar; the
`[d| ... |]` indicates that we are quoting a declaration.) The meta-level function *signed* is
the interface to our library: it maps an abstract type definition in the signing style to an equiv-
alent definition in the sealing style. (We use an italic font to distinguish meta-level identifiers
like *signed* from object-level identifiers like `Complex`, which are set in typewriter face.) The
set of declarations passed to *signed* should include a type declaration and a number of function
and value bindings, each with a type signature. The type signatures are mandatory, since they
indicate at which points the representation type should be made abstract; that is, at which points
the generated code should wrap or unwrap values in a constructor. It would be convenient to
give type class instances for `Complex` within the *signed* block, but a technical difficulty pre-
vents this: type class instances cannot be defined for type aliases, and so Template Haskell
rejects instances for `Complex` before the quoted declarations are passed to *signed* for transfor-
mation. However, it is possible to define instances outside the *signed* block using functions in
the interface. For example, we can write

```
instance Num Complex where
    (+) = plus
```

and so on.

The *signed* function produces the following output for the signing-style definition of complex numbers:

```
newtype Complex = In { out :: (Float, Float) }

(make, real, imag, conj, plus)
  = (make', real', imag', conj', plus')
```

$$\text{map}_{\text{Float}} = \text{id}$$
$$\text{map}_{\text{Complex}} = \text{id}$$
$$\text{map}_{(,)} = \lambda f\ g\ (x,\ y) \rightarrow (f\ x,\ g\ y)$$
$$\text{map}_{\rightarrow}\ f\ g\ h = g \cdot h \cdot f$$

$$(\text{in}_{\text{Complex}},\ \text{out}_{\text{Complex}}) = (\lambda x \rightarrow \text{map}_{\text{Complex}}\ (\text{In}\ x),$$
$$\lambda x \rightarrow \text{out}\ (\text{map}_{\text{Complex}}\ x))$$

$$(\text{make}',\ \text{real}',\ \text{imag}',\ \text{conj}',\ \text{plus}')$$
$$= (\text{map}_{\rightarrow}\ (\text{map}_{(,)}\ \text{map}_{\text{Float}}\ \text{map}_{\text{Float}})\ \text{in}_{\text{Complex}}\ \text{make},$$
$$\text{map}_{\rightarrow}\ \text{out}_{\text{Complex}}\ \text{map}_{\text{Float}}\ \text{real},$$
$$\text{map}_{\rightarrow}\ \text{out}_{\text{Complex}}\ \text{map}_{\text{Float}}\ \text{imag},$$
$$\text{map}_{\rightarrow}\ \text{out}_{\text{Complex}}\ \text{in}_{\text{Complex}}\ \text{conj},$$
$$\text{map}_{\rightarrow}\ \text{out}_{\text{Complex}}\ (\text{map}_{\rightarrow}\ \text{out}_{\text{Complex}}\ \text{in}_{\text{Complex}})\ \text{plus})$$

```
  where
    make (x, y) = (x, y)
    real (x, y) = x
    imag (x, y) = y
    conj c = (real c, −imag c)
    plus (u, v) (x, y) = (u + x, v + y)
```

(Our actual implementation takes care to avoid introducing visible bindings. For example, the names here written conj′, imag′, etc., are, in the output of the implementation, private names, drawn from a vocabulary which is not available for use in Haskell source. In particular, the constructor and destructor In and out that witness the type isomorphism are private; thus there is no way to create or examine values of type Complex except via the five functions in the interface.)

Although the generated code appears somewhat heavyweight, straightforward equational reasoning reveals that the generated definition is in fact equivalent to the hand-written sealing-style definition at the beginning of this section. It is no surprise to discover that the Glasgow Haskell Compiler (GHC) generates absolutely equivalent object code for the two definitions when basic optimisation is enabled.

Section 5.5.4 describes the translation scheme in the general case. We confine our remarks

here to a couple of key points.

The most obvious distinction between the code generated by the Template Haskell extension and the PolyPCF code generated by the translation between tagged and untagged representations of abstract types (Section 5.3.2) is the large number of calls to map_T functions in the former, which do not appear at all in the latter. The explanation is simple: Haskell allows the definition of new type constructors using the **type**, **data** and **newtype** constructs, whereas type constructors in PolyPCF are limited to a predefined set, namely `list`, $\rightarrow$, $T(-)$ and $\forall$. Each map_T function is used to traverse structures of type $T\ \tau_1 \ldots \tau_n$ in order to add tags to any values of type `Complex` located within. In the PolyPCF translation of Section 5.3.2 these traversals are simply written out inline for each type constructor.

The type alias (defined with **type**) used in the input declaration is replaced in the output with a **newtype** definition. The $\text{in}_{\text{Complex}}$ and $\text{out}_{\text{Complex}}$ functions are used to add and remove the constructor for the `Complex` type at places where the type signatures in the input definition indicate that the type should be abstract. They serve two functions: $\text{in}_{\text{Complex}}$ both adds a constructor to a `Complex` value and traverses the value in order to add constructors to any `Complex` values found within. (In this case the traversal is trivial, since there can never be a `Complex` within a `Complex`; for parameterised abstract types the traversal is non-trivial, as shown in Section 5.5.4.3.) The primed variables are used to distinguish the transformed functions such as `real'` which act upon the abstract type (`Complex`) from the original functions such as `real` which act upon its representation ((`Float, Float`)). Thus, the reference to `real` within the definition of `conj` refers to the original, not the transformed, version of the function. (In Haskell, unlike in PolyPCF, bindings may generally be mutually recursive. We take advantage of this facility here, using the `real` and `imag` functions to implement `conj`.)

5.5.3 Example: a sudoku solver

The `Complex` example illustrates the connection between our Template Haskell library and the ML and PolyPCF programs in Sections 5.1–5.4. We now give a further example that highlights some distinctive properties of the Template Haskell implementation and shows its applicability to an existing program.

Bird (2006) presents a Haskell program for solving Sudoku puzzles. The development begin by defining types to represent boards. A Sudoku board is a square matrix of characters, which is represented as a list of lists:

```
type Matrix α = [[α]]
type Board    = Matrix Char
```

The representation is particularly convenient because Haskell provides a large library of list functions, besides special syntax for creating and dissecting list values. Thus, for instance, the `choices` function which replaces blank entries in a Sudoku board with all possible choices for that entry has a characteristically concise implementation:

```
choices :: Board -> Matrix [Char]
choices = map (map choose)
```

Many of the other functions involved in the program are similarly elegant.

Unfortunately, when we come to print the result of the program we discover a minor inconvenience. Since a `Board` is encoded as a list of lists, the standard Haskell functions for printing, which are based on the type class `Show`, display a board using the standard list formatting. We would like to see this:

```
Sudoku> print board
2 . . . . 1 . 3 8
. . . . . . . . 5
. 7 . . . 6 . . .
. . . . . . . 1 3
. 9 8 1 . . 2 5 7
3 1 . . . . 8 . .
9 . . 8 . . . 2 .
. 5 . . 6 9 7 8 4
4 . . 2 5 . . . .
```

but instead we see this:

```
Sudoku> print board
["2....1.38","........5",".7...6...",".......13",".981\
..257","31....8..","9..8...2.",".5..69784","4..25...."]
```

We can, naturally, write an additional function

```
showBoard :: Board → String
```

for printing boards in the format of our choosing. However, there is no way to connect `showBoard` with the `Show` class: only one instance of `Show` for lists is allowed. Consequently, if we wish to print a list of boards (which is quite probable, since that is the type of the result of the Sudoku solver), we must first apply our `showBoard` function to every element in the list:

```
Sudoku> print (map showBoard boards)
```

If there were a way to give an instance of the `Show` class for boards then printing a list of boards would be no harder than printing a single board:

```
Sudoku> print boards
```

Of course, there is a way to give distinct `Show` instances for boards and lists: we can make

Board a distinct type (using **newtype**) rather than an alias (using **type**). Unfortunately, while this solves the printing problem, it makes the implementation of the solver much less elegant. Suppose that the type of boards is defined as follows:

```
newtype Board = MkBoard { unBoard :: Matrix Char }
```

We must now add code to many of the functions that implement the Sudoku solver to add and remove the `MkBoard` constructor, resulting in an undesirable increase in the size of the code. For example, the `choices` function becomes

```
choices :: Board -> Matrix [Char]
choices = map (map choose) · unBoard
```

The idea of the extension described in this section is to allow us the advantages of both approaches: we can define boards directly in terms of lists, enabling an elegant implementation of the solver, while making the type `Board` distinct from the list type, allowing us to customise the behaviour of overloaded functions such as `show`. A further benefit of moving from a type alias to an abstract type is the potential to enforce constraints, such as board squareness, that are not captured by the representation type.

5.5.4   The translation

We now turn to the specification of the translation.

Translation of the definition of an abstract type constructor `T` from the signing style to the sealing style involves four meta-level functions, which we describe in Sections 5.5.4.1–5.5.4.4. The first function, $tynames$, (Section 5.5.4.1) determines the set of type constructors involved in the representation type and operations of `T`. The second function, map, (Section 5.5.4.2) uses the definition of each type constructor to generate a corresponding map function. The third function, tag, (Section 5.5.4.3) creates functions that convert between values of the abstract type `T` and values of the corresponding representation type. The final function, tr, (Section 5.5.4.4) generates functions that convert each operation in the abstract type definition from the signing style to the sealing style. The meta-level function $signed$ (Section 5.5.4.5) which translates from signing-style to a sealing-style definition, is defined in terms of these four functions.

5.5.4.1   Finding type constructors ($tynames$)

Both the representation of an abstract type and the signatures of the operations on the type are given in terms of type constructors which are defined elsewhere in the program or provided as standard. For example, the representation of the `Complex` type is built from the type con-

structor for pairs, written $(,)$, and the nullary type constructor `Float`. The operations on the `Complex` type also use the function type constructor, $\rightarrow$. Inserting and removing tags involves traversing values whose shape is specified by the definitions of these type constructors; in order to traverse these values we must therefore generate a map_T function for each type constructor T. The definition of map_T depends on the definition of T, to which we therefore require access; retrieving the definition of a type constructor (or other identifier) is the purpose of the *reify* function supplied with Template Haskell. Building on *reify*, we can retrieve the set of all type constructors used in a type expression, whether they appear in the expression itself or in the definition of some other type constructor which is used by the expression. We begin by giving three Template Haskell functions — $tynames_D$, $tynames_C$ and $tynames_E$ — that compute the set of type constructors used, respectively, in a type definition, a data constructor specification, and a type expression.

The $tynames_D$ function takes two arguments: a type declaration, written between brackets, and a set r of the type constructors already seen. For **data** and **newtype** declarations we must collect both the type constructor T on the left hand side and any type constructors used in the constructor specifications. We add T to r when examining constructor specifications, since these may use T if the type is recursive.

$$tynames_D \; [\textbf{data} \; T \; \alpha_1 \ldots \alpha_n = c_1 \mid \ldots \mid c_n] \; r$$
$$= \{T\} \cup tynames_C \; [c_1] \; (r \cup \{T\}) \cup \ldots \cup tynames_C \; [c_n] \; (r \cup \{T\})$$

$$tynames_D \; [\textbf{newtype} \; T \; \alpha_1 \ldots \alpha_n = C \, \{x :: \tau\}] \; r$$
$$= \{T\} \cup tynames_E \; [\tau] \; (r \cup \{T\})$$

A **type** declaration cannot be recursive, so we pass through r unchanged. The result consists of the type constructor on the left hand side of the declaration and any type constructors used on the right hand side.

$$tynames_D \; [\textbf{type} \; T \; \alpha_1 \ldots \alpha_n = \tau] \; r = \{T\} \cup tynames_E \; [\tau] \; r$$

Finding the type constructors used in a constructor specification is simply a matter of examining the type expressions which specify the domain of the constructor.

$$tynames_C \; [C \; \tau_1 \ldots \tau_n] \; r \quad = \quad tynames_E \; [\tau_1] \; r \ldots tynames_E \; [\tau_n] \; r$$

The $tynames_E$ function finds the type constructors used, directly or indirectly, in a type expression. A type expression consisting of a type variable does not use any type constructors:

$$tynames_E \; [\alpha] \; r \quad = \quad \{\}$$

For a type expression consisting of a type constructor T the result depends on whether T is found in r. If it is then we are in the process of examining the definition of T already, so there

is nothing more to do. If not, then the set of type constructors used consists of both T and all of the type constructors used in the definition of T, which we retrieve using *reify*.

$$
\begin{aligned}
tynames_{\mathsf{E}}\ [\mathsf{T}]\ r\quad &=\quad \{\}\qquad\qquad\qquad\qquad\qquad \textit{if}\ \mathsf{T}\in r\\
&\qquad\{\mathsf{T}\}\cup tynames_{\mathsf{D}}\ [\textit{reify}\ \mathsf{T}]\ r\quad \textit{otherwise}
\end{aligned}
$$

For a type application we simply examine the sub-expressions to determine the set of type constructors involved.

$$
tynames_{\mathsf{E}}\ [\tau_1\ \tau_2]\ r\quad =\quad tynames_{\mathsf{E}}\ \tau_1\ r\cup tynames_{\mathsf{E}}\ \tau_2\ r
$$

5.5.4.2   Generating map functions (map)

The next step is to generate functions $\mathsf{map}_{\mathsf{T}}$ for each type constructor T used in the definition of the abstract type. These perform the bulk of the work in translating from the signing style to the sealing style. The type of each $\mathsf{map}_{\mathsf{T}}$ function depends the way that the type parameters to T are used in its definition. A type constructor T with n parameters, all of which are used both positively and negatively within the definition of T, results in a function $\mathsf{map}_{\mathsf{T}}$ of the type

$$
(\alpha_1\to\beta_1,\beta_1\to\alpha_1)\ \to\ldots\to\ (\alpha_n\to\beta_n,\beta_n\to\alpha_n)\ \to\ \mathsf{T}\,\alpha_1\ldots\alpha_n\to\mathsf{T}\,\beta_1\ldots\beta_n
$$

— that is, with a pair of functions for each parameter of T, to transform positive and negative occurrences of the parameter. If a type parameter is only used positively then the second component of the pair is omitted in the corresponding parameter to the generated $\mathsf{map}_{\mathsf{T}}$ function. For type parameters that are only used negatively, the first component is omitted. For "phantom" type parameters, not used at all in the definition of T, there is no corresponding function parameter.

There are four type constructors involved in the definition of `Complex`: `Float`, `Complex`, `(,)` and $\to$. The first two of these, `Float` and `Complex`, are nullary, and so the associated `map` functions are equivalent to the identity function. The map function for the pair type, $\mathsf{map}_{(,)}$, has type:

$$
(\alpha\to\gamma)\to(\beta\to\delta)\to(\alpha,\beta)\to(\gamma,\delta)
$$

In this case the parameters to $\mathsf{map}_{(,)}$ are functions for transforming positive occurrences of the corresponding type variables, since the type parameters to the pair type only occur positively in its definition. We must distinguish between positive and negative occurrences, since at positive occurrences of the type constructor `Complex` we will insert calls to `In` to convert from the representation to the abstract type, and at negative occurrences we will insert calls to `out`. The $\mathsf{map}_{\to}$ function has two function parameters, reflecting the two type parameters to the function type constructor $\to$, the first (denoting the argument type on the function) negative and the

second (denoting the return type) positive. Consequently, the generated map function has the following type:

$$(\gamma \to \alpha) \to (\beta \to \delta) \to (\alpha \to \beta) \to (\gamma \to \delta)$$

We use an auxiliary function, $param$, to construct parameters for generated functions, writing

$param(\alpha_i, \mathtt{T}) \quad = \quad (\mathtt{p}_i, \mathtt{n}_i)$

   if α_i occurs positively and negatively within T's definition

$param(\alpha_i, \mathtt{T}) \quad = \quad \mathtt{p}_i$

   if α_i occurs only positively within T's definition

$param(\alpha_i, \mathtt{T}) \quad = \quad \mathtt{n}_i$

   if α_i occurs only negatively within T's definition

$param(\alpha_i, \mathtt{T}) \quad = \quad$ *(i.e. nothing)*

   if α_i does not occur within T's definition

Each parameter $\mathtt{p}_i$ of the generated function corresponds to a function which transforms the positive occurrences of the $\mathtt{i}^{\mathrm{th}}$ type parameter, α_i; each $\mathtt{n}_i$ acts similarly on negative occurrences. If there are no positive occurrences of α_i then $\mathtt{p}_i$ is omitted; similarly, $\mathtt{n}_i$ is omitted if α_i does not occur negatively.

We now turn to the definitions of the Template Haskell functions which generate the `map` functions. Once again we give three Template Haskell functions, treating type definitions, type expressions, and data constructor specifications. The map_{D} function generates a `map` function for a type definition. There are three cases, one for each type-constructor-introducing keyword.

For a **data** declaration we generate a case match over the constructors of the declared datatype, delegating generation of the matches to the map_{C} function. The substitution $[\alpha_i \mapsto (\mathtt{p}_i, \mathtt{n}_i)]$ passed to map_{C} records the correspondence between type variables and parameters.

$map_{\mathrm{D}}[\mathbf{data}\ \mathtt{T}\ \alpha_1 \ldots \alpha_n = \mathtt{c}_1 \mid \ldots \mid \mathtt{c}_n]$

$= \lambda\ param(\alpha_1, \mathtt{T}) \ldots param(\alpha_n, \mathtt{T})\ \mathtt{x} \to$

   case $\mathtt{x}$ **of**

   $map_{\mathrm{C}}[\mathtt{c}_1][\alpha_i \mapsto (\mathtt{p}_i, \mathtt{n}_i)]$

   $\ldots$

   $map_{\mathrm{C}}[\mathtt{c}_n][\alpha_i \mapsto (\mathtt{p}_i, \mathtt{n}_i)]$

The treatment of **newtype** is similar, save that there is only one constructor to consider.

$map_{\mathrm{D}}[\mathbf{newtype}\ \mathtt{T}\ \alpha_1 \ldots \alpha_n = \mathtt{C}\ \{\mathtt{unC} :: \tau\}]$

$= \lambda\ param(\alpha_1, \mathtt{T}) \ldots param(\alpha_n, \mathtt{T})\ \mathtt{x} \to$

   case $\mathtt{x}$ **of**

   $map_{\mathrm{C}}[\mathtt{C}\ \tau][\alpha_i \mapsto (\mathtt{p}_i, \mathtt{n}_i)]$

For alias declarations, declared with **type**, the map_E function does all of the work.

$$map_D [\textbf{type } T \; \alpha_1 \ldots \alpha_n = \tau]$$
$$= \lambda \; param(\alpha_1, T) \ldots param(\alpha_n, T) \; \texttt{x} \rightarrow$$
$$map_E^+ [\tau_n][\alpha_i \mapsto (\texttt{p}_i, \texttt{n}_i)] \; \texttt{x}$$

The map_C function treats constructor specifications, generating a case match for a data constructor. The $\texttt{x}_i$ variables bind the constructor parameters, and we use map_E to transform each parameter, finally applying the constructor C.

$$map_C \; [\texttt{C} \; \tau_1 \ldots \tau_n] \; \texttt{r} \; = \; \texttt{C} \; \texttt{x}_1 \ldots \texttt{x}_n \rightarrow \texttt{C} \; (map_E^+ \; [\tau_1] \; \texttt{r} \; \texttt{x}_1) \ldots (map_E^+ \; [\tau_n] \; \texttt{r} \; \texttt{x}_n)$$

The map_E function generates, for a type τ with free variables $\alpha_1 \ldots \alpha_n$, a term t with free variables $\texttt{p}_1 \ldots \texttt{p}_n, \texttt{n}_1 \ldots \texttt{n}_n$. If each of the $\texttt{p}_i$ has type $\alpha_i \rightarrow \beta_i$ and each of the $\texttt{n}_i$ has type $\beta_i \rightarrow \alpha_i$ then t will have type $\tau \rightarrow \tau[\vec{\beta_i}/\vec{\alpha_i}]$. Besides the substitution parameter r and the parameter between brackets which denotes a type expression there is a further parameter to map_C representing the "polarity" of the expression, i.e. whether type variables occurring in the expression represent input or output parameters. At a type variable α_i we project out the first or second component of the corresponding pair of functions $(\texttt{p}_i, \texttt{n}_i)$, depending on the polarity.

$$map_E^+ \; [\alpha_i] \; \texttt{r} = \texttt{p}_i \qquad\qquad \textit{where } \texttt{r}(\alpha_i) = (\texttt{p}_i, \texttt{n}_i)$$
$$map_E^- \; [\alpha_i] \; \texttt{r} = \texttt{n}_i \qquad\qquad \textit{where } \texttt{r}(\alpha_i) = (\texttt{p}_i, \texttt{n}_i)$$

At a type constructor T we generate a (possibly recursive) call to the map function for T.

$$map_E^p \; [T] \; \texttt{r} = \texttt{map}_T$$

At type applications $\tau_1 \; \tau_2$ we generate function applications. The function component is generated from τ_1 by map_E. The argument that is generated depends on the type of the generated function expression (which, in turn, depends on the variance of the type constructor that is involved). We write $\bar{p}$ to reverse the polarity of p. Then we generate the following application expressions from type applications:

$$map_E^p \; [\tau_1 \; \tau_2] \; \texttt{r} \quad = \quad map_E^p \; [\tau_1] \; \texttt{r} \; (map_E^p \; [\tau_2] \; \texttt{r}, map_E^{\bar{p}} \; [\tau_2] \; \texttt{r})$$

    if τ_2 is the argument for a type parameter that is used both positively and negatively.

$$map_E^p \; [\tau_1 \; \tau_2] \; \texttt{r} \quad = \quad map_E^p \; [\tau_1] \; \texttt{r} \; (map_E^p \; [\tau_2] \; \texttt{r})$$

    if τ_2 is the argument for a type parameter that is used only positively.

$$map_E^p \; [\tau_1 \; \tau_2] \; \texttt{r} \quad = \quad map_E^p \; [\tau_1] \; \texttt{r} \; (map_E^{\bar{p}} \; [\tau_2] \; \texttt{r})$$

    if τ_2 is the argument for a type parameter that is used only negatively.

$$map_E^p \; [\tau_1 \; \tau_2] \; \texttt{r} \quad = \quad map_E^p \; [\tau_1] \; \texttt{r}$$

    if τ_2 is the argument for a phantom type parameter.

5.5.4.3 Generating tagging and untagging functions (tag)

The operations of adding and removing constructors form the core of the translation between the signing and the sealing style of abstract type definition. The tag function generates object-level functions in_T and out_T for an abstract type constructor T. The in_T function performs the dual functions of constructing a value v of type $T \tau_1 \ldots \tau_n$ from a value of the corresponding representation type and traversing v using the map function generated for T. The out_T function performs the inverse operation, converting from the abstract type to the corresponding representation type.

For example, for the Matrix type defined in Section 5.5.3,

```
newtype Matrix α = MkMatrix {unMatrix :: [[α]]}
```

the following terms are generated:

$$in_{Matrix} = \lambda p\ x \rightarrow map_{Matrix}\ p\ (\text{MkMatrix } x)$$
$$out_{Matrix} = \lambda p\ x \rightarrow \text{unMatrix } (map_{Matrix}\ p\ x)$$

The tag function takes a single parameter — a **newtype** definition for T — and generates a pair of terms corresponding to (in_T, out_T). If the type constructor T has n non-phantom parameters then each component of the pair is a function taking $n+1$ parameters: one for each non-phantom parameter α_i, and a value x of either the representation type τ (for in_T) or the abstract type $T \tau_1 \ldots \tau_n$ (for out_T). The in_T function seals the value x and traverses the result using map_T (Section 5.5.4.2) and the functions $h_1 \ldots h_n$. The generated out_T function unseals the result of traversing the sealed value x.

$$
\begin{aligned}
&tag\ [\textbf{newtype}\ \text{T } \alpha_1\ \ldots\ \alpha_1\ =\ \text{In } \{\text{out :: } \tau\}] \\
&= (\lambda param\,(\alpha_1,\ \text{T})\ \ldots\ param\,(\alpha_n,\ \text{T})\ \text{x}\ \rightarrow \\
&\qquad map_T\ param\,(\alpha_1,\ \text{T})\ \ldots\ param\,(\alpha_n,\ \text{T})\ (\text{In x}), \\
&\quad\ \lambda param\,(\alpha_1,\ \text{T})\ \ldots\ param\,(\alpha_n,\ \text{T})\ \text{x}\ \rightarrow \\
&\qquad \text{out } (map_T\ param\,(\alpha_1,\ \text{T})\ \ldots\ param\,(\alpha_n,\ \text{T})\ \text{x}))
\end{aligned}
$$

For a nullary type constructor such as Complex, in_T and out_T are equivalent to the constructor and destructor for the type.

5.5.4.4 Converting signing-style functions to sealing-style functions (tr)

The final step in translating from the signing style to the sealing style involves translating each operation in the definition of the abstract type.

For each operation f in the definition of the abstract type we generate a function that converts from the signing-style version of f to the sealing-style version. For example, for the function real, the generated transformation function is the following

$$tr^+_{\mathtt{Complex}} [\mathtt{Complex} \to \mathtt{Float}] = \mathtt{map}_\to \ \mathtt{out}_{\mathtt{Complex}} \ \mathtt{map}_{\mathtt{Float}}$$

with the type

```
((Float, Float)  →  Float)  →  (Complex  →  Float)
```

i.e. it accepts a function of type (Float, Float) $\to$ Float and returns a function of type Complex $\to$ Float.

The function tr generates transformation functions. There are three parameters: the name of the abstract type constructor (written as a subscript), a polarity (written as a superscript) and the type τ which to be transformed (written in brackets following tr). The polarity is used to keep track of whether we should insert constructors or destructors at a particular occurrence of the abstract type constructor. The important cases occur when we encounter the constructor of the abstract type T; here we insert calls to $\mathtt{in}_\mathtt{T}$ if we are treating a positive occurrence of T and $\mathtt{out}_\mathtt{T}$ for a negative occurrence.

$$
\begin{aligned}
tr^+_\mathtt{T}[\mathtt{T}] &= \mathtt{in}_\mathtt{T} \\
tr^-_\mathtt{T}[\mathtt{T}] &= \mathtt{out}_\mathtt{T}
\end{aligned}
$$

The remainder of the cases involve traversing values of the type to find occurrences of T. Since T may not occur within type variables, the transformation function at a type variable is the identity function.

$$tr^\mathtt{p}_\mathtt{T}[\alpha] = \mathtt{id}$$

At type constructors other than T the transformation is based on the map function corresponding to the constructor.

$$tr^\mathtt{p}_\mathtt{T}[\mathtt{T'}] = \mathtt{map}_{\mathtt{T'}} \quad (\textit{where } \mathtt{T} \neq \mathtt{T}')$$

At type applications $\tau_1 \ \tau_2$ the generated code is an application of the function generated for τ_1. As with the code that map_E generates for type applications, the argument that is generated depends on the type of the generated function expression (and ultimately on the variance of the type constructor that is involved). We generate the following application expressions from type applications:

$$tr^\mathtt{p}_\mathtt{T}[\tau_1 \ \tau_2] = tr^\mathtt{p}_\mathtt{T}[\tau_1] \, (tr^\mathtt{p}_\mathtt{T}[\tau_2], \, tr^{\bar{\mathtt{p}}}_\mathtt{T}[\tau_2])$$

if τ_2 is the argument for a type parameter that is used both positively and negatively.

$$tr^\mathtt{p}_\mathtt{T}[\tau_1 \ \tau_2] = tr^\mathtt{p}_\mathtt{T}[\tau_1] \, (tr^\mathtt{p}_\mathtt{T}[\tau_2])$$

if τ_2 is the argument for a type parameter that is used only positively.

$$tr^\mathtt{p}_\mathtt{T}[\tau_1 \ \tau_2] = tr^\mathtt{p}_\mathtt{T}[\tau_1] \, (tr^{\bar{\mathtt{p}}}_\mathtt{T}[\tau_2])$$

if τ_2 is the argument for a type parameter that is used only negatively.

$$tr^\mathtt{p}_\mathtt{T}[\tau_1 \ \tau_2] = tr^\mathtt{p}_\mathtt{T}[\tau_1]$$

if τ_2 is the argument for a phantom type parameter.

5.5.4.5 General translation scheme ($signed$)

The general scheme for translating signing-style definitions to sealing-style is as follows.

```
signed
    [d| type T α₁ ... α₁ = τ
        x₁ :: τ₁
        x₁ = e₁
        ...
        xₙ :: τₙ
        xₙ = eₙ |]
  =
    [d| newtype T α₁ ... α₁ = In {out :: τ}
        (inₜ, outₜ) = tag[newtype T α₁ ... α₁ = In {out :: τ}]
        mapₜ   = map_D[newtype T α₁ ... α₁ = In {out :: τ}]
        map_{T₁} = map_D[reify T₁]
        ...
        map_{Tₘ} = map_D[reify Tₘ]
        (x₁', ... xₙ') = (tr⁺ₜ[τ₁] x₁, ... tr⁺ₜ[τₙ] xₙ)
            where x₁ = e₁; ... xₙ = eₙ
        x₁ :: τ₁
        x₁ = x₁'
        ...
        xₙ :: τₙ
        xₙ = xₙ'   |]
```

where

$$\{\mathtt{T_1}, \ldots, \mathtt{T_m}\} = tynames_E\,[\tau]\,\{\} \cup tynames_E\,[\tau_1]\,\{\} \cup \ldots \cup tynames_E\,[\tau_n]\,\{\}$$

The definition of $signed$ is largely a straightforward combination of the functions described in Sections 5.5.4.1–5.5.4.4.

The bindings in the generated code are carefully arranged so that references to bindings from inside the abstraction resolve to the untransformed versions of the functions in the interface, while references to the bindings from outside resolve to the transformed versions. In particular, the call to `real` in the definition of `conj` resolves to the function with type `(Float, Float) → Float`, not to the version with type `Complex → Float` that is exposed to the user.

5.6 Related work

The sealing style for creating abstract types dates back to Morris (Morris, 1973), who informally outlines the connection with the signing style. More recently, several authors have given

formal translations between static and dynamic schemes for preserving types. It is not surpris-
ing that the translations are similar in each case, although their motivations (and consequently
the precise properties that they investigate) differ considerably.

Sumii and Pierce (Sumii and Pierce, 2004) have examined the relationship between type
abstraction and cryptography, developing a theory of relational parametricity for their crypto-
graphic λ-calculus. They give an encoding of type abstraction into encryption (but not vice
versa); their encryption primitives, which give a variant of dynamic sealing, are significantly
more expressive than our tags, so the inverse translation is not straightforward.

Matthews and Findler (Matthews and Findler, 2009) investigate the semantics of programs
written in multiple languages — partly in Scheme and partly in ML. They investigate two ways
to encode foreign values: as opaque "lumps" that must be explicitly passed to the language
which created them each time they are used, or as values that have been wrapped using a type-
directed strategy, so that they can be used directly. The wrappings, which dynamically check
that the values they wrap have the appropriate shapes, are an instance of higher-order contracts
(Findler and Felleisen, 2002); the type-directed scheme for inserting guards is analogous to our
translation from signing-style to sealing-style abstract types.

Matthews and Ahmed (Matthews and Ahmed, 2008) extend Matthews and Findler's system
with polymorphism, and prove a parametricity property. In order to such prevent violations
of parametricity as can arise from dynamic inspection of values by the Scheme portion of a
program, they dynamically seal values of abstract type before passing them to Scheme.

5.7   Future work

Abstract types   The central result of Sections 5.1–5.4 establishes an equivalence between
translations of the two styles of abstract type within the context of a single program (namely
the continuation M of the **signtype** and **sealtype** constructs). More traditionally, abstract
types are represented using existential types; we might obtain a stronger result by translating
signtype to an existential type and **sealtype** to some other appropriate type (perhaps
extending PolyPCF with scoped data constructors, rather than globally-available tags), then
showing that the two are isomorphic up to observational equivalence.

We have, in this chapter, investigated a comparatively simple notion of abstract type. Mod-
ern ML-family languages (Milner, Tofte, Harper, and MacQueen, 1997, Romanenko, Russo,
and Sestoft, 2000, Leroy, 2008) typically support considerably more elaborate mechanisms for
creating abstract types: nested signatures, parameterised modules (functors), recursive mod-
ules, and so on. Further, in our work the number of seals is statically fixed for each program.

This concords with the features provided by Haskell and SML, but is less general than mechanisms used in Scheme, where seals may be created dynamically. The question naturally arises whether the scheme given here can be extended to treat these more general systems.

Overloading   We have shown that the two styles of type abstraction commonly found in functional programming languages are equivalent and inter-definable. The language designer is therefore free to provide either style, without danger of loss of expressiveness. There is a similar dichotomy in the design space of overloaded functions: should we base method resolution on the types of the program, or on its values? The designers of Haskell have chosen the first option: values are typically represented uniformly, and method resolution in Haskell can be influenced by type annotations. Object-oriented languages typically take the second route, inspecting values at runtime to select a suitable method implementation. To what extent are these approaches compatible? Can we design a system which admits either compilation strategy? Odersky, Wadler, and Wehr (1995) resolve some of these questions, by requiring that the type variable denoting the instance type appear as the first argument of every overloaded method. It remains to be seen whether a system such as Odersky et al.'s (1995) can incorporate features of the modern system of Haskell type classes, such as constructor classes and associated types (Chakravarty, Keller, and Peyton Jones, 2005a, Chakravarty, Keller, Peyton Jones, and Marlow, 2005b).

Chapter 6

Conclusion

6.1 Contributions

Our study of three interfaces to computation — idioms, arrows, and monads — revealed that idioms are the weakest of the three, an adjustment to the informal claims of McBride and Paterson (2008). We have proved that the arrow calculus, a variant of Paterson's notation, is in equational correspondence with the classical presentation of arrows. Using the arrow calculus, we have shown that idioms correspond to those arrows which are isomorphic to oblivious computations that return functions:

$$A \leadsto B \cong 1 \leadsto (A \to B)$$

and that monads correspond to those arrows which are isomorphic to functions that construct computations:

$$A \leadsto B \cong A \to (1 \leadsto B)$$

Additionally, we have used the laws of each interface to construct normalising transformers for idioms, arrows and monads in Haskell.

We used our results about idioms, arrows and monads to guide the design of a library, *formlets*, for compositional construction of HTML forms. The semantics of formlets are given in terms of three primitive idioms, which capture the effects involved in form construction: fresh name generation, XML construction, and reading values from an environment. While each of these idioms is also a monad, their composition as the formlet idiom is not a monad; in fact, we argue that idioms, the weakest of the three interfaces, are the most suitable basis for formlets. Our description of the formlet library is intended to reveal the "essence" of form abstraction, and the result is accordingly simple. Nevertheless, formlets extend readily to more realistic settings. We have shown how to support validation of user input by incorporating an idiom

197

that captures failure, and how to give static guarantees of XHTML validity by using *indexed* idioms, an analogue of indexed monads (Wadler and Thiemann, 2003, Abadi, 2007). Formlets also come equipped with a convenient syntax; we have given a desugaring of this syntax into applications of the idiom operations and an alternative, more efficient, desugaring based on multi-holed contexts and *parameterised* idioms (an analogue of Atkey's (2009) parameterised monads).

The formlets library depends on continuations that persist between requests. Links uses a *stateless-server* approach, embedding serialised continuations within the page sent to the client, to be sent back to the server with subsequent requests. Code for serialisation typically recapitulates the structure of the type to be serialised; we avoid this in the Links implementation by using an analogue of Haskell's *deriving* mechanism for automatically constructing overloaded function instances from type declarations. The design of our *deriving* for OCaml (the Links implementation language) is guided by the well-known correspondence between type classes and ML-style modules. We have described the implementation of a structure-sharing serialiser, `Pickle`, in the *deriving* framework, and shown how giving custom instances for equality and hashing (superclasses of `Pickle`) can improve the compactness of its output without the need to change any serialisation code. We illustrated this using a serialiser for lambda terms that is customised to consider alpha-equivalent terms equal, resulting in more compact output than serialisers which treat terms as simple trees.

Our final chapter investigated the two common styles of abstract type definition, in which the abstraction boundary is drawn either in the terms, using a private data constructor ("sealing"), or in the types, using a signature ("signing"). We extended Pitts's (2000) partial polymorphic lambda calculus, PolyPCF, with constructs for both styles and used parametricity to show that they are inter-definable. To demonstrate the utility of the equivalence we gave a implementation of the signing style in Template Haskell, showing how programs written in the signing style can be translated into the sealing style, addressing concerns about ambiguity that led the Haskell designers to choose the latter over the former.

6.2   The status of Links

The ideas described in this dissertation were developed in the context of the Links project. The reader may therefore be interested in a brief summary of the current status of Links.

There is a small community of Links users, with a mailing list of around seventy members. Work on the implementation continues, with further releases planned for the near future. Researchers in Maryland and Cambridge have developed variants of Links with integrated

support for security (Swamy, Hicks, and Tsang, 2007, Corcoran, Swamy, and Hicks, 2007, Swamy, Corcoran, and Hicks, 2008, Corcoran, Swamy, and Hicks, 2009, Baltopoulos and Gordon, 2009). Finally, researchers in Edinburgh are applying for funding to continue development of Links.

Appendix A

Arrow proofs
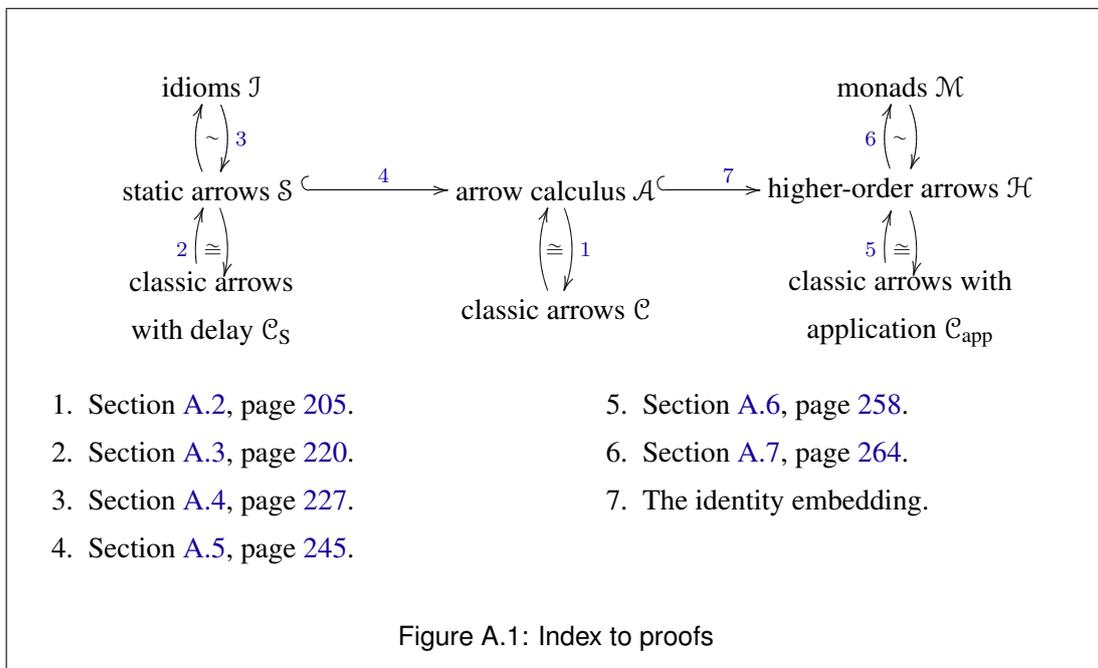
Figure A.1: Index to proofs

This appendix contains proofs of various statements connected with arrows made in Chapter 2. Section A.1 supplies correctness proofs for the normalising arrow transformer of Section 2.2.5.1. Sections A.2–A.7 give proofs of the relationships between the equational theories introduced in Sections 2.3 and 2.4. Figure A.1 is an index to these proofs. Section A.8 gives a proof that the $(\leadsto_{H2})$ law of classic arrows with apply is redundant.

Throughout this appendix we make implicit use of $\leadsto_3$, the associativity law for arrow composition, writing $L \ggg M \ggg N$ for both $(L \ggg M) \ggg N$ and $L \ggg (M \ggg N)$.

A.1 Arrow normalisation

This section gives proofs of various statements related to the normalising arrow transformer of Section 2.2.5.1.

We will use the law numbers given in the formal development of classic arrows in Figure 2.5, which correspond to the order in which the laws were presented in Section 2.2.2.3.

We require the following auxiliary law

$$\mathtt{arr}\,(\mathtt{id} \times \mathtt{f}) = \mathtt{second}\,(\mathtt{arr}\,\mathtt{f})$$

which is a companion to the fifth arrow law. Its proof is as follows:

```
    second (arr f)
=        (def second)
    arr swap ⋙ first (arr f) ⋙ arr swap
=        (⤳₅)
    arr swap ⋙ arr (f × id) ⋙ arr swap
=        (⤳₄)
    arr (swap · f × id · swap)
=        (swap · f × id · swap = id × f)
    arr (id × f)
```

The following three statements occur in the proof of the correctness of the normalising arrow transformer. The first statement is

$$\mathtt{arr}\,\mathtt{f} \ggg ((\mathtt{arr}\,\mathtt{g} \ggg \mathtt{c})\ \&\&\&\ \mathtt{arr}\,\mathtt{id}) \ggg \mathtt{h}$$

$$=$$

$$((\mathtt{arr}\,(\mathtt{g} \cdot \mathtt{f}) \ggg \mathtt{c})\ \&\&\&\ \mathtt{arr}\,\mathtt{id}) \ggg (\mathtt{arr}\,(\mathtt{id} \times \mathtt{f}) \ggg \mathtt{h})$$

and its proof is as follows:

```
    arr f ⋙ ((arr g ⋙ c) &&& arr id) ⋙ h
=        (def &&&, ⤳₄, ⤳₁)
    arr f ⋙ arr dup ⋙ first (arr g ⋙ c) ⋙ h
=        (⤳₆)
    arr f ⋙ arr dup ⋙ first (arr g) ⋙ first c ⋙ h
=        (⤳₅)
    arr f ⋙ arr dup ⋙ arr (g × id) ⋙ first c ⋙ h
=        (⤳₄)
    arr (g × id · dup · f) ⋙ first c ⋙ h
=        ((g × id) · dup · f = (id × f) · ((g · f) × id) · dup)
    arr ((id × f) · ((g · f) × id) · dup) ⋙ first c ⋙ h
=        (⤳₄)
    arr dup ⋙ arr ((g · f) × id) ⋙ arr (id × f) ⋙ first c ⋙ h
```

(continued on next page)

(continued from previous page)

$\quad$ arr dup $\ggg$ arr $((g \cdot f) \times id) \ggg$ arr $(id \times f) \ggg$ first c $\ggg$ h

$=\qquad (\mathtt{second}\,(\mathtt{arr\,f}) = \mathtt{arr}\,(\mathtt{id} \times \mathtt{f}))$

$\quad$ arr dup $\ggg$ arr $((g \cdot f) \times id) \ggg$ second (arr f) $\ggg$ first c $\ggg$ h

$=\qquad (\rightsquigarrow_5, \rightsquigarrow_7)$

$\quad$ arr dup $\ggg$ first $(\mathtt{arr}\,(g \cdot f)) \ggg$ first c $\ggg$ second (arr f) $\ggg$ h

$=\qquad (\rightsquigarrow_6, \mathtt{second}\,(\mathtt{arr\,f}) = \mathtt{arr}\,(\mathtt{id} \times \mathtt{f}))$

$\quad$ arr dup $\ggg$ first $(\mathtt{arr}\,(g \cdot f) \ggg c) \ggg$ arr $(id \times f) \ggg$ h

$=\qquad (\mathtt{def}\,\&\!\&\!\&, \rightsquigarrow_4, \rightsquigarrow_1)$

$\quad ((\mathtt{arr}\,(g \cdot f) \ggg c)\,\&\!\&\!\&\,\mathtt{arr\,id}) \ggg$ arr $(id \times f) \ggg$ h

The second statement is

$\quad$ first $(((\mathtt{arr\,g} \ggg c)\,\&\!\&\!\&\,\mathtt{arr\,id}) \ggg h)$

$\quad =$

$\quad ((\mathtt{arr}\,(g \cdot \mathtt{fst}) \ggg c)\,\&\!\&\!\&\,\mathtt{arr\,id}) \ggg \mathtt{arr\,assoc}^{-1} \ggg \mathtt{first\,h}$

and its proof is as follows:

$\quad$ first $(((\mathtt{arr\,g} \ggg c)\,\&\!\&\!\&\,\mathtt{arr\,id}) \ggg h)$

$=\qquad (\mathtt{def}\,\&\!\&\!\&)$

$\quad$ first (arr dup $\ggg$ first (arr g $\ggg c) \ggg$ arr swap

$\quad\quad \ggg$ first (arr id) $\ggg$ arr swap $\ggg h)$

$=\qquad (\rightsquigarrow_6)$

$\quad$ first (arr dup $\ggg$ first (arr g) $\ggg$ first c $\ggg$ arr swap

$\quad\quad \ggg$ first (arr id) $\ggg$ arr swap $\ggg h)$

$=\qquad (\rightsquigarrow_5)$

$\quad$ first (arr dup $\ggg$ arr $(g \times id) \ggg$ first c $\ggg$ arr swap

$\quad\quad \ggg$ arr $(id \times id) \ggg$ arr swap $\ggg h)$

$=\qquad (\rightsquigarrow_4, \mathtt{swap} \cdot \mathtt{id} \times \mathtt{id} \cdot \mathtt{swap} = \mathtt{id})$

$\quad$ first (arr $((g \times id) \cdot \mathtt{dup}) \ggg$ first c $\ggg$ arr id $\ggg h)$

$=\qquad (\rightsquigarrow_1)$

$\quad$ first (arr $((g \times id) \cdot \mathtt{dup}) \ggg$ first c $\ggg h)$

$=\qquad (\rightsquigarrow_6)$

$\quad$ first $(\mathtt{arr}\,((g \times id) \cdot \mathtt{dup})) \ggg$ first (first c) $\ggg$ first h

$=\qquad (\rightsquigarrow_5)$

$\quad$ arr $(((g \times id) \cdot \mathtt{dup}) \times id) \ggg$ first (first c) $\ggg$ first h

$=\qquad (\mathtt{assoc}^{-1} \cdot \mathtt{assoc} = \mathtt{id}, \rightsquigarrow_1)$

$\quad$ arr $(((g \times id) \cdot \mathtt{dup}) \times id) \ggg$ first (first c) $\ggg$ arr $(\mathtt{assoc}^{-1} \cdot \mathtt{assoc})$

$\quad\quad \ggg$ first h

(continued from previous page)
$\texttt{arr}\ (((\texttt{g} \times \texttt{id}) \cdot \texttt{dup}) \times \texttt{id}) \ggg \texttt{first}\ (\texttt{first}\ \texttt{c}) \ggg \texttt{arr}\ (\texttt{assoc}^{-1} \cdot \texttt{assoc})$
     $\ggg \texttt{first}\ \texttt{h}$
= $\qquad (\leadsto_4)$
$\texttt{arr}\ (((\texttt{g} \times \texttt{id}) \cdot \texttt{dup}) \times \texttt{id}) \ggg \texttt{first}\ (\texttt{first}\ \texttt{c}) \ggg \texttt{arr}\ \texttt{assoc} \ggg \texttt{arr}\ \texttt{assoc}^{-1}$
     $\ggg \texttt{first}\ \texttt{h}$
= $\qquad (\leadsto_9)$
$\texttt{arr}\ (((\texttt{g} \times \texttt{id}) \cdot \texttt{dup}) \times \texttt{id}) \ggg \texttt{arr}\ \texttt{assoc} \ggg \texttt{first}\ \texttt{c} \ggg \texttt{arr}\ \texttt{assoc}^{-1} \ggg \texttt{first}\ \texttt{h}$
= $\qquad (\leadsto_4, \texttt{assoc} \cdot (((\texttt{g} \times \texttt{id}) \cdot \texttt{dup}) \times \texttt{id}) = ((\texttt{g} \cdot \texttt{fst}) \times \texttt{id}) \cdot \texttt{dup})$
$\texttt{arr}\ (\texttt{assoc} \cdot (((\texttt{g} \times \texttt{id}) \cdot \texttt{dup}) \times \texttt{id})) \ggg \texttt{first}\ \texttt{c} \ggg \texttt{arr}\ \texttt{assoc}^{-1} \ggg \texttt{first}\ \texttt{h}$
= $\qquad (\leadsto_4, \texttt{swap} \cdot \texttt{id} \times \texttt{id} \cdot \texttt{swap} \cdot \texttt{f} = \texttt{f})$
$\texttt{arr}\ \texttt{dup} \ggg \texttt{arr}\ ((\texttt{g} \cdot \texttt{fst}) \times \texttt{id}) \ggg \texttt{first}\ \texttt{c} \ggg \texttt{arr}\ \texttt{swap} \ggg \texttt{arr}\ (\texttt{id} \times \texttt{id})$
     $\ggg \texttt{arr}\ \texttt{swap} \ggg \texttt{arr}\ \texttt{assoc}^{-1} \ggg \texttt{first}\ \texttt{h}$
= $\qquad (\leadsto_5)$
$\texttt{arr}\ \texttt{dup} \ggg \texttt{first}\ (\texttt{arr}\ (\texttt{g} \cdot \texttt{fst})) \ggg \texttt{first}\ \texttt{c} \ggg \texttt{arr}\ \texttt{swap} \ggg \texttt{first}\ (\texttt{arr}\ \texttt{id})$
     $\ggg \texttt{arr}\ \texttt{swap} \ggg \texttt{arr}\ \texttt{assoc}^{-1} \ggg \texttt{first}\ \texttt{h}$
= $\qquad (\leadsto_6)$
$\texttt{arr}\ \texttt{dup} \ggg \texttt{first}\ (\texttt{arr}\ (\texttt{g} \cdot \texttt{fst}) \ggg \texttt{c}) \ggg \texttt{arr}\ \texttt{swap} \ggg \texttt{first}\ (\texttt{arr}\ \texttt{id})$
     $\ggg \texttt{arr}\ \texttt{swap} \ggg \texttt{arr}\ \texttt{assoc}^{-1} \ggg \texttt{first}\ \texttt{h}$
= $\qquad (\text{def}\ \&\!\&\!\&)$
$((\texttt{arr}\ (\texttt{g} \cdot \texttt{fst}) \ggg \texttt{c})\ \&\!\&\!\&\ \texttt{arr}\ \texttt{id}) \ggg \texttt{arr}\ \texttt{assoc}^{-1} \ggg \texttt{first}\ \texttt{h}$

The third statement is

$\quad \texttt{f} = ((\texttt{arr}\ \texttt{id} \ggg \texttt{f})\ \&\!\&\!\&\ \texttt{arr}\ \texttt{id}) \ggg \texttt{arr}\ \texttt{fst}$

and its proof is as follows:

$\quad ((\texttt{arr}\ \texttt{id} \ggg \texttt{f})\ \&\!\&\!\&\ \texttt{arr}\ \texttt{id}) \ggg \texttt{arr}\ \texttt{fst}$
= $\qquad (\text{def}\ \&\!\&\!\&)$
$\texttt{arr}\ \texttt{dup} \ggg \texttt{first}\ (\texttt{arr}\ \texttt{id} \ggg \texttt{f}) \ggg \texttt{arr}\ \texttt{fst}$
= $\qquad (\leadsto_6)$
$\texttt{arr}\ \texttt{dup} \ggg \texttt{first}\ (\texttt{arr}\ \texttt{id}) \ggg \texttt{first}\ \texttt{f} \ggg \texttt{arr}\ \texttt{fst}$
= $\qquad (\leadsto_5)$
$\texttt{arr}\ \texttt{dup} \ggg \texttt{arr}\ (\texttt{id} \times \texttt{id}) \ggg \texttt{first}\ \texttt{f} \ggg \texttt{arr}\ \texttt{fst}$
= $\qquad (\texttt{id} \times \texttt{id} = \texttt{id}, \leadsto_1)$
$\texttt{arr}\ \texttt{dup} \ggg \texttt{first}\ \texttt{f} \ggg \texttt{arr}\ \texttt{fst}$
= $\qquad (\leadsto_8)$
$\texttt{arr}\ \texttt{dup} \ggg \texttt{arr}\ \texttt{fst} \ggg \texttt{f}$
= $\qquad (\leadsto_4, \texttt{fst} \cdot \texttt{dup} = \texttt{id})$
$\texttt{arr}\ \texttt{id} \ggg \texttt{f}$
= $\qquad (\leadsto_1)$
$\texttt{f}$

A.2 Equational correspondence between $\mathcal{A}$ and $\mathcal{C}$

This section gives a proof of Proposition 11 (page 61).

The proof depends on two lemmas. Lemma 8 justifies the meaning of translation on a substitution. Lemma 10 gives the translation of weakening for commands of $\mathcal{A}$.

A.2.1 Proofs of Lemmas 8 and 10

We begin with proofs of Lemma 8 and 10, which give the translations of substitution and weakening.

Proof of Lemma 8 (Translating substitution from $\mathcal{A}$ to $\mathcal{C}$)   .

The translations of substitution on terms and commands from $\mathcal{A}$ to $\mathcal{C}$ are as follows.

$$\llbracket M[x := N] \rrbracket = \llbracket M \rrbracket [x := \llbracket N \rrbracket]$$

$$\llbracket P[x := N] \rrbracket_\Delta = \texttt{arr}\ (\lambda\Delta.(\Delta, \llbracket N \rrbracket)) \ggg \llbracket P \rrbracket_{\Delta,x}$$

Proof. By mutual induction on the derivations of P and M. There is one case for each term form and each command form. We give only the cases for command forms here.

1. Case $L \bullet M$

$$
\begin{aligned}
&\llbracket (L \bullet M)[x := N] \rrbracket_\Delta \\
=\ &\qquad (\text{def } \texttt{substitution}) \\
&\llbracket L \bullet (M[x := N]) \rrbracket_\Delta \\
=\ &\qquad (\text{def } \llbracket - \rrbracket_\Delta) \\
&\texttt{arr}\ (\lambda\Delta.\ \llbracket M[x := N] \rrbracket) \ggg \llbracket L \rrbracket \\
=\ &\qquad (\text{induction hypothesis}) \\
&\texttt{arr}\ (\lambda\Delta.\ \llbracket M \rrbracket [x := \llbracket N \rrbracket]) \ggg \llbracket L \rrbracket \\
=\ &\qquad (\leadsto_4) \\
&\texttt{arr}\ (\lambda\Delta.\ \langle\Delta, \llbracket N \rrbracket\rangle) \ggg \texttt{arr}\ (\lambda\langle\Delta, x\rangle.\ \llbracket M \rrbracket) \ggg \llbracket L \rrbracket \\
=\ &\qquad (\text{def } \llbracket - \rrbracket_{\Delta,x}) \\
&\texttt{arr}\ (\lambda\Delta.\ \langle\Delta, \llbracket N \rrbracket\rangle) \ggg \llbracket L \bullet M \rrbracket_{\Delta,x}
\end{aligned}
$$

2. Case $[M]$

$$
\begin{aligned}
&\llbracket [M][x := N] \rrbracket_\Delta \\
=\ &\qquad (\text{def } \texttt{substitution}) \\
&\llbracket [M[x := N]] \rrbracket_\Delta \\
=\ &\qquad (\text{def } \llbracket - \rrbracket_\Delta) \\
&\texttt{arr}\ (\lambda\Delta.\ \llbracket M[x := N] \rrbracket)
\end{aligned}
$$

(continued on next page)

(continued from previous page)
$\texttt{arr}\,(\lambda\Delta.\,[\![\texttt{M}[\texttt{x}:=\texttt{N}]]\!])$
$=$ (induction hypothesis)
$\texttt{arr}\,(\lambda\Delta.\,[\![\texttt{M}]\!][\texttt{x}:=[\![\texttt{N}]\!]])$
$=$ $(\rightsquigarrow_4)$
$\texttt{arr}\,(\lambda\Delta.\,\langle\Delta,[\![\texttt{N}]\!]\rangle) \ggg \texttt{arr}\,(\lambda\langle\Delta,\texttt{x}\rangle.\,[\![\texttt{M}]\!])$
$=$ $(\text{def }[\![-]\!]_{\Delta,\texttt{x}})$
$\texttt{arr}\,(\lambda\Delta.\,\langle\Delta,[\![\texttt{N}]\!]\rangle) \ggg [\![\texttt{M}]\!]_{\Delta,\texttt{x}}$

3. Case **let** $\texttt{y} \Leftarrow \texttt{P}$ **in** $\texttt{Q}$

$[\![(\textbf{let}\,\texttt{y} \Leftarrow \texttt{P}\,\textbf{in}\,\texttt{Q})[\texttt{x}:=\texttt{N}]]\!]_{\Delta}$
$=$ (def substitution)
$[\![\textbf{let}\,\texttt{y} \Leftarrow \texttt{P}[\texttt{x}:=\texttt{N}]\,\textbf{in}\,\texttt{Q}[\texttt{x}:=\texttt{N}]]\!]_{\Delta}$
$=$ $(\text{def }[\![-]\!]_{\Delta})$
$(\texttt{arr id}\,\&\!\&\!\&\,[\![\texttt{P}[\texttt{x}:=\texttt{N}]]\!]_{\Delta}) \ggg [\![\texttt{Q}[\texttt{x}:=\texttt{N}]]\!]_{\Delta,\texttt{y}}$
$=$ (induction hypothesis)
$(\texttt{arr id}\,\&\!\&\!\&\,(\texttt{arr}\,(\lambda\Delta.\,\langle\Delta,[\![\texttt{N}]\!]\rangle) \ggg [\![\texttt{P}]\!]_{\Delta,\texttt{x}}))$
  $\ggg \texttt{arr}\,(\lambda\langle\Delta,\texttt{y}\rangle.\,\langle\langle\Delta,\texttt{y}\rangle,[\![\texttt{N}]\!]\rangle) \ggg [\![\texttt{Q}]\!]_{\Delta,\texttt{y},\texttt{x}}$
$=$ $(\text{def }\&\!\&\!\&)$
$\texttt{arr dup} \ggg \texttt{first}\,(\texttt{arr id}) \ggg \texttt{arr swap}$
  $\ggg \texttt{first}\,(\texttt{arr}\,(\lambda\Delta.\,\langle\Delta,[\![\texttt{N}]\!]\rangle) \ggg [\![\texttt{P}]\!]_{\Delta,\texttt{x}}) \ggg \texttt{arr swap}$
  $\ggg \texttt{arr}\,(\lambda\langle\Delta,\texttt{y}\rangle.\,\langle\langle\Delta,\texttt{y}\rangle,[\![\texttt{N}]\!]\rangle) \ggg [\![\texttt{Q}]\!]_{\Delta,\texttt{y},\texttt{x}}$
$=$ $(\rightsquigarrow_6)$
$\texttt{arr dup} \ggg \texttt{first}\,(\texttt{arr id}) \ggg \texttt{arr swap} \ggg \texttt{first}\,(\texttt{arr}\,(\lambda\Delta.\,\langle\Delta,[\![\texttt{N}]\!]\rangle))$
  $\ggg \texttt{first}\,[\![\texttt{P}]\!]_{\Delta,\texttt{x}} \ggg \texttt{arr swap} \ggg \texttt{arr}\,(\lambda\langle\Delta,\texttt{y}\rangle.\,\langle\langle\Delta,\texttt{y}\rangle,[\![\texttt{N}]\!]\rangle)$
  $\ggg [\![\texttt{Q}]\!]_{\Delta,\texttt{y},\texttt{x}}$
$=$ $(\rightsquigarrow_5)$
$\texttt{arr dup} \ggg \texttt{arr}\,(\texttt{id} \times \texttt{id}) \ggg \texttt{arr swap} \ggg \texttt{arr}\,((\lambda\Delta.\,\langle\Delta,[\![\texttt{N}]\!]\rangle) \times \texttt{id})$
  $\ggg \texttt{first}\,[\![\texttt{P}]\!]_{\Delta,\texttt{x}} \ggg \texttt{arr swap} \ggg \texttt{arr}\,(\lambda\langle\Delta,\texttt{y}\rangle.\,\langle\langle\Delta,\texttt{y}\rangle,[\![\texttt{N}]\!]\rangle) \ggg [\![\texttt{Q}]\!]_{\Delta,\texttt{y},\texttt{x}}$
$=$ $(\rightsquigarrow_4,\ \texttt{swap}\cdot\texttt{id}\times\texttt{id}\cdot\texttt{dup}=\texttt{dup})$
$\texttt{arr}\,(\lambda\Delta.\,\langle\langle\Delta,[\![\texttt{N}]\!]\rangle,\Delta\rangle) \ggg \texttt{first}\,[\![\texttt{P}]\!]_{\Delta,\texttt{x}} \ggg \texttt{arr}\,(\lambda\langle\texttt{y},\Delta\rangle.\,\langle\langle\Delta,\texttt{y}\rangle,[\![\texttt{N}]\!]\rangle) \ggg [\![\texttt{Q}]\!]_{\Delta,\texttt{y},\texttt{x}}$
$=$ (Lemma 10)
$\texttt{arr}\,(\lambda\Delta.\,\langle\langle\Delta,[\![\texttt{N}]\!]\rangle,\Delta\rangle) \ggg \texttt{first}\,[\![\texttt{P}]\!]_{\Delta,\texttt{x}} \ggg \texttt{arr}\,(\lambda\langle\texttt{y},\Delta\rangle.\,\langle\langle\Delta,\texttt{y}\rangle,[\![\texttt{N}]\!]\rangle)$
  $\ggg \texttt{arr}\,(\lambda\langle\langle\Delta,\texttt{y}\rangle,\texttt{x}\rangle.\,\langle\langle\Delta,\texttt{x}\rangle,\texttt{y}\rangle) \ggg [\![\texttt{Q}]\!]_{\Delta,\texttt{x},\texttt{y}}$
$=$ $(\rightsquigarrow_4,\ \beta^{\rightarrow})$
$\texttt{arr}\,(\lambda\Delta.\,\langle\langle\Delta,[\![\texttt{N}]\!]\rangle,\langle\Delta,[\![\texttt{N}]\!]\rangle\rangle) \ggg \texttt{arr}\,(\texttt{id} \times \texttt{fst}) \ggg \texttt{first}\,[\![\texttt{P}]\!]_{\Delta,\texttt{x}}$
  $\ggg \texttt{arr}\,(\lambda\langle\texttt{y},\Delta\rangle.\,\langle\langle\Delta,[\![\texttt{N}]\!]\rangle,\texttt{y}\rangle) \ggg [\![\texttt{Q}]\!]_{\Delta,\texttt{x},\texttt{y}}$

(continued from previous page)

$\text{arr}\,(\lambda\Delta.\,\langle\langle\Delta, [\![N]\!]\rangle, \langle\Delta, [\![N]\!]\rangle\rangle) \ggg \text{arr}\,(\text{id} \times \text{fst}) \ggg \text{first}\, [\![P]\!]_{\Delta,\text{x}}$
$\quad \ggg \text{arr}\,(\lambda\langle\text{y}, \Delta\rangle.\,\langle\langle\Delta, [\![N]\!]\rangle, \text{y}\rangle) \ggg [\![Q]\!]_{\Delta,\text{x},\text{y}}$

$=\qquad (\rightsquigarrow_7)$

$\text{arr}\,(\lambda\Delta.\,\langle\langle\Delta, [\![N]\!]\rangle, \langle\Delta, [\![N]\!]\rangle\rangle) \ggg \text{first}\, [\![P]\!]_{\Delta,\text{x}} \ggg \text{arr}\,(\text{id} \times \text{fst})$
$\quad \ggg \text{arr}\,(\lambda\langle\text{y}, \Delta\rangle.\,\langle\langle\Delta, [\![N]\!]\rangle, \text{y}\rangle) \ggg [\![Q]\!]_{\Delta,\text{x},\text{y}}$

$=\qquad (\rightsquigarrow_4)$

$\text{arr}\,(\lambda\Delta.\,\langle\langle\Delta, [\![N]\!]\rangle, \langle\Delta, [\![N]\!]\rangle\rangle) \ggg \text{first}\, [\![P]\!]_{\Delta,\text{x}} \ggg \text{arr}\,(\text{id} \times (\lambda\langle\Delta, \text{x}\rangle.\,\langle\Delta, [\![N]\!]\rangle))$
$\quad \ggg \text{arr}\,\text{swap} \ggg [\![Q]\!]_{\Delta,\text{x},\text{y}}$

$=\qquad (\rightsquigarrow_7)$

$\text{arr}\,(\lambda\Delta.\,\langle\langle\Delta, [\![N]\!]\rangle, \langle\Delta, [\![N]\!]\rangle\rangle) \ggg \text{arr}\,(\text{id} \times (\lambda\langle\Delta, \text{x}\rangle.\,\langle\Delta, [\![N]\!]\rangle)) \ggg \text{first}\, [\![P]\!]_{\Delta,\text{x}}$
$\quad \ggg \text{arr}\,\text{swap} \ggg [\![Q]\!]_{\Delta,\text{x},\text{y}}$

$=\qquad (\rightsquigarrow_4, \beta^{\rightarrow})$

$\text{arr}\,(\lambda\Delta.\,\langle\langle\Delta, [\![N]\!]\rangle, \langle\Delta, [\![N]\!]\rangle\rangle) \ggg \text{first}\, [\![P]\!]_{\Delta,\text{x}} \ggg \text{arr}\,\text{swap} \ggg [\![Q]\!]_{\Delta,\text{x},\text{y}}$

$=\qquad (\rightsquigarrow_4, \text{swap} \cdot \text{id} \times \text{id} \cdot \text{dup} = \text{dup})$

$\text{arr}\,(\lambda\Delta.\,\langle\Delta, [\![N]\!]\rangle) \ggg \text{arr}\,\text{dup} \ggg \text{arr}\,(\text{id} \times \text{id}) \ggg \text{arr}\,\text{swap} \ggg \text{first}\, [\![P]\!]_{\Delta,\text{x}}$
$\quad \ggg \text{arr}\,\text{swap} \ggg [\![Q]\!]_{\Delta,\text{x},\text{y}}$

$=\qquad (\rightsquigarrow_5)$

$\text{arr}\,(\lambda\Delta.\,\langle\Delta, [\![N]\!]\rangle) \ggg \text{arr}\,\text{dup} \ggg \text{first}\,(\text{arr}\,\text{id}) \ggg \text{arr}\,\text{swap} \ggg \text{first}\, [\![P]\!]_{\Delta,\text{x}}$
$\quad \ggg \text{arr}\,\text{swap} \ggg [\![Q]\!]_{\Delta,\text{x},\text{y}}$

$=\qquad (\text{def}\,\&\!\&\!\&)$

$\text{arr}\,(\lambda\Delta.\,\langle\Delta, [\![N]\!]\rangle) \ggg (\text{arr}\,\text{id}\,\&\!\&\!\&\, [\![P]\!]_{\Delta,\text{x}}) \ggg [\![Q]\!]_{\Delta,\text{x},\text{y}}$

$=\qquad (\text{def}\,[\![-]\!]_{\Delta,\text{x}})$

$\text{arr}\,(\lambda\Delta.\,\langle\Delta, [\![N]\!]\rangle) \ggg [\![\textbf{let}\,\text{y} \Leftarrow P\,\textbf{in}\,Q]\!]_{\Delta,\text{x}}$

$\square$

Proof of Lemma 10 (Translating weakening from $\mathcal{A}$ to $\mathcal{C}$)   .

The translation of weakening from $\mathcal{A}$ to $\mathcal{C}$ for commands is as follows.

$$\left[\!\!\left[\frac{\Gamma;\, \Delta \vdash Q\,!\,B}{\Gamma';\, \Delta' \vdash Q\,!\,B}\right]\!\!\right] \quad = \quad \frac{\Gamma \vdash [\![Q]\!]_\Delta : \Delta \rightsquigarrow B}{\Gamma' \vdash \text{arr}\,(\lambda\Delta'.\Delta) \ggg [\![Q]\!]_\Delta : \Delta' \rightsquigarrow B}$$

Proof. By induction on the derivation of Q. There is one case for each command form.

1. Case L • M.

$$\llbracket \text{L} \bullet \text{M} \rrbracket_{\Delta'}$$
$$= \qquad (\text{def } \llbracket - \rrbracket_{\Delta'})$$
$$\text{arr } (\lambda\Delta'. \llbracket \text{M} \rrbracket) \ggg \llbracket \text{L} \rrbracket$$
$$= \qquad ((\lambda\Delta.\text{M}) \cdot (\lambda\Delta'.\Delta) = (\lambda\Delta'.\text{M}))$$
$$\text{arr } ((\lambda\Delta. \llbracket \text{M} \rrbracket) \cdot (\lambda\Delta'.\Delta)) \ggg \llbracket \text{L} \rrbracket$$
$$= \qquad (\leadsto_4)$$
$$\text{arr } (\lambda\Delta'.\Delta) \ggg \text{arr } (\lambda\Delta. \llbracket \text{M} \rrbracket) \ggg \llbracket \text{L} \rrbracket$$
$$= \qquad (\text{def } \llbracket - \rrbracket_{\Delta})$$
$$\text{arr } (\lambda\Delta'.\Delta) \ggg \llbracket \text{L} \bullet \text{M} \rrbracket_{\Delta}$$

2. Case [M].

$$\llbracket [\text{M}] \rrbracket_{\Delta'}$$
$$= \qquad (\text{def } \llbracket - \rrbracket_{\Delta'})$$
$$\text{arr } (\lambda\Delta'. \llbracket \text{M} \rrbracket)$$
$$= \qquad ((\lambda\Delta.\text{M}) \cdot (\lambda\Delta'.\Delta) = (\lambda\Delta'.\text{M}))$$
$$\text{arr } ((\lambda\Delta. \llbracket \text{M} \rrbracket) \cdot (\lambda\Delta'.\Delta))$$
$$= \qquad (\leadsto_4)$$
$$\text{arr } (\lambda\Delta'.\Delta) \ggg \text{arr } (\lambda\Delta. \llbracket \text{M} \rrbracket)$$
$$= \qquad (\text{def } \llbracket - \rrbracket_{\Delta})$$
$$\text{arr } (\lambda\Delta'.\Delta) \ggg \llbracket [\text{M}] \rrbracket_{\Delta}$$

3. Case **let** x ⇐ P **in** Q.

$$\llbracket \textbf{let } \text{x} \Leftarrow \text{P } \textbf{in } \text{Q} \rrbracket_{\Delta'}$$
$$= \qquad (\text{def } \llbracket - \rrbracket_{\Delta'})$$
$$(\text{arr id } \&\&\& \llbracket \text{P} \rrbracket_{\Delta'}) \ggg \llbracket \text{Q} \rrbracket_{\Delta',\text{x}}$$
$$= \qquad (\text{induction hypothesis})$$
$$(\text{arr id } \&\&\& (\text{arr } (\lambda\Delta'.\Delta) \ggg \llbracket \text{P} \rrbracket_{\Delta})) \ggg \text{arr } ((\lambda\Delta'.\Delta) \times \text{id}) \ggg \llbracket \text{Q} \rrbracket_{\Delta,\text{x}}$$
$$= \qquad (\text{def } \&\&\&)$$
$$\text{arr dup} \ggg \text{first } (\text{arr id}) \ggg \text{arr swap} \ggg \text{first } (\text{arr } (\lambda\Delta'.\Delta) \ggg \llbracket \text{P} \rrbracket_{\Delta})$$
$$\qquad \ggg \text{arr swap} \ggg \text{arr } ((\lambda\Delta'.\Delta) \times \text{id}) \ggg \llbracket \text{Q} \rrbracket_{\Delta,\text{x}}$$
$$= \qquad (\leadsto_6)$$
$$\text{arr dup} \ggg \text{first } (\text{arr id}) \ggg \text{arr swap} \ggg \text{first } (\text{arr } (\lambda\Delta'.\Delta))$$
$$\qquad \ggg \text{first } \llbracket \text{P} \rrbracket_{\Delta} \ggg \text{arr swap} \ggg \text{arr } ((\lambda\Delta'.\Delta) \times \text{id}) \ggg \llbracket \text{Q} \rrbracket_{\Delta,\text{x}}$$
$$= \qquad (\leadsto_5)$$
$$\text{arr dup} \ggg \text{arr } (\text{id} \times \text{id}) \ggg \text{arr swap} \ggg \text{arr } ((\lambda\Delta'.\Delta) \times \text{id})$$
$$\qquad \ggg \text{first } \llbracket \text{P} \rrbracket_{\Delta} \ggg \text{arr swap} \ggg \text{arr } ((\lambda\Delta'.\Delta) \times \text{id}) \ggg \llbracket \text{Q} \rrbracket_{\Delta,\text{x}}$$
$$= \qquad (\leadsto_4, ((\lambda\Delta'.\Delta) \times \text{id}) \cdot \text{swap} \cdot (\text{id} \times \text{id}) \cdot \text{dup} = (\lambda\Delta'.\langle\Delta,\Delta'\rangle))$$
$$\text{arr } (\lambda\Delta'.\langle\Delta,\Delta'\rangle) \ggg \text{first } \llbracket \text{P} \rrbracket_{\Delta} \ggg \text{arr } (((\lambda\Delta'.\Delta) \times \text{id}) \cdot \text{swap}) \ggg \llbracket \text{Q} \rrbracket_{\Delta,\text{x}}$$

(continued on next page)

(continued from previous page)
$$\text{arr}\,(\lambda\Delta'.\,\langle\Delta,\Delta'\rangle) \ggg \text{first}\,[\![P]\!]_\Delta \ggg \text{arr}\,(((\lambda\Delta'.\,\Delta)\times\text{id})\cdot\text{swap}) \ggg [\![Q]\!]_{\Delta,x}$$

$=\qquad ((f\times g)\cdot\text{swap}=\text{swap}\cdot(\text{id}\times g))$

$$\text{arr}\,(\lambda\Delta'.\,\langle\Delta,\Delta'\rangle) \ggg \text{first}\,[\![P]\!]_\Delta \ggg \text{arr}\,(\text{swap}\cdot(\text{id}\times(\lambda\Delta'.\,\Delta))) \ggg [\![Q]\!]_{\Delta,x}$$

$=\qquad (\leadsto_4)$

$$\text{arr}\,(\lambda\Delta'.\,\langle\Delta,\Delta'\rangle) \ggg \text{first}\,[\![P]\!]_\Delta \ggg \text{arr}\,(\text{id}\times(\lambda\Delta'.\,\Delta)) \ggg \text{arr}\,\text{swap} \ggg [\![Q]\!]_{\Delta,x}$$

$=\qquad (\leadsto_7)$

$$\text{arr}\,(\lambda\Delta'.\,\langle\Delta,\Delta'\rangle) \ggg \text{arr}\,(\text{id}\times(\lambda\Delta'.\,\Delta)) \ggg \text{first}\,[\![P]\!]_\Delta \ggg \text{arr}\,\text{swap} \ggg [\![Q]\!]_{\Delta,x}$$

$=\qquad (\leadsto_4,\,\text{id}\times(\lambda\Delta'.\,\Delta)\cdot(\lambda\Delta'.\,\langle\Delta,\Delta'\rangle)=(\lambda\Delta'.\,\langle\Delta,\Delta\rangle))$

$$\text{arr}\,(\lambda\Delta'.\,\langle\Delta,\Delta\rangle) \ggg \text{first}\,[\![P]\!]_\Delta \ggg \text{arr}\,\text{swap} \ggg [\![Q]\!]_{\Delta,x}$$

$=\qquad (\text{swap}\cdot(\text{id}\times\text{id})\cdot\text{dup}\cdot(\lambda\Delta'.\,\Delta)=(\lambda\Delta'.\,\langle\Delta,\Delta\rangle))$

$$\text{arr}\,(\text{swap}\cdot(\text{id}\times\text{id})\cdot\text{dup}\cdot(\lambda\Delta'.\,\Delta)) \ggg \text{first}\,[\![P]\!]_\Delta \ggg \text{arr}\,\text{swap} \ggg [\![Q]\!]_{\Delta,x}$$

$=\qquad (\leadsto_4)$

$$\text{arr}\,(\lambda\Delta'.\,\Delta) \ggg \text{arr}\,\text{dup} \ggg \text{arr}\,(\text{id}\times\text{id}) \ggg \text{arr}\,\text{swap} \ggg \text{first}\,[\![P]\!]_\Delta$$
$$\ggg \text{arr}\,\text{swap} \ggg [\![Q]\!]_{\Delta,x}$$

$=\qquad (\leadsto_5)$

$$\text{arr}\,(\lambda\Delta'.\,\Delta) \ggg \text{arr}\,\text{dup} \ggg \text{first}\,(\text{arr}\,\text{id}) \ggg \text{arr}\,\text{swap} \ggg \text{first}\,[\![P]\!]_\Delta$$
$$\ggg \text{arr}\,\text{swap} \ggg [\![Q]\!]_{\Delta,x}$$

$=\qquad (\text{def}\,\&\&\&)$

$$\text{arr}\,(\lambda\Delta'.\,\Delta) \ggg (\text{arr}\,\text{id}\,\&\&\&\,[\![P]\!]_\Delta) \ggg [\![Q]\!]_{\Delta,x}$$

$=\qquad (\text{def}\,[\![-]\!]_\Delta)$

$$\text{arr}\,(\lambda\Delta'.\,\Delta) \ggg [\![\textbf{let}\ x \Leftarrow P\ \textbf{in}\ Q]\!]_\Delta$$

$\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\square$

A.2.2 The laws of $\mathcal{A}$ follow from the laws of $\mathcal{C}$

For each arrow calculus law $M = N$ we must show $[\![M]\!] = [\![N]\!]$.

1. $(\beta^{\leadsto})$

$$[\![(\lambda^\bullet x.\,Q)\bullet M]\!]_\Delta$$

$=\qquad (\text{def}\,[\![-]\!]_\Delta)$

$$\text{arr}\,(\lambda\Delta.\,[\![M]\!]) \ggg [\![Q]\!]_x$$

$=\qquad (\leadsto_4)$

$$\text{arr}\,(\lambda\Delta.\,\langle\Delta,[\![M]\!]\rangle) \ggg \text{arr}\,(\lambda\langle\Delta,x\rangle.\,x) \ggg [\![Q]\!]_x$$

$=\qquad (\text{Lemma } 10)$

$$\text{arr}\,(\lambda\Delta.\,\langle\Delta,[\![M]\!]\rangle) \ggg [\![Q]\!]_{\Delta,x}$$

$=\qquad (\text{Lemma } 8)$

$$[\![Q[x:=M]]\!]_\Delta$$

2. $(\eta^{\leadsto})$

$$\llbracket \lambda^{\bullet} x.\, L \bullet x \rrbracket$$
$$=\qquad (\text{def } \llbracket - \rrbracket)$$
$$\text{arr id} \ggg \llbracket L \rrbracket$$
$$=\qquad (\leadsto_1)$$
$$\llbracket L \rrbracket$$

3. (left)

$$\llbracket \mathbf{let}\ x \Leftarrow [M]\ \mathbf{in}\ Q \rrbracket_{\Delta}$$
$$=\qquad (\text{def } \llbracket - \rrbracket_{\Delta})$$
$$\text{arr dup} \ggg \texttt{first}\, (\text{arr}\, (\lambda\Delta.\, \llbracket M \rrbracket)) \ggg \text{arr swap} \ggg \llbracket Q \rrbracket_{\Delta,x}$$
$$=\qquad (\leadsto_5)$$
$$\text{arr dup} \ggg \text{arr}\, ((\lambda\Delta.\, \llbracket M \rrbracket) \times \text{id}) \ggg \text{arr swap} \ggg \llbracket Q \rrbracket_{\Delta,x}$$
$$=\qquad (\leadsto_4)$$
$$\text{arr}\, (\text{swap} \cdot ((\lambda\Delta.\, \llbracket M \rrbracket) \times \text{id}) \cdot \text{dup}) \ggg \llbracket Q \rrbracket_{\Delta,x}$$
$$=\qquad (\text{swap} \cdot ((\lambda\Delta.\, \llbracket M \rrbracket) \times \text{id}) \cdot \text{dup} = (\lambda\Delta.\, \langle \Delta, \llbracket M \rrbracket \rangle))$$
$$\text{arr}\, (\lambda\Delta.\, \langle \Delta, \llbracket M \rrbracket \rangle) \ggg \llbracket Q \rrbracket_{\Delta,x}$$
$$=\qquad (\text{Lemma } 8)$$
$$\llbracket Q[x := M] \rrbracket_{\Delta}$$

4. (right)

$$\llbracket \mathbf{let}\ x \Leftarrow P\ \mathbf{in}\ [x] \rrbracket_{\Delta}$$
$$=\qquad (\text{def } \llbracket - \rrbracket)$$
$$(\text{arr id} \;\&\&\!\&\; \llbracket P \rrbracket_{\Delta}) \ggg \text{arr snd}$$
$$=\qquad (\text{def } \&\&\!\&)$$
$$\text{arr dup} \ggg \texttt{first}\, (\text{arr id}) \ggg \texttt{second}\, \llbracket P \rrbracket_{\Delta} \ggg \text{arr snd}$$
$$=\qquad (\leadsto_5, \text{id} \times \text{id} = \text{id})$$
$$\text{arr dup} \ggg \text{arr id} \ggg \texttt{second}\, \llbracket P \rrbracket_{\Delta} \ggg \text{arr snd}$$
$$=\qquad (\leadsto_1)$$
$$\text{arr dup} \ggg \texttt{second}\, \llbracket P \rrbracket_{\Delta} \ggg \text{arr snd}$$
$$=\qquad (\text{def second})$$
$$\text{arr dup} \ggg \text{arr swap} \ggg \texttt{first}\, \llbracket P \rrbracket_{\Delta} \ggg \text{arr swap} \ggg \text{arr snd}$$
$$=\qquad (\leadsto_4, \text{swap} \cdot \text{dup} = \text{dup}, \text{swap} \cdot \text{snd} = \text{fst})$$
$$\text{arr dup} \ggg \texttt{first}\, \llbracket P \rrbracket_{\Delta} \ggg \text{arr fst}$$
$$=\qquad (\leadsto_8)$$
$$\text{arr dup} \ggg \text{arr fst} \ggg \llbracket P \rrbracket_{\Delta}$$
$$=\qquad (\leadsto_4, \text{fst} \cdot \text{dup} = \text{id})$$
$$\text{arr id} \ggg \llbracket P \rrbracket_{\Delta}$$
$$=\qquad (\leadsto_1)$$
$$\llbracket P \rrbracket_{\Delta}$$

5. (assoc)

$$\llbracket \textbf{let}\ y \Leftarrow \textbf{let}\ x \Leftarrow \texttt{P}\ \textbf{in}\ \texttt{Q}\ \textbf{in}\ \texttt{R} \rrbracket_\Delta$$

$=$ $(\text{def}\ \llbracket - \rrbracket_\Delta)$

$\texttt{arr dup} \ggg \texttt{first}\ (\texttt{arr dup} \ggg \texttt{first}\ \llbracket \texttt{P} \rrbracket_\Delta \ggg \texttt{arr swap} \ggg \llbracket \texttt{Q} \rrbracket_{\Delta,\texttt{x}})$
   $\ggg \texttt{arr swap} \ggg \llbracket \texttt{R} \rrbracket_{\Delta,\texttt{x}}$

$=$ $(\leadsto_6)$

$\texttt{arr dup} \ggg \texttt{first}\ (\texttt{arr dup}) \ggg \texttt{first}\ (\texttt{first}\ \llbracket \texttt{P} \rrbracket_\Delta) \ggg \texttt{first}\ (\texttt{arr swap})$
   $\ggg \texttt{first}\ \llbracket \texttt{Q} \rrbracket_{\Delta,\texttt{x}} \ggg \texttt{arr swap} \ggg \llbracket \texttt{R} \rrbracket_{\Delta,\texttt{x}}$

$=$ $(\leadsto_5)$

$\texttt{arr dup} \ggg \texttt{arr}\ (\texttt{dup} \times \texttt{id}) \ggg \texttt{first}\ (\texttt{first}\ \llbracket \texttt{P} \rrbracket_\Delta) \ggg \texttt{arr}\ (\texttt{swap} \times \texttt{id})$
   $\ggg \texttt{first}\ \llbracket \texttt{Q} \rrbracket_{\Delta,\texttt{x}} \ggg \texttt{arr swap} \ggg \llbracket \texttt{R} \rrbracket_{\Delta,\texttt{x}}$

$=$ $(\leadsto_4)$

$\texttt{arr}\ (\texttt{dup} \times \texttt{id} \cdot \texttt{dup}) \ggg \texttt{first}\ (\texttt{first}\ \llbracket \texttt{P} \rrbracket_\Delta) \ggg \texttt{arr}\ (\texttt{swap} \times \texttt{id})$
   $\ggg \texttt{first}\ \llbracket \texttt{Q} \rrbracket_{\Delta,\texttt{x}} \ggg \texttt{arr swap} \ggg \llbracket \texttt{R} \rrbracket_{\Delta,\texttt{x}}$

$=$ $(\texttt{swap} \times \texttt{id} = (\lambda\langle \texttt{x}, \langle \texttt{y}, \texttt{z} \rangle \rangle.\ \langle \langle \texttt{y}, \texttt{x} \rangle, \texttt{z} \rangle) \cdot \texttt{assoc})$

$\texttt{arr}\ (\texttt{dup} \times \texttt{id} \cdot \texttt{dup}) \ggg \texttt{first}\ (\texttt{first}\ \llbracket \texttt{P} \rrbracket_\Delta)$
   $\ggg \texttt{arr}\ ((\lambda\langle \texttt{x}, \langle \texttt{y}, \texttt{z} \rangle \rangle.\ \langle \langle \texttt{y}, \texttt{x} \rangle, \texttt{z} \rangle) \cdot \texttt{assoc})$
   $\ggg \texttt{first}\ \llbracket \texttt{Q} \rrbracket_{\Delta,\texttt{x}} \ggg \texttt{arr swap} \ggg \llbracket \texttt{R} \rrbracket_{\Delta,\texttt{x}}$

$=$ $(\leadsto_4)$

$\texttt{arr}\ (\texttt{dup} \times \texttt{id} \cdot \texttt{dup}) \ggg \texttt{first}\ (\texttt{first}\ \llbracket \texttt{P} \rrbracket_\Delta) \ggg \texttt{arr assoc}$
   $\ggg \texttt{arr}\ (\lambda\langle \texttt{x}, \langle \texttt{y}, \texttt{z} \rangle \rangle.\ \langle \langle \texttt{y}, \texttt{x} \rangle, \texttt{z} \rangle)$
   $\ggg \texttt{first}\ \llbracket \texttt{Q} \rrbracket_{\Delta,\texttt{x}} \ggg \texttt{arr swap} \ggg \llbracket \texttt{R} \rrbracket_{\Delta,\texttt{x}}$

$=$ $(\leadsto_9)$

$\texttt{arr}\ (\texttt{dup} \times \texttt{id} \cdot \texttt{dup}) \ggg \texttt{arr assoc} \ggg \texttt{first}\ \llbracket \texttt{P} \rrbracket_\Delta$
   $\ggg \texttt{arr}\ (\lambda\langle \texttt{x}, \langle \texttt{y}, \texttt{z} \rangle \rangle.\ \langle \langle \texttt{y}, \texttt{x} \rangle, \texttt{z} \rangle)$
   $\ggg \texttt{first}\ \llbracket \texttt{Q} \rrbracket_{\Delta,\texttt{x}} \ggg \texttt{arr swap} \ggg \llbracket \texttt{R} \rrbracket_{\Delta,\texttt{x}}$

$=$ $(\leadsto_4, \texttt{assoc} \cdot (\texttt{dup} \times \texttt{id}) \cdot \texttt{dup} = (\texttt{id} \times \texttt{dup}) \cdot \texttt{dup})$

$\texttt{arr}\ (\texttt{id} \times \texttt{dup} \cdot \texttt{dup}) \ggg \texttt{first}\ \llbracket \texttt{P} \rrbracket_\Delta \ggg \texttt{arr}\ (\lambda\langle \texttt{x}, \langle \texttt{y}, \texttt{z} \rangle \rangle.\ \langle \langle \texttt{y}, \texttt{x} \rangle, \texttt{z} \rangle)$
   $\ggg \texttt{first}\ \llbracket \texttt{Q} \rrbracket_{\Delta,\texttt{x}} \ggg \texttt{arr swap} \ggg \llbracket \texttt{R} \rrbracket_{\Delta,\texttt{x}}$

$=$ $(\leadsto_4)$

$\texttt{arr dup} \ggg \texttt{arr}\ (\texttt{id} \times \texttt{dup}) \ggg \texttt{first}\ \llbracket \texttt{P} \rrbracket_\Delta \ggg \texttt{arr}\ (\lambda\langle \texttt{x}, \langle \texttt{y}, \texttt{z} \rangle \rangle.\ \langle \langle \texttt{y}, \texttt{x} \rangle, \texttt{z} \rangle)$
   $\ggg \texttt{first}\ \llbracket \texttt{Q} \rrbracket_{\Delta,\texttt{x}} \ggg \texttt{arr swap} \ggg \llbracket \texttt{R} \rrbracket_{\Delta,\texttt{x}}$

$=$ $(\leadsto_7)$

$\texttt{arr dup} \ggg \texttt{first}\ \llbracket \texttt{P} \rrbracket_\Delta \ggg \texttt{arr}\ (\texttt{id} \times \texttt{dup}) \ggg \texttt{arr}\ (\lambda\langle \texttt{x}, \langle \texttt{y}, \texttt{z} \rangle \rangle.\ \langle \langle \texttt{y}, \texttt{x} \rangle, \texttt{z} \rangle)$
   $\ggg \texttt{first}\ \llbracket \texttt{Q} \rrbracket_{\Delta,\texttt{x}} \ggg \texttt{arr swap} \ggg \llbracket \texttt{R} \rrbracket_{\Delta,\texttt{x}}$

$=$ $(\leadsto_4)$

$\texttt{arr dup} \ggg \texttt{first}\ \llbracket \texttt{P} \rrbracket_\Delta \ggg \texttt{arr}\ ((\lambda\langle \texttt{x}, \langle \texttt{y}, \texttt{z} \rangle \rangle.\ \langle \langle \texttt{y}, \texttt{x} \rangle, \texttt{z} \rangle) \cdot (\texttt{id} \times \texttt{dup}))$
   $\ggg \texttt{first}\ \llbracket \texttt{Q} \rrbracket_{\Delta,\texttt{x}} \ggg \texttt{arr swap} \ggg \llbracket \texttt{R} \rrbracket_{\Delta,\texttt{x}}$

$=$ $((\lambda\langle \texttt{x}, \langle \texttt{y}, \texttt{z} \rangle \rangle.\ \langle \langle \texttt{y}, \texttt{x} \rangle, \texttt{z} \rangle) \cdot (\texttt{id} \times \texttt{dup}) = (\texttt{id} \times \texttt{fst}) \cdot (\texttt{dup} \cdot \texttt{swap}))$

$\texttt{arr dup} \ggg \texttt{first}\ \llbracket \texttt{P} \rrbracket_\Delta \ggg \texttt{arr}\ ((\texttt{id} \times \texttt{fst}) \cdot (\texttt{dup} \cdot \texttt{swap}))$
   $\ggg \texttt{first}\ \llbracket \texttt{Q} \rrbracket_{\Delta,\texttt{x}} \ggg \texttt{arr swap} \ggg \llbracket \texttt{R} \rrbracket_{\Delta,\texttt{x}}$

(continued from previous page)

$\text{arr dup} \ggg \text{first } [\![P]\!]_\Delta \ggg \text{arr} \left((\text{id} \times \text{fst}) \cdot (\text{dup} \cdot \text{swap})\right)$
$\qquad \ggg \text{first } [\![Q]\!]_{\Delta,x} \ggg \text{arr swap} \ggg [\![R]\!]_{\Delta,x}$

$= \qquad (\leadsto_4)$

$\text{arr dup} \ggg \text{first } [\![P]\!]_\Delta \ggg \text{arr} (\text{dup} \cdot \text{swap}) \ggg \text{arr} (\text{id} \times \text{fst})$
$\qquad \ggg \text{first } [\![Q]\!]_{\Delta,x} \ggg \text{arr swap} \ggg [\![R]\!]_{\Delta,x}$

$= \qquad (\leadsto_7)$

$\text{arr dup} \ggg \text{first } [\![P]\!]_\Delta \ggg \text{arr} (\text{dup} \cdot \text{swap}) \ggg \text{first } [\![Q]\!]_{\Delta,x}$
$\qquad \ggg \text{arr} (\text{id} \times \text{fst}) \ggg \text{arr swap} \ggg [\![R]\!]_{\Delta,x}$

$= \qquad (\leadsto_4, \text{swap} \cdot (\text{id} \times \text{fst}) = (\lambda\langle\langle\Delta,x\rangle,y\rangle.\langle\Delta,y\rangle) \cdot \text{swap})$

$\text{arr dup} \ggg \text{first } [\![P]\!]_\Delta \ggg \text{arr} (\text{dup} \cdot \text{swap}) \ggg \text{first } [\![Q]\!]_{\Delta,x}$
$\qquad \ggg \text{arr swap} \ggg \text{arr} (\lambda\langle\langle\Delta,x\rangle,y\rangle.\langle y,\Delta\rangle) \ggg [\![R]\!]_{\Delta,x}$

$= \qquad (\text{Lemma } 10)$

$\text{arr dup} \ggg \text{first } [\![P]\!]_\Delta \ggg \text{arr} (\text{dup} \cdot \text{swap}) \ggg \text{first } [\![Q]\!]_{\Delta,x}$
$\qquad \ggg \text{arr swap} \ggg [\![R]\!]_{\Delta,x,y}$

$= \qquad (\leadsto_4)$

$\text{arr dup} \ggg \text{first } [\![P]\!]_\Delta \ggg \text{arr swap} \ggg \text{arr dup} \ggg \text{first } [\![Q]\!]_{\Delta,x}$
$\qquad \ggg \text{arr swap} \ggg [\![R]\!]_{\Delta,x,y}$

$= \qquad (\text{def } [\![-]\!]_\Delta)$

$[\![\textbf{let } x \Leftarrow P \textbf{ in let } x \Leftarrow Q \textbf{ in } R]\!]_\Delta$

A.2.3 The laws of $\mathcal{C}$ follow from the laws of $\mathcal{A}$

For each classic arrows law $M = N$ we must show $\langle\!\langle M \rangle\!\rangle = \langle\!\langle N \rangle\!\rangle$.

1. $(\leadsto_1)$

$\langle\!\langle \text{arr id} \ggg L \rangle\!\rangle$

$= \qquad (\text{def } \langle\!\langle - \rangle\!\rangle)$

$\lambda^\bullet x. \textbf{ let } y \Leftarrow (\lambda^\bullet z. [\text{id } z]) \bullet x \textbf{ in } \langle\!\langle L \rangle\!\rangle \bullet y$

$= \qquad (\beta^\rightarrow)$

$\lambda^\bullet x. \textbf{ let } y \Leftarrow (\lambda^\bullet z. [z]) \bullet x \textbf{ in } \langle\!\langle L \rangle\!\rangle \bullet y$

$= \qquad (\beta^{\leadsto})$

$\lambda^\bullet x. \textbf{ let } y \Leftarrow [x] \textbf{ in } \langle\!\langle L \rangle\!\rangle \bullet y$

$= \qquad (\text{left})$

$\lambda^\bullet x. \langle\!\langle L \rangle\!\rangle \bullet x$

$= \qquad (\eta^{\leadsto})$

$\langle\!\langle L \rangle\!\rangle$

2. $(\leadsto_2)$

$\quad\quad \langle\!| L \ggg \mathtt{arr\ id} |\!\rangle$
$= \quad\quad\quad (\text{def } \langle\!|-|\!\rangle)$
$\quad\quad \lambda^\bullet x.\ \mathbf{let}\ y \Leftarrow \langle\!| L |\!\rangle \bullet x\ \mathbf{in}\ (\lambda^\bullet z.\ [\mathtt{id}\ z]) \bullet y$
$= \quad\quad\quad (\beta^\rightarrow)$
$\quad\quad \lambda^\bullet x.\ \mathbf{let}\ y \Leftarrow \langle\!| L |\!\rangle \bullet x\ \mathbf{in}\ (\lambda^\bullet z.\ [z]) \bullet y$
$= \quad\quad\quad (\beta^{\leadsto})$
$\quad\quad \lambda^\bullet x.\ \mathbf{let}\ y \Leftarrow \langle\!| L |\!\rangle \bullet x\ \mathbf{in}\ [y]$
$= \quad\quad\quad (\text{right})$
$\quad\quad \lambda^\bullet x.\ \langle\!| L |\!\rangle \bullet x$
$= \quad\quad\quad (\eta^{\leadsto})$
$\quad\quad \langle\!| L |\!\rangle$

3. $(\leadsto_3)$

$\quad\quad \langle\!| (L \ggg M) \ggg N |\!\rangle$
$= \quad\quad\quad (\text{def } \langle\!|-|\!\rangle)$
$\quad\quad \lambda^\bullet z.\ \mathbf{let}\ w \Leftarrow (\lambda^\bullet x.\ \mathbf{let}\ y \Leftarrow \langle\!| L |\!\rangle \bullet x\ \mathbf{in}\ \langle\!| M |\!\rangle \bullet y) \bullet z\ \mathbf{in}\ \langle\!| N |\!\rangle \bullet w$
$= \quad\quad\quad (\beta^{\leadsto})$
$\quad\quad \lambda^\bullet z.\ \mathbf{let}\ w \Leftarrow \mathbf{let}\ y \Leftarrow \langle\!| L |\!\rangle \bullet z\ \mathbf{in}\ \langle\!| M |\!\rangle \bullet y\ \mathbf{in}\ \langle\!| N |\!\rangle \bullet w$
$= \quad\quad\quad (\text{assoc})$
$\quad\quad \lambda^\bullet z.\ \mathbf{let}\ y \Leftarrow \langle\!| L |\!\rangle \bullet z\ \mathbf{in}\ \mathbf{let}\ w \Leftarrow \langle\!| M |\!\rangle \bullet y\ \mathbf{in}\ \langle\!| N |\!\rangle \bullet w$
$= \quad\quad\quad (\beta^{\leadsto})$
$\quad\quad \lambda^\bullet z.\ \mathbf{let}\ y \Leftarrow \langle\!| L |\!\rangle \bullet z\ \mathbf{in}\ (\lambda^\bullet z.\ \mathbf{let}\ w \Leftarrow \langle\!| M |\!\rangle \bullet z\ \mathbf{in}\ \langle\!| N |\!\rangle \bullet w) \bullet y$
$= \quad\quad\quad (\text{def } \langle\!|-|\!\rangle)$
$\quad\quad \langle\!| L \ggg (M \ggg N) |\!\rangle$

4. $(\leadsto_4)$

$\quad\quad \langle\!| \mathtt{arr}\ (M \cdot L) |\!\rangle$
$= \quad\quad\quad (\text{def } \langle\!|-|\!\rangle)$
$\quad\quad \lambda^\bullet x.\ [(\langle\!| M |\!\rangle \cdot \langle\!| L |\!\rangle)\ x]$
$= \quad\quad\quad (\text{def } \cdot)$
$\quad\quad \lambda^\bullet x.\ [M\ (L\ x)]$
$= \quad\quad\quad (\text{left})$
$\quad\quad \lambda^\bullet x.\ \mathbf{let}\ y \Leftarrow [L\ x]\ \mathbf{in}\ [M\ y]$
$= \quad\quad\quad (\beta^{\leadsto}, \beta^{\leadsto})$
$\quad\quad \lambda^\bullet x.\ \mathbf{let}\ y \Leftarrow (\lambda^\bullet x.\ [L\ x]) \bullet x\ \mathbf{in}\ (\lambda^\bullet x.\ [M\ x]) \bullet y$
$= \quad\quad\quad (\text{def } \langle\!|-|\!\rangle)$
$\quad\quad \langle\!| \mathtt{arr}\ L \ggg \mathtt{arr}\ M |\!\rangle$

5. $(\leadsto_5)$

$$\langle\!| \texttt{first (arr L)} |\!\rangle$$
$=$ (def $\langle\!|-|\!\rangle$, $\beta^{\leadsto}$)
$$\lambda^{\bullet}\texttt{z}.\ \textbf{let}\ \texttt{x} \Leftarrow [\langle\!| \texttt{L} |\!\rangle\ (\textbf{fst}\ \texttt{z})]\ \textbf{in}\ [\langle \texttt{x}, \textbf{snd}\ \texttt{z}\rangle]$$
$=$ (left)
$$\lambda^{\bullet}\texttt{z}.\ [\langle \langle\!| \texttt{L} |\!\rangle\ (\textbf{fst}\ \texttt{z}), \textbf{snd}\ \texttt{z}\rangle]$$
$=$ (def $\times$)
$$\lambda^{\bullet}\texttt{z}.\ [(\langle\!| \texttt{L} |\!\rangle \times \texttt{id})\ \texttt{z}]$$
$=$ (def $\langle\!|-|\!\rangle$)
$$\langle\!| \texttt{arr (L} \times \texttt{id)} |\!\rangle$$

6. $(\leadsto_6)$

$$\langle\!| \texttt{first (L} \ggg \texttt{M)} |\!\rangle$$
$=$ (def $\langle\!|-|\!\rangle$, $\beta^{\leadsto}$)
$$\lambda^{\bullet}\texttt{z}.\ \textbf{let}\ \texttt{w} \Leftarrow \textbf{let}\ \texttt{y} \Leftarrow \langle\!| \texttt{L} |\!\rangle \bullet (\textbf{fst}\ \texttt{z})\ \textbf{in}\ \langle\!| \texttt{M} |\!\rangle \bullet \texttt{y}\ \textbf{in}\ [\langle \texttt{w}, \textbf{snd}\ \texttt{z}\rangle]$$
$=$ (assoc)
$$\lambda^{\bullet}\texttt{z}.\ \textbf{let}\ \texttt{y} \Leftarrow \langle\!| \texttt{L} |\!\rangle \bullet \textbf{fst}\ \texttt{z}\ \textbf{in}\ \textbf{let}\ \texttt{w} \Leftarrow \langle\!| \texttt{M} |\!\rangle \bullet \texttt{y}\ \textbf{in}\ [\langle \texttt{w}, \textbf{snd}\ \texttt{z}\rangle]$$
$=$ ($\beta_1^{\times}$, $\beta_2^{\times}$)
$$\lambda^{\bullet}\texttt{z}.\ \textbf{let}\ \texttt{y} \Leftarrow \langle\!| \texttt{L} |\!\rangle \bullet \textbf{fst}\ \texttt{z}\ \textbf{in}$$
$$\quad \textbf{let}\ \texttt{w} \Leftarrow \langle\!| \texttt{M} |\!\rangle \bullet \textbf{fst}\ \langle \texttt{y}, \textbf{snd}\ \texttt{z}\rangle\ \textbf{in}$$
$$\quad\quad [\langle \texttt{w}, \textbf{snd}\ \langle \texttt{y}, \textbf{snd}\ \texttt{z}\rangle\rangle]$$
$=$ (left)
$$\lambda^{\bullet}\texttt{z}.\ \textbf{let}\ \texttt{y} \Leftarrow \langle\!| \texttt{L} |\!\rangle \bullet \textbf{fst}\ \texttt{z}\ \textbf{in}$$
$$\quad \textbf{let}\ \texttt{v} \Leftarrow [\langle \texttt{y}, \textbf{snd}\ \texttt{z}\rangle]\ \textbf{in}$$
$$\quad \textbf{let}\ \texttt{w} \Leftarrow \langle\!| \texttt{M} |\!\rangle \bullet \textbf{fst}\ \texttt{v}\ \textbf{in}$$
$$\quad\quad [\langle \texttt{w}, \textbf{snd}\ \texttt{v}\rangle]$$
$=$ (assoc)
$$\lambda^{\bullet}\texttt{z}.\ \textbf{let}\ \texttt{v} \Leftarrow \textbf{let}\ \texttt{y} \Leftarrow \langle\!| \texttt{L} |\!\rangle \bullet \textbf{fst}\ \texttt{z}\ \textbf{in}\ [\langle \texttt{y}, \textbf{snd}\ \texttt{z}\rangle]\ \textbf{in}$$
$$\quad \textbf{let}\ \texttt{w} \Leftarrow \langle\!| \texttt{M} |\!\rangle \bullet \textbf{fst}\ \texttt{v}\ \textbf{in}$$
$$\quad\quad [\langle \texttt{w}, \textbf{snd}\ \texttt{v}\rangle]$$
$=$ (def $\langle\!|-|\!\rangle$, $\beta^{\leadsto}$)
$$\langle\!| \texttt{first L} \ggg \texttt{first M} |\!\rangle$$

7. $(\leadsto_7)$

$$\langle\!| \texttt{first L} \ggg \texttt{arr (id} \times \texttt{M)} |\!\rangle$$
$=$ (def $\langle\!|-|\!\rangle$, $\beta^{\leadsto}$)
$$\lambda^{\bullet}\texttt{x}.\ \textbf{let}\ \texttt{y} \Leftarrow \textbf{let}\ \texttt{v} \Leftarrow \langle\!| \texttt{L} |\!\rangle \bullet \textbf{fst}\ \texttt{x}\ \textbf{in}\ [\langle \texttt{v}, \textbf{snd}\ \texttt{z}\rangle]\ \textbf{in}\ [(\texttt{id} \times \texttt{M})\ \texttt{y}]$$
$=$ (assoc)
$$\lambda^{\bullet}\texttt{x}.\ \textbf{let}\ \texttt{v} \Leftarrow \langle\!| \texttt{L} |\!\rangle \bullet \textbf{fst}\ \texttt{x}\ \textbf{in}\ \textbf{let}\ \texttt{y} \Leftarrow [\langle \texttt{v}, \textbf{snd}\ \texttt{z}\rangle]\ \textbf{in}\ [(\texttt{id} \times \texttt{M})\ \texttt{y}]$$

(continued on next page)

(continued from previous page)

$\lambda^{\bullet}$x. **let** v $\Leftarrow \langle\!|$L$|\!\rangle \bullet$ **fst** x **in let** y $\Leftarrow [\langle$v, **snd** z$\rangle]$ **in** $[(\mathrm{id} \times$ M$)$ y$]$

=   (left)

$\lambda^{\bullet}$x. **let** v $\Leftarrow \langle\!|$L$|\!\rangle \bullet$ **fst** x **in** $[(\mathrm{id} \times$ M$) \langle$v, **snd** z$\rangle]$

=   (def $\times$)

$\lambda^{\bullet}$x. **let** v $\Leftarrow \langle\!|$L$|\!\rangle \bullet$ **fst** $((\mathrm{id} \times$ M$)$ x$)$ **in** $[\langle$v, **snd** $((\mathrm{id} \times$ M$)$ x$)\rangle]$

=   (left)

$\lambda^{\bullet}$x. **let** y $\Leftarrow [(\mathrm{id} \times$ M$)$ x$]$ **in let** v $\Leftarrow \langle\!|$L$|\!\rangle \bullet$ **fst** y **in** $[\langle$v, **snd** y$\rangle]$

=   (def $\langle\!|-|\!\rangle, \beta^{\rightsquigarrow}$)

$\langle\!|$ **arr** $(\mathrm{id} \times$ M$) \ggg$ **first** L$|\!\rangle$

8. ($\rightsquigarrow_8$)

$\langle\!|$ **first** L $\ggg$ **arr fst**$|\!\rangle$

=   (def $\langle\!|-|\!\rangle$)

$\lambda^{\bullet}$x. **let** y $\Leftarrow (\lambda^{\bullet}$z. **let** w $\Leftarrow \langle\!|$L$|\!\rangle \bullet$ **fst** z **in** $[\langle$w, **snd** z$\rangle]) \bullet$ x **in**
     $(\lambda^{\bullet}$z. $[$**fst** z$]) \bullet$ y

=   ($\beta^{\rightsquigarrow}$)

$\lambda^{\bullet}$x. **let** y $\Leftarrow$ **let** w $\Leftarrow \langle\!|$L$|\!\rangle \bullet$ **fst** x **in** $[\langle$w, **snd** x$\rangle]$ **in** $[$**fst** y$]$

=   (assoc)

$\lambda^{\bullet}$x. **let** w $\Leftarrow \langle\!|$L$|\!\rangle \bullet$ **fst** x **in let** y $\Leftarrow [\langle$w, **snd** x$\rangle]$ **in** $[$**fst** y$]$

=   (left)

$\lambda^{\bullet}$x. **let** w $\Leftarrow \langle\!|$L$|\!\rangle \bullet$ **fst** x **in** $[$**fst** $\langle$w, **snd** x$\rangle]$

=   ($\beta_1^{\times}$)

$\lambda^{\bullet}$x. **let** w $\Leftarrow \langle\!|$L$|\!\rangle \bullet$ **fst** x **in** $[$w$]$

=   (right)

$\lambda^{\bullet}$x. $\langle\!|$L$|\!\rangle \bullet$ **fst** x

=   (left)

$\lambda^{\bullet}$x. **let** y $\Leftarrow [$x$]$ **in** $\langle\!|$L$|\!\rangle \bullet$ y

=   ($\beta^{\rightsquigarrow}$)

$\lambda^{\bullet}$x. **let** y $\Leftarrow (\lambda^{\bullet}$z. $[$**fst** z$]) \bullet$ x **in** $\langle\!|$L$|\!\rangle \bullet$ y

=   (def $\langle\!|-|\!\rangle$)

$\langle\!|$ **arr fst** $\ggg$ L$|\!\rangle$

9. ($\rightsquigarrow_9$)

$\langle\!|$ **first** $($**first** L$) \ggg$ **arr assoc**$|\!\rangle$

=   (def $\langle\!|-|\!\rangle, \beta^{\rightsquigarrow}$)

$\lambda^{\bullet}$a. **let** b $\Leftarrow$ **let** x $\Leftarrow$ **let** r $\Leftarrow \langle\!|$L$|\!\rangle \bullet$ **fst** $($**fst** a$)$ **in**
                             $[\langle$r, **snd** $($**fst** a$)\rangle]$ **in** $[\langle$x, **snd** a$\rangle]$ **in**
         $[$**assoc** b$]$

(continued on next page)

(continued from previous page)
$\lambda^{\bullet}$a. **let** b $\Leftarrow$ **let** x $\Leftarrow$ **let** r $\Leftarrow$ $\langle\!\langle$L$\rangle\!\rangle$ $\bullet$ **fst** (**fst** a) **in**
$\qquad\qquad\qquad\qquad$ [$\langle$r, **snd** (**fst** a)$\rangle$] **in** [$\langle$x, **snd** a$\rangle$] **in**
$\qquad$ [assoc b]
=$\qquad$ (assoc)
$\lambda^{\bullet}$a. **let** r $\Leftarrow$ $\langle\!\langle$L$\rangle\!\rangle$ $\bullet$ **fst** (**fst** a) **in**
$\qquad$ **let** x $\Leftarrow$ [$\langle$r, **snd** (**fst** a)$\rangle$] **in**
$\qquad$ **let** b $\Leftarrow$ [$\langle$x, **snd** a$\rangle$] **in**
$\qquad$ [assoc b]
=$\qquad$ (left)
$\lambda^{\bullet}$a. **let** r $\Leftarrow$ $\langle\!\langle$L$\rangle\!\rangle$ $\bullet$ **fst** (**fst** a) **in** [assoc $\langle\langle$r, **snd** (**fst** a)$\rangle$, **snd** a$\rangle$]
=$\qquad$ (def assoc)
$\lambda^{\bullet}$a. **let** r $\Leftarrow$ $\langle\!\langle$L$\rangle\!\rangle$ $\bullet$ **fst** (**fst** a) **in** [$\langle$r, $\langle$**snd** (**fst** a), **snd** a$\rangle\rangle$]
=$\qquad$ (fst (assoc a) = fst (fst a), $\langle$**snd** (**fst** a), **snd** a$\rangle$ = **snd** (assoc a))
$\lambda^{\bullet}$a. **let** r $\Leftarrow$ $\langle\!\langle$L$\rangle\!\rangle$ $\bullet$ **fst** (assoc a) **in** [$\langle$r, **snd** (assoc a)$\rangle$]
=$\qquad$ (left)
$\lambda^{\bullet}$a. **let** b $\Leftarrow$ [assoc a] **in let** r $\Leftarrow$ $\langle\!\langle$L$\rangle\!\rangle$ $\bullet$ **fst** b **in** [$\langle$r, **snd** b$\rangle$]
=$\qquad$ (def $\langle\!\langle-\rangle\!\rangle$, $\beta^{\rightsquigarrow}$)
$\langle\!\langle$arr assoc $\ggg$ first L$\rangle\!\rangle$

A.2.4   Translating $\mathcal{A}$ to $\mathcal{C}$ and back

For each arrow calculus term M, $\langle\!\langle \llbracket M \rrbracket \rangle\!\rangle$ = M. For each arrow calculus command P, $\langle\!\langle \llbracket P \rrbracket_{\Delta} \rangle\!\rangle \bullet \Delta$ = P. The proof is by mutual induction on the derivations of M and P.

1. Case $\lambda^{\bullet}$x. Q

$\qquad$ $\langle\!\langle \llbracket \lambda^{\bullet}\text{x. Q} \rrbracket \rangle\!\rangle$
=$\qquad$ (def $\llbracket - \rrbracket$)
$\qquad$ $\langle\!\langle \llbracket \text{Q} \rrbracket_{\text{x}} \rangle\!\rangle$
=$\qquad$ (induction hypothesis)
$\qquad$ $\lambda^{\bullet}$x. Q

2. Case L $\bullet$ M

$\qquad$ $\langle\!\langle \llbracket \text{L} \bullet \text{M} \rrbracket_{\Delta} \rangle\!\rangle$
=$\qquad$ (def $\llbracket - \rrbracket_{\Delta}$)
$\qquad$ $\langle\!\langle$arr $(\lambda\Delta. \llbracket \text{M} \rrbracket) \ggg \llbracket \text{L} \rrbracket\rangle\!\rangle$
=$\qquad$ (def $\langle\!\langle - \rangle\!\rangle$)
$\qquad$ $\lambda^{\bullet}\Delta$. **let** y $\Leftarrow$ $(\lambda^{\bullet}$z. [$(\lambda\Delta. \langle\!\langle \llbracket \text{M} \rrbracket \rangle\!\rangle)$ z]) $\bullet$ Δ **in** $\langle\!\langle \llbracket \text{L} \rrbracket \rangle\!\rangle$ $\bullet$ y

(continued on next page)

(continued from previous page)
$$\lambda^\bullet \Delta.\ \textbf{let } y \Leftarrow (\lambda^\bullet z.\ [(\lambda \Delta.\ \langle\!\langle [\![M]\!] \rangle\!\rangle)\ z]) \bullet \Delta \textbf{ in } \langle\!\langle [\![L]\!] \rangle\!\rangle \bullet y$$
$=$ (induction hypothesis)
$$\lambda^\bullet \Delta.\ \textbf{let } y \Leftarrow (\lambda^\bullet z.\ [(\lambda \Delta.\ \texttt{M})\ z]) \bullet \Delta \textbf{ in } L \bullet y$$
$=$ $(\beta^{\rightsquigarrow}, \beta^{\rightarrow})$
$$\lambda^\bullet \Delta.\ \textbf{let } y \Leftarrow [\texttt{M}] \textbf{ in } L \bullet y$$
$=$ (left)
$$\lambda^\bullet \Delta.\ L \bullet \texttt{M}$$

3. Case $[\texttt{M}]$

$$\langle\!\langle [\![\texttt{M}]\!]_\Delta \rangle\!\rangle$$
$=$ $(\text{def } [\![-]\!]_\Delta)$
$$\langle\!\langle \texttt{arr } (\lambda \Delta.\ [\![\texttt{M}]\!]) \rangle\!\rangle$$
$=$ $(\text{def } \langle\!\langle -\rangle\!\rangle)$
$$\lambda^\bullet \Delta.\ [(\lambda \Delta.\ \langle\!\langle [\![\texttt{M}]\!] \rangle\!\rangle)\ \Delta]$$
$=$ (induction hypothesis)
$$\lambda^\bullet \Delta.\ [(\lambda \Delta.\ \texttt{M})\ \Delta]$$
$=$ $(\beta^{\rightarrow})$
$$\lambda^\bullet \Delta.\ [\texttt{M}]$$

4. Case $\textbf{let } x \Leftarrow P \textbf{ in } Q$

$$\langle\!\langle [\![\textbf{let } x \Leftarrow P \textbf{ in } Q]\!] \rangle\!\rangle$$
$=$ $(\text{def } [\![-]\!])$
$$\langle\!\langle (\texttt{arr id \&\&\& } [\![P]\!]_\Delta) \ggg [\![Q]\!]_{\Delta,x} \rangle\!\rangle$$
$=$ $(\text{def } \&\&\&)$
$$\langle\!\langle \texttt{arr dup} \ggg \texttt{first (arr id)} \ggg \texttt{arr swap} \ggg \texttt{first } [\![P]\!]_\Delta \ggg \texttt{arr swap}$$
$$\ggg [\![Q]\!]_{\Delta,x} \rangle\!\rangle$$
$=$ $(\rightsquigarrow_5)$
$$\langle\!\langle \texttt{arr dup} \ggg \texttt{arr (id} \times \texttt{id)} \ggg \texttt{arr swap} \ggg \texttt{first } [\![P]\!]_\Delta \ggg \texttt{arr swap}$$
$$\ggg [\![Q]\!]_{\Delta,x} \rangle\!\rangle$$
$=$ $(\text{id} \times \text{id} = \text{id})$
$$\langle\!\langle \texttt{arr dup} \ggg \texttt{arr id} \ggg \texttt{arr swap} \ggg \texttt{first } [\![P]\!]_\Delta \ggg \texttt{arr swap} \ggg [\![Q]\!]_{\Delta,x} \rangle\!\rangle$$
$=$ $(\rightsquigarrow_1)$
$$\langle\!\langle \texttt{arr dup} \ggg \texttt{arr swap} \ggg \texttt{first } [\![P]\!]_\Delta \ggg \texttt{arr swap} \ggg [\![Q]\!]_{\Delta,x} \rangle\!\rangle$$
$=$ $(\rightsquigarrow_4, \text{swap} \cdot \text{dup} = \text{dup})$
$$\langle\!\langle \texttt{arr dup} \ggg \texttt{first } [\![P]\!]_\Delta \ggg \texttt{arr swap} \ggg [\![Q]\!]_{\Delta,x} \rangle\!\rangle$$
$=$ $(\text{def } \langle\!\langle -\rangle\!\rangle, \text{def dup}, \beta^{\rightsquigarrow})$
$$\lambda^\bullet \Delta.\ \textbf{let } x \Leftarrow [\langle \Delta, \Delta \rangle] \textbf{ in let } w \Leftarrow \textbf{let } v \Leftarrow \langle\!\langle [\![P]\!]_\Delta \rangle\!\rangle \bullet \textbf{fst } x \textbf{ in } [\langle v, \textbf{snd } x \rangle] \textbf{ in}$$
$$\textbf{let } y \Leftarrow [\texttt{swap } w] \textbf{ in } \langle\!\langle [\![Q]\!]_{\Delta,x} \rangle\!\rangle \bullet y$$

(continued on next page)

(continued from previous page)

$\lambda^\bullet \Delta.\ \mathbf{let}\ x \Leftarrow [\langle \Delta, \Delta \rangle]\ \mathbf{in\ let}\ w \Leftarrow \mathbf{let}\ v \Leftarrow \langle\!\langle [\![P]\!]_\Delta \rangle\!\rangle \bullet \mathbf{fst}\ x\ \mathbf{in}\ [\langle v, \mathbf{snd}\ x \rangle]\ \mathbf{in}$
$\qquad \mathbf{let}\ y \Leftarrow [\mathrm{swap}\ w]\ \mathbf{in}\ \langle\!\langle [\![Q]\!]_{\Delta,x} \rangle\!\rangle \bullet y$

$=\qquad (\mathrm{left}, \beta_1^\times, \beta_2^\times)$

$\lambda^\bullet \Delta.\ \mathbf{let}\ w \Leftarrow \mathbf{let}\ v \Leftarrow \langle\!\langle [\![P]\!]_\Delta \rangle\!\rangle \bullet \Delta\ \mathbf{in}\ [\langle v, \Delta \rangle]\ \mathbf{in\ let}\ y \Leftarrow [\mathrm{swap}\ w]\ \mathbf{in}\ \langle\!\langle [\![Q]\!]_{\Delta,x} \rangle\!\rangle \bullet y$

$=\qquad (\mathrm{assoc})$

$\lambda^\bullet \Delta.\ \mathbf{let}\ v \Leftarrow \langle\!\langle [\![P]\!]_\Delta \rangle\!\rangle \bullet \Delta\ \mathbf{in\ let}\ w \Leftarrow [\langle v, \Delta \rangle]\ \mathbf{in\ let}\ y \Leftarrow [\mathrm{swap}\ w]\ \mathbf{in}\ \langle\!\langle [\![Q]\!]_{\Delta,x} \rangle\!\rangle \bullet y$

$=\qquad (\mathrm{left}, \mathrm{def\ swap})$

$\lambda^\bullet \Delta.\ \mathbf{let}\ v \Leftarrow \langle\!\langle [\![P]\!]_\Delta \rangle\!\rangle \bullet \Delta\ \mathbf{in}\ \langle\!\langle [\![Q]\!]_{\Delta,x} \rangle\!\rangle \bullet \langle \Delta, v \rangle$

$=\qquad (\mathrm{induction\ hypothesis})$

$\lambda^\bullet \Delta.\ \mathbf{let}\ v \Leftarrow (\lambda^\bullet \Delta.\ P) \bullet \Delta\ \mathbf{in}\ (\lambda^\bullet \langle \Delta, x \rangle.\ Q) \bullet \langle \Delta, v \rangle$

$=\qquad (\beta^{\leadsto})$

$\lambda^\bullet \Delta.\ \mathbf{let}\ v \Leftarrow P\ \mathbf{in}\ Q$

A.2.5 Translating $\mathcal{C}$ to $\mathcal{A}$ and back

For each classic arrows term M, $[\![\langle\!\langle M \rangle\!\rangle]\!] = M$. The proof is by induction on the derivation of M. There are three cases, one for each constant of $\mathcal{C}$.

1. Case arr

$\qquad [\![\langle\!\langle \mathrm{arr} \rangle\!\rangle]\!]$

$=\qquad (\mathrm{def}\ \langle\!\langle - \rangle\!\rangle, \beta^{\rightarrow})$

$\qquad [\![\lambda f.\ \lambda^\bullet x.\ [f\ x]]\!]$

$=\qquad (\mathrm{def}\ [\![-]\!])$

$\qquad \lambda f.\ \mathrm{arr}\ (\lambda x.\ f\ x)$

$=\qquad (\eta^{\rightarrow})$

$\qquad \mathrm{arr}$

2. Case $(\ggg)$

$\qquad [\![\langle\!\langle (\ggg) \rangle\!\rangle]\!]$

$=\qquad (\mathrm{def}\ \langle\!\langle - \rangle\!\rangle, \beta^{\rightarrow})$

$\qquad [\![\lambda f.\ \lambda g.\ \lambda^\bullet x.\ \mathbf{let}\ y \Leftarrow f \bullet x\ \mathbf{in}\ g \bullet y]\!]$

$=\qquad (\mathrm{def}\ [\![-]\!], \leadsto_1)$

$\qquad \lambda f.\ \lambda g.\ (\mathrm{arr}\ \mathrm{id}\ \&\!\&\!\& f) \ggg \mathrm{arr}\ \mathrm{snd} \ggg g$

$=\qquad (\mathrm{def}\ \&\!\&\!\&)$

$\qquad \lambda f.\ \lambda g.\ \mathrm{arr}\ \mathrm{dup} \ggg \mathrm{first}\ (\mathrm{arr}\ \mathrm{id}) \ggg \mathrm{arr}\ \mathrm{swap} \ggg \mathrm{first}\ f \ggg \mathrm{arr}\ \mathrm{swap}$
$\qquad\quad \ggg \mathrm{arr}\ \mathrm{snd} \ggg g$

(continued on next page)

(continued from previous page)

$\lambda\mathtt{f}.\,\lambda\mathtt{g}.\,\mathrm{arr\ dup} \ggg \mathtt{first}\,(\mathrm{arr\ id}) \ggg \mathrm{arr\ swap} \ggg \mathtt{first\ f} \ggg \mathrm{arr\ swap}$
    $\ggg \mathrm{arr\ snd} \ggg \mathtt{g}$

$=$ $(\leadsto_5)$

$\lambda\mathtt{f}.\,\lambda\mathtt{g}.\,\mathrm{arr\ dup} \ggg \mathrm{arr}\,(\mathrm{id} \times \mathrm{id}) \ggg \mathrm{arr\ swap} \ggg \mathtt{first\ f} \ggg \mathrm{arr\ swap}$
    $\ggg \mathrm{arr\ snd} \ggg \mathtt{g}$

$=$ $(\mathrm{id} \times \mathrm{id} = \mathrm{id})$

$\lambda\mathtt{f}.\,\lambda\mathtt{g}.\,\mathrm{arr\ dup} \ggg \mathrm{arr\ id} \ggg \mathrm{arr\ swap} \ggg \mathtt{first\ f} \ggg \mathrm{arr\ swap}$
    $\ggg \mathrm{arr\ snd} \ggg \mathtt{g}$

$=$ $(\leadsto_2)$

$\lambda\mathtt{f}.\,\lambda\mathtt{g}.\,\mathrm{arr\ dup} \ggg \mathrm{arr\ swap} \ggg \mathtt{first\ f} \ggg \mathrm{arr\ swap} \ggg \mathrm{arr\ snd} \ggg \mathtt{g}$

$=$ $(\leadsto_4, \mathrm{swap} \cdot \mathrm{dup} = \mathrm{dup}, \mathrm{snd} \cdot \mathrm{swap} = \mathrm{fst})$

$\lambda\mathtt{f}.\,\lambda\mathtt{g}.\,\mathrm{arr\ dup} \ggg \mathtt{first\ f} \ggg \mathrm{arr\ fst} \ggg \mathtt{g}$

$=$ $(\leadsto_8)$

$\lambda\mathtt{f}.\,\lambda\mathtt{g}.\,\mathrm{arr\ dup} \ggg \mathrm{arr\ fst} \ggg \mathtt{f} \ggg \mathtt{g}$

$=$ $(\leadsto_4, \mathrm{fst} \cdot \mathrm{dup} = \mathrm{id})$

$\lambda\mathtt{f}.\,\lambda\mathtt{g}.\,\mathrm{arr\ id} \ggg \mathtt{f} \ggg \mathtt{g}$

$=$ $(\leadsto_1, \eta^{\rightarrow})$

    $(\ggg)$

3. Case `first`

    $[\![\langle\!|\mathtt{first}|\!\rangle]\!]$

$=$ $(\mathrm{def}\ \langle\!|-|\!\rangle, \beta^{\rightarrow})$

    $[\![\lambda\mathtt{f}.\,\lambda^{\bullet}\mathtt{z}.\,\mathbf{let}\ \mathtt{x} \Leftarrow \mathtt{f} \bullet \mathbf{fst}\ \mathtt{z}\ \mathbf{in}\ [\langle\mathtt{x}, \mathbf{snd}\,\mathtt{z}\rangle]]\!]$

$=$ $(\mathrm{def}\ [\![-]\!], \mathrm{def\ fst}, \mathrm{def}\ \&\!\&\!\&)$

$\lambda\mathtt{f}.\,\mathrm{arr\ dup} \ggg \mathtt{first}\,(\mathrm{arr\ id}) \ggg \mathrm{arr\ swap} \ggg \mathtt{first}\,(\mathrm{arr\ fst} \ggg \mathtt{f})$
    $\ggg \mathrm{arr\ swap} \ggg \mathrm{arr}\,(\lambda\mathtt{v}.\,\langle\mathbf{snd}\,\mathtt{v}, \mathbf{snd}\,(\mathbf{fst}\,\mathtt{v})\rangle)$

$=$ $(\leadsto_6)$

$\lambda\mathtt{f}.\,\mathrm{arr\ dup} \ggg \mathtt{first}\,(\mathrm{arr\ id}) \ggg \mathrm{arr\ swap} \ggg \mathtt{first}\,(\mathrm{arr\ fst}) \ggg \mathtt{first\ f}$
    $\ggg \mathrm{arr\ swap} \ggg \mathrm{arr}\,(\lambda\mathtt{v}.\,\langle\mathbf{snd}\,\mathtt{v}, \mathbf{snd}\,(\mathbf{fst}\,\mathtt{v})\rangle)$

$=$ $(\leadsto_5)$

$\lambda\mathtt{f}.\,\mathrm{arr\ dup} \ggg \mathrm{arr}\,(\mathrm{id} \times \mathrm{id}) \ggg \mathrm{arr\ swap} \ggg \mathrm{arr}\,(\mathrm{fst} \times \mathrm{id}) \ggg \mathtt{first\ f}$
    $\ggg \mathrm{arr\ swap} \ggg \mathrm{arr}\,(\lambda\mathtt{v}.\,\langle\mathbf{snd}\,\mathtt{v}, \mathbf{snd}\,(\mathbf{fst}\,\mathtt{v})\rangle)$

$=$ $(\leadsto_4, \mathrm{swap} \cdot \mathrm{id} \times \mathrm{id} \cdot \mathrm{dup} = \mathrm{dup}, ((\lambda\mathtt{v}.\,\langle\mathbf{snd}\,\mathtt{v}, \mathbf{snd}\,(\mathbf{fst}\,\mathtt{v})\rangle) \cdot \mathrm{swap}) = \mathrm{id} \times \mathrm{snd})$

$\lambda\mathtt{f}.\,\mathrm{arr}\,(\mathrm{fst} \times \mathrm{id} \cdot \mathrm{dup}) \ggg \mathtt{first\ f} \ggg \mathrm{arr}\,(\mathrm{id} \times \mathrm{snd})$

$=$ $(\leadsto_7)$

$\lambda\mathtt{f}.\,\mathrm{arr}\,(\mathrm{fst} \times \mathrm{id} \cdot \mathrm{dup}) \ggg \mathrm{arr}\,(\mathrm{id} \times \mathrm{snd}) \ggg \mathtt{first\ f}$

$=$ $(\leadsto_4)$

$\lambda\mathtt{f}.\,\mathrm{arr}\,(\mathrm{id} \times \mathrm{snd} \cdot \mathrm{dup}) \ggg \mathtt{first\ f}$

$=$ $(\mathrm{id} \times \mathrm{snd} \cdot \mathrm{fst} \times \mathrm{id} \cdot \mathrm{dup} = \mathrm{id})$

$\lambda\mathtt{f}.\,\mathrm{arr\ id} \ggg \mathtt{first\ f}$

$=$ $(\leadsto_1, \eta^{\rightarrow})$

    `first`

A.3   Equational correspondence between $\mathcal{S}$ and $\mathcal{C}_{\mathbf{S}}$

This section gives a proof of Proposition 12 (page 69). The proofs here extend the proofs of
the equational correspondence between $\mathcal{A}$ and $\mathcal{C}$ in Section A.2.

A.3.1   Extensions of Lemmas 8 and 10

We must first extend Lemmas 8 and 10 to show that the translations of substitution and weak-
ening hold for the **run** L construct. The proof of Lemma 8 for **run** L is as follows:

$$
\begin{aligned}
&\quad [\![(\mathbf{run}\ \mathrm{L})[\mathrm{x} := \mathrm{N}]]\!]_\Delta \\
&= \qquad (\text{def substitution}) \\
&\quad [\![\mathbf{run}\ \mathrm{L}]\!]_\Delta \\
&= \qquad (\text{def } [\![-]\!]_\Delta) \\
&\quad \mathrm{arr}\,(\lambda\Delta.\,\langle\rangle) \ggg \mathrm{delay}\ [\![\mathrm{L}]\!] \\
&= \qquad (\rightsquigarrow_4, \text{def } \cdot) \\
&\quad \mathrm{arr}\,(\lambda\Delta.\,\langle\Delta, [\![\mathrm{N}]\!]\rangle) \ggg \mathrm{arr}\,(\lambda\langle\Delta, \mathrm{x}\rangle.\,\langle\rangle) \ggg \mathrm{delay}\ [\![\mathrm{L}]\!] \\
&= \qquad (\text{def } [\![-]\!]_{\Delta,\mathrm{x}}) \\
&\quad \mathrm{arr}\,(\lambda\Delta.\,\langle\Delta, [\![\mathrm{N}]\!]\rangle) \ggg [\![\mathbf{run}\ \mathrm{L}]\!]_{\Delta,\mathrm{x}}
\end{aligned}
$$

The proof of Lemma 10 for **run** L is as follows:

$$
\begin{aligned}
&\quad [\![\mathbf{run}\ \mathrm{L}]\!]_{\Delta'} \\
&= \qquad (\text{def } [\![-]\!]_{\Delta'}) \\
&\quad \mathrm{arr}\,(\lambda\Delta.\,\langle\rangle) \ggg \mathrm{delay}\ [\![\mathrm{L}]\!] \\
&= \qquad (\rightsquigarrow_4, \text{def } \cdot) \\
&\quad \mathrm{arr}\,(\lambda\Delta'.\,\Delta) \ggg \mathrm{arr}\,(\lambda\Delta.\,\langle\rangle) \ggg \mathrm{delay}\ [\![\mathrm{L}]\!] \\
&= \qquad (\text{def } [\![-]\!]_\Delta) \\
&\quad \mathrm{arr}\,(\lambda\Delta'.\,\Delta) \ggg [\![\mathbf{run}\ \mathrm{L}]\!]_\Delta
\end{aligned}
$$

A.3.2   The laws of $\mathcal{S}$ follow from the laws of $\mathcal{C}_{\mathbf{S}}$

For each static arrows law on terms $\mathrm{M} = \mathrm{N}$ we must show $[\![\mathrm{M}]\!] = [\![\mathrm{N}]\!]$ and for each static
arrows law on commands $\mathrm{P} = \mathrm{Q}$ we must show $[\![\mathrm{P}]\!]_\Delta = [\![\mathrm{Q}]\!]_\Delta$.

1. (run_1)

$\qquad \llbracket \text{L} \bullet \text{M} \rrbracket_\Delta$
$= \qquad (\text{def } \llbracket - \rrbracket_\Delta)$
$\qquad \text{arr } (\lambda\Delta.\, \llbracket \text{M} \rrbracket) \ggg \llbracket \text{L} \rrbracket$
$= \qquad (\leadsto_{S1})$
$\qquad \text{arr } (\lambda\Delta.\, \llbracket \text{M} \rrbracket) \ggg \text{force } (\text{delay } \llbracket \text{L} \rrbracket)$
$= \qquad (\text{def force})$
$\qquad \text{arr } (\lambda\Delta.\, \llbracket \text{M} \rrbracket) \ggg \text{arr } (\lambda\text{a}.\, \langle\langle\rangle, \text{a}\rangle) \ggg \text{first } (\text{delay } \llbracket \text{L} \rrbracket) \ggg \text{arr } (\lambda\langle\text{f}, \text{a}\rangle.\, \text{f a})$
$= \qquad (\leadsto_4, \beta^\rightarrow)$
$\qquad \text{arr } (\lambda\text{a}.\, \langle\langle\rangle, \text{a}\rangle) \ggg \text{arr } (\text{id} \times (\lambda\Delta.\, \llbracket \text{M} \rrbracket)) \ggg \text{first } (\text{delay } \llbracket \text{L} \rrbracket)$
$\qquad \quad \ggg \text{arr } (\lambda\langle\text{f}, \text{a}\rangle.\, \text{f a})$
$= \qquad (\leadsto_7)$
$\qquad \text{arr } (\lambda\text{a}.\, \langle\langle\rangle, \text{a}\rangle) \ggg \text{first } (\text{delay } \llbracket \text{L} \rrbracket)$
$\qquad \quad \ggg \text{arr } (\text{id} \times (\lambda\Delta.\, \llbracket \text{M} \rrbracket)) \ggg \text{arr } (\lambda\langle\text{f}, \text{a}\rangle.\, \text{f a})$
$= \qquad (\leadsto_4)$
$\qquad \text{arr } (\lambda\text{a}.\, \langle\langle\rangle, \text{a}\rangle) \ggg \text{first } (\text{delay } \llbracket \text{L} \rrbracket) \ggg \text{arr } ((\lambda\langle\text{f}, \text{a}\rangle.\, \text{f a}) \cdot (\text{id} \times (\lambda\Delta.\, \llbracket \text{M} \rrbracket)))$
$= \qquad (\beta^\rightarrow)$
$\qquad \text{arr } (\lambda\text{a}.\, \langle\langle\rangle, \text{a}\rangle) \ggg \text{first } (\text{delay } \llbracket \text{L} \rrbracket) \ggg \text{arr } (\lambda\langle\text{f}, \Delta\rangle.\, \text{f } \llbracket \text{M} \rrbracket)$
$= \qquad (\leadsto_4, \beta^\rightarrow)$
$\qquad \text{arr dup} \ggg \text{arr } (\text{id} \times \text{id}) \ggg \text{arr swap} \ggg \text{arr } ((\lambda\Delta.\, \langle\rangle) \times \text{id})$
$\qquad \quad \ggg \text{first } (\text{delay } \llbracket \text{L} \rrbracket) \ggg \text{arr swap} \ggg \text{arr } (\lambda\langle\Delta, \text{f}\rangle.\, \text{f } \llbracket \text{M} \rrbracket)$
$= \qquad (\leadsto_5)$
$\qquad \text{arr dup} \ggg \text{first } (\text{arr id}) \ggg \text{arr swap} \ggg \text{first } (\text{arr } (\lambda\Delta.\, \langle\rangle))$
$\qquad \quad \ggg \text{first } (\text{delay } \llbracket \text{L} \rrbracket) \ggg \text{arr swap} \ggg \text{arr } (\lambda\langle\Delta, \text{f}\rangle.\, \text{f } \llbracket \text{M} \rrbracket)$
$= \qquad (\leadsto_6)$
$\qquad \text{arr dup} \ggg \text{first } (\text{arr id}) \ggg \text{arr swap} \ggg \text{first } (\text{arr } (\lambda\Delta.\, \langle\rangle) \ggg \text{delay } \llbracket \text{L} \rrbracket)$
$\qquad \quad \ggg \text{arr swap} \ggg \text{arr } (\lambda\langle\Delta, \text{f}\rangle.\, \text{f } \llbracket \text{M} \rrbracket)$
$= \qquad (\text{def } \&\&\&)$
$\qquad (\text{arr id} \&\&\& (\text{arr } (\lambda\Delta.\, \langle\rangle) \ggg \text{delay } \llbracket \text{L} \rrbracket)) \ggg \text{arr } (\lambda\langle\Delta, \text{f}\rangle.\, \text{f } \llbracket \text{M} \rrbracket)$
$= \qquad (\text{def } \llbracket - \rrbracket_\Delta)$
$\qquad \llbracket \textbf{let } \text{f} \Leftarrow \textbf{run } \text{L } \textbf{in } [\text{f M}] \rrbracket_\Delta$

2. (run_2)

$\qquad \llbracket \textbf{run } (\lambda^\bullet \text{x}.\, [\text{M}]) \rrbracket_\Delta$
$= \qquad (\text{def } \llbracket - \rrbracket_\Delta)$
$\qquad \text{arr } (\lambda\Delta.\, \langle\rangle) \ggg \text{delay } (\text{arr } (\lambda\text{x}.\, \llbracket \text{M} \rrbracket))$
$= \qquad ((\lambda\text{x}.\, \llbracket \text{M} \rrbracket) = (\text{apply} \cdot ((\lambda\langle\rangle.\, \lambda\text{x}.\, \llbracket \text{M} \rrbracket) \times \text{id}) \cdot (\lambda\text{x}.\, \langle\langle\rangle, \text{x}\rangle)))$
$\qquad \text{arr } (\lambda\Delta.\, \langle\rangle) \ggg \text{delay } (\text{arr } (\text{apply} \cdot ((\lambda\langle\rangle.\, \lambda\text{x}.\, \llbracket \text{M} \rrbracket) \times \text{id}) \cdot (\lambda\text{x}.\, \langle\langle\rangle, \text{x}\rangle)))$
$= \qquad (\leadsto_4)$
$\qquad \text{arr } (\lambda\Delta.\, \langle\rangle) \ggg \text{delay } (\text{arr } (\lambda\text{x}.\, \langle\langle\rangle, \text{x}\rangle) \ggg \text{arr } ((\lambda\langle\rangle.\, \lambda\text{x}.\, \llbracket \text{M} \rrbracket) \times \text{id}) \ggg \text{arr apply})$
$= \qquad (\leadsto_5)$
$\qquad \text{arr } (\lambda\Delta.\, \langle\rangle) \ggg \text{delay } (\text{arr } (\lambda\text{x}.\, \langle\langle\rangle, \text{x}\rangle) \ggg \text{first } (\text{arr } (\lambda\langle\rangle.\, \lambda\text{x}.\, \llbracket \text{M} \rrbracket)) \ggg \text{arr apply})$

(continued on next page)

(continued from previous page)
$\text{arr}\,(\lambda\Delta.\,\langle\rangle) \ggg \text{delay}\,(\text{arr}\,(\lambda x.\,\langle\langle\rangle, x\rangle)) \ggg \text{first}\,(\text{arr}\,(\lambda\langle\rangle.\,\lambda x.\,[\![M]\!])) \ggg \text{arr apply})$

$=\qquad (\text{def force})$

$\text{arr}\,(\lambda\Delta.\,\langle\rangle) \ggg \text{delay}\,(\text{force}\,(\text{arr}\,(\lambda\langle\rangle.\,\lambda x.\,[\![M]\!])))$

$=\qquad (\leadsto_{S2})$

$\text{arr}\,(\lambda\Delta.\,\langle\rangle) \ggg \text{arr}\,(\lambda\langle\rangle.\,\lambda x.\,[\![M]\!])$

$=\qquad (\leadsto_4, \beta^{\rightarrow})$

$\text{arr}\,(\lambda\Delta.\,\lambda x.\,[\![M]\!])$

$=\qquad (\text{def}\,[\![-]\!]_{\Delta})$

$[\![[\lambda x.\,M]]\!]_{\Delta}$

3. (run_3)

$[\![\textbf{run}\,(\lambda^{\bullet}x.\,\textbf{let}\,y \Leftarrow P\,\textbf{in}\,Q)]\!]_{\Delta}$

$=\qquad (\text{def}\,[\![-]\!]_{\Delta})$

$\text{arr}\,(\lambda\Delta.\,\langle\rangle) \ggg \text{delay}\,((\text{arr id} \,\&\!\&\!\&\, [\![P]\!]_x) \ggg [\![Q]\!]_{x,y})$

$=\qquad (\text{Lemma 10})$

$\text{arr}\,(\lambda\Delta.\,\langle\rangle) \ggg \text{delay}\,((\text{arr id} \,\&\!\&\!\&\, (\text{arr}\,(\lambda x.\,\langle\rangle) \ggg [\![P]\!]_{\langle\rangle})) \ggg [\![Q]\!]_{x,y})$

$=\qquad (\text{def}\,\&\!\&\!\&)$

$\text{arr}\,(\lambda\Delta.\,\langle\rangle) \ggg \text{delay}\,(\text{arr dup} \ggg \text{first}\,(\text{arr id}) \ggg \text{arr swap}$
$\qquad\qquad\qquad\qquad \ggg \text{first}\,(\text{arr}\,(\lambda x.\,\langle\rangle) \ggg [\![P]\!]_{\langle\rangle} \ggg \text{arr swap}) \ggg [\![Q]\!]_{x,y})$

$=\qquad (\leadsto_6)$

$\text{arr}\,(\lambda\Delta.\,\langle\rangle)$
$\quad \ggg \text{delay}\,(\text{arr dup} \ggg \text{first}\,(\text{arr id}) \ggg \text{arr swap}$
$\qquad\qquad \ggg \text{first}\,(\text{arr}\,(\lambda x.\,\langle\rangle)) \ggg \text{first}\,[\![P]\!]_{\langle\rangle} \ggg \text{arr swap} \ggg [\![Q]\!]_{x,y})$

$=\qquad (\leadsto_5)$

$\text{arr}\,(\lambda\Delta.\,\langle\rangle)$
$\quad \ggg \text{delay}\,(\text{arr dup} \ggg \text{arr}\,(\text{id} \times \text{id}) \ggg \text{arr swap}$
$\qquad\qquad \ggg \text{arr}\,((\lambda x.\,\langle\rangle) \times \text{id}) \ggg \text{first}\,[\![P]\!]_{\langle\rangle} \ggg \text{arr swap} \ggg [\![Q]\!]_{x,y})$

$=\qquad (\leadsto_4, ((\lambda x.\,\langle\rangle) \times \text{id}) \cdot \text{swap} \cdot \text{id} \times \text{id} \cdot \text{dup} = \text{dup})$

$\text{arr}\,(\lambda\Delta.\,\langle\rangle)$
$\quad \ggg \text{delay}\,(\text{arr}\,(\lambda x.\,\langle\langle\rangle, x\rangle) \ggg \text{first}\,[\![P]\!]_{\langle\rangle} \ggg \text{arr swap} \ggg [\![Q]\!]_{x,y})$

$=\qquad (\leadsto_{S1})$

$\text{arr}\,(\lambda\Delta.\,\langle\rangle)$
$\quad \ggg \text{delay}\,(\text{arr}\,(\lambda x.\,\langle\langle\rangle, x\rangle) \ggg \text{first}\,[\![P]\!]_{\langle\rangle} \ggg \text{arr swap}$
$\qquad\qquad \ggg \text{force}\,(\text{delay}\,[\![Q]\!]_{x,y}))$

$=\qquad (\text{def force})$

$\text{arr}\,(\lambda\Delta.\,\langle\rangle)$
$\quad \ggg \text{delay}\,(\text{arr}\,(\lambda x.\,\langle\langle\rangle, x\rangle) \ggg \text{first}\,[\![P]\!]_{\langle\rangle} \ggg \text{arr swap}$
$\qquad\qquad \ggg \text{arr}\,(\lambda x.\,\langle\langle\rangle, x\rangle) \ggg \text{first}\,(\text{delay}\,[\![Q]\!]_{x,y}) \ggg \text{arr}\,(\lambda\langle f, x\rangle.\,f\,x))$

(continued from previous page)

$\mathtt{arr}\ (\lambda\Delta.\,\langle\rangle)$
 $\ggg \mathtt{delay}\ (\mathtt{arr}\ (\lambda\mathtt{x}.\,\langle\langle\rangle,\mathtt{x}\rangle) \ggg \mathtt{first}\ [\![\mathtt{P}]\!]_{\langle\rangle} \ggg \mathtt{arr}\ \mathtt{swap}$
     $\ggg \mathtt{arr}\ (\lambda\mathtt{x}.\,\langle\langle\rangle,\mathtt{x}\rangle) \ggg \mathtt{first}\ (\mathtt{delay}\ [\![\mathtt{Q}]\!]_{\mathtt{x,y}}) \ggg \mathtt{arr}\ (\lambda\langle\mathtt{f},\mathtt{x}\rangle.\,\mathtt{f}\ \mathtt{x}))$

$=$ $(\leadsto_4)$

$\mathtt{arr}\ (\lambda\Delta.\,\langle\rangle)$
 $\ggg \mathtt{delay}\ (\mathtt{arr}\ (\lambda\mathtt{x}.\,\langle\langle\rangle,\mathtt{x}\rangle) \ggg \mathtt{first}\ [\![\mathtt{P}]\!]_{\langle\rangle} \ggg \mathtt{arr}\ (\lambda\langle\mathtt{y},\mathtt{x}\rangle.\,\langle\langle\rangle,\langle\mathtt{x},\mathtt{y}\rangle\rangle)$
     $\ggg \mathtt{first}\ (\mathtt{delay}\ [\![\mathtt{Q}]\!]_{\mathtt{x,y}}) \ggg \mathtt{arr}\ (\lambda\langle\mathtt{f},\mathtt{x}\rangle.\,\mathtt{f}\ \mathtt{x}))$

$=$ $(\leadsto_4)$

$\mathtt{arr}\ (\lambda\Delta.\,\langle\rangle)$
 $\ggg \mathtt{delay}\ (\mathtt{arr}\ (\lambda\mathtt{x}.\,\langle\langle\rangle,\mathtt{x}\rangle) \ggg \mathtt{first}\ [\![\mathtt{P}]\!]_{\langle\rangle} \ggg \mathtt{arr}\ (\lambda\langle\mathtt{y},\mathtt{x}\rangle.\,\langle\langle\langle\rangle,\mathtt{x}\rangle,\mathtt{y}\rangle)$
     $\ggg \mathtt{arr}\ \mathtt{assoc} \ggg \mathtt{first}\ (\mathtt{delay}\ [\![\mathtt{Q}]\!]_{\mathtt{x,y}}) \ggg \mathtt{arr}\ (\lambda\langle\mathtt{f},\mathtt{x}\rangle.\,\mathtt{f}\ \mathtt{x}))$

$=$ $(\leadsto_9)$

$\mathtt{arr}\ (\lambda\Delta.\,\langle\rangle)$
 $\ggg \mathtt{delay}\ (\mathtt{arr}\ (\lambda\mathtt{x}.\,\langle\langle\rangle,\mathtt{x}\rangle) \ggg \mathtt{first}\ [\![\mathtt{P}]\!]_{\langle\rangle} \ggg \mathtt{arr}\ (\lambda\langle\mathtt{y},\mathtt{x}\rangle.\,\langle\langle\langle\rangle,\mathtt{x}\rangle,\mathtt{y}\rangle)$
     $\ggg \mathtt{first}\ (\mathtt{first}\ (\mathtt{delay}\ [\![\mathtt{Q}]\!]_{\mathtt{x,y}})) \ggg \mathtt{arr}\ \mathtt{assoc}$
     $\ggg \mathtt{arr}\ (\lambda\langle\mathtt{f},\mathtt{x}\rangle.\,\mathtt{f}\ \mathtt{x}))$

$=$ $(\leadsto_4)$

$\mathtt{arr}\ (\lambda\Delta.\,\langle\rangle)$
 $\ggg \mathtt{delay}\ (\mathtt{arr}\ (\lambda\mathtt{x}.\,\langle\langle\rangle,\mathtt{x}\rangle) \ggg \mathtt{first}\ [\![\mathtt{P}]\!]_{\langle\rangle} \ggg \mathtt{arr}\ (\lambda\langle\mathtt{y},\mathtt{x}\rangle.\,\langle\langle\langle\rangle,\mathtt{y}\rangle,\mathtt{x}\rangle)$
     $\ggg \mathtt{first}\ (\mathtt{first}\ (\mathtt{delay}\ [\![\mathtt{Q}]\!]_{\mathtt{x,y}})) \ggg \mathtt{arr}\ (\lambda\langle\langle\mathtt{f},\mathtt{y}\rangle,\mathtt{x}\rangle.\,(\mathtt{f}\ \langle\mathtt{x},\mathtt{y}\rangle)))$

$=$ $(\leadsto_5)$

$\mathtt{arr}\ (\lambda\Delta.\,\langle\rangle)$
 $\ggg \mathtt{delay}\ (\mathtt{arr}\ (\lambda\mathtt{x}.\,\langle\langle\rangle,\mathtt{x}\rangle) \ggg \mathtt{first}\ [\![\mathtt{P}]\!]_{\langle\rangle} \ggg \mathtt{first}\ (\mathtt{arr}\ (\lambda\mathtt{y}.\,\langle\langle\rangle,\mathtt{y}\rangle))$
     $\ggg \mathtt{first}\ (\mathtt{first}\ (\mathtt{delay}\ [\![\mathtt{Q}]\!]_{\mathtt{x,y}})) \ggg \mathtt{arr}\ (\lambda\langle\langle\mathtt{f},\mathtt{y}\rangle,\mathtt{x}\rangle.\,(\mathtt{f}\ \langle\mathtt{x},\mathtt{y}\rangle)))$

$=$ $(\leadsto_6)$

$\mathtt{arr}\ (\lambda\Delta.\,\langle\rangle)$
 $\ggg \mathtt{delay}\ (\mathtt{arr}\ (\lambda\mathtt{x}.\,\langle\langle\rangle,\mathtt{x}\rangle)$
    $\ggg \mathtt{first}\ ([\![\mathtt{P}]\!]_{\langle\rangle} \ggg \mathtt{arr}\ (\lambda\mathtt{y}.\,\langle\langle\rangle,\mathtt{y}\rangle) \ggg \mathtt{first}\ (\mathtt{delay}\ [\![\mathtt{Q}]\!]_{\mathtt{x,y}}))$
    $\ggg \mathtt{arr}\ (\lambda\langle\langle\mathtt{f},\mathtt{y}\rangle,\mathtt{x}\rangle.\,(\mathtt{f}\ \langle\mathtt{x},\mathtt{y}\rangle)))$

$=$ $(\leadsto_5,\leadsto_4)$

$\mathtt{arr}\ (\lambda\Delta.\,\langle\rangle) \ggg \mathtt{delay}\ (\mathtt{arr}\ (\lambda\mathtt{x}.\,\langle\langle\rangle,\mathtt{x}\rangle)$
    $\ggg \mathtt{first}\ ([\![\mathtt{P}]\!]_{\langle\rangle} \ggg \mathtt{arr}\ (\lambda\mathtt{y}.\,\langle\langle\rangle,\mathtt{y}\rangle) \ggg \mathtt{first}\ (\mathtt{delay}\ [\![\mathtt{Q}]\!]_{\mathtt{x,y}}))$
    $\ggg \mathtt{first}\ (\mathtt{arr}\ (\lambda\langle\mathtt{f},\mathtt{y}\rangle.\,\lambda\mathtt{x}.\,\mathtt{f}\ \langle\mathtt{x},\mathtt{y}\rangle)) \ggg \mathtt{arr}\ (\lambda\langle\mathtt{f},\mathtt{a}\rangle.\,\mathtt{f}\ \mathtt{a}))$

$=$ $(\leadsto_6)$

$\mathtt{arr}\ (\lambda\Delta.\,\langle\rangle) \ggg \mathtt{delay}\ (\mathtt{arr}\ (\lambda\mathtt{x}.\,\langle\langle\rangle,\mathtt{x}\rangle)$
    $\ggg \mathtt{first}\ ([\![\mathtt{P}]\!]_{\langle\rangle} \ggg \mathtt{arr}\ (\lambda\mathtt{y}.\,\langle\langle\rangle,\mathtt{y}\rangle) \ggg \mathtt{first}\ (\mathtt{delay}\ [\![\mathtt{Q}]\!]_{\mathtt{x,y}})$
       $\ggg \mathtt{arr}\ (\lambda\langle\mathtt{f},\mathtt{y}\rangle.\,\lambda\mathtt{x}.\,\mathtt{f}\ \langle\mathtt{x},\mathtt{y}\rangle)) \ggg \mathtt{arr}\ (\lambda\langle\mathtt{f},\mathtt{a}\rangle.\,\mathtt{f}\ \mathtt{a}))$

$=$ $(\mathrm{def}\ \mathtt{force})$

$\mathtt{arr}\ (\lambda\Delta.\,\langle\rangle) \ggg \mathtt{delay}\ (\mathtt{force}\ ([\![\mathtt{P}]\!]_{\langle\rangle} \ggg \mathtt{arr}\ (\lambda\mathtt{y}.\,\langle\langle\rangle,\mathtt{y}\rangle) \ggg \mathtt{first}\ (\mathtt{delay}\ [\![\mathtt{Q}]\!]_{\mathtt{x,y}})$
       $\ggg \mathtt{arr}\ (\lambda\langle\mathtt{f},\mathtt{y}\rangle.\,\lambda\mathtt{x}.\,\mathtt{f}\ \langle\mathtt{x},\mathtt{y}\rangle)))$

(continued from previous page)
$\mathtt{arr}\,(\lambda\Delta.\,\langle\rangle) \ggg \mathtt{delay}\,(\mathtt{force}\,(\llbracket P\rrbracket_{\langle\rangle} \ggg \mathtt{arr}\,(\lambda\mathtt{y}.\,\langle\langle\rangle,\mathtt{y}\rangle) \ggg \mathtt{first}\,(\mathtt{delay}\,\llbracket Q\rrbracket_{\mathtt{x,y}})$
$\qquad\qquad\qquad\qquad\qquad \ggg \mathtt{arr}\,(\lambda\langle\mathtt{f},\mathtt{y}\rangle.\,\lambda\mathtt{x}.\,\mathtt{f}\,\langle\mathtt{x},\mathtt{y}\rangle)))$

$=\qquad\quad(\leadsto_{S2})$

$\mathtt{arr}\,(\lambda\Delta.\,\langle\rangle) \ggg \llbracket P\rrbracket_{\langle\rangle} \ggg \mathtt{arr}\,(\lambda\mathtt{y}.\,\langle\langle\rangle,\mathtt{y}\rangle) \ggg \mathtt{first}\,(\mathtt{delay}\,\llbracket Q\rrbracket_{\mathtt{x,y}})$
$\quad \ggg \mathtt{arr}\,(\lambda\langle\mathtt{f},\mathtt{y}\rangle.\,\lambda\mathtt{x}.\,\mathtt{f}\,\langle\mathtt{x},\mathtt{y}\rangle)$

$=\qquad\quad(\leadsto_4,\beta^{\to})$

$\mathtt{arr}\,(\lambda\Delta.\,\langle\langle\rangle,\Delta\rangle) \ggg \mathtt{arr}\,\mathtt{fst} \ggg \llbracket P\rrbracket_{\langle\rangle} \ggg \mathtt{arr}\,(\lambda\mathtt{y}.\,\langle\langle\rangle,\mathtt{y}\rangle)$
$\quad \ggg \mathtt{first}\,(\mathtt{delay}\,\llbracket Q\rrbracket_{\mathtt{x,y}}) \ggg \mathtt{arr}\,(\lambda\langle\mathtt{f},\mathtt{y}\rangle.\,\lambda\mathtt{x}.\,\mathtt{f}\,\langle\mathtt{x},\mathtt{y}\rangle)$

$=\qquad\quad(\leadsto_8)$

$\mathtt{arr}\,(\lambda\Delta.\,\langle\langle\rangle,\Delta\rangle) \ggg \mathtt{first}\,\llbracket P\rrbracket_{\langle\rangle} \ggg \mathtt{arr}\,\mathtt{fst} \ggg \mathtt{arr}\,(\lambda\mathtt{y}.\,\langle\langle\rangle,\mathtt{y}\rangle)$
$\quad \ggg \mathtt{first}\,(\mathtt{delay}\,\llbracket Q\rrbracket_{\mathtt{x,y}}) \ggg \mathtt{arr}\,(\lambda\langle\mathtt{f},\mathtt{y}\rangle.\,\lambda\mathtt{x}.\,\mathtt{f}\,\langle\mathtt{x},\mathtt{y}\rangle)$

$=\qquad\quad(\leadsto_4,\beta^{\to})$

$\mathtt{arr}\,(\lambda\Delta.\,\langle\langle\rangle,\Delta\rangle) \ggg \mathtt{first}\,\llbracket P\rrbracket_{\langle\rangle} \ggg \mathtt{arr}\,(\lambda\langle\mathtt{y},\Delta\rangle.\,\langle\langle\rangle,\langle\Delta,\mathtt{y}\rangle\rangle) \ggg \mathtt{arr}\,(\mathtt{id}\times\mathtt{snd})$
$\quad \ggg \mathtt{first}\,(\mathtt{delay}\,\llbracket Q\rrbracket_{\mathtt{x,y}}) \ggg \mathtt{arr}\,(\lambda\langle\mathtt{f},\mathtt{y}\rangle.\,\lambda\mathtt{x}.\,\mathtt{f}\,\langle\mathtt{x},\mathtt{y}\rangle)$

$=\qquad\quad(\leadsto_7)$

$\mathtt{arr}\,(\lambda\Delta.\,\langle\langle\rangle,\Delta\rangle) \ggg \mathtt{first}\,\llbracket P\rrbracket_{\langle\rangle} \ggg \mathtt{arr}\,(\lambda\langle\mathtt{y},\Delta\rangle.\,\langle\langle\rangle,\langle\Delta,\mathtt{y}\rangle\rangle)$
$\quad \ggg \mathtt{first}\,(\mathtt{delay}\,\llbracket Q\rrbracket_{\mathtt{x,y}}) \ggg \mathtt{arr}\,(\mathtt{id}\times\mathtt{snd}) \ggg \mathtt{arr}\,(\lambda\langle\mathtt{f},\mathtt{y}\rangle.\,\lambda\mathtt{x}.\,\mathtt{f}\,\langle\mathtt{x},\mathtt{y}\rangle)$

$=\qquad\quad(\leadsto_4,\beta^{\to})$

$\mathtt{arr}\,\mathtt{dup} \ggg \mathtt{arr}\,(\mathtt{id}\times\mathtt{id}) \ggg \mathtt{arr}\,\mathtt{swap} \ggg \mathtt{arr}\,((\lambda\Delta.\,\langle\rangle)\times\mathtt{id}) \ggg \mathtt{first}\,\llbracket P\rrbracket_{\langle\rangle}$
$\quad \ggg \mathtt{arr}\,\mathtt{swap} \ggg \mathtt{arr}\,\mathtt{dup} \ggg \mathtt{arr}\,(\mathtt{id}\times\mathtt{id}) \ggg \mathtt{arr}\,\mathtt{swap}$
$\quad \ggg \mathtt{arr}\,((\lambda\langle\Delta,\mathtt{y}\rangle.\,\langle\rangle)\times\mathtt{id}) \ggg \mathtt{first}\,(\mathtt{delay}\,\llbracket Q\rrbracket_{\mathtt{x,y}})$
$\quad \ggg \mathtt{arr}\,\mathtt{swap} \ggg \mathtt{arr}\,(\lambda\langle\langle\Delta,\mathtt{y}\rangle,\mathtt{f}\rangle.\,\lambda\mathtt{x}.\,\mathtt{f}\,\langle\mathtt{x},\mathtt{y}\rangle)$

$=\qquad\quad(\leadsto_5)$

$\mathtt{arr}\,\mathtt{dup} \ggg \mathtt{first}\,(\mathtt{arr}\,\mathtt{id}) \ggg \mathtt{arr}\,\mathtt{swap} \ggg \mathtt{first}\,(\mathtt{arr}\,(\lambda\Delta.\,\langle\rangle))$
$\quad \ggg \mathtt{first}\,\llbracket P\rrbracket_{\langle\rangle} \ggg \mathtt{arr}\,\mathtt{swap} \ggg \mathtt{arr}\,\mathtt{dup} \ggg \mathtt{first}\,(\mathtt{arr}\,\mathtt{id})$
$\quad \ggg \mathtt{arr}\,\mathtt{swap} \ggg \mathtt{first}\,(\mathtt{arr}\,(\lambda\langle\Delta,\mathtt{y}\rangle.\,\langle\rangle)) \ggg \mathtt{first}\,(\mathtt{delay}\,\llbracket Q\rrbracket_{\mathtt{x,y}})$
$\quad \ggg \mathtt{arr}\,\mathtt{swap} \ggg \mathtt{arr}\,(\lambda\langle\langle\Delta,\mathtt{y}\rangle,\mathtt{f}\rangle.\,\lambda\mathtt{x}.\,\mathtt{f}\,\langle\mathtt{x},\mathtt{y}\rangle)$

$=\qquad\quad(\leadsto_6)$

$\mathtt{arr}\,\mathtt{dup} \ggg \mathtt{first}\,(\mathtt{arr}\,\mathtt{id}) \ggg \mathtt{arr}\,\mathtt{swap} \ggg \mathtt{first}\,(\mathtt{arr}\,(\lambda\Delta.\,\langle\rangle) \ggg \llbracket P\rrbracket_{\langle\rangle})$
$\quad \ggg \mathtt{arr}\,\mathtt{swap} \ggg \mathtt{arr}\,\mathtt{dup} \ggg \mathtt{first}\,(\mathtt{arr}\,\mathtt{id}) \ggg \mathtt{arr}\,\mathtt{swap}$
$\quad \ggg \mathtt{first}\,(\mathtt{arr}\,(\lambda\langle\Delta,\mathtt{y}\rangle.\,\langle\rangle) \ggg \mathtt{delay}\,\llbracket Q\rrbracket_{\mathtt{x,y}}) \ggg \mathtt{arr}\,\mathtt{swap}$
$\quad \ggg \mathtt{arr}\,(\lambda\langle\langle\Delta,\mathtt{y}\rangle,\mathtt{f}\rangle.\,\lambda\mathtt{x}.\,\mathtt{f}\,\langle\mathtt{x},\mathtt{y}\rangle)$

$=\qquad\quad(\text{def}\,\&\!\&\!\&)$

$(\mathtt{arr}\,\mathtt{id}\,\&\!\&\!\&\,(\mathtt{arr}\,(\lambda\Delta.\,\langle\rangle) \ggg \llbracket P\rrbracket_{\langle\rangle}))$
$\quad \ggg (\mathtt{arr}\,\mathtt{id}\,\&\!\&\!\&\,(\mathtt{arr}\,(\lambda\langle\Delta,\mathtt{y}\rangle.\,\langle\rangle) \ggg \mathtt{delay}\,\llbracket Q\rrbracket_{\mathtt{x,y}})) \ggg \mathtt{arr}\,(\lambda\langle\langle\Delta,\mathtt{y}\rangle,\mathtt{f}\rangle.\,\lambda\mathtt{x}.\,\mathtt{f}\,\langle\mathtt{x},\mathtt{y}\rangle)$

$=\qquad\quad(\text{Lemma }10)$

$(\mathtt{arr}\,\mathtt{id}\,\&\!\&\!\&\,\llbracket P\rrbracket_{\Delta}) \ggg (\mathtt{arr}\,\mathtt{id}\,\&\!\&\!\&\,(\mathtt{arr}\,(\lambda\langle\Delta,\mathtt{y}\rangle.\,\langle\rangle) \ggg \mathtt{delay}\,\llbracket Q\rrbracket_{\mathtt{x,y}}))$
$\quad \ggg \mathtt{arr}\,(\lambda\langle\langle\Delta,\mathtt{y}\rangle,\mathtt{f}\rangle.\,\lambda\mathtt{x}.\,\mathtt{f}\,\langle\mathtt{x},\mathtt{y}\rangle)$

$=\qquad\quad(\text{def}\,\llbracket-\rrbracket_{\Delta})$

$\llbracket\mathbf{let}\,\mathtt{y} \Leftarrow P\,\mathbf{in}\,\mathbf{let}\,\mathtt{f} \Leftarrow \mathbf{run}\,(\lambda^{\bullet}\langle\mathtt{x},\mathtt{y}\rangle.\,Q)\,\mathbf{in}\,[\lambda\mathtt{x}.\,\mathtt{f}\,\langle\mathtt{x},\mathtt{y}\rangle]\rrbracket_{\Delta}$

A.3.3 The laws of $\mathbb{C}_S$ follow from the laws of $\mathbb{S}$

For each law $M = N$ of classic arrows with delay we must show $\langle\!\langle M \rangle\!\rangle = \langle\!\langle N \rangle\!\rangle$.

1. $(\leadsto_{S1})$

$$\langle\!\langle \mathtt{force\ (delay\ a)} \rangle\!\rangle$$
$= \quad$ (def $\mathtt{force}$)
$$\langle\!\langle \mathtt{arr}\ (\lambda x.\ \langle\langle\rangle, x\rangle) \ggg \mathtt{delay\ a} \ggg \mathtt{arr}\ (\lambda z.\ \mathbf{fst}\ z\ (\mathbf{snd}\ z)) \rangle\!\rangle$$
$= \quad$ (def $\langle\!\langle - \rangle\!\rangle, \beta^{\leadsto}$)
$$\lambda^{\bullet} x.\,\mathbf{let}\ y \Leftarrow [\langle\langle\rangle, x\rangle]\ \mathbf{in}$$
$$\qquad \mathbf{let}\ z \Leftarrow \mathbf{let}\ w \Leftarrow \mathbf{run}\ a\ \mathbf{in}\ [\langle w, \mathbf{snd}\ y\rangle]\ \mathbf{in}$$
$$\qquad\quad [\mathbf{fst}\ z\ (\mathbf{snd}\ z)]$$
$= \quad$ (left, $\beta_2^{\times}$)
$$\lambda^{\bullet} x.\,\mathbf{let}\ z \Leftarrow \mathbf{let}\ w \Leftarrow \mathbf{run}\ a\ \mathbf{in}\ [\langle w, x\rangle]\ \mathbf{in}\ [\mathbf{fst}\ z\ (\mathbf{snd}\ z)]$$
$= \quad$ (assoc)
$$\lambda^{\bullet} x.\,\mathbf{let}\ w \Leftarrow \mathbf{run}\ a\ \mathbf{in}\ \mathbf{let}\ z \Leftarrow [\langle w, x\rangle]\ \mathbf{in}\ [\mathbf{fst}\ z\ (\mathbf{snd}\ z)]$$
$= \quad$ (left, $\beta_1^{\times}, \beta_2^{\times}$)
$$\lambda^{\bullet} x.\,\mathbf{let}\ w \Leftarrow \mathbf{run}\ a\ \mathbf{in}\ [w\ x]$$
$= \quad$ (run_1)
$$\lambda^{\bullet} x.\, a \bullet x$$
$= \quad$ ($\eta^{\leadsto}$)
$$a$$
$= \quad$ (def $\langle\!\langle - \rangle\!\rangle$)
$$\langle\!\langle a \rangle\!\rangle$$

2. $(\leadsto_{S2})$

$$\langle\!\langle \mathtt{delay\ (force\ a)} \rangle\!\rangle$$
$= \quad$ (def $\mathtt{force}$)
$$\langle\!\langle \mathtt{delay}\ (\mathtt{arr}\ (\lambda x.\ \langle\langle\rangle, x\rangle) \ggg a \ggg \mathtt{arr}\ (\lambda z.\ (\mathtt{fst}\ z\ (\mathtt{snd}\ z)))) \rangle\!\rangle$$
$= \quad$ (def $\langle\!\langle - \rangle\!\rangle$)
$$\lambda^{\bullet} \langle\rangle.\,\mathbf{run}\ (\lambda^{\bullet} x.\,\mathbf{let}\ y \Leftarrow [\langle\langle\rangle, x\rangle]\ \mathbf{in}$$
$$\qquad\qquad \mathbf{let}\ z \Leftarrow \mathbf{let}\ w \Leftarrow a \bullet (\mathbf{fst}\ y)\ \mathbf{in}\ [\langle w, \mathbf{snd}\ y\rangle]\ \mathbf{in}\ [(\mathbf{fst}\ z\ (\mathbf{snd}\ z))])$$
$= \quad$ (left, $\beta_1^{\times}, \beta_2^{\times}$)
$$\lambda^{\bullet} \langle\rangle.\,\mathbf{run}\ (\lambda^{\bullet} x.\,\mathbf{let}\ z \Leftarrow \mathbf{let}\ w \Leftarrow a \bullet \langle\rangle\ \mathbf{in}\ [\langle w, x\rangle]\ \mathbf{in}\ [(\mathbf{fst}\ z\ (\mathbf{snd}\ z))])$$
$= \quad$ (assoc)
$$\lambda^{\bullet} \langle\rangle.\,\mathbf{run}\ (\lambda^{\bullet} x.\,\mathbf{let}\ w \Leftarrow a \bullet \langle\rangle\ \mathbf{in}\ \mathbf{let}\ z \Leftarrow [\langle w, x\rangle]\ \mathbf{in}\ [(\mathbf{fst}\ z\ (\mathbf{snd}\ z))])$$
$= \quad$ (left, $\beta_1^{\times}, \beta_2^{\times}$)
$$\lambda^{\bullet} \langle\rangle.\,\mathbf{run}\ (\lambda^{\bullet} x.\,\mathbf{let}\ w \Leftarrow a \bullet \langle\rangle\ \mathbf{in}\ [w\ x])$$
$= \quad$ (run_3)
$$\lambda^{\bullet} \langle\rangle.\,\mathbf{let}\ w \Leftarrow a \bullet \langle\rangle\ \mathbf{in}\ \mathbf{let}\ f \Leftarrow \mathbf{run}\ (\lambda^{\bullet} \langle x, w\rangle.\ [w\ x])\ \mathbf{in}\ [\lambda x.\ f\ \langle x, w\rangle]$$
$= \quad$ (run_2)
$$\lambda^{\bullet} \langle\rangle.\,\mathbf{let}\ w \Leftarrow a \bullet \langle\rangle\ \mathbf{in}\ \mathbf{let}\ f \Leftarrow [\lambda\langle x, w\rangle.\ w\ x]\ \mathbf{in}\ [\lambda x.\ f\ \langle x, w\rangle]$$

(continued on next page)

(continued from previous page)
$\lambda^\bullet\langle\rangle.\ \mathbf{let}\ \mathtt{w} \Leftarrow \mathtt{a} \bullet \langle\rangle\ \mathbf{in}\ \mathbf{let}\ \mathtt{f} \Leftarrow [\lambda\langle\mathtt{x},\mathtt{w}\rangle.\,\mathtt{w}\,\mathtt{x}]\ \mathbf{in}\ [\lambda\mathtt{x}.\,\mathtt{f}\ \langle\mathtt{x},\mathtt{w}\rangle]$

$=\qquad$ (left, $\beta^\rightarrow$)

$\lambda^\bullet\langle\rangle.\ \mathbf{let}\ \mathtt{w} \Leftarrow \mathtt{a} \bullet \langle\rangle\ \mathbf{in}\ [\lambda\mathtt{x}.\,\mathtt{w}\,\mathtt{x}]$

$=\qquad$ ($\eta^\rightarrow$)

$\lambda^\bullet\langle\rangle.\ \mathbf{let}\ \mathtt{w} \Leftarrow \mathtt{a} \bullet \langle\rangle\ \mathbf{in}\ [\mathtt{w}]$

$=\qquad$ (right)

$\lambda^\bullet\langle\rangle.\ \mathtt{a} \bullet \langle\rangle$

$=\qquad$ (def $\langle\!\langle-\rangle\!\rangle, \eta^{\rightsquigarrow}$)

$\langle\!\langle\mathtt{a}\rangle\!\rangle$

A.3.4   Translating $\mathcal{S}$ to $\mathcal{C}_\mathbf{S}$ and back

For each static arrows command P we must show $\langle\!\langle\,[\![\mathrm{P}]\!]_\Delta\,\rangle\!\rangle = \lambda^\bullet\Delta.\,\mathrm{P}$.

1. Case $\mathbf{run}\ \mathrm{L}$

$\langle\!\langle[\![\mathbf{run}\ \mathrm{L}]\!]_\Delta\rangle\!\rangle$

$=\qquad$ (def $[\![-]\!]_\Delta$)

$\langle\!\langle\mathtt{arr}\ (\lambda\Delta.\,\langle\rangle) \ggg \mathtt{delay}\ [\![\mathrm{L}]\!]\rangle\!\rangle$

$=\qquad$ (def $\langle\!\langle-\rangle\!\rangle, \beta^\rightarrow, \beta^{\rightsquigarrow}$)

$\lambda^\bullet\Delta.\ \mathbf{let}\ \mathtt{x} \Leftarrow [\langle\rangle]\ \mathbf{in}\ \mathbf{run}\ \langle\!\langle[\![\mathrm{L}]\!]\rangle\!\rangle$

$=\qquad$ (induction hypothesis)

$\lambda^\bullet\Delta.\ \mathbf{let}\ \mathtt{x} \Leftarrow [\langle\rangle]\ \mathbf{in}\ \mathbf{run}\ \mathrm{L}$

$=\qquad$ (left)

$\lambda^\bullet\Delta.\ \mathbf{run}\ \mathrm{L}$

A.3.5   Translating $\mathcal{C}_\mathbf{S}$ to $\mathcal{S}$ and back

For each term M of classic arrows with delay we must show $[\![\,\langle\!\langle\mathrm{M}\rangle\!\rangle\,]\!] = \mathrm{M}$.

1. Case `delay`:

$[\![\langle\!\langle\mathtt{delay}\rangle\!\rangle]\!]$

$=\qquad$ (def $\langle\!\langle-\rangle\!\rangle$)

$[\![\lambda\mathtt{a}.\,\lambda^\bullet\langle\rangle.\,\mathbf{run}\ \mathtt{a}]\!]$

$=\qquad$ (def $[\![-]\!]$)

$\lambda\mathtt{a}.\,\mathtt{arr}\ (\lambda\langle\rangle.\,\langle\rangle) \ggg \mathtt{delay}\ \mathtt{a}$

$=\qquad$ ($\rightsquigarrow_1, \eta^\rightarrow$)

$\mathtt{delay}$

A.4   Equational equivalence between $\mathfrak{I}$ and $\mathcal{S}$

This section gives a proof of Proposition 15 (page 70).

A.4.1   Proofs of Lemmas 13 and 14

We begin with proofs of Lemma 13 and 14, which give the translations of substitution and weakening.

Proof of Lemma 13 (Translating substitution from $\mathcal{S}$ to $\mathfrak{I}$)   .

The translations of substitution on terms and commands from $\mathcal{S}$ to $\mathfrak{I}$ are as follows.

$$\langle\!\langle M[x := N]\rangle\!\rangle = \langle\!\langle M\rangle\!\rangle[x := \langle\!\langle N\rangle\!\rangle]$$

$$\langle\!\langle (\!\mid\!) Q[x := N])_\Delta = \texttt{pure } (\lambda g.\lambda\Delta.g \ (\Delta, \langle\!\langle N\rangle\!\rangle)) \circledast \langle\!\langle Q\rangle\!\rangle_{\Delta,x}$$

Proof. By mutual induction on the derivations of P and M. There is one case for each term form and each command form. We give only the cases for command forms here.

1. Case $L \bullet M$

$$\llbracket (L \bullet M)[x := N]\rrbracket_\Delta$$
$=$ (def $\texttt{substitution}$)
$$\llbracket L \bullet (M[x := N])\rrbracket_\Delta$$
$=$ (def $\llbracket - \rrbracket_\Delta$)
$$\texttt{pure } (\lambda l.\lambda\Delta.l \ \llbracket M[x := N]\rrbracket) \circledast \llbracket L\rrbracket$$
$=$ (induction hypothesis)
$$\texttt{pure } (\lambda l.\lambda\Delta.l \ (\llbracket M\rrbracket[x := \llbracket N\rrbracket])) \circledast \llbracket L\rrbracket$$
$=$ $(I_3, I_2, \beta^\rightarrow)$
$$\texttt{pure } (\lambda g.\lambda\Delta.g \ \langle\Delta, \llbracket N\rrbracket\rangle) \circledast (\texttt{pure } (\lambda l.\lambda\langle\Delta, x\rangle.l \ \llbracket M\rrbracket) \circledast \llbracket L\rrbracket)$$
$=$ (def $\llbracket - \rrbracket_{\Delta,x}$)
$$\texttt{pure } (\lambda g.\lambda\Delta.g \ \langle\Delta, \llbracket N\rrbracket\rangle) \circledast \llbracket L \bullet M\rrbracket_{\Delta,x}$$

2. Case $[M]$

$$\llbracket [M][x := N]\rrbracket_\Delta$$
$=$ (def $\texttt{substitution}$)
$$\llbracket [M[x := N]]\rrbracket_\Delta$$
$=$ (def $\llbracket - \rrbracket_\Delta$)
$$\texttt{pure } (\lambda\Delta.\llbracket M[x := N]\rrbracket)$$
$=$ (induction hypothesis)
$$\texttt{pure } (\lambda\Delta.\llbracket M\rrbracket[x := \llbracket N\rrbracket])$$

(continued on next page)

(continued from previous page)
$\text{pure}\,(\lambda\Delta.\,[\![M]\!][x := [\![N]\!]])$

$=\qquad (I_2,\,\beta^\rightarrow)$

$\text{pure}\,(\lambda g.\,\lambda\Delta.\,g\,\langle\Delta,[\![N]\!]\rangle)\circledast\text{pure}\,(\lambda\langle\Delta,x\rangle.\,[\![M]\!])$

$=\qquad (\text{def }[\![-]\!]_{\Delta,x})$

$\text{pure}\,(\lambda g.\,\lambda\Delta.\,g\,\langle\Delta,[\![N]\!]\rangle)\circledast[\![[M]\!]]_{\Delta,x}$

3. Case **let** $y \Leftarrow P$ **in** Q

$[\![(\mathbf{let}\ y \Leftarrow P\ \mathbf{in}\ Q)[x := N]]\!]_\Delta$

$=\qquad (\text{def substitution})$

$[\![\mathbf{let}\ y \Leftarrow P[x := N]\ \mathbf{in}\ Q[x := N]]\!]_\Delta$

$=\qquad (\text{def }[\![-]\!]_\Delta)$

$\text{pure}\,(\lambda p.\,\lambda q.\,\lambda\Delta.\,q\,\langle\Delta,p\,\Delta\rangle)\circledast[\![P[x := N]]\!]_\Delta\circledast[\![Q[x := N]]\!]_{\Delta,y}$

$=\qquad (\text{induction hypothesis})$

$\text{pure}\,(\lambda p.\,\lambda q.\,\lambda\Delta.\,q\,\langle\Delta,p\,\Delta\rangle)$
$\quad\circledast(\text{pure}\,(\lambda g.\,\lambda\Delta.\,g\,\langle\Delta,[\![N]\!]\rangle)\circledast[\![P]\!]_{\Delta,x})$
$\quad\circledast(\text{pure}\,(\lambda g.\,\lambda\Delta.\,g\,\langle\Delta,[\![N]\!]\rangle)\circledast[\![Q]\!]_{\Delta,y,x})$

$=\qquad (\text{Lemma }14)$

$\text{pure}\,(\lambda p.\,\lambda q.\,\lambda\Delta.\,q\,\langle\Delta,p\,\Delta\rangle)$
$\quad\circledast(\text{pure}\,(\lambda g.\,\lambda\Delta.\,g\,\langle\Delta,[\![N]\!]\rangle)\circledast[\![P]\!]_{\Delta,x})$
$\quad\circledast(\text{pure}\,(\lambda g.\,\lambda\Delta.\,g\,\langle\Delta,[\![N]\!]\rangle)\circledast(\text{pure}\,(\lambda g.\,\lambda\langle\langle\Delta,y\rangle,x\rangle.\,g\,\langle\langle\Delta,x\rangle,y\rangle)\circledast[\![Q]\!]_{\Delta,x,y}))$

$=\qquad (I_3,\,I_2,\,\beta^\rightarrow)$

$\text{pure}\,(\lambda g.\,\lambda q.\,\lambda\Delta.\,q\,\langle\Delta,g\,\langle\Delta,[\![N]\!]\rangle\rangle)\circledast[\![P]\!]_{\Delta,x}$
$\quad\circledast(\text{pure}\,(\lambda g.\,\lambda\Delta.\,g\,\langle\Delta,[\![N]\!]\rangle)\circledast(\text{pure}\,(\lambda g.\,\lambda\langle\langle\Delta,y\rangle,x\rangle.\,g\,\langle\langle\Delta,x\rangle,y\rangle)\circledast[\![Q]\!]_{\Delta,x,y}))$

$=\qquad (I_3,\,I_2,\,\beta^\rightarrow)$

$\text{pure}\,(\lambda g.\,\lambda q.\,\lambda\Delta.\,q\,\langle\Delta,g\,\langle\Delta,[\![N]\!]\rangle\rangle)\circledast[\![P]\!]_{\Delta,x}$
$\quad\circledast(\text{pure}\,(\lambda g.\,\lambda\langle\Delta,y\rangle.\,g\,\langle\langle\Delta,[\![N]\!]\rangle,y\rangle)\circledast[\![Q]\!]_{\Delta,x,y})$

$=\qquad (I_3)$

$\text{pure}\,(\cdot)\circledast(\text{pure}\,(\lambda g.\,\lambda q.\,\lambda\Delta.\,q\,\langle\Delta,g\,\langle\Delta,[\![N]\!]\rangle\rangle)\circledast[\![P]\!]_{\Delta,x})$
$\quad\circledast\text{pure}\,(\lambda g.\,\lambda\langle\Delta,y\rangle.\,g\,\langle\langle\Delta,[\![N]\!]\rangle,y\rangle)\circledast[\![Q]\!]_{\Delta,x,y}$

$=\qquad (I_3,\,I_2,\,\beta^\rightarrow)$

$\text{pure}\,(\lambda p.\,\lambda s.\,\lambda q.\,\lambda\Delta.\,s\,q\,\langle\Delta,p\,\langle\Delta,[\![N]\!]\rangle\rangle)\circledast[\![P]\!]_{\Delta,x}$
$\quad\circledast\text{pure}\,(\lambda g.\,\lambda\langle\Delta,y\rangle.\,g\,\langle\langle\Delta,[\![N]\!]\rangle,y\rangle)\circledast[\![Q]\!]_{\Delta,x,y}$

$=\qquad (I_4)$

$\text{pure}\,(\lambda h.\,h\,(\lambda g.\,\lambda\langle\Delta,y\rangle.\,g\,\langle\langle\Delta,[\![N]\!]\rangle,y\rangle))$
$\quad\circledast(\text{pure}\,(\lambda p.\,\lambda s.\,\lambda q.\,\lambda\Delta.\,s\,q\,\langle\Delta,p\,\langle\Delta,[\![N]\!]\rangle\rangle)\circledast[\![P]\!]_{\Delta,x})\circledast[\![Q]\!]_{\Delta,x,y}$

$=\qquad (I_3,\,I_2,\,\beta^\rightarrow)$

$\text{pure}\,(\lambda p.\,\lambda q.\,\lambda\Delta.\,q\,\langle\langle\Delta,[\![N]\!]\rangle,p\,\langle\Delta,[\![N]\!]\rangle\rangle)\circledast[\![P]\!]_{\Delta,x}\circledast[\![Q]\!]_{\Delta,x,y}$

(continued on next page)

(continued from previous page)
$\texttt{pure}\ (\lambda\texttt{p}.\ \lambda\texttt{q}.\ \lambda\Delta.\ \texttt{q}\ \langle\langle\Delta, \llbracket\texttt{N}\rrbracket\rangle, \texttt{p}\ \langle\Delta, \llbracket\texttt{N}\rrbracket\rangle\rangle)\ \circledast\ \llbracket\texttt{P}\rrbracket_{\Delta,\texttt{x}}\ \circledast\ \llbracket\texttt{Q}\rrbracket_{\Delta,\texttt{x},\texttt{y}}$

$=\qquad (\text{I}_3, \text{I}_2, \beta^{\rightarrow})$

$\texttt{pure}\ (\lambda\texttt{b}.\ \lambda\texttt{q}.\ \lambda\Delta.\ \texttt{b}\ \texttt{q}\ \langle\Delta, \llbracket\texttt{N}\rrbracket\rangle)$
$\qquad \circledast\ (\texttt{pure}\ (\lambda\texttt{p}.\ \lambda\texttt{q}.\ \lambda\Delta.\ \texttt{q}\ \langle\Delta, \texttt{p}\ \Delta\rangle)\ \circledast\ \llbracket\texttt{P}\rrbracket_{\Delta,\texttt{x}})\ \circledast\ \llbracket\texttt{Q}\rrbracket_{\Delta,\texttt{x},\texttt{y}}$

$=\qquad (\text{I}_3, \text{I}_2, \beta^{\rightarrow})$

$\texttt{pure}\ (\lambda\texttt{g}.\ \lambda\Delta.\ \texttt{g}\ \langle\Delta, \texttt{n}\rangle)\ \circledast\ (\texttt{pure}\ (\lambda\texttt{p}.\ \lambda\texttt{q}.\ \lambda\Delta.\ \texttt{q}\ \langle\Delta, \texttt{p}\ \Delta\rangle)\ \circledast\ \llbracket\texttt{P}\rrbracket_{\Delta,\texttt{x}}\ \circledast\ \llbracket\texttt{Q}\rrbracket_{\Delta,\texttt{x},\texttt{y}})$

$=\qquad (\text{def}\ \llbracket-\rrbracket_{\Delta,\texttt{x}})$

$\texttt{pure}\ (\lambda\texttt{g}.\ \lambda\Delta.\ \texttt{g}\ \langle\Delta, \llbracket\texttt{N}\rrbracket\rangle)\ \circledast\ \llbracket\textbf{let}\ \texttt{y} \Leftarrow \texttt{P}\ \textbf{in}\ \texttt{Q}\rrbracket_{\Delta,\texttt{x}}$

4. Case **run** L

$\llbracket(\textbf{run}\ \texttt{L})[\texttt{x} := \texttt{N}]\rrbracket_{\Delta}$

$=\qquad (\text{def substitution})$

$\llbracket\textbf{run}\ \texttt{L}\rrbracket_{\Delta}$

$=\qquad (\text{def}\ \llbracket-\rrbracket_{\Delta})$

$\texttt{pure}\ (\lambda\texttt{l}.\ \lambda\Delta.\ \texttt{l})\ \circledast\ \llbracket\texttt{L}\rrbracket$

$=\qquad (\text{I}_3, \text{I}_2, \beta^{\rightarrow})$

$\texttt{pure}\ (\lambda\texttt{g}.\ \lambda\Delta.\ \texttt{g}\ \langle\Delta, \llbracket\texttt{N}\rrbracket\rangle)\ \circledast\ (\texttt{pure}\ (\lambda\texttt{l}.\ \lambda\langle\Delta, \texttt{x}\rangle.\ \texttt{l})\ \circledast\ \llbracket\texttt{L}\rrbracket)$

$=\qquad (\text{def}\ \llbracket-\rrbracket_{\Delta,\texttt{x}})$

$\texttt{pure}\ (\lambda\texttt{g}.\ \lambda\Delta.\ \texttt{g}\ \langle\Delta, \llbracket\texttt{N}\rrbracket\rangle)\ \circledast\ \llbracket\textbf{run}\ \texttt{L}\rrbracket_{\Delta,\texttt{x}}$

$\square$

Proof of Lemma 14 (Translating weakening from $\mathbb{S}$ **to** $\mathbb{J}$**)** .

The translation of weakening from $\mathbb{S}$ to $\mathbb{J}$ for commands is as follows.

$$\left\langle\!\!\left[\frac{\Gamma;\ \Delta \vdash \texttt{Q}\ !\ \texttt{B}}{\Gamma';\ \Delta' \vdash \texttt{Q}\ !\ \texttt{B}}\right]\!\!\right\rangle \qquad = \qquad \frac{\langle\!\langle\Gamma\rangle\!\rangle \vdash \langle\!\langle\texttt{Q}\rangle\!\rangle_{\Delta} : \text{I}(\langle\!\langle\Delta\rangle\!\rangle \rightarrow \langle\!\langle\texttt{B}\rangle\!\rangle)}{\langle\!\langle\Gamma'\rangle\!\rangle \vdash \texttt{pure}\ (\lambda\texttt{g}.\ \lambda\Delta'.\texttt{g}\ \Delta)\ \circledast\ \langle\!\langle\texttt{Q}\rangle\!\rangle_{\Delta} : \text{I}(\langle\!\langle\Delta'\rangle\!\rangle \rightarrow \langle\!\langle\texttt{B}\rangle\!\rangle)}$$

Proof. By induction on the derivation of Q. There is one case for each command form.

1. Case L $\bullet$ M.

$\llbracket\texttt{L} \bullet \texttt{M}\rrbracket_{\Delta'}$

$=\qquad (\text{def}\ \llbracket-\rrbracket_{\Delta'})$

$\texttt{pure}\ (\lambda\texttt{l}.\ \lambda\Delta'.\ \texttt{l}\ \llbracket\texttt{M}\rrbracket)\ \circledast\ \llbracket\texttt{L}\rrbracket$

$=\qquad (\text{I}_3, \text{I}_2, \beta^{\rightarrow})$

$\texttt{pure}\ (\lambda\texttt{g}.\ \lambda\Delta'.\ \texttt{g}\ \Delta)\ \circledast\ (\texttt{pure}\ (\lambda\texttt{l}.\ \lambda\Delta.\ \texttt{l}\ \llbracket\texttt{M}\rrbracket)\ \circledast\ \llbracket\texttt{L}\rrbracket)$

$=\qquad (\text{def}\ \llbracket-\rrbracket_{\Delta})$

$\texttt{pure}\ (\lambda\texttt{g}.\ \lambda\Delta'.\ \texttt{g}\ \Delta)\ \circledast\ \llbracket\texttt{L} \bullet \texttt{M}\rrbracket_{\Delta}$

2. Case [M].

$$\llbracket[\mathrm{M}]\rrbracket_{\Delta'}$$
$$=\quad (\text{def } \llbracket-\rrbracket_{\Delta'})$$
$$\mathtt{pure}\ (\lambda\Delta'.\ \llbracket\mathrm{M}\rrbracket)$$
$$=\quad (\mathrm{I}_2,\beta^{\to})$$
$$\mathtt{pure}\ (\lambda\mathrm{g}.\ \lambda\Delta'.\ \mathrm{g}\ \Delta)\circledast \mathtt{pure}\ (\lambda\Delta.\ \llbracket\mathrm{M}\rrbracket)$$
$$=\quad (\text{def } \llbracket-\rrbracket_{\Delta})$$
$$\mathtt{pure}\ (\lambda\mathrm{g}.\ \lambda\Delta'.\ \mathrm{g}\ \Delta)\circledast \llbracket[\mathrm{M}]\rrbracket_{\Delta}$$

3. Case **let** $\mathrm{x}\Leftarrow\mathrm{P}$ **in** Q.

$$\llbracket\mathbf{let}\ \mathrm{x}\Leftarrow\mathrm{P}\ \mathbf{in}\ \mathrm{Q}\rrbracket_{\Delta'}$$
$$=\quad (\text{def } \llbracket-\rrbracket_{\Delta'})$$
$$\mathtt{pure}\ (\lambda\mathrm{p}.\ \lambda\mathrm{q}.\ \lambda\Delta'.\ \mathrm{q}\ \langle\Delta',\mathrm{p}\ \Delta'\rangle)\circledast\llbracket\mathrm{P}\rrbracket_{\Delta'}\circledast\llbracket\mathrm{Q}\rrbracket_{\Delta',\mathrm{x}}$$
$$=\quad (\text{induction hypothesis})$$
$$\mathtt{pure}\ (\lambda\mathrm{p}.\ \lambda\mathrm{q}.\ \lambda\Delta'.\ \mathrm{q}\ \langle\Delta',\mathrm{p}\ \Delta'\rangle)\circledast(\mathtt{pure}\ (\lambda\mathrm{g}.\ \lambda\Delta'.\ \mathrm{g}\ \Delta)\circledast\llbracket\mathrm{P}\rrbracket_{\Delta})$$
$$\quad\circledast(\mathtt{pure}\ (\lambda\mathrm{g}.\ \lambda\langle\Delta',\mathrm{x}\rangle.\ \mathrm{g}\ \langle\Delta,\mathrm{x}\rangle)\circledast\llbracket\mathrm{Q}\rrbracket_{\Delta,\mathrm{x}})$$
$$=\quad (\mathrm{I}_3,\mathrm{I}_2,\beta^{\to})$$
$$\mathtt{pure}\ (\lambda\mathrm{g}.\ \lambda\mathrm{q}.\ \lambda\Delta'.\ \mathrm{q}\ \langle\Delta',\mathrm{g}\ \Delta\rangle)\circledast\llbracket\mathrm{P}\rrbracket_{\Delta}\circledast(\mathtt{pure}\ (\lambda\mathrm{g}.\ \lambda\langle\Delta',\mathrm{x}\rangle.\ \mathrm{g}\ \langle\Delta,\mathrm{x}\rangle)\circledast\llbracket\mathrm{Q}\rrbracket_{\Delta,\mathrm{x}})$$
$$=\quad (\mathrm{I}_3)$$
$$\mathtt{pure}\ (\cdot)\circledast(\mathtt{pure}\ (\lambda\mathrm{g}.\ \lambda\mathrm{q}.\ \lambda\Delta'.\ \mathrm{q}\ \langle\Delta',\mathrm{g}\ \Delta\rangle)\circledast\llbracket\mathrm{P}\rrbracket_{\Delta})$$
$$\quad\circledast\mathtt{pure}\ (\lambda\mathrm{g}.\ \lambda\langle\Delta',\mathrm{x}\rangle.\ \mathrm{g}\ \langle\Delta,\mathrm{x}\rangle)\circledast\llbracket\mathrm{Q}\rrbracket_{\Delta,\mathrm{x}}$$
$$=\quad (\mathrm{I}_3,\mathrm{I}_2,\beta^{\to})$$
$$\mathtt{pure}\ (\lambda\mathrm{g}.\ \lambda\mathrm{b}.\ \lambda\mathrm{c}.\ \lambda\Delta'.\ \mathrm{b}\ \mathrm{c}\ \langle\Delta',\mathrm{g}\ \Delta\rangle)\circledast\llbracket\mathrm{P}\rrbracket_{\Delta}$$
$$\quad\circledast\mathtt{pure}\ (\lambda\mathrm{g}.\ \lambda\langle\Delta',\mathrm{x}\rangle.\ \mathrm{g}\ \langle\Delta,\mathrm{x}\rangle)\circledast\llbracket\mathrm{Q}\rrbracket_{\Delta,\mathrm{x}}$$
$$=\quad (\mathrm{I}_4)$$
$$\mathtt{pure}\ (\lambda\mathrm{h}.\ \mathrm{h}\ (\lambda\mathrm{g}.\ \lambda\langle\Delta',\mathrm{x}\rangle.\ \mathrm{g}\ \langle\Delta,\mathrm{x}\rangle))$$
$$\quad\circledast(\mathtt{pure}\ (\lambda\mathrm{g}.\ \lambda\mathrm{b}.\ \lambda\mathrm{c}.\ \lambda\Delta'.\ \mathrm{b}\ \mathrm{c}\ \langle\Delta',\mathrm{g}\ \Delta\rangle)\circledast\llbracket\mathrm{P}\rrbracket_{\Delta})\circledast\llbracket\mathrm{Q}\rrbracket_{\Delta,\mathrm{x}}$$
$$=\quad (\mathrm{I}_3,\mathrm{I}_2,\beta^{\to})$$
$$\mathtt{pure}\ (\lambda\mathrm{p}.\ \lambda\mathrm{q}.\ \lambda\Delta'.\ \mathrm{q}\ \langle\Delta,\mathrm{p}\ \Delta\rangle)\circledast\llbracket\mathrm{P}\rrbracket_{\Delta}\circledast\llbracket\mathrm{Q}\rrbracket_{\Delta,\mathrm{x}}$$
$$=\quad (\mathrm{I}_3,\mathrm{I}_2,\beta^{\to})$$
$$\mathtt{pure}\ (\lambda\mathrm{p}.\ \lambda\mathrm{q}.\ \lambda\Delta'.\ \mathrm{p}\ \mathrm{q}\ \Delta)\circledast(\mathtt{pure}\ (\lambda\mathrm{p}.\ \lambda\mathrm{q}.\ \lambda\Delta'.\ \mathrm{q}\ \langle\Delta',\mathrm{p}\ \Delta'\rangle)\circledast\llbracket\mathrm{P}\rrbracket_{\Delta})\circledast\llbracket\mathrm{Q}\rrbracket_{\Delta,\mathrm{x}}$$
$$=\quad (\mathrm{I}_3,\mathrm{I}_2,\beta^{\to})$$
$$\mathtt{pure}\ (\lambda\mathrm{g}.\ \lambda\Delta'.\ \mathrm{g}\ \Delta)\circledast(\mathtt{pure}\ (\lambda\mathrm{p}.\ \lambda\mathrm{q}.\ \lambda\Delta'.\ \mathrm{q}\ \langle\Delta',\mathrm{p}\ \Delta'\rangle)\circledast\llbracket\mathrm{P}\rrbracket_{\Delta}\circledast\llbracket\mathrm{Q}\rrbracket_{\Delta,\mathrm{x}})$$
$$=\quad (\text{def } \llbracket-\rrbracket_{\Delta})$$
$$\mathtt{pure}\ (\lambda\mathrm{g}.\ \lambda\Delta'.\ \mathrm{g}\ \Delta)\circledast\llbracket\mathbf{let}\ \mathrm{x}\Leftarrow\mathrm{P}\ \mathbf{in}\ \mathrm{Q}\rrbracket_{\Delta}$$

4. Case **run** L.

$$\llbracket \mathbf{run}\ \mathrm{L} \rrbracket_{\Delta'}$$
$$= \qquad (\mathrm{def}\ \llbracket - \rrbracket_{\Delta'})$$
$$\mathtt{pure}\ (\lambda \mathrm{l}.\ \lambda \Delta'.\ \mathrm{l}) \circledast \llbracket \mathrm{L} \rrbracket$$
$$= \qquad (\mathrm{I}_3, \mathrm{I}_2, \beta^{\rightarrow})$$
$$\mathtt{pure}\ (\lambda \mathrm{g}.\ \lambda \Delta'.\ \mathrm{g}\ \Delta) \circledast (\mathtt{pure}\ (\lambda \mathrm{l}.\ \lambda \Delta.\ \mathrm{l}) \circledast \llbracket \mathrm{L} \rrbracket)$$
$$= \qquad (\mathrm{def}\ \llbracket - \rrbracket_{\Delta',\mathrm{x}})$$
$$\mathtt{pure}\ (\lambda \mathrm{g}.\ \lambda \Delta'.\ \mathrm{g}\ \Delta) \circledast \llbracket \mathbf{run}\ \mathrm{L} \rrbracket_{\Delta}$$

$\square$

A.4.2   The laws of $\mathcal{J}$ follow from the laws of $\mathcal{S}$

For each law $\mathrm{M} = \mathrm{N}$ of $\mathcal{J}$ we must show $\llbracket \mathrm{M} \rrbracket = \llbracket \mathrm{N} \rrbracket$.

1. (I_1)

$$\llbracket \mathtt{pure\ id} \circledast \mathrm{M} \rrbracket$$
$$= \qquad (\mathrm{def}\ \llbracket - \rrbracket, \beta^{\rightarrow})$$
$$\lambda^{\bullet} \langle \rangle.\ \mathbf{let}\ \mathrm{k} \Leftarrow (\lambda^{\bullet} \langle \rangle.\ [\mathrm{id}]) \bullet \langle \rangle\ \mathbf{in}\ \mathbf{let}\ \mathrm{x} \Leftarrow \llbracket \mathrm{M} \rrbracket \bullet \langle \rangle\ \mathbf{in}\ [\mathrm{k}\ \mathrm{x}]$$
$$= \qquad (\beta^{\rightsquigarrow})$$
$$\lambda^{\bullet} \langle \rangle.\ \mathbf{let}\ \mathrm{k} \Leftarrow [\mathrm{id}]\ \mathbf{in}\ \mathbf{let}\ \mathrm{x} \Leftarrow \llbracket \mathrm{M} \rrbracket \bullet \langle \rangle\ \mathbf{in}\ [\mathrm{k}\ \mathrm{x}]$$
$$= \qquad (\mathrm{left}, \mathrm{def}\ \mathrm{id})$$
$$\lambda^{\bullet} \langle \rangle.\ \mathbf{let}\ \mathrm{x} \Leftarrow \llbracket \mathrm{M} \rrbracket \bullet \langle \rangle\ \mathbf{in}\ [\mathrm{x}]$$
$$= \qquad (\mathrm{right})$$
$$\lambda^{\bullet} \langle \rangle.\ \llbracket \mathrm{M} \rrbracket \bullet \langle \rangle$$
$$= \qquad (\eta^{\rightsquigarrow})$$
$$\llbracket \mathrm{M} \rrbracket$$

2. (I_2)

$$\llbracket \mathtt{pure}\ \mathrm{M} \circledast \mathtt{pure}\ \mathrm{N} \rrbracket$$
$$= \qquad (\mathrm{def}\ \llbracket - \rrbracket, \beta^{\rightarrow})$$
$$\lambda^{\bullet} \langle \rangle.\ \mathbf{let}\ \mathrm{k} \Leftarrow (\lambda^{\bullet} \langle \rangle.\ [\llbracket \mathrm{M} \rrbracket]) \bullet \langle \rangle\ \mathbf{in}\ \mathbf{let}\ \mathrm{x} \Leftarrow (\lambda^{\bullet} \langle \rangle.\ [\llbracket \mathrm{N} \rrbracket]) \bullet \langle \rangle\ \mathbf{in}\ [\mathrm{k}\ \mathrm{x}]$$
$$= \qquad (\beta^{\rightsquigarrow})$$
$$\lambda^{\bullet} \langle \rangle.\ \mathbf{let}\ \mathrm{k} \Leftarrow [\llbracket \mathrm{M} \rrbracket]\ \mathbf{in}\ \mathbf{let}\ \mathrm{x} \Leftarrow [\llbracket \mathrm{N} \rrbracket]\ \mathbf{in}\ [\mathrm{k}\ \mathrm{x}]$$
$$= \qquad (\mathrm{left})$$
$$\lambda^{\bullet} \langle \rangle.\ [\llbracket \mathrm{M} \rrbracket\ \llbracket \mathrm{N} \rrbracket]$$
$$= \qquad (\mathrm{def}\ \llbracket - \rrbracket)$$
$$\llbracket \mathtt{pure}\ (\mathrm{M}\ \mathrm{N}) \rrbracket$$

3. (I_3)

$\llbracket \texttt{pure}\,(\cdot) \circledast \texttt{L} \circledast \texttt{M} \circledast \texttt{N} \rrbracket$

$= \quad (\text{def } \llbracket - \rrbracket, \beta^{\rightarrow})$

$\lambda^{\bullet}\langle\rangle.\,\textbf{let}\,\texttt{l} \Leftarrow (\lambda^{\bullet}\langle\rangle.\,\textbf{let}\,\texttt{k} \Leftarrow (\lambda^{\bullet}\langle\rangle.\,\textbf{let}\,\texttt{j} \Leftarrow (\lambda^{\bullet}\langle\rangle.\,[(\cdot)]) \bullet \langle\rangle\,\textbf{in}$
$\qquad\qquad\qquad\qquad\qquad\qquad\qquad \textbf{let}\,\texttt{y} \Leftarrow \llbracket \texttt{L} \rrbracket \bullet \langle\rangle\,\textbf{in}\,[\texttt{j}\,\texttt{y}]) \bullet \langle\rangle\,\textbf{in}$
$\qquad\qquad\qquad\qquad \textbf{let}\,\texttt{x} \Leftarrow \llbracket \texttt{M} \rrbracket \bullet \langle\rangle\,\textbf{in}\,[\texttt{k}\,\texttt{x}]) \bullet \langle\rangle\,\textbf{in}$
$\qquad \textbf{let}\,\texttt{m} \Leftarrow \llbracket \texttt{N} \rrbracket \bullet \langle\rangle\,\textbf{in}\,[\texttt{l}\,\texttt{m}]$

$= \quad (\beta^{\rightsquigarrow})$

$\lambda^{\bullet}\langle\rangle.\,\textbf{let}\,\texttt{l} \Leftarrow \textbf{let}\,\texttt{k} \Leftarrow \textbf{let}\,\texttt{j} \Leftarrow [(\cdot)]\,\textbf{in}$
$\qquad\qquad\qquad\qquad\qquad\qquad \textbf{let}\,\texttt{y} \Leftarrow \llbracket \texttt{L} \rrbracket \bullet \langle\rangle\,\textbf{in}\,[\texttt{j}\,\texttt{y}]\,\textbf{in}$
$\qquad\qquad\qquad\qquad \textbf{let}\,\texttt{x} \Leftarrow \llbracket \texttt{M} \rrbracket \bullet \langle\rangle\,\textbf{in}\,[\texttt{k}\,\texttt{x}]\,\textbf{in}$
$\qquad \textbf{let}\,\texttt{m} \Leftarrow \llbracket \texttt{N} \rrbracket \bullet \langle\rangle\,\textbf{in}\,[\texttt{l}\,\texttt{m}]$

$= \quad (\text{assoc})$

$\lambda^{\bullet}\langle\rangle.\,\textbf{let}\,\texttt{j} \Leftarrow [(\cdot)]\,\textbf{in}$
$\qquad \textbf{let}\,\texttt{y} \Leftarrow \llbracket \texttt{L} \rrbracket \bullet \langle\rangle\,\textbf{in}$
$\qquad \textbf{let}\,\texttt{k} \Leftarrow [\texttt{j}\,\texttt{y}]\,\textbf{in}$
$\qquad \textbf{let}\,\texttt{x} \Leftarrow \llbracket \texttt{M} \rrbracket \bullet \langle\rangle\,\textbf{in}$
$\qquad \textbf{let}\,\texttt{l} \Leftarrow [\texttt{k}\,\texttt{x}]\,\textbf{in}$
$\qquad \textbf{let}\,\texttt{m} \Leftarrow \llbracket \texttt{N} \rrbracket \bullet \langle\rangle\,\textbf{in}\,[\texttt{l}\,\texttt{m}]$

$= \quad (\text{left})$

$\lambda^{\bullet}\langle\rangle.\,\textbf{let}\,\texttt{y} \Leftarrow \llbracket \texttt{L} \rrbracket \bullet \langle\rangle\,\textbf{in}$
$\qquad \textbf{let}\,\texttt{x} \Leftarrow \llbracket \texttt{M} \rrbracket \bullet \langle\rangle\,\textbf{in}$
$\qquad \textbf{let}\,\texttt{m} \Leftarrow \llbracket \texttt{N} \rrbracket \bullet \langle\rangle\,\textbf{in}\,[(\cdot)\,\texttt{y}\,\texttt{x}\,\texttt{m}]$

$= \quad (\text{def}\,\cdot)$

$\lambda^{\bullet}\langle\rangle.\,\textbf{let}\,\texttt{k} \Leftarrow \llbracket \texttt{L} \rrbracket \bullet \langle\rangle\,\textbf{in}$
$\qquad \textbf{let}\,\texttt{j} \Leftarrow \llbracket \texttt{M} \rrbracket \bullet \langle\rangle\,\textbf{in}$
$\qquad \textbf{let}\,\texttt{y} \Leftarrow \llbracket \texttt{N} \rrbracket \bullet \langle\rangle\,\textbf{in}\,[\texttt{k}\,(\texttt{j}\,\texttt{y})]$

$= \quad (\text{left})$

$\lambda^{\bullet}\langle\rangle.\,\textbf{let}\,\texttt{k} \Leftarrow \llbracket \texttt{L} \rrbracket \bullet \langle\rangle\,\textbf{in}$
$\qquad \textbf{let}\,\texttt{j} \Leftarrow \llbracket \texttt{M} \rrbracket \bullet \langle\rangle\,\textbf{in}$
$\qquad \textbf{let}\,\texttt{y} \Leftarrow \llbracket \texttt{N} \rrbracket \bullet \langle\rangle\,\textbf{in}$
$\qquad \textbf{let}\,\texttt{x} \Leftarrow [\texttt{j}\,\texttt{y}]\,\textbf{in}\,[\texttt{k}\,\texttt{x}]$

$= \quad (\text{assoc})$

$\lambda^{\bullet}\langle\rangle.\,\textbf{let}\,\texttt{k} \Leftarrow \llbracket \texttt{L} \rrbracket \bullet \langle\rangle\,\textbf{in}$
$\qquad \textbf{let}\,\texttt{x} \Leftarrow \textbf{let}\,\texttt{j} \Leftarrow \llbracket \texttt{M} \rrbracket \bullet \langle\rangle\,\textbf{in}\,\textbf{let}\,\texttt{y} \Leftarrow \llbracket \texttt{N} \rrbracket \bullet \langle\rangle\,\textbf{in}\,[\texttt{j}\,\texttt{y}]\,\textbf{in}$
$\qquad\quad [\texttt{k}\,\texttt{x}]$

$= \quad (\beta^{\rightsquigarrow})$

$\lambda^{\bullet}\langle\rangle.\,\textbf{let}\,\texttt{k} \Leftarrow \llbracket \texttt{L} \rrbracket \bullet \langle\rangle\,\textbf{in}$
$\qquad \textbf{let}\,\texttt{x} \Leftarrow (\lambda^{\bullet}\langle\rangle.\,\textbf{let}\,\texttt{j} \Leftarrow \llbracket \texttt{M} \rrbracket \bullet \langle\rangle\,\textbf{in}\,\textbf{let}\,\texttt{y} \Leftarrow \llbracket \texttt{N} \rrbracket \bullet \langle\rangle\,\textbf{in}\,[\texttt{j}\,\texttt{y}]) \bullet \langle\rangle\,\textbf{in}$
$\qquad\quad [\texttt{k}\,\texttt{x}]$

$= \quad (\text{def } \llbracket - \rrbracket, \beta^{\rightarrow})$

$\llbracket \texttt{L} \circledast (\texttt{M} \circledast \texttt{N}) \rrbracket$

4. (I_4)

$$\llbracket M \circledast \text{pure } N \rrbracket$$
$=$ $(\text{def } \llbracket - \rrbracket, \beta^\rightarrow)$
$$\lambda^\bullet \langle \rangle. \, \mathtt{let} \; k \Leftarrow \llbracket M \rrbracket \bullet \langle \rangle \; \mathtt{in} \; \mathtt{let} \; x \Leftarrow (\lambda^\bullet \langle \rangle. \, [\llbracket N \rrbracket]) \bullet \langle \rangle \; \mathtt{in} \; [k \; x]$$
$=$ $(\beta^{\rightsquigarrow})$
$$\lambda^\bullet \langle \rangle. \, \mathtt{let} \; k \Leftarrow \llbracket M \rrbracket \bullet \langle \rangle \; \mathtt{in} \; \mathtt{let} \; x \Leftarrow [\llbracket N \rrbracket] \; \mathtt{in} \; [k \; x]$$
$=$ (left)
$$\lambda^\bullet \langle \rangle. \, \mathtt{let} \; k \Leftarrow \llbracket M \rrbracket \bullet \langle \rangle \; \mathtt{in} \; [k \; \llbracket N \rrbracket]$$
$=$ $(\beta^\rightarrow)$
$$\lambda^\bullet \langle \rangle. \, \mathtt{let} \; x \Leftarrow \llbracket M \rrbracket \bullet \langle \rangle \; \mathtt{in} \; [(\lambda f. \, f \; \llbracket N \rrbracket) \; x]$$
$=$ (left)
$$\lambda^\bullet \langle \rangle. \, \mathtt{let} \; k \Leftarrow [\lambda f. \, f \; \llbracket N \rrbracket] \; \mathtt{in} \; \mathtt{let} \; x \Leftarrow \llbracket M \rrbracket \bullet \langle \rangle \; \mathtt{in} \; [k \; x]$$
$=$ $(\beta^{\rightsquigarrow})$
$$\lambda^\bullet \langle \rangle. \, \mathtt{let} \; k \Leftarrow (\lambda^\bullet \langle \rangle. \, [\lambda f. \, f \; \llbracket N \rrbracket]) \bullet \langle \rangle \; \mathtt{in} \; \mathtt{let} \; x \Leftarrow \llbracket M \rrbracket \bullet \langle \rangle \; \mathtt{in} \; [k \; x]$$
$=$ $(\text{def } \llbracket - \rrbracket, \beta^\rightarrow)$
$$\llbracket \text{pure } (\lambda f. \, f \; N) \circledast M \rrbracket$$

A.4.3   The laws of $\mathbb{S}$ follow from the laws of $\mathbb{J}$

For each law $P = Q$ of $\mathbb{S}$ we must show $\langle\!\langle P \rangle\!\rangle_\Delta = \langle\!\langle Q \rangle\!\rangle_\Delta$.

1. $(\beta^{\rightsquigarrow})$

$$\langle\!\langle (\lambda^\bullet x. \, Q) \bullet M \rangle\!\rangle_\Delta$$
$=$ $(\text{def } \langle\!\langle - \rangle\!\rangle_\Delta)$
$$\text{pure} \, (\lambda l. \, \lambda \Delta. \, l \; \langle\!\langle M \rangle\!\rangle) \circledast \langle\!\langle Q \rangle\!\rangle_x$$
$=$ $(I_2, \beta^\rightarrow)$
$$\text{pure} \, (\cdot) \circledast \text{pure} \, (\lambda g. \, \lambda \Delta. \, g \; \langle \Delta, \langle\!\langle M \rangle\!\rangle \rangle) \circledast \text{pure} \, (\lambda g. \, \lambda \langle \Delta, x \rangle. \, g \; x) \circledast \langle\!\langle Q \rangle\!\rangle_x$$
$=$ (I_3)
$$\text{pure} \, (\lambda g. \, \lambda \Delta. \, g \; \langle \Delta, \langle\!\langle M \rangle\!\rangle \rangle) \circledast (\text{pure} \, (\lambda g. \, \lambda \langle \Delta, x \rangle. \, g \; x) \circledast \langle\!\langle Q \rangle\!\rangle_x)$$
$=$ $(\text{Lemma } 14)$
$$\text{pure} \, (\lambda g. \, \lambda \Delta. \, g \; \langle \Delta, \langle\!\langle M \rangle\!\rangle \rangle) \circledast \langle\!\langle Q \rangle\!\rangle_{\Delta, x}$$
$=$ $(\text{Lemma } 13)$
$$\langle\!\langle Q[x := M] \rangle\!\rangle_\Delta$$

2. $(\eta^{\rightsquigarrow})$

$$\langle\!\langle \lambda^\bullet x. \, L \bullet x \rangle\!\rangle$$
$=$ $(\text{def } \langle\!\langle - \rangle\!\rangle)$
$$\text{pure} \, (\lambda l. \, \lambda x. \, l \; x) \circledast \langle\!\langle L \rangle\!\rangle$$
$=$ $((\lambda l. \, \lambda x. \, l \; x) = \text{id}, I_1)$
$$\langle\!\langle L \rangle\!\rangle$$

3. (left)

$$\langle\!\langle \textbf{let } \texttt{x} \Leftarrow [\texttt{M}] \textbf{ in } \texttt{Q}\rangle\!\rangle_\Delta$$
$$= \qquad (\text{def } \langle\!\langle - \rangle\!\rangle_\Delta)$$
$$\texttt{pure } (\lambda\texttt{p}.\,\lambda\texttt{q}.\,\lambda\Delta.\,\texttt{q}\,\langle\Delta, \texttt{p}\,\Delta\rangle) \circledast \texttt{pure } (\lambda\Delta.\,\langle\!\langle\texttt{M}\rangle\!\rangle) \circledast \langle\!\langle\texttt{Q}\rangle\!\rangle_{\Delta,\texttt{x}}$$
$$= \qquad (\text{I}_2, \beta^\rightarrow)$$
$$\texttt{pure } (\lambda\texttt{q}.\,\lambda\Delta.\,\texttt{q}\,\langle\Delta, \langle\!\langle\texttt{M}\rangle\!\rangle\rangle) \circledast \langle\!\langle\texttt{Q}\rangle\!\rangle_{\Delta,\texttt{x}}$$
$$= \qquad (\text{Lemma } 13)$$
$$\langle\!\langle\texttt{Q}[\texttt{x} := \texttt{M}]\rangle\!\rangle_\Delta$$

4. (right)

$$\langle\!\langle \textbf{let } \texttt{x} \Leftarrow \texttt{P} \textbf{ in } [\texttt{x}]\rangle\!\rangle_\Delta$$
$$= \qquad (\text{def } \langle\!\langle - \rangle\!\rangle_\Delta)$$
$$\texttt{pure } (\lambda\texttt{p}.\,\lambda\texttt{q}.\,\lambda\Delta.\,\texttt{q}\,\langle\Delta, \texttt{p}\,\Delta\rangle) \circledast \langle\!\langle\texttt{P}\rangle\!\rangle_\Delta \circledast \texttt{pure } (\lambda\langle\Delta, \texttt{x}\rangle.\,\texttt{x})$$
$$= \qquad (\text{I}_4)$$
$$\texttt{pure } (\lambda\texttt{f}.\,\texttt{f}\,(\lambda\langle\Delta, \texttt{x}\rangle.\,\texttt{x})) \circledast (\texttt{pure } (\lambda\texttt{p}.\,\lambda\texttt{q}.\,\lambda\Delta.\,\texttt{q}\,\langle\Delta, \texttt{p}\,\Delta\rangle) \circledast \langle\!\langle\texttt{P}\rangle\!\rangle_\Delta)$$
$$= \qquad (\text{I}_3)$$
$$\texttt{pure } (\cdot) \circledast \texttt{pure } (\lambda\texttt{f}.\,\texttt{f}\,(\lambda\langle\Delta, \texttt{x}\rangle.\,\texttt{x})) \circledast \texttt{pure } (\lambda\texttt{p}.\,\lambda\texttt{q}.\,\lambda\Delta.\,\texttt{q}\,\langle\Delta, \texttt{p}\,\Delta\rangle) \circledast \langle\!\langle\texttt{P}\rangle\!\rangle_\Delta$$
$$= \qquad (\text{I}_2, \beta^\rightarrow)$$
$$\texttt{pure } (\lambda\texttt{c}.\,\lambda\Delta.\,\texttt{c}\,\Delta) \circledast \langle\!\langle\texttt{P}\rangle\!\rangle_\Delta$$
$$= \qquad ((\lambda\texttt{c}.\,\lambda\Delta.\,\texttt{c}\,\Delta) = \texttt{id}, \text{I}_1)$$
$$\langle\!\langle\texttt{P}\rangle\!\rangle_\Delta$$

5. (assoc)

$$\langle\!\langle \textbf{let } \texttt{y} \Leftarrow (\textbf{let } \texttt{x} \Leftarrow \texttt{P} \textbf{ in } \texttt{Q}) \textbf{ in } \texttt{R}\rangle\!\rangle_\Delta$$
$$= \qquad (\text{def } \langle\!\langle - \rangle\!\rangle_\Delta)$$
$$\texttt{pure } (\lambda\texttt{s}.\,\lambda\texttt{r}.\,\lambda\Delta.\,\texttt{r}\,\langle\Delta, \texttt{s}\,\Delta\rangle) \circledast (\texttt{pure } (\lambda\texttt{p}.\,\lambda\texttt{q}.\,\lambda\Delta.\,\texttt{q}\,\langle\Delta, \texttt{p}\,\Delta\rangle) \circledast \langle\!\langle\texttt{P}\rangle\!\rangle_\Delta \circledast \langle\!\langle\texttt{Q}\rangle\!\rangle_{\Delta,\texttt{x}})$$
$$\qquad \circledast \langle\!\langle\texttt{R}\rangle\!\rangle_{\Delta,\texttt{y}}$$
$$= \qquad (\text{I}_3, \text{I}_2, \beta^\rightarrow)$$
$$\texttt{pure } (\lambda\texttt{b}.\,\lambda\texttt{c}.\,\lambda\texttt{r}.\,\lambda\Delta.\,\texttt{r}\,\langle\Delta, (\texttt{b c})\,\Delta\rangle) \circledast (\texttt{pure } (\lambda\texttt{p}.\,\lambda\texttt{q}.\,\lambda\Delta.\,\texttt{q}\,\langle\Delta, \texttt{p}\,\Delta\rangle) \circledast \langle\!\langle\texttt{P}\rangle\!\rangle_\Delta)$$
$$\qquad \circledast \langle\!\langle\texttt{Q}\rangle\!\rangle_{\Delta,\texttt{x}} \circledast \langle\!\langle\texttt{R}\rangle\!\rangle_{\Delta,\texttt{y}}$$
$$= \qquad (\text{I}_3, \text{I}_2, \beta^\rightarrow)$$
$$\texttt{pure } (\lambda\texttt{l}.\,\lambda\texttt{c}.\,\lambda\texttt{r}.\,\lambda\Delta.\,\texttt{r}\,\langle\Delta, (\texttt{c}\,\langle\Delta, \texttt{l}\,\Delta\rangle)\rangle) \circledast \langle\!\langle\texttt{P}\rangle\!\rangle_\Delta \circledast \langle\!\langle\texttt{Q}\rangle\!\rangle_{\Delta,\texttt{x}} \circledast \langle\!\langle\texttt{R}\rangle\!\rangle_{\Delta,\texttt{y}}$$
$$= \qquad (\text{I}_3, \text{I}_2, \beta^\rightarrow)$$
$$\texttt{pure } (\lambda\texttt{t}.\,\lambda\texttt{e}.\,\lambda\texttt{i}.\,\texttt{t e}\,(\lambda\langle\langle\Delta, \texttt{x}\rangle, \texttt{y}\rangle.\,\texttt{i}\,\langle\Delta, \texttt{y}\rangle))$$
$$\qquad \circledast (\texttt{pure } (\lambda\texttt{c}.\,\lambda\texttt{g}.\,\lambda\texttt{l}.\,\lambda\Delta.\,\texttt{l}\,\langle\langle\Delta, \texttt{c}\,\Delta\rangle, \texttt{g}\,\langle\Delta, \texttt{c}\,\Delta\rangle\rangle) \circledast \langle\!\langle\texttt{P}\rangle\!\rangle_\Delta)$$
$$\qquad \circledast \langle\!\langle\texttt{Q}\rangle\!\rangle_{\Delta,\texttt{x}} \circledast \langle\!\langle\texttt{R}\rangle\!\rangle_{\Delta,\texttt{y}}$$
$$= \qquad (\text{I}_3, \text{I}_2, \beta^\rightarrow)$$
$$\texttt{pure } (\lambda\texttt{b}.\,\lambda\texttt{e}.\,\texttt{b e}\,(\lambda\texttt{g}.\,\lambda\langle\langle\Delta, \texttt{x}\rangle, \texttt{y}\rangle.\,\texttt{g}\,\langle\Delta, \texttt{y}\rangle))$$
$$\qquad \circledast (\texttt{pure } ((\cdot)\,(\cdot)) \circledast (\texttt{pure } (\lambda\texttt{c}.\,\lambda\texttt{g}.\,\lambda\texttt{l}.\,\lambda\Delta.\,\texttt{l}\,\langle\langle\Delta, \texttt{c}\,\Delta\rangle, \texttt{g}\,\langle\Delta, \texttt{c}\,\Delta\rangle\rangle) \circledast \langle\!\langle\texttt{P}\rangle\!\rangle_\Delta))$$
$$\qquad \circledast \langle\!\langle\texttt{Q}\rangle\!\rangle_{\Delta,\texttt{x}} \circledast \langle\!\langle\texttt{R}\rangle\!\rangle_{\Delta,\texttt{y}}$$

(continued on next page)

(continued from previous page)
$$\texttt{pure } (\lambda\texttt{b}.\,\lambda\texttt{e}.\,\texttt{b e } (\lambda\texttt{g}.\,\lambda\langle\langle\Delta,\texttt{x}\rangle,\texttt{y}\rangle.\,\texttt{g } \langle\Delta,\texttt{y}\rangle))$$
$$\circledast \;(\texttt{pure } ((\cdot)\,(\cdot)) \circledast (\texttt{pure } (\lambda\texttt{c}.\,\lambda\texttt{g}.\,\lambda\texttt{l}.\,\lambda\Delta.\,\texttt{l } \langle\langle\Delta,\texttt{c } \Delta\rangle,\texttt{g } \langle\Delta,\texttt{c } \Delta\rangle\rangle) \circledast ⦅\texttt{P}⦆_\Delta))$$
$$\circledast \;⦅\texttt{Q}⦆_{\Delta,\texttt{x}} \circledast ⦅\texttt{R}⦆_{\Delta,\texttt{y}}$$
$$= \qquad (\texttt{I}_3,\texttt{I}_2,\beta^\rightarrow)$$
$$\texttt{pure } (\lambda\texttt{h}.\,\texttt{h } (\lambda\texttt{g}.\,\lambda\langle\langle\Delta,\texttt{x}\rangle,\texttt{y}\rangle.\,\texttt{g } \langle\Delta,\texttt{y}\rangle))$$
$$\circledast \;(\texttt{pure } ((\cdot)\,(\cdot)) \circledast (\texttt{pure } (\lambda\texttt{c}.\,\lambda\texttt{g}.\,\lambda\texttt{l}.\,\lambda\Delta.\,\texttt{l } \langle\langle\Delta,\texttt{c } \Delta\rangle,\texttt{g } \langle\Delta,\texttt{c } \Delta\rangle\rangle) \circledast ⦅\texttt{P}⦆_\Delta)$$
$$\circledast \;⦅\texttt{Q}⦆_{\Delta,\texttt{x}})$$
$$\circledast \;⦅\texttt{R}⦆_{\Delta,\texttt{y}}$$
$$= \qquad (\texttt{I}_4)$$
$$\texttt{pure } ((\cdot)\,(\cdot)) \circledast (\texttt{pure } (\lambda\texttt{c}.\,\lambda\texttt{g}.\,\lambda\texttt{l}.\,\lambda\Delta.\,\texttt{l } \langle\langle\Delta,\texttt{c } \Delta\rangle,\texttt{g } \langle\Delta,\texttt{c } \Delta\rangle\rangle) \circledast ⦅\texttt{P}⦆_\Delta)$$
$$\circledast \;⦅\texttt{Q}⦆_{\Delta,\texttt{x}} \circledast \texttt{pure } (\lambda\texttt{g}.\,\lambda\langle\langle\Delta,\texttt{x}\rangle,\texttt{y}\rangle.\,\texttt{g } \langle\Delta,\texttt{y}\rangle) \circledast ⦅\texttt{R}⦆_{\Delta,\texttt{y}}$$
$$= \qquad (\texttt{I}_3,\texttt{I}_2,\beta^\rightarrow)$$
$$\texttt{pure } (\cdot) \circledast (\texttt{pure } (\lambda\texttt{c}.\,\lambda\texttt{g}.\,\lambda\texttt{l}.\,\lambda\Delta.\,\texttt{l } \langle\langle\Delta,\texttt{c } \Delta\rangle,\texttt{g } \langle\Delta,\texttt{c } \Delta\rangle\rangle) \circledast ⦅\texttt{P}⦆_\Delta \circledast ⦅\texttt{Q}⦆_{\Delta,\texttt{x}})$$
$$\circledast \;\texttt{pure } (\lambda\texttt{g}.\,\lambda\langle\langle\Delta,\texttt{x}\rangle,\texttt{y}\rangle.\,\texttt{g } \langle\Delta,\texttt{y}\rangle) \circledast ⦅\texttt{R}⦆_{\Delta,\texttt{y}}$$
$$= \qquad (\texttt{I}_3)$$
$$\texttt{pure } (\lambda\texttt{c}.\,\lambda\texttt{g}.\,\lambda\texttt{l}.\,\lambda\Delta.\,\texttt{l } \langle\langle\Delta,\texttt{c } \Delta\rangle,\texttt{g } \langle\Delta,\texttt{c } \Delta\rangle\rangle) \circledast ⦅\texttt{P}⦆_\Delta \circledast ⦅\texttt{Q}⦆_{\Delta,\texttt{x}}$$
$$\circledast \;(\texttt{pure } (\lambda\texttt{g}.\,\lambda\langle\langle\Delta,\texttt{x}\rangle,\texttt{y}\rangle.\,\texttt{g } \langle\Delta,\texttt{y}\rangle) \circledast ⦅\texttt{R}⦆_{\Delta,\texttt{y}})$$
$$= \qquad (\texttt{I}_3,\texttt{I}_2,\beta^\rightarrow)$$
$$\texttt{pure } (\lambda\texttt{f}.\,\texttt{f } (\lambda\texttt{q}.\,\lambda\texttt{r}.\,\lambda\Delta.\,\texttt{r } \langle\Delta,\texttt{q } \Delta\rangle))$$
$$\circledast \;(\texttt{pure } (\lambda\texttt{e}.\,\lambda\texttt{f}.\,\lambda\texttt{g}.\,\lambda\texttt{l}.\,\lambda\Delta.\,\texttt{f g l } \langle\Delta,\texttt{e } \Delta\rangle) \circledast ⦅\texttt{P}⦆_\Delta)$$
$$\circledast \;⦅\texttt{Q}⦆_{\Delta,\texttt{x}} \circledast (\texttt{pure } (\lambda\texttt{g}.\,\lambda\langle\langle\Delta,\texttt{x}\rangle,\texttt{y}\rangle.\,\texttt{g } \langle\Delta,\texttt{y}\rangle) \circledast ⦅\texttt{R}⦆_{\Delta,\texttt{y}})$$
$$= \qquad (\texttt{I}_4)$$
$$\texttt{pure } (\lambda\texttt{e}.\,\lambda\texttt{f}.\,\lambda\texttt{g}.\,\lambda\texttt{l}.\,\lambda\Delta.\,\texttt{f g l } \langle\Delta,\texttt{e } \Delta\rangle) \circledast ⦅\texttt{P}⦆_\Delta$$
$$\circledast \;\texttt{pure } (\lambda\texttt{q}.\,\lambda\texttt{r}.\,\lambda\Delta.\,\texttt{r } \langle\Delta,\texttt{q } \Delta\rangle)$$
$$\circledast \;⦅\texttt{Q}⦆_{\Delta,\texttt{x}} \circledast (\texttt{pure } (\lambda\texttt{g}.\,\lambda\langle\langle\Delta,\texttt{x}\rangle,\texttt{y}\rangle.\,\texttt{g } \langle\Delta,\texttt{y}\rangle) \circledast ⦅\texttt{R}⦆_{\Delta,\texttt{y}})$$
$$= \qquad (\texttt{I}_3,\texttt{I}_2,\beta^\rightarrow)$$
$$\texttt{pure } (\cdot) \circledast (\texttt{pure } (\lambda\texttt{c}.\,\lambda\texttt{k}.\,\lambda\texttt{l}.\,\lambda\Delta.\,\texttt{k l } \langle\Delta,\texttt{c } \Delta\rangle) \circledast ⦅\texttt{P}⦆_\Delta)$$
$$\circledast \;\texttt{pure } (\lambda\texttt{q}.\,\lambda\texttt{r}.\,\lambda\Delta.\,\texttt{r } \langle\Delta,\texttt{q } \Delta\rangle)$$
$$\circledast \;⦅\texttt{Q}⦆_{\Delta,\texttt{x}} \circledast (\texttt{pure } (\lambda\texttt{g}.\,\lambda\langle\langle\Delta,\texttt{x}\rangle,\texttt{y}\rangle.\,\texttt{g } \langle\Delta,\texttt{y}\rangle) \circledast ⦅\texttt{R}⦆_{\Delta,\texttt{y}})$$
$$= \qquad (\texttt{I}_3)$$
$$\texttt{pure } (\lambda\texttt{c}.\,\lambda\texttt{k}.\,\lambda\texttt{l}.\,\lambda\Delta.\,\texttt{k l } \langle\Delta,\texttt{c } \Delta\rangle) \circledast ⦅\texttt{P}⦆_\Delta$$
$$\circledast \;(\texttt{pure } (\lambda\texttt{q}.\,\lambda\texttt{r}.\,\lambda\Delta.\,\texttt{r } \langle\Delta,\texttt{q } \Delta\rangle) \circledast ⦅\texttt{Q}⦆_{\Delta,\texttt{x}})$$
$$\circledast \;(\texttt{pure } (\lambda\texttt{g}.\,\lambda\langle\langle\Delta,\texttt{x}\rangle,\texttt{y}\rangle.\,\texttt{g } \langle\Delta,\texttt{y}\rangle) \circledast ⦅\texttt{R}⦆_{\Delta,\texttt{y}})$$
$$= \qquad (\texttt{I}_3,\texttt{I}_2,\beta^\rightarrow)$$
$$\texttt{pure } (\cdot) \circledast (\texttt{pure } (\lambda\texttt{p}.\,\lambda\texttt{q}.\,\lambda\Delta.\,\texttt{q } \langle\Delta,\texttt{p } \Delta\rangle) \circledast ⦅\texttt{P}⦆_\Delta)$$
$$\circledast \;(\texttt{pure } (\lambda\texttt{q}.\,\lambda\texttt{r}.\,\lambda\Delta.\,\texttt{r } \langle\Delta,\texttt{q } \Delta\rangle) \circledast ⦅\texttt{Q}⦆_{\Delta,\texttt{x}})$$
$$\circledast \;(\texttt{pure } (\lambda\texttt{g}.\,\lambda\langle\langle\Delta,\texttt{x}\rangle,\texttt{y}\rangle.\,\texttt{g } \langle\Delta,\texttt{y}\rangle) \circledast ⦅\texttt{R}⦆_{\Delta,\texttt{y}})$$

(continued from previous page)
$$\texttt{pure}\ (\cdot) \circledast (\texttt{pure}\ (\lambda\texttt{p}.\ \lambda\texttt{q}.\ \lambda\Delta.\ \texttt{q}\ \langle\Delta, \texttt{p}\ \Delta\rangle) \circledast \langle\!\langle\texttt{P}\rangle\!\rangle_\Delta)$$
$$\circledast\ (\texttt{pure}\ (\lambda\texttt{q}.\ \lambda\texttt{r}.\ \lambda\Delta.\ \texttt{r}\ \langle\Delta, \texttt{q}\ \Delta\rangle) \circledast \langle\!\langle\texttt{Q}\rangle\!\rangle_{\Delta,\texttt{x}})$$
$$\circledast\ (\texttt{pure}\ (\lambda\texttt{g}.\ \lambda\langle\langle\Delta,\texttt{x}\rangle,\texttt{y}\rangle.\ \texttt{g}\ \langle\Delta,\texttt{y}\rangle) \circledast \langle\!\langle\texttt{R}\rangle\!\rangle_{\Delta,\texttt{y}})$$

$=$ (I$_3$)
$$\texttt{pure}\ (\lambda\texttt{p}.\ \lambda\texttt{q}.\ \lambda\Delta.\ \texttt{q}\ \langle\Delta, \texttt{p}\ \Delta\rangle) \circledast \langle\!\langle\texttt{P}\rangle\!\rangle_\Delta$$
$$\circledast\ (\texttt{pure}\ (\lambda\texttt{q}.\ \lambda\texttt{r}.\ \lambda\Delta.\ \texttt{r}\ \langle\Delta, \texttt{q}\ \Delta\rangle) \circledast \langle\!\langle\texttt{Q}\rangle\!\rangle_{\Delta,\texttt{x}}$$
$$\circledast\ (\texttt{pure}\ (\lambda\texttt{g}.\ \lambda\langle\langle\Delta,\texttt{x}\rangle,\texttt{y}\rangle.\ \texttt{g}\ \langle\Delta,\texttt{y}\rangle) \circledast \langle\!\langle\texttt{R}\rangle\!\rangle_{\Delta,\texttt{y}}))$$

$=$ (Lemma 14)
$$\texttt{pure}\ (\lambda\texttt{p}.\ \lambda\texttt{q}.\ \lambda\Delta.\ \texttt{q}\ \langle\Delta, \texttt{p}\ \Delta\rangle) \circledast \langle\!\langle\texttt{P}\rangle\!\rangle_\Delta$$
$$\circledast\ (\texttt{pure}\ (\lambda\texttt{q}.\ \lambda\texttt{r}.\ \lambda\Delta.\ \texttt{r}\ \langle\Delta, \texttt{q}\ \Delta\rangle) \circledast \langle\!\langle\texttt{Q}\rangle\!\rangle_{\Delta,\texttt{x}} \circledast \langle\!\langle\texttt{R}\rangle\!\rangle_{\Delta,\texttt{x},\texttt{y}})$$

$=$ (def $\langle\!\langle-\rangle\!\rangle_\Delta$)
$$\langle\!\langle\textbf{let}\ \texttt{x} \Leftarrow \texttt{P}\ \textbf{in let}\ \texttt{y} \Leftarrow \texttt{Q}\ \textbf{in R}\rangle\!\rangle_\Delta$$

6. (run_1)

$$\langle\!\langle\texttt{L} \bullet \texttt{M}\rangle\!\rangle_\Delta$$
$=$ (def $\langle\!\langle-\rangle\!\rangle_\Delta$)
$$\texttt{pure}\ (\lambda\texttt{l}.\ \lambda\Delta.\ \texttt{l}\ \langle\!\langle\texttt{M}\rangle\!\rangle) \circledast \langle\!\langle\texttt{L}\rangle\!\rangle$$
$=$ (I$_2$, $\beta^\rightarrow$)
$$\texttt{pure}\ (\cdot) \circledast \texttt{pure}\ (\lambda\texttt{g}.\ \texttt{g}\ (\lambda\langle\Delta,\texttt{f}\rangle.\ \texttt{f}\ \langle\!\langle\texttt{M}\rangle\!\rangle)) \circledast \texttt{pure}\ (\lambda\texttt{c}.\ \lambda\texttt{q}.\ \lambda\Delta.\ \texttt{q}\ \langle\Delta,\texttt{c}\rangle) \circledast \langle\!\langle\texttt{L}\rangle\!\rangle$$
$=$ (I$_3$)
$$\texttt{pure}\ (\lambda\texttt{g}.\ \texttt{g}\ (\lambda\langle\Delta,\texttt{f}\rangle.\ \texttt{f}\ \langle\!\langle\texttt{M}\rangle\!\rangle)) \circledast (\texttt{pure}\ (\lambda\texttt{c}.\ \lambda\texttt{q}.\ \lambda\Delta.\ \texttt{q}\ \langle\Delta,\texttt{c}\rangle) \circledast \langle\!\langle\texttt{L}\rangle\!\rangle)$$
$=$ (I$_4$)
$$\texttt{pure}\ (\lambda\texttt{c}.\ \lambda\texttt{q}.\ \lambda\Delta.\ \texttt{q}\ \langle\Delta,\texttt{c}\rangle) \circledast \langle\!\langle\texttt{L}\rangle\!\rangle \circledast \texttt{pure}\ (\lambda\langle\Delta,\texttt{f}\rangle.\ \texttt{f}\ \langle\!\langle\texttt{M}\rangle\!\rangle)$$
$=$ (I$_3$, I$_2$, $\beta^\rightarrow$)
$$\texttt{pure}\ (\lambda\texttt{p}.\ \lambda\texttt{q}.\ \lambda\Delta.\ \texttt{q}\ \langle\Delta,\texttt{p}\ \Delta\rangle) \circledast (\texttt{pure}\ (\lambda\texttt{l}.\ \lambda\Delta.\ \texttt{l}) \circledast \langle\!\langle\texttt{L}\rangle\!\rangle) \circledast \texttt{pure}\ (\lambda\langle\Delta,\texttt{f}\rangle.\ \texttt{f}\ \langle\!\langle\texttt{M}\rangle\!\rangle)$$
$=$ (def $\langle\!\langle-\rangle\!\rangle_\Delta$)
$$\langle\!\langle\textbf{let}\ \texttt{f} \Leftarrow \textbf{run}\ \texttt{L}\ \textbf{in}\ [\texttt{f M}]\rangle\!\rangle$$

7. (run_2)

$$\langle\!\langle\textbf{run}\ (\lambda^\bullet\texttt{x}.\ [\texttt{M}])\rangle\!\rangle_\Delta$$
$=$ (def $\langle\!\langle-\rangle\!\rangle_\Delta$)
$$\texttt{pure}\ (\lambda\texttt{l}.\ \lambda\Delta.\ \texttt{l}) \circledast \texttt{pure}\ (\lambda\texttt{x}.\ \langle\!\langle\texttt{M}\rangle\!\rangle)$$
$=$ (I$_2$)
$$\texttt{pure}\ (\lambda\Delta.\ \lambda\texttt{x}.\ \texttt{M})$$
$=$ (def $\langle\!\langle-\rangle\!\rangle_\Delta$)
$$\langle\!\langle[[\lambda\texttt{x}.\ \texttt{M}]]\rangle\!\rangle_\Delta$$

8. (run_3)

$$\langle\!\langle \mathbf{run}\ (\lambda^\bullet x.\ \mathbf{let}\ y \Leftarrow P\ \mathbf{in}\ Q) \rangle\!\rangle_\Delta$$

$= \quad (\text{def } \langle\!\langle - \rangle\!\rangle_\Delta)$

$$\texttt{pure const} \circledast (\texttt{pure}\ (\lambda p.\ \lambda q.\ \lambda x.\ q\ \langle x, p\ x\rangle) \circledast \langle\!\langle P \rangle\!\rangle_x \circledast \langle\!\langle Q \rangle\!\rangle_{x,y})$$

$= \quad (\text{Lemma } 14)$

$$\texttt{pure const} \circledast (\texttt{pure}\ (\lambda p.\ \lambda q.\ \lambda x.\ q\ \langle x, p\ x\rangle) \circledast (\texttt{pure}\ (\lambda g.\ \lambda x.\ g\ \langle\rangle) \circledast \langle\!\langle P \rangle\!\rangle_{\langle\rangle}) \circledast \langle\!\langle Q \rangle\!\rangle_{x,y})$$

$= \quad (\text{I}_3, \text{I}_2, \beta^\rightarrow)$

$$\texttt{pure const} \circledast (\texttt{pure}\ (\lambda g.\ (\lambda q.\ \lambda x.\ q\ \langle x, g\ \langle\rangle\rangle)) \circledast \langle\!\langle P \rangle\!\rangle_{\langle\rangle} \circledast \langle\!\langle Q \rangle\!\rangle_{x,y})$$

$= \quad (\text{I}_3, \text{I}_2)$

$$\texttt{pure}\ (\lambda h.\ \texttt{const} \cdot h) \circledast (\texttt{pure}\ (\lambda g.\ (\lambda q.\ \lambda x.\ q\ \langle x, g\ \langle\rangle\rangle)) \circledast \langle\!\langle P \rangle\!\rangle_{\langle\rangle}) \circledast \langle\!\langle Q \rangle\!\rangle_{x,y}$$

$= \quad (\text{I}_3, \text{I}_2, \beta^\rightarrow)$

$$\texttt{pure}\ (\lambda p.\ \lambda q.\ \lambda d.\ \lambda x.\ q\ \langle x, p\ \langle\rangle\rangle) \circledast \langle\!\langle P \rangle\!\rangle_{\langle\rangle} \circledast \langle\!\langle Q \rangle\!\rangle_{x,y}$$

$= \quad (\text{I}_3, \text{I}_2)$

$$\texttt{pure}\ ((\cdot)\ \texttt{const}) \circledast (\texttt{pure}\ (\lambda p.\ \lambda q.\ \lambda x.\ q\ \langle x, p\ \langle\rangle\rangle) \circledast \langle\!\langle P \rangle\!\rangle_{\langle\rangle}) \circledast \langle\!\langle Q \rangle\!\rangle_{x,y}$$

$= \quad (\text{I}_3, \text{I}_2, \beta^\rightarrow)$

$$\texttt{pure}\ (\lambda p.\ \lambda q.\ \lambda z.\ \lambda x.\ q\ \langle x, p\ \langle\rangle\rangle) \circledast \langle\!\langle P \rangle\!\rangle_{\langle\rangle} \circledast \langle\!\langle Q \rangle\!\rangle_{x,y}$$

$= \quad (\text{I}_3, \text{I}_2, \beta^\rightarrow)$

$$\texttt{pure}\ (\lambda h.\ h\ (\lambda f.\ \lambda\langle\Delta, y\rangle.\ \lambda x.\ f\ \langle x, y\rangle))$$
$$\circledast (\texttt{pure}\ (\lambda p.\ \lambda b.\ \lambda q.\ \lambda z.\ b\ q\ \langle z, p\ \langle\rangle\rangle) \circledast \langle\!\langle P \rangle\!\rangle_{\langle\rangle})$$
$$\circledast \langle\!\langle Q \rangle\!\rangle_{x,y}$$

$= \quad (\text{I}_4)$

$$\texttt{pure}\ (\lambda p.\ \lambda b.\ \lambda q.\ \lambda z.\ b\ q\ \langle z, p\ \langle\rangle\rangle) \circledast \langle\!\langle P \rangle\!\rangle_{\langle\rangle}$$
$$\circledast \texttt{pure}\ (\lambda f.\ \lambda\langle\Delta, y\rangle.\ \lambda x.\ f\ \langle x, y\rangle) \circledast \langle\!\langle Q \rangle\!\rangle_{x,y}$$

$= \quad (\text{I}_3, \text{I}_2, \beta^\rightarrow)$

$$\texttt{pure}\ (\lambda p.\ (\lambda q.\ \lambda z.\ q\ \langle z, p\ \langle\rangle\rangle)) \circledast \langle\!\langle P \rangle\!\rangle_{\langle\rangle}$$
$$\circledast (\texttt{pure}\ (\lambda h.\ h\ (\lambda\langle\langle\Delta, y\rangle, f\rangle.\ \lambda x.\ f\ \langle x, y\rangle))$$
$$\circledast (\texttt{pure}\ (\lambda n.\ \lambda s.\ \lambda\langle\Delta, y\rangle.\ s\ \langle\langle\Delta, y\rangle, n\rangle) \circledast \langle\!\langle Q \rangle\!\rangle_{x,y}))$$

$= \quad (\text{I}_4)$

$$\texttt{pure}\ (\lambda p.\ (\lambda q.\ \lambda\Delta.\ q\ \langle\Delta, p\ \langle\rangle\rangle)) \circledast \langle\!\langle P \rangle\!\rangle_{\langle\rangle}$$
$$\circledast (\texttt{pure}\ (\lambda n.\ \lambda s.\ \lambda\langle\Delta, y\rangle.\ s\ \langle\langle\Delta, y\rangle, n\rangle) \circledast \langle\!\langle Q \rangle\!\rangle_{x,y}$$
$$\circledast \texttt{pure}\ (\lambda\langle\langle\Delta, y\rangle, f\rangle.\ \lambda x.\ f\ \langle x, y\rangle))$$

$= \quad (\text{I}_3, \text{I}_2, \beta^\rightarrow)$

$$\texttt{pure}\ (\lambda p.\ \lambda q.\ \lambda\Delta.\ q\ \langle\Delta, p\ \Delta\rangle) \circledast (\texttt{pure}\ (\lambda g.\ \lambda\Delta.\ g\ \langle\rangle) \circledast \langle\!\langle P \rangle\!\rangle_{\langle\rangle})$$
$$\circledast (\texttt{pure}\ (\lambda r.\ \lambda s.\ \lambda\langle\Delta, y\rangle.\ s\ \langle\langle\Delta, y\rangle, r\ \langle\Delta, y\rangle\rangle) \circledast (\texttt{pure const} \circledast \langle\!\langle Q \rangle\!\rangle_{x,y})$$
$$\circledast \texttt{pure}\ (\lambda\langle\langle\Delta, y\rangle, f\rangle.\ \lambda x.\ f\ \langle x, y\rangle))$$

$= \quad (\text{Lemma } 14)$

$$\texttt{pure}\ (\lambda p.\ \lambda q.\ \lambda\Delta.\ q\ \langle\Delta, p\ \Delta\rangle) \circledast \langle\!\langle P \rangle\!\rangle_\Delta$$
$$\circledast (\texttt{pure}\ (\lambda r.\ \lambda s.\ \lambda\langle\Delta, y\rangle.\ s\ \langle\langle\Delta, y\rangle, r\ \langle\Delta, y\rangle\rangle) \circledast (\texttt{pure const} \circledast \langle\!\langle Q \rangle\!\rangle_{x,y})$$
$$\circledast \texttt{pure}\ (\lambda\langle\langle\Delta, y\rangle, f\rangle.\ \lambda x.\ f\ \langle x, y\rangle))$$

$= \quad (\text{def } \langle\!\langle - \rangle\!\rangle_\Delta)$

$$\langle\!\langle \mathbf{let}\ y \Leftarrow P\ \mathbf{in}\ \mathbf{let}\ f \Leftarrow \mathbf{run}\ (\lambda^\bullet\langle x, y\rangle.\ Q)\ \mathbf{in}\ [\lambda x.\ f\ \langle x, y\rangle] \rangle\!\rangle_\Delta$$

A.4.4 Translating $\mathcal{J}$ to $\mathcal{S}$ and back

For each term M of $\mathcal{J}$ we must show $\langle\![\![M]\!]\rangle\!\rangle = f_A(M)$. The proof is by induction on the derivation of M.

1. Case $\mathtt{pure}$

$$
\begin{aligned}
&f^{-1}_{A\to I(A)}\ \langle\![\![\mathtt{pure}]\!]\rangle\!\rangle \\
=\quad & (\text{def } [\![-]\!], \text{def } \langle\!(-)\!\rangle) \\
&f^{-1}_{A\to I(A)}\ (\lambda x.\, \mathtt{pure}\ (\lambda\langle\rangle.\, x)) \\
=\quad & (\text{def } f^{-1}_{A\to I(A)}, \beta^\to) \\
&\lambda v.\, \mathtt{pure}\ (\lambda x.\, f^{-1}_A\ (x\ \langle\rangle)) \circledast \mathtt{pure}\ (\lambda\langle\rangle.\, (f_A\ v)) \\
=\quad & (\mathrm{I}_2, \beta^\to) \\
&\lambda v.\, \mathtt{pure}\ (f^{-1}_A\ (f_A\ v)) \\
=\quad & (f^{-1}_A\ (f_A\ v) = v, \eta^\to) \\
&\mathtt{pure}
\end{aligned}
$$

2. Case $(\circledast)$

$$
\begin{aligned}
&f^{-1}_{I(A\to B)\to IA\to IB} \\
&\quad \langle\![\![(\circledast)]\!]\rangle\!\rangle \\
=\quad & (\text{def } [\![-]\!]) \\
&f^{-1}_{I(A\to B)\to IA\to IB} \\
&\quad \langle\!\langle \lambda h.\, \lambda a.\, \lambda^\bullet\langle\rangle.\, \mathbf{let}\ k \Leftarrow h \bullet \langle\rangle\ \mathbf{in}\ \mathbf{let}\ x \Leftarrow a \bullet \langle\rangle\ \mathbf{in}\ [k\ x] \rangle\!\rangle \\
=\quad & (\text{def } \langle\!(-)\!\rangle) \\
&f^{-1}_{I(A\to B)\to IA\to IB} \\
&\quad (\lambda h.\, \lambda a.\, \mathtt{pure}\ (\lambda p.\, \lambda q.\, \lambda\langle\rangle.\, q\ \langle\langle\rangle, p\ \langle\rangle\rangle) \circledast (\mathtt{pure}\ (\lambda l.\, \lambda d.\, l\ \langle\rangle) \circledast h) \\
&\qquad \circledast (\mathtt{pure}\ (\lambda p.\, \lambda q.\, \lambda\langle\langle\rangle, k\rangle.\, q\ \langle\langle\langle\rangle, k\rangle, p\ \langle\langle\rangle, k\rangle\rangle) \circledast (\mathtt{pure}\ (\lambda l.\, \lambda\langle\langle\rangle, k\rangle.\, l\ \langle\rangle) \circledast a) \\
&\qquad\quad \circledast \mathtt{pure}\ (\lambda\langle\langle\langle\rangle, k\rangle, x\rangle.\, k\ x))) \\
=\quad & (\mathrm{I}_3, \mathrm{I}_2, \beta^\to) \\
&f^{-1}_{I(A\to B)\to IA\to IB} \\
&\quad (\lambda h.\, \lambda a.\, \mathtt{pure}\ (\lambda c.\, \lambda q.\, \lambda\langle\rangle.\, q\ \langle\langle\rangle, (c\ \langle\rangle)\rangle) \circledast h \\
&\qquad \circledast (\mathtt{pure}\ (\lambda p.\, \lambda q.\, \lambda\langle\langle\rangle, k\rangle.\, q\ \langle\langle\langle\rangle, k\rangle, p\ \langle\langle\rangle, k\rangle\rangle) \circledast (\mathtt{pure}\ (\lambda l.\, \lambda\langle\langle\rangle, k\rangle.\, l\ \langle\rangle) \circledast a) \\
&\qquad\quad \circledast \mathtt{pure}\ (\lambda\langle\langle\langle\rangle, k\rangle, x\rangle.\, k\ x))) \\
=\quad & (\mathrm{I}_4) \\
&f^{-1}_{I(A\to B)\to IA\to IB} \\
&\quad (\lambda h.\, \lambda a.\, \mathtt{pure}\ (\lambda c.\, \lambda q.\, \lambda\langle\rangle.\, q\ \langle\langle\rangle, (c\ \langle\rangle)\rangle) \circledast h \\
&\qquad \circledast (\mathtt{pure}\ (\lambda f.\, f\ (\lambda\langle\langle\langle\rangle, k\rangle, x\rangle.\, k\ x)) \circledast (\mathtt{pure}\ (\lambda p.\, \lambda q.\, \lambda\langle\langle\rangle, k\rangle.\, q\ \langle\langle\langle\rangle, k\rangle, p\ \langle\langle\rangle, k\rangle\rangle) \\
&\qquad\quad \circledast (\mathtt{pure}\ (\lambda l.\, \lambda\langle\langle\rangle, k\rangle.\, l\ \langle\rangle) \circledast a))))
\end{aligned}
$$

(continued on next page)

(continued from previous page)

$\mathtt{f}^{-1}_{\mathrm{I}(A\to B)\to IA\to IB}$
$\quad(\lambda\mathtt{h}.\,\lambda\mathtt{a}.\,\mathtt{pure}\;(\lambda\mathtt{c}.\,\lambda\mathtt{q}.\,\lambda\langle\rangle.\,\mathtt{q}\;\langle\langle\rangle,(\mathtt{c}\;\langle\rangle)\rangle)\otimes\mathtt{h}$
$\quad\quad\otimes\;(\mathtt{pure}\;(\lambda\mathtt{f}.\,\mathtt{f}\;(\lambda\langle\langle\langle\rangle,\mathtt{k}\rangle,\mathtt{x}\rangle.\,\mathtt{k}\;\mathtt{x}))\otimes(\mathtt{pure}\;(\lambda\mathtt{p}.\,\lambda\mathtt{q}.\,\lambda\langle\langle\rangle,\mathtt{k}\rangle.\,\mathtt{q}\;\langle\langle\langle\rangle,\mathtt{k}\rangle,\mathtt{p}\;\langle\langle\rangle,\mathtt{k}\rangle\rangle)$
$\quad\quad\quad\otimes\;(\mathtt{pure}\;(\lambda\mathtt{l}.\,\lambda\langle\langle\rangle,\mathtt{k}\rangle.\,\mathtt{l}\;\langle\rangle)\otimes\mathtt{a}))))$
$=\quad\quad(\mathrm{I}_3,\mathrm{I}_2,\beta^\to)$

$\mathtt{f}^{-1}_{\mathrm{I}(A\to B)\to IA\to IB}$
$\quad(\lambda\mathtt{h}.\,\lambda\mathtt{a}.\,\mathtt{pure}\;(\cdot)\otimes(\mathtt{pure}\;(\lambda\mathtt{c}.\,\lambda\mathtt{q}.\,\lambda\langle\rangle.\,\mathtt{q}\;\langle\langle\rangle,(\mathtt{c}\;\langle\rangle)\rangle)\otimes\mathtt{h})$
$\quad\quad\otimes\;\mathtt{pure}\;(\lambda\mathtt{f}.\,\mathtt{f}\;(\lambda\langle\langle\langle\rangle,\mathtt{k}\rangle,\mathtt{x}\rangle.\,\mathtt{k}\;\mathtt{x}))$
$\quad\quad\otimes\;(\mathtt{pure}\;(\lambda\mathtt{c}.\,\lambda\mathtt{q}.\,\lambda\langle\langle\rangle,\mathtt{k}\rangle.\,\mathtt{q}\;\langle\langle\langle\rangle,\mathtt{k}\rangle,(\mathtt{c}\;\langle\rangle)\rangle)\otimes\mathtt{a}))$
$=\quad\quad(\mathrm{I}_3,\mathrm{I}_2,\beta^\to)$

$\mathtt{f}^{-1}_{\mathrm{I}(A\to B)\to IA\to IB}$
$\quad(\lambda\mathtt{h}.\,\lambda\mathtt{a}.\,\mathtt{pure}\;(\lambda\mathtt{t}.\,\lambda\mathtt{b}.\,\lambda\mathtt{c}.\,\lambda\langle\rangle.\,(\mathtt{b}\;\mathtt{c})\;\langle\langle\rangle,(\mathtt{t}\;\langle\rangle)\rangle)\otimes\mathtt{h}$
$\quad\quad\otimes\;\mathtt{pure}\;(\lambda\mathtt{f}.\,\mathtt{f}\;(\lambda\langle\langle\langle\rangle,\mathtt{k}\rangle,\mathtt{x}\rangle.\,\mathtt{k}\;\mathtt{x}))$
$\quad\quad\otimes\;(\mathtt{pure}\;(\lambda\mathtt{c}.\,\lambda\mathtt{q}.\,\lambda\langle\langle\rangle,\mathtt{k}\rangle.\,\mathtt{q}\;\langle\langle\langle\rangle,\mathtt{k}\rangle,(\mathtt{c}\;\langle\rangle)\rangle)\otimes\mathtt{a}))$
$=\quad\quad(\mathrm{I}_4)$

$\mathtt{f}^{-1}_{\mathrm{I}(A\to B)\to IA\to IB}$
$\quad(\lambda\mathtt{h}.\,\lambda\mathtt{a}.\,\mathtt{pure}\;(\lambda\mathtt{g}.\,\mathtt{g}\;(\lambda\mathtt{f}.\,\mathtt{f}\;(\lambda\langle\langle\langle\rangle,\mathtt{k}\rangle,\mathtt{x}\rangle.\,\mathtt{k}\;\mathtt{x})))$
$\quad\quad\otimes\;(\mathtt{pure}\;(\lambda\mathtt{t}.\,\lambda\mathtt{b}.\,\lambda\mathtt{c}.\,\lambda\langle\rangle.\,(\mathtt{b}\;\mathtt{c})\;\langle\langle\rangle,(\mathtt{t}\;\langle\rangle)\rangle)\otimes\mathtt{h})$
$\quad\quad\otimes\;(\mathtt{pure}\;(\lambda\mathtt{c}.\,\lambda\mathtt{q}.\,\lambda\langle\langle\rangle,\mathtt{k}\rangle.\,\mathtt{q}\;\langle\langle\langle\rangle,\mathtt{k}\rangle,(\mathtt{c}\;\langle\rangle)\rangle)\otimes\mathtt{a}))$
$=\quad\quad(\mathrm{I}_3,\mathrm{I}_2,\beta^\to)$

$\mathtt{f}^{-1}_{\mathrm{I}(A\to B)\to IA\to IB}$
$\quad(\lambda\mathtt{h}.\,\lambda\mathtt{a}.\,\mathtt{pure}\;(\lambda\mathtt{n}.\,\lambda\mathtt{c}.\,\lambda\langle\rangle.\,\mathtt{c}\;(\lambda\langle\langle\langle\rangle,\mathtt{k}\rangle,\mathtt{x}\rangle.\,\mathtt{k}\;\mathtt{x})\;\langle\langle\rangle,(\mathtt{n}\;\langle\rangle)\rangle)\otimes\mathtt{h}$
$\quad\quad\otimes\;(\mathtt{pure}\;(\lambda\mathtt{c}.\,\lambda\mathtt{q}.\,\lambda\langle\langle\rangle,\mathtt{k}\rangle.\,\mathtt{q}\;\langle\langle\langle\rangle,\mathtt{k}\rangle,(\mathtt{c}\;\langle\rangle)\rangle)\otimes\mathtt{a}))$
$=\quad\quad(\mathrm{I}_3,\mathrm{I}_2,\beta^\to)$

$\mathtt{f}^{-1}_{\mathrm{I}(A\to B)\to IA\to IB}$
$\quad(\lambda\mathtt{h}.\,\lambda\mathtt{a}.\,\mathtt{pure}\;((\lambda\mathtt{k}.\,\lambda\mathtt{e}.\,\lambda\mathtt{f}.\,\lambda\langle\rangle.\,\mathtt{e}\;\mathtt{f}\;(\lambda\langle\langle\langle\rangle,\mathtt{p}\rangle,\mathtt{x}\rangle.\,\mathtt{p}\;\mathtt{x})\;\langle\langle\rangle,(\mathtt{k}\;\langle\rangle)\rangle))\otimes\mathtt{h}$
$\quad\quad\otimes\;\mathtt{pure}\;(\lambda\mathtt{c}.\,\lambda\mathtt{q}.\,\lambda\langle\langle\rangle,\mathtt{k}\rangle.\,\mathtt{q}\;\langle\langle\langle\rangle,\mathtt{k}\rangle,(\mathtt{c}\;\langle\rangle)\rangle)\otimes\mathtt{a})$
$=\quad\quad(\mathrm{I}_4)$

$\mathtt{f}^{-1}_{\mathrm{I}(A\to B)\to IA\to IB}$
$\quad(\lambda\mathtt{h}.\,\lambda\mathtt{a}.\,\mathtt{pure}\;(\lambda\mathtt{g}.\,\mathtt{g}\;(\lambda\mathtt{c}.\,\lambda\mathtt{q}.\,\lambda\langle\langle\rangle,\mathtt{k}\rangle.\,\mathtt{q}\;\langle\langle\langle\rangle,\mathtt{k}\rangle,(\mathtt{c}\;\langle\rangle)\rangle))$
$\quad\quad\otimes\;(\mathtt{pure}\;((\lambda\mathtt{k}.\,\lambda\mathtt{e}.\,\lambda\mathtt{f}.\,\lambda\langle\rangle.\,\mathtt{e}\;\mathtt{f}\;(\lambda\langle\langle\langle\rangle,\mathtt{p}\rangle,\mathtt{x}\rangle.\,\mathtt{p}\;\mathtt{x})\;\langle\langle\rangle,(\mathtt{k}\;\langle\rangle)\rangle))\otimes\mathtt{h})\otimes\mathtt{a})$
$=\quad\quad(\mathrm{I}_3,\mathrm{I}_2,\beta^\to)$

$\mathtt{f}^{-1}_{\mathrm{I}(A\to B)\to IA\to IB}$
$\quad(\lambda\mathtt{h}.\,\lambda\mathtt{a}.\,\mathtt{pure}\;(\lambda\mathtt{p}.\,\lambda\mathtt{q}.\,\lambda\langle\rangle.\,\mathtt{p}\;\langle\rangle\;(\mathtt{q}\;\langle\rangle))\otimes\mathtt{h}\otimes\mathtt{a})$
$=\quad\quad(\mathrm{def}\;\mathtt{f}^{-1}_{\mathrm{I}(A\to B)\to IA\to IB},\beta^\to)$
$(\lambda\mathtt{s}.\,\lambda\mathtt{d}.\,\mathtt{pure}\;(\lambda\mathtt{y}.\,\mathtt{f}^{-1}_\mathtt{B}\;(\mathtt{y}\;\langle\rangle))$
$\quad\quad\otimes\;(\mathtt{pure}\;(\lambda\mathtt{p}.\,\lambda\mathtt{q}.\,\lambda\langle\rangle.\,\mathtt{p}\;\langle\rangle\;(\mathtt{q}\;\langle\rangle))\otimes(\mathtt{pure}\;(\lambda\mathtt{z}.\,\lambda\langle\rangle.\,\mathtt{f}_\mathtt{B}\cdot\mathtt{z}\cdot\mathtt{f}^{-1}_\mathtt{A})\otimes\mathtt{s})$
$\quad\quad\quad\otimes\;(\mathtt{pure}\;(\lambda\mathtt{x}.\,\lambda\langle\rangle.\,\mathtt{f}_\mathtt{A}\;\mathtt{x})\otimes\mathtt{d})))$
$=\quad\quad(\mathrm{I}_3,\mathrm{I}_2,\beta^\to)$
$(\lambda\mathtt{s}.\,\lambda\mathtt{d}.\,\mathtt{pure}\;(\lambda\mathtt{c}.\,\lambda\mathtt{m}.\,\lambda\mathtt{n}.\,\mathtt{f}^{-1}_\mathtt{B}\;(\mathtt{f}_\mathtt{B}\;(\mathtt{c}\;(\mathtt{f}^{-1}_\mathtt{A}\;(\mathtt{m}\;\mathtt{n}\;\langle\rangle)))))\otimes\mathtt{s}$
$\quad\quad\otimes\;\mathtt{pure}\;(\lambda\mathtt{x}.\,\lambda\langle\rangle.\,\mathtt{f}_\mathtt{A}\;\mathtt{x})\otimes\mathtt{d})$
$=\quad\quad(\mathrm{I}_4)$
$(\lambda\mathtt{s}.\,\lambda\mathtt{d}.\,\mathtt{pure}\;(\lambda\mathtt{g}.\,\mathtt{g}\;(\lambda\mathtt{x}.\,\lambda\langle\rangle.\,\mathtt{f}_\mathtt{A}\;\mathtt{x}))$
$\quad\quad\otimes\;(\mathtt{pure}\;(\lambda\mathtt{c}.\,\lambda\mathtt{m}.\,\lambda\mathtt{n}.\,\mathtt{f}^{-1}_\mathtt{B}\;(\mathtt{f}_\mathtt{B}\;(\mathtt{c}\;(\mathtt{f}^{-1}_\mathtt{A}\;(\mathtt{m}\;\mathtt{n}\;\langle\rangle)))))\otimes\mathtt{s})\otimes\mathtt{d})$

(continued from previous page)
$(\lambda s. \lambda d. \mathtt{pure}\ (\lambda g.\ g\ (\lambda x.\ \lambda\langle\rangle.\ f_A\ x))$
$\qquad \circledast\ (\mathtt{pure}\ (\lambda c.\ \lambda m.\ \lambda n.\ f_B^{-1}\ (f_B\ (c\ (f_A^{-1}\ (m\ n\ \langle\rangle)))))) \circledast s) \circledast d)$
$=\qquad (I_3, I_2, \beta^{\rightarrow})$
$(\lambda s. \lambda d. \mathtt{pure}\ (\lambda y.\ \lambda n.\ f_B^{-1}\ (f_B\ (y\ (f_A^{-1}\ (f_A\ n))))) \circledast s \circledast d)$
$=\qquad (f_B^{-1}\ (f_B\ v) = v, f_A^{-1}\ (f_A\ v) = v)$
$(\lambda s. \lambda d. \mathtt{pure}\ (\lambda y.\ \lambda n.\ y\ n) \circledast s \circledast d)$
$=\qquad (\eta^{\rightarrow})$
$(\lambda s. \lambda d. \mathtt{pure}\ \mathtt{id} \circledast s \circledast d)$
$=\qquad (I_1)$
$(\lambda s. \lambda d.\ s \circledast d)$
$=\qquad (\eta^{\rightarrow})$
$(\circledast)$

A.4.5 Translating $\mathcal{S}$ to $\mathcal{J}$ and back

For each term M of $\mathcal{S}$ we must show $[\![\langle\!\langle M\rangle\!\rangle]\!] = g_A(M)$. For each command P of $\mathcal{S}$ we must show $[\![\langle\!\langle P\rangle\!\rangle_\Delta]\!] = g_{\Delta\rightsquigarrow A}(\lambda^\bullet\Delta.\ P)$. The proof is by mutual induction on the derivations of M and P.

1. Case $\lambda^\bullet x.\ Q$

   $[\![\langle\!\langle\lambda^\bullet x.\ Q\rangle\!\rangle]\!]$
   $=\qquad (\mathrm{def}\ \langle\!\langle-\rangle\!\rangle)$
   $[\![\langle\!\langle Q\rangle\!\rangle_x]\!]$
   $=\qquad (\mathrm{induction\ hypothesis})$
   $g_{A\rightsquigarrow B}\ (\lambda^\bullet x.\ Q)$

2. Case $L \bullet M$

   $g_{D\rightsquigarrow B}^{-1}\ [\![\langle\!\langle L \bullet M\rangle\!\rangle_\Delta]\!]$
   $=\qquad (\mathrm{def}\ \langle\!\langle-\rangle\!\rangle_\Delta)$
   $g_{D\rightsquigarrow B}^{-1}\ [\![\mathtt{pure}\ (\lambda l.\ \lambda\Delta.\ l\ \langle\!\langle M\rangle\!\rangle) \circledast \langle\!\langle L\rangle\!\rangle]\!]$
   $=\qquad (\mathrm{def}\ [\![-]\!])$
   $g_{D\rightsquigarrow B}^{-1}\ (\lambda^\bullet\langle\rangle.\ \mathbf{let}\ k \Leftarrow (\lambda^\bullet\langle\rangle.\ [\lambda l.\ \lambda\Delta.\ l\ [\![\langle\!\langle M\rangle\!\rangle]\!]]) \bullet \langle\rangle\ \mathbf{in}\ \mathbf{let}\ l \Leftarrow [\![\langle\!\langle L\rangle\!\rangle]\!] \bullet \langle\rangle\ \mathbf{in}\ [k\ l])$
   $=\qquad (\beta^{\rightsquigarrow})$
   $g_{D\rightsquigarrow B}^{-1}\ (\lambda^\bullet\langle\rangle.\ \mathbf{let}\ k \Leftarrow [\lambda l.\ \lambda\Delta.\ l\ [\![\langle\!\langle M\rangle\!\rangle]\!]]\ \mathbf{in}\ \mathbf{let}\ l \Leftarrow [\![\langle\!\langle L\rangle\!\rangle]\!] \bullet \langle\rangle\ \mathbf{in}\ [k\ l])$
   $=\qquad (\mathrm{left},\ \beta^{\rightarrow})$
   $g_{D\rightsquigarrow B}^{-1}\ (\lambda^\bullet\langle\rangle.\ \mathbf{let}\ l \Leftarrow [\![\langle\!\langle L\rangle\!\rangle]\!] \bullet \langle\rangle\ \mathbf{in}\ [\lambda\Delta.\ l\ [\![\langle\!\langle M\rangle\!\rangle]\!]])$

   (continued on next page)

(continued from previous page)

$g_{D \rightsquigarrow B}^{-1} \ (\lambda^\bullet \langle\rangle. \ \textbf{let } \texttt{l} \Leftarrow [\![\langle L \rangle]\!] \bullet \langle\rangle \ \textbf{in } [\lambda\Delta. \ \texttt{l} \ [\![\langle M \rangle]\!]])$

$=\qquad$ (induction hypothesis)

$g_{D \rightsquigarrow B}^{-1} \ (\lambda^\bullet \langle\rangle. \ \textbf{let } \texttt{l} \Leftarrow (g_{A \rightsquigarrow B} \ L) \bullet \langle\rangle \ \textbf{in } [\lambda\Delta. \ \texttt{l} \ g_A \ M])$

$=\qquad$ (def $g_{D \rightsquigarrow B}^{-1}$, def $g_{A \rightsquigarrow B}^{-1}$, $\beta^\rightarrow$)

$(\lambda^\bullet \Delta. \ \textbf{let } \texttt{h} \Leftarrow (\lambda^\bullet \langle\rangle. \ \textbf{let } \texttt{l} \Leftarrow (\lambda^\bullet \langle\rangle. \ \textbf{let } \texttt{f} \Leftarrow \textbf{run } L \ \textbf{in}$
$\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad [\lambda x. \, g_B \ (\texttt{f} \ (g_A^{-1} \ x))]) \bullet \langle\rangle \ \textbf{in}$
$\qquad\qquad\qquad\qquad [\lambda\Delta. \ \texttt{l} \ g_A \ M]) \bullet \langle\rangle \ \textbf{in}$
$\qquad\qquad [g_B^{-1} \ (\texttt{h} \ (g_D \ \Delta))]])$

$=\qquad$ ($\beta^{\rightsquigarrow}$)

$(\lambda^\bullet \Delta. \ \textbf{let } \texttt{h} \Leftarrow (\textbf{let } \texttt{l} \Leftarrow (\textbf{let } \texttt{f} \Leftarrow \textbf{run } L \ \textbf{in } [\lambda x. \, g_B \ (\texttt{f} \ (g_A^{-1} \ x))]) \ \textbf{in}$
$\qquad\qquad\qquad\qquad [\lambda\Delta. \ \texttt{l} \ g_A \ M]) \ \textbf{in}$
$\qquad\qquad [g_B^{-1} \ (\texttt{h} \ (g_D \ \Delta))]])$

$=\qquad$ (assoc)

$(\lambda^\bullet \Delta. \ \textbf{let } \texttt{f} \Leftarrow \textbf{run } L \ \textbf{in}$
$\qquad \textbf{let } \texttt{l} \Leftarrow [\lambda x. \, g_B \ (\texttt{f} \ (g_A^{-1} \ x))] \ \textbf{in}$
$\qquad \textbf{let } \texttt{h} \Leftarrow [\lambda\Delta. \ \texttt{l} \ g_A \ M] \ \textbf{in}$
$\qquad [g_B^{-1} \ (\texttt{h} \ (g_D \ \Delta))]])$

$=\qquad$ (left, $\beta^\rightarrow$)

$(\lambda^\bullet \Delta. \ \textbf{let } \texttt{f} \Leftarrow \textbf{run } L \ \textbf{in } [g_B^{-1} \ (g_B \ (\texttt{f} \ (g_A^{-1} \ (g_A \ M))))]])$

$=\qquad$ ($g_A^{-1} \ (g_A \ x) = x, g_B^{-1} \ (g_B \ x) = x$)

$(\lambda^\bullet \Delta. \ \textbf{let } \texttt{f} \Leftarrow \textbf{run } L \ \textbf{in } [\texttt{f} \ M])$

$=\qquad$ (run_1)

$(\lambda^\bullet \Delta. \ L \bullet M)$

3. Case [M]

$\qquad [\![\langle [M] \rangle_\Delta]\!]$

$=\qquad$ (def $\langle\!\!\!\; - \!\!\!\;\rangle_\Delta$)

$\qquad [\![\texttt{pure} \ (\lambda\Delta. \ \langle M \rangle)]\!]$

$=\qquad$ (def $[\![-]\!]$)

$\qquad \lambda^\bullet \langle\rangle. \ [\lambda\Delta. \ [\![\langle M \rangle]\!]]$

$=\qquad$ (induction hypothesis)

$\qquad \lambda^\bullet \langle\rangle. \ [\lambda\Delta. \ g_A \ M]$

$=\qquad$ (left)

$\qquad \lambda^\bullet \langle\rangle. \ \textbf{let } \texttt{f} \Leftarrow [\lambda\Delta. \ M] \ \textbf{in } [\lambda\Delta. \, g_A \ (\texttt{f} \ (g_D^{-1} \ \Delta))]$

$=\qquad$ (run_2)

$\qquad \lambda^\bullet \langle\rangle. \ \textbf{let } \texttt{f} \Leftarrow \textbf{run } (\lambda^\bullet \Delta. \ [M]) \ \textbf{in } [\lambda\Delta. \, g_A \ (\texttt{f} \ (g_D^{-1} \ \Delta))]$

$=\qquad$ (def $g_{D \rightsquigarrow A}$)

$\qquad g_{D \rightsquigarrow A} \ (\lambda^\bullet \Delta. \ [M])$

4. Case **let** $x \Leftarrow P$ **in** Q

$\qquad g^{-1}_{D \rightsquigarrow B} \, [\![\langle\!\langle \textbf{let } x \Leftarrow P \textbf{ in } Q \rangle\!\rangle_\Delta]\!]$

$= \qquad (\text{def } \langle\!\langle - \rangle\!\rangle_\Delta)$

$\qquad g^{-1}_{D \rightsquigarrow B} \, [\![\text{pure } (\lambda p.\, \lambda q.\, \lambda\Delta.\, q \, \langle \Delta, p \, q \rangle) \circledast \langle\!\langle P \rangle\!\rangle_\Delta \circledast \langle\!\langle Q \rangle\!\rangle_{\Delta,x}]\!]$

$= \qquad (\text{def } [\![-]\!])$

$\qquad g^{-1}_{D \rightsquigarrow B} \, (\lambda^\bullet \langle\rangle.\, \textbf{let } j \Leftarrow (\lambda^\bullet \langle\rangle.\, \textbf{let } k \Leftarrow (\lambda^\bullet \langle\rangle.\, [\lambda p.\, \lambda q.\, \lambda\Delta.\, q \, \langle \Delta, p \, \Delta \rangle]) \bullet \langle\rangle \textbf{ in}$
$\qquad\qquad\qquad\qquad\qquad\qquad\qquad \textbf{let } x \Leftarrow [\![\langle\!\langle P \rangle\!\rangle_\Delta]\!] \bullet \langle\rangle \textbf{ in } [k \, x]) \bullet \langle\rangle \textbf{ in}$
$\qquad\qquad\qquad\quad \textbf{let } y \Leftarrow [\![\langle\!\langle Q \rangle\!\rangle_{\Delta,x}]\!] \bullet \langle\rangle \textbf{ in}$
$\qquad\qquad\qquad\qquad [j \, y])$

$= \qquad (\beta^{\rightsquigarrow})$

$\qquad g^{-1}_{D \rightsquigarrow B} \, (\lambda^\bullet \langle\rangle.\, \textbf{let } j \Leftarrow (\textbf{let } k \Leftarrow [\lambda p.\, \lambda q.\, \lambda\Delta.\, q \, \langle \Delta, p \, \Delta \rangle] \textbf{ in}$
$\qquad\qquad\qquad\qquad\qquad\qquad \textbf{let } x \Leftarrow [\![\langle\!\langle P \rangle\!\rangle_\Delta]\!] \bullet \langle\rangle \textbf{ in}$
$\qquad\qquad\qquad\qquad\qquad\qquad\quad [k \, x]) \textbf{ in}$
$\qquad\qquad\qquad\quad \textbf{let } y \Leftarrow [\![\langle\!\langle Q \rangle\!\rangle_{\Delta,x}]\!] \bullet \langle\rangle \textbf{ in } [j \, y])$

$= \qquad (\text{assoc})$

$\qquad g^{-1}_{D \rightsquigarrow B} \, (\lambda^\bullet \langle\rangle.\, \textbf{let } k \Leftarrow [\lambda p.\, \lambda q.\, \lambda\Delta.\, q \, \langle \Delta, p \, \Delta \rangle] \textbf{ in}$
$\qquad\qquad\qquad\quad \textbf{let } x \Leftarrow [\![\langle\!\langle P \rangle\!\rangle_\Delta]\!] \bullet \langle\rangle \textbf{ in}$
$\qquad\qquad\qquad\quad \textbf{let } j \Leftarrow [k \, x] \textbf{ in}$
$\qquad\qquad\qquad\quad \textbf{let } y \Leftarrow [\![\langle\!\langle Q \rangle\!\rangle_{\Delta,x}]\!] \bullet \langle\rangle \textbf{ in}$
$\qquad\qquad\qquad\qquad [j \, y])$

$= \qquad (\text{left}, \beta^{\rightarrow})$

$\qquad g^{-1}_{D \rightsquigarrow B} \, (\lambda^\bullet \langle\rangle.\, \textbf{let } x \Leftarrow [\![\langle\!\langle P \rangle\!\rangle_\Delta]\!] \bullet \langle\rangle \textbf{ in}$
$\qquad\qquad\qquad\quad \textbf{let } y \Leftarrow [\![\langle\!\langle Q \rangle\!\rangle_{\Delta,x}]\!] \bullet \langle\rangle \textbf{ in}$
$\qquad\qquad\qquad\qquad [\lambda\Delta.\, y \, \langle \Delta, x \, \Delta \rangle])$

$= \qquad (\text{induction hypothesis})$

$\qquad g^{-1}_{D \rightsquigarrow B} \, (\lambda^\bullet \langle\rangle.\, \textbf{let } x \Leftarrow (g_{D \rightsquigarrow A} \, (\lambda^\bullet \Delta.\, P)) \bullet \langle\rangle \textbf{ in}$
$\qquad\qquad\qquad\quad \textbf{let } y \Leftarrow (g_{(D \times A) \rightsquigarrow B} \, (\lambda^\bullet \langle \Delta, x \rangle.\, Q)) \bullet \langle\rangle \textbf{ in}$
$\qquad\qquad\qquad\qquad [\lambda\Delta.\, y \, \langle \Delta, x \, \Delta \rangle])$

$= \qquad (\text{def } g^{-1}_{D \rightsquigarrow B}, \text{def } g_{D \rightsquigarrow A}, \text{def } g_{(D \times A) \rightsquigarrow B}, \beta^{\rightarrow})$

$\qquad (\lambda^\bullet \Delta.\, \textbf{let } h \Leftarrow (\lambda^\bullet \langle\rangle.\, \textbf{let } x \Leftarrow (\lambda^\bullet \langle\rangle.\, \textbf{let } f \Leftarrow \textbf{run } (\lambda^\bullet \Delta.\, P) \textbf{ in}$
$\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad [\lambda x.\, g_A \, (f \, (g^{-1}_D \, x))]) \bullet \langle\rangle \textbf{ in}$
$\qquad\qquad\qquad\qquad\quad \textbf{let } y \Leftarrow (\lambda^\bullet \langle\rangle.\, \textbf{let } f \Leftarrow \textbf{run } (\lambda^\bullet \langle \Delta, x \rangle.\, Q) \textbf{ in}$
$\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad [\lambda x.\, g_B \, (f \, (g^{-1}_{D \times A} \, x))]) \bullet \langle\rangle \textbf{ in}$
$\qquad\qquad\qquad\qquad\qquad [\lambda\Delta.\, y \, \langle \Delta, x \, \Delta \rangle]) \bullet \langle\rangle \textbf{ in}$
$\qquad\qquad\quad [g^{-1}_B \, (h \, (g_D \, \Delta))]])$

$= \qquad (\beta^{\rightsquigarrow})$

$\qquad (\lambda^\bullet \Delta.\, \textbf{let } h \Leftarrow (\textbf{let } x \Leftarrow (\textbf{let } f \Leftarrow \textbf{run } (\lambda^\bullet \Delta.\, P) \textbf{ in } [\lambda x.\, g_A \, (f \, (g^{-1}_D \, x))]) \textbf{ in}$
$\qquad\qquad\qquad\quad \textbf{let } y \Leftarrow (\textbf{let } f \Leftarrow \textbf{run } (\lambda^\bullet \langle \Delta, x \rangle.\, Q) \textbf{ in } [\lambda x.\, g_B \, (f \, (g^{-1}_{D \times A} \, x))]) \textbf{ in}$
$\qquad\qquad\qquad\qquad [\lambda\Delta.\, y \, \langle \Delta, x \, \Delta \rangle]) \textbf{ in}$
$\qquad\qquad\quad [g^{-1}_B \, (h \, (g_D \, \Delta))]])$

(continued on next page)

(continued from previous page)

$(\lambda^\bullet\Delta.\, \mathbf{let}\ h \Leftarrow (\mathbf{let}\ x \Leftarrow (\mathbf{let}\ f \Leftarrow \mathbf{run}\ (\lambda^\bullet\Delta.\, P)\ \mathbf{in}\ [\lambda x.\, g_A\ (f\ (g_D^{-1}\ x))])\ \mathbf{in}$

$\qquad\qquad\quad \mathbf{let}\ y \Leftarrow (\mathbf{let}\ f \Leftarrow \mathbf{run}\ (\lambda^\bullet\langle\Delta, x\rangle.\, Q)\ \mathbf{in}\ [\lambda x.\, g_B\ (f\ (g_{D\times A}^{-1}\ x))])\ \mathbf{in}$

$\qquad\qquad\qquad [\lambda\Delta.\, y\ \langle\Delta, x\ \Delta\rangle])\ \mathbf{in}$

$\qquad\qquad [g_B^{-1}\ (h\ (g_D\ \Delta))]])$

$=$ (assoc)

$(\lambda^\bullet\Delta.\, \mathbf{let}\ f \Leftarrow \mathbf{run}\ (\lambda^\bullet\Delta.\, P)\ \mathbf{in}$

$\qquad \mathbf{let}\ x \Leftarrow [\lambda x.\, g_A\ (f\ (g_D^{-1}\ x))]\ \mathbf{in}$

$\qquad \mathbf{let}\ j \Leftarrow \mathbf{run}\ (\lambda^\bullet\langle\Delta, x\rangle.\, Q)\ \mathbf{in}$

$\qquad \mathbf{let}\ y \Leftarrow [\lambda x.\, g_B\ (j\ (g_{D\times A}^{-1}\ x))]\ \mathbf{in}$

$\qquad \mathbf{let}\ h \Leftarrow [\lambda\Delta.\, y\ \langle\Delta, x\ \Delta\rangle]\ \mathbf{in}$

$\qquad\qquad [g_B^{-1}\ (h\ (g_D\ \Delta))]])$

$=$ (left, $\beta^\rightarrow$)

$(\lambda^\bullet\Delta.\, \mathbf{let}\ f \Leftarrow \mathbf{run}\ (\lambda^\bullet\Delta.\, P)\ \mathbf{in}$

$\qquad \mathbf{let}\ j \Leftarrow \mathbf{run}\ (\lambda^\bullet\langle\Delta, x\rangle.\, Q)\ \mathbf{in}$

$\qquad\qquad [g_B^{-1}\ (g_B\ (j\ (g_{D\times A}^{-1}\ \langle g_D\ \Delta, g_A\ (f\ (g_D^{-1}\ (g_D\ \Delta)))\rangle)))]])$

$=$ (def $g_{D\times A}^{-1}$)

$(\lambda^\bullet\Delta.\, \mathbf{let}\ f \Leftarrow \mathbf{run}\ (\lambda^\bullet\Delta.\, P)\ \mathbf{in}$

$\qquad \mathbf{let}\ j \Leftarrow \mathbf{run}\ (\lambda^\bullet\langle\Delta, x\rangle.\, Q)\ \mathbf{in}$

$\qquad\qquad [g_B^{-1}\ (g_B\ (j\ \langle g_D^{-1}\ (g_D\ \Delta), g_A^{-1}\ (g_A\ (f\ (g_D^{-1}\ (g_D\ \Delta))))\rangle))]])$

$=$ $(g_D^{-1}\ (g_D\ x) = x, g_B^{-1}\ (g_B\ x) = x, g_A^{-1}\ (g_A\ x) = x)$

$(\lambda^\bullet\Delta.\, \mathbf{let}\ f \Leftarrow \mathbf{run}\ (\lambda^\bullet\Delta.\, P)\ \mathbf{in}\ \mathbf{let}\ j \Leftarrow \mathbf{run}\ (\lambda^\bullet\langle\Delta, x\rangle.\, Q)\ \mathbf{in}\ [j\ \langle\Delta, f\ \Delta\rangle])$

$=$ (run_1)

$(\lambda^\bullet\Delta.\, \mathbf{let}\ f \Leftarrow \mathbf{run}\ (\lambda^\bullet\Delta.\, P)\ \mathbf{in}\ (\lambda^\bullet\langle\Delta, x\rangle.\, Q) \bullet \langle\Delta, f\ \Delta\rangle)$

$=$ $(\beta^{\rightsquigarrow})$

$(\lambda^\bullet\Delta.\, \mathbf{let}\ f \Leftarrow \mathbf{run}\ (\lambda^\bullet\Delta.\, P)\ \mathbf{in}\ Q[x := f\ \Delta])$

$=$ (left)

$(\lambda^\bullet\Delta.\, \mathbf{let}\ f \Leftarrow \mathbf{run}\ (\lambda^\bullet\Delta.\, P)\ \mathbf{in}\ \mathbf{let}\ x \Leftarrow [f\ \Delta]\ \mathbf{in}\ Q)$

$=$ (assoc)

$(\lambda^\bullet\Delta.\, \mathbf{let}\ x \Leftarrow \mathbf{let}\ f \Leftarrow \mathbf{run}\ (\lambda^\bullet\Delta.\, P)\ \mathbf{in}\ [f\ \Delta]\ \mathbf{in}\ Q)$

$=$ (run_1)

$(\lambda^\bullet\Delta.\, \mathbf{let}\ x \Leftarrow (\lambda^\bullet\Delta.\, P) \bullet \Delta\ \mathbf{in}\ Q)$

$=$ $(\beta^{\rightsquigarrow})$

$(\lambda^\bullet\Delta.\, \mathbf{let}\ x \Leftarrow P\ \mathbf{in}\ Q)$

5. Case **run** L

$$g^{-1}_{D \rightsquigarrow (A \rightarrow B)} [\![\langle\!\langle \textbf{run } L \rangle\!\rangle_\Delta]\!]$$

$=$ (def $\langle\!\langle - \rangle\!\rangle_\Delta$)

$$g^{-1}_{D \rightsquigarrow (A \rightarrow B)} [\![\texttt{pure const} \circledast \langle\!\langle L \rangle\!\rangle]\!]$$

$=$ (def $[\![-]\!]$)

$$g^{-1}_{D \rightsquigarrow (A \rightarrow B)} (\lambda^\bullet \langle \rangle. \textbf{ let } k \Leftarrow (\lambda^\bullet \langle \rangle. [\text{const}]) \bullet \langle \rangle \textbf{ in let } x \Leftarrow [\![\langle\!\langle L \rangle\!\rangle]\!] \bullet \langle \rangle \textbf{ in } [k\, x])$$

$=$ ($\beta^{\rightsquigarrow}$)

$$g^{-1}_{D \rightsquigarrow (A \rightarrow B)} (\lambda^\bullet \langle \rangle. \textbf{ let } k \Leftarrow [\text{const}] \textbf{ in let } x \Leftarrow [\![\langle\!\langle L \rangle\!\rangle]\!] \bullet \langle \rangle \textbf{ in } [k\, x])$$

$=$ (left)

$$g^{-1}_{D \rightsquigarrow (A \rightarrow B)} (\lambda^\bullet \langle \rangle. \textbf{ let } x \Leftarrow [\![\langle\!\langle L \rangle\!\rangle]\!] \bullet \langle \rangle \textbf{ in } [\text{const } x])$$

$=$ (def $g^{-1}_{D \rightsquigarrow (A \rightarrow B)}$)

$$\lambda^\bullet \Delta. \textbf{ let } h \Leftarrow (\lambda^\bullet \langle \rangle. \textbf{ let } x \Leftarrow [\![\langle\!\langle L \rangle\!\rangle]\!] \bullet \langle \rangle \textbf{ in } [\text{const } x]) \bullet \langle \rangle \textbf{ in } [g^{-1}_{A \rightarrow B} (h\, (g_D\, x))]$$

$=$ ($\beta^{\rightsquigarrow}$)

$$\lambda^\bullet \Delta. \textbf{ let } h \Leftarrow \textbf{ let } x \Leftarrow [\![\langle\!\langle L \rangle\!\rangle]\!] \bullet \langle \rangle \textbf{ in } [\text{const } x] \textbf{ in } [g^{-1}_{A \rightarrow B} (h\, (g_D\, x))]$$

$=$ (assoc)

$$\lambda^\bullet \Delta. \textbf{ let } x \Leftarrow [\![\langle\!\langle L \rangle\!\rangle]\!] \bullet \langle \rangle \textbf{ in let } h \Leftarrow [\text{const } x] \textbf{ in } [g^{-1}_{A \rightarrow B} (h\, (g_D\, x))]$$

$=$ (left)

$$\lambda^\bullet \Delta. \textbf{ let } x \Leftarrow [\![\langle\!\langle L \rangle\!\rangle]\!] \bullet \langle \rangle \textbf{ in } [g^{-1}_{A \rightarrow B}\, x]$$

$=$ (induction hypothesis)

$$\lambda^\bullet \Delta. \textbf{ let } x \Leftarrow (g_{A \rightsquigarrow B}\, L) \bullet \langle \rangle \textbf{ in } [g^{-1}_{A \rightarrow B}\, x]$$

$=$ (def $g_{A \rightsquigarrow B}$)

$$\lambda^\bullet \Delta. \textbf{ let } x \Leftarrow (\lambda^\bullet \langle \rangle. \textbf{ let } f \Leftarrow \textbf{run } L \textbf{ in } [\lambda x. g_B\, (f\, (g_A^{-1}\, x))]) \bullet \langle \rangle \textbf{ in } [g^{-1}_{A \rightarrow B}\, x]$$

$=$ ($\beta^{\rightsquigarrow}$)

$$\lambda^\bullet \Delta. \textbf{ let } x \Leftarrow \textbf{ let } f \Leftarrow \textbf{run } L \textbf{ in } [\lambda x. g_B\, (f\, (g_A^{-1}\, x))] \textbf{ in } [g^{-1}_{A \rightarrow B}\, x]$$

$=$ (assoc)

$$\lambda^\bullet \Delta. \textbf{ let } f \Leftarrow \textbf{run } L \textbf{ in let } x \Leftarrow [\lambda x. g_B\, (f\, (g_A^{-1}\, x))] \textbf{ in } [g^{-1}_{A \rightarrow B}\, x]$$

$=$ (left)

$$\lambda^\bullet \Delta. \textbf{ let } f \Leftarrow \textbf{run } L \textbf{ in } [g^{-1}_{A \rightarrow B}\, (\lambda x. g_B\, (f\, (g_A^{-1}\, x)))]$$

$=$ (def $g^{-1}_{A \rightarrow B}$)

$$\lambda^\bullet \Delta. \textbf{ let } f \Leftarrow \textbf{run } L \textbf{ in } [\lambda s. g_B^{-1}\, (g_B\, (f\, (g_A^{-1}\, (g_A\, s))))]$$

$=$ ($g_A^{-1}\, (g_A\, v) = v, g_B^{-1}\, (g_B\, v) = v$)

$$\lambda^\bullet \Delta. \textbf{ let } f \Leftarrow \textbf{run } L \textbf{ in } [\lambda s. f\, s]$$

$=$ ($\eta^{\rightarrow}$)

$$\lambda^\bullet \Delta. \textbf{ let } f \Leftarrow \textbf{run } L \textbf{ in } [f]$$

$=$ (right)

$$\lambda^\bullet \Delta. \textbf{run } L$$

A.5   Equational embedding of $\mathbb{S}$ into $\mathcal{A}$

This section gives a proof of Proposition 19 (page 75).

A.5.1   Proofs of Lemmas 16 and 18

We begin with proofs of Lemma 16 and 18, which give the translations of substitution and weakening.

Proof of Lemma 16 (Translating substitution from $\mathbb{S}$ to $\mathcal{A}$)   .

The translations of substitution on terms and commands from $\mathbb{S}$ to $\mathcal{A}$ are as follows.

$$[\![M[x := N]]\!] = [\![M]\!][x := [\![N]\!]]$$

$$[\![Q[x := N]]\!] = \mathtt{let}\ q \Leftarrow [\![Q]\!]_{\Delta,x}\ \mathtt{in}\ [\lambda\Delta.q\,(\Delta, [\![N]\!])]$$

Proof. By mutual induction on the derivations of P and M. There is one case for each term form and each command form. We give only the cases for command forms here.

1. Case $L \bullet M$

$$[\![(L \bullet M)[x := N]]\!]_{\Delta}$$
= (def substitution)
$$[\![L \bullet (M[x := N])]\!]_{\Delta}$$
= (def $[\![-]\!]_{\Delta}$)
$$\mathtt{let}\ l \Leftarrow [\![L]\!] \bullet \langle\rangle\ \mathtt{in}\ [\lambda\Delta.\,l\,[\![M[x := N]]\!]]$$
= (induction hypothesis)
$$\mathtt{let}\ l \Leftarrow [\![L]\!] \bullet \langle\rangle\ \mathtt{in}\ [\lambda\Delta.\,l\,([\![M]\!][x := [\![N]\!]])]$$
= (left)
$$\mathtt{let}\ l \Leftarrow [\![L]\!] \bullet \langle\rangle\ \mathtt{in}\ \mathtt{let}\ q \Leftarrow [\lambda\langle\Delta,x\rangle.\,l\,[\![M]\!]]\ \mathtt{in}\ [\lambda\Delta.\,q\,\langle\Delta, [\![N]\!]\rangle]$$
= (assoc)
$$\mathtt{let}\ q \Leftarrow (\mathtt{let}\ l \Leftarrow [\![L]\!] \bullet \langle\rangle\ \mathtt{in}\ [\lambda\langle\Delta,x\rangle.\,l\,[\![M]\!]])\ \mathtt{in}\ [\lambda\Delta.\,q\,\langle\Delta, [\![N]\!]\rangle]$$
= (def $[\![-]\!]_{\Delta,x}$)
$$\mathtt{let}\ q \Leftarrow [\![L \bullet M]\!]_{\Delta,x}\ \mathtt{in}\ [\lambda\Delta.\,q\,\langle\Delta, [\![N]\!]\rangle]$$

2. Case $[M]$

$$\llbracket[M][x := N]\rrbracket_\Delta$$
$=$ (def substitution)
$$\llbracket[M[x := N]]\rrbracket_\Delta$$
$=$ (def $\llbracket-\rrbracket_\Delta$)
$$[\lambda\Delta.\,\llbracket M[x := N]\rrbracket]$$
$=$ (induction hypothesis)
$$[\lambda\Delta.\,M[x := \llbracket N\rrbracket]]$$
$=$ (left)
let $q \Leftarrow [\lambda\langle\Delta, x\rangle.\,\llbracket M\rrbracket]$ **in** $[\lambda\Delta.\,q\,\langle\Delta, \llbracket N\rrbracket\rangle]$
$=$ (def $\llbracket-\rrbracket_{\Delta, x}$)
let $q \Leftarrow \llbracket[M]\rrbracket_{\Delta, x}$ **in** $[\lambda\Delta.\,q\,\langle\Delta, \llbracket N\rrbracket\rangle]$

3. Case **let** $y \Leftarrow P$ **in** Q

$$\llbracket(\textbf{let}\ y \Leftarrow P\ \textbf{in}\ Q)[x := N]\rrbracket_\Delta$$
$=$ (def substitution)
$$\llbracket\textbf{let}\ y \Leftarrow P[x := N]\ \textbf{in}\ Q[x := N]\rrbracket_\Delta$$
$=$ (def $\llbracket-\rrbracket_\Delta$)
let $p \Leftarrow \llbracket P[x := N]\rrbracket_\Delta$ **in let** $q \Leftarrow \llbracket Q[x := N]\rrbracket_{\Delta, y}$ **in** $[\lambda\Delta.\,q\,\langle\Delta, p\,\Delta\rangle]$
$=$ (induction hypothesis)
let $p \Leftarrow (\textbf{let}\ r \Leftarrow \llbracket P\rrbracket_{\Delta, x}\ \textbf{in}\ [\lambda\Delta.\,r\,\langle\Delta, \llbracket N\rrbracket\rangle])$ **in**
let $q \Leftarrow (\textbf{let}\ s \Leftarrow \llbracket Q\rrbracket_{\Delta, y, x}\ \textbf{in}\ [\lambda\langle\Delta, y\rangle.\,s\,\langle\langle\Delta, y\rangle, \llbracket N\rrbracket\rangle])$ **in**
    $[\lambda\Delta.\,q\,\langle\Delta, p\,\Delta\rangle]$
$=$ (assoc)
let $r \Leftarrow \llbracket P\rrbracket_{\Delta, x}$ **in**
let $p \Leftarrow [\lambda\Delta.\,r\,\langle\Delta, \llbracket N\rrbracket\rangle]$ **in**
let $s \Leftarrow \llbracket Q\rrbracket_{\Delta, y, x}$ **in**
let $q \Leftarrow [\lambda\langle\Delta, y\rangle.\,s\,\langle\langle\Delta, y\rangle, \llbracket N\rrbracket\rangle]$ **in**
  $[\lambda\Delta.\,q\,\langle\Delta, p\,\Delta\rangle]$
$=$ (left, $\beta^\rightarrow$)
let $r \Leftarrow \llbracket P\rrbracket_{\Delta, x}$ **in let** $s \Leftarrow \llbracket Q\rrbracket_{\Delta, y, x}$ **in** $[\lambda\Delta.\,s\,\langle\langle\Delta, r\,\langle\Delta, \llbracket N\rrbracket\rangle\rangle, \llbracket N\rrbracket\rangle]$
$=$ (Lemma 18)
let $r \Leftarrow \llbracket P\rrbracket_{\Delta, x}$ **in**
let $s \Leftarrow (\textbf{let}\ q \Leftarrow \llbracket Q\rrbracket_{\Delta, x, y}\ \textbf{in}\ [\lambda\langle\langle\Delta, y\rangle, x\rangle.\,q\,\langle\langle\Delta, x\rangle, x\rangle])$ **in**
  $[\lambda\Delta.\,s\,\langle\langle\Delta, r\,\langle\Delta, \llbracket N\rrbracket\rangle\rangle, \llbracket N\rrbracket\rangle]$
$=$ (assoc)
let $r \Leftarrow \llbracket P\rrbracket_{\Delta, x}$ **in**
let $q \Leftarrow \llbracket Q\rrbracket_{\Delta, x, y}$ **in**
let $s \Leftarrow [\lambda\langle\langle\Delta, y\rangle, x\rangle.\,q\,\langle\langle\Delta, x\rangle, y\rangle]$ **in**
  $[\lambda\Delta.\,s\,\langle\langle\Delta, r\,\langle\Delta, \llbracket N\rrbracket\rangle\rangle, \llbracket N\rrbracket\rangle]$

(continued on next page)

(continued from previous page)
let $r \Leftarrow [\![P]\!]_{\Delta,x}$ **in**
let $q \Leftarrow [\![Q]\!]_{\Delta,x,y}$ **in**
let $s \Leftarrow [\lambda \langle \langle \Delta, y \rangle, x \rangle. q \langle \langle \Delta, x \rangle, y \rangle]$ **in**
  $[\lambda\Delta. s \langle \langle \Delta, r \langle \Delta, [\![N]\!] \rangle \rangle, [\![N]\!] \rangle]$
$=$ (left, $\beta^{\rightarrow}$)
let $p \Leftarrow [\![P]\!]_{\Delta,x}$ **in**
let $q \Leftarrow [\![Q]\!]_{\Delta,x,y}$ **in**
  $[\lambda\Delta. q \langle \langle \Delta, [\![N]\!] \rangle, p \langle \Delta, [\![N]\!] \rangle \rangle]$
$=$ (left, $\beta^{\rightarrow}$)
let $p \Leftarrow [\![P]\!]_{\Delta,x}$ **in**
let $q \Leftarrow [\![Q]\!]_{\Delta,x,y}$ **in**
let $s \Leftarrow [\lambda\langle \Delta, x \rangle. q \langle \langle \Delta, x \rangle, p \langle \Delta, x \rangle \rangle]$ **in**
  $[\lambda\Delta. s \langle \Delta, [\![N]\!] \rangle]$
$=$ (assoc)
let $s \Leftarrow ($**let** $p \Leftarrow [\![P]\!]_{\Delta,x}$ **in**
       let $q \Leftarrow [\![Q]\!]_{\Delta,x,y}$ **in**
         $[\![\lambda\langle \Delta, x \rangle. q \langle \langle \Delta, x \rangle, p \langle \Delta, x \rangle \rangle]\!])$ **in**
  $[\lambda\Delta. s \langle \Delta, [\![N]\!] \rangle]$
$=$ (def $[\![-]\!]_{\Delta,x}$)
let $s \Leftarrow [\![$**let** $y \Leftarrow P$ **in** $Q]\!]_{\Delta,x}$ **in** $[\lambda\Delta. s \langle \Delta, [\![N]\!] \rangle]$

4. Case **run** L

$[\![(\textbf{run } L)[x := N]]\!]_{\Delta}$
$=$ (def substitution)
$[\![\textbf{run } L]\!]_{\Delta}$
$=$ (def $[\![-]\!]_{\Delta}$)
let $h \Leftarrow [\![L]\!] \bullet \langle \rangle$ **in** $[\lambda\Delta. h]$
$=$ (left)
let $h \Leftarrow [\![L]\!] \bullet \langle \rangle$ **in let** $q \Leftarrow [\lambda\langle \Delta, x \rangle. h]$ **in** $[\lambda\Delta. q \langle \Delta, [\![N]\!] \rangle]$
$=$ (assoc)
let $q \Leftarrow ($**let** $h \Leftarrow [\![L]\!] \bullet \langle \rangle$ **in** $[\lambda\langle \Delta, x \rangle. h])$ **in** $[\lambda\Delta. q \langle \Delta, [\![N]\!] \rangle]$
$=$ (def $[\![-]\!]_{\Delta,x}$)
let $q \Leftarrow [\![\textbf{run } L]\!]_{\Delta,x}$ **in** $[\lambda\Delta. q \langle \Delta, [\![N]\!] \rangle]$

$\square$

Proof of Lemma 18 (Translating weakening from $\mathcal{S}$ to $\mathcal{A}$)   The translation of weakening from $\mathcal{S}$ to $\mathcal{A}$ for commands is as follows.

$$
\left[\!\!\left[\frac{\Gamma;\, \Delta \vdash Q\,!\,B}{\Gamma';\, \Delta' \vdash Q\,!\,B} \right]\!\!\right] \quad = \quad \frac{[\![\Gamma]\!];\, \cdot \vdash [\![Q]\!]_\Delta \,!\, [\![\Delta]\!] \to [\![B]\!]}{[\![\Gamma']\!];\, \cdot \vdash \textbf{let } q \Leftarrow [\![Q]\!]_\Delta \textbf{ in } [\lambda\Delta'.q\,\Delta] \,!\, [\![\Delta']\!] \to [\![B]\!]}
$$

Proof. By induction on the derivation of Q. There is one case for each command form.

1. Case $L \bullet M$.

$$
\begin{aligned}
& [\![L \bullet M]\!]_{\Delta'} \\
=\ & \qquad (\text{def } [\![-]\!]_{\Delta'}) \\
& \textbf{let } l \Leftarrow [\![L]\!] \bullet \langle\rangle \textbf{ in } [\lambda\Delta'.\,l\,[\![M]\!]] \\
=\ & \qquad (\text{left}, \beta^\to) \\
& \textbf{let } l \Leftarrow [\![L]\!] \bullet \langle\rangle \textbf{ in let } q \Leftarrow [\lambda\Delta.\,l\,[\![M]\!]] \textbf{ in } [\lambda\Delta'.\,q\,\Delta] \\
=\ & \qquad (\text{assoc}) \\
& \textbf{let } q \Leftarrow \textbf{let } l \Leftarrow [\![L]\!] \bullet \langle\rangle \textbf{ in } [\lambda\Delta.\,l\,[\![M]\!]] \textbf{ in } [\lambda\Delta'.\,q\,\Delta] \\
=\ & \qquad (\text{def } [\![-]\!]_\Delta) \\
& \textbf{let } q \Leftarrow [\![L \bullet M]\!]_\Delta \textbf{ in } [\lambda\Delta'.\,q\,\Delta]
\end{aligned}
$$

2. Case $[M]$.

$$
\begin{aligned}
& [\![[M]]\!]_{\Delta'} \\
=\ & \qquad (\text{def } [\![-]\!]_{\Delta'}) \\
& [\lambda\Delta'.\,[\![M]\!]] \\
=\ & \qquad (\text{left}, \beta^\to) \\
& \textbf{let } q \Leftarrow [\lambda\Delta.\,[\![M]\!]] \textbf{ in } [\lambda\Delta'.\,q\,\Delta] \\
=\ & \qquad (\text{def } [\![-]\!]_\Delta) \\
& \textbf{let } q \Leftarrow [\![[M]]\!]_\Delta \textbf{ in } [\lambda\Delta'.\,q\,\Delta]
\end{aligned}
$$

3. Case $\textbf{let } x \Leftarrow P \textbf{ in } Q$.

$$
\begin{aligned}
& [\![\textbf{let } x \Leftarrow P \textbf{ in } Q]\!]_{\Delta'} \\
=\ & \qquad (\text{def } [\![-]\!]_{\Delta'}) \\
& \textbf{let } p \Leftarrow [\![P]\!]_{\Delta'} \textbf{ in let } q \Leftarrow [\![Q]\!]_{\Delta',x} \textbf{ in } [\lambda\Delta'.\,q\,\langle\Delta',p\,\Delta'\rangle] \\
=\ & \qquad (\text{induction hypothesis}) \\
& \textbf{let } r \Leftarrow \textbf{let } p \Leftarrow [\![P]\!]_\Delta \textbf{ in } [\lambda\Delta'.\,p\,\Delta] \textbf{ in} \\
& \textbf{let } s \Leftarrow \textbf{let } q \Leftarrow [\![Q]\!]_{\Delta',x} \textbf{ in } [\lambda\langle\Delta',x\rangle.\,q\,\langle\Delta,x\rangle] \textbf{ in} \\
& \quad [\lambda\Delta'.\,s\,\langle\Delta',r\,\Delta'\rangle]
\end{aligned}
$$

(continued from previous page)
let $r \Leftarrow$ **let** $p \Leftarrow [\![P]\!]_\Delta$ **in** $[\lambda\Delta'.\, p\, \Delta]$ **in**
let $s \Leftarrow$ **let** $q \Leftarrow [\![Q]\!]_{\Delta',x}$ **in** $[\lambda\langle\Delta', x\rangle.\, q\,\langle\Delta, x\rangle]$ **in**
   $[\lambda\Delta'.\, s\,\langle\Delta', r\,\Delta'\rangle]$

$=$ (assoc)
let $p \Leftarrow [\![P]\!]_\Delta$ **in**
let $r \Leftarrow [\lambda\Delta'.\, p\,\Delta]$ **in**
let $q \Leftarrow [\![Q]\!]_{\Delta',x}$ **in**
let $s \Leftarrow [\lambda\langle\Delta', x\rangle.\, q\,\langle\Delta, x\rangle]$ **in**
   $[\lambda\Delta'.\, s\,\langle\Delta', r\,\Delta'\rangle]$

$=$ (left, $\beta^\rightarrow$)
let $p \Leftarrow [\![P]\!]_\Delta$ **in let** $q \Leftarrow [\![Q]\!]_{\Delta',x}$ **in** $[\lambda\Delta'.\, q\,\langle\Delta, p\,\Delta\rangle]$

$=$ (left, $\beta^\rightarrow$)
let $p \Leftarrow [\![P]\!]_\Delta$ **in let** $q \Leftarrow [\![Q]\!]_{\Delta,x}$ **in let** $q \Leftarrow [\lambda\Delta.\, q\,\langle\Delta, p\,\Delta\rangle]$ **in** $[\lambda\Delta'.\, q\,\Delta]$

$=$ (assoc)
let $q \Leftarrow ($**let** $p \Leftarrow [\![P]\!]_\Delta$ **in let** $q \Leftarrow [\![Q]\!]_{\Delta,x}$ **in** $[\lambda\Delta.\, q\,\langle\Delta, p\,\Delta\rangle])$ **in** $[\lambda\Delta'.\, q\,\Delta]$

$=$ (def $[\![-]\!]_\Delta$)
let $q \Leftarrow [\![$**let** $x \Leftarrow P$ **in** $Q]\!]_\Delta$ **in** $[\lambda\Delta'.\, q\,\Delta]$

4. Case **run** L.

   $[\![$**run** $L]\!]_{\Delta'}$
$=$ (def $[\![-]\!]_{\Delta'}$)
let $h \Leftarrow [\![L]\!] \bullet \langle\rangle$ **in** $[\lambda\Delta'.\, h]$
$=$ (left, $\beta^\rightarrow$)
let $h \Leftarrow [\![L]\!] \bullet \langle\rangle$ **in let** $q \Leftarrow [\lambda\Delta.\, h]$ **in** $[\lambda\Delta'.\, q\,\Delta]$
$=$ (assoc)
let $q \Leftarrow$ **let** $h \Leftarrow [\![L]\!] \bullet \langle\rangle$ **in** $[\lambda\Delta.\, h]$ **in** $[\lambda\Delta'.\, q\,\Delta]$
$=$ (def $[\![-]\!]_\Delta$)
let $q \Leftarrow [\![$**run** $L]\!]_\Delta$ **in** $[\lambda\Delta'.\, q\,\Delta]$

$\square$

A.5.2 The laws of $\mathcal{S}$ follow from the laws of $\mathcal{A}$

For each law $M = N$ or $P = Q$ of $\mathcal{S}$ we must show $[\![M]\!] = [\![N]\!]$ or $[\![P]\!]_\Delta = [\![Q]\!]_\Delta$.

1. $(\beta^{\rightsquigarrow})$

$$[\![(\lambda^\bullet x. Q) \bullet M]\!]_\Delta$$
$$= \quad (\text{def } [\![-]\!]_\Delta)$$
$$\mathbf{let}\ l \Leftarrow (\lambda^\bullet \langle\rangle. [\![Q]\!]_x) \bullet \langle\rangle\ \mathbf{in}\ [\lambda\Delta. l\ [\![M]\!]]$$
$$= \quad (\beta^{\rightsquigarrow})$$
$$\mathbf{let}\ l \Leftarrow [\![Q]\!]_x\ \mathbf{in}\ [\lambda\Delta. l\ [\![M]\!]]$$
$$= \quad (\text{Lemma } 16)$$
$$[\![Q[x := M]]\!]$$

2. $(\eta^{\rightsquigarrow})$

$$[\![\lambda^\bullet x. L \bullet x]\!]$$
$$= \quad (\text{def } [\![-]\!])$$
$$\lambda^\bullet \langle\rangle. \mathbf{let}\ l \Leftarrow [\![L]\!] \bullet \langle\rangle\ \mathbf{in}\ [\lambda x. l\ x]$$
$$= \quad (\eta^{\rightarrow})$$
$$\lambda^\bullet \langle\rangle. \mathbf{let}\ l \Leftarrow [\![L]\!] \bullet \langle\rangle\ \mathbf{in}\ [l]$$
$$= \quad (\text{right})$$
$$\lambda^\bullet \langle\rangle. [\![L]\!] \bullet \langle\rangle$$
$$= \quad (\eta^{\rightsquigarrow})$$
$$[\![L]\!]$$

3. (left)

$$[\![\mathbf{let}\ x \Leftarrow [M]\ \mathbf{in}\ Q]\!]_\Delta$$
$$= \quad (\text{def } [\![-]\!]_\Delta)$$
$$\mathbf{let}\ p \Leftarrow [\lambda\Delta. [\![M]\!]]\ \mathbf{in}\ \mathbf{let}\ q \Leftarrow [\![Q]\!]_{\Delta,x}\ \mathbf{in}\ [\lambda\Delta. q\ \langle\Delta, p\ \Delta\rangle]$$
$$= \quad (\text{left}, \beta^{\rightarrow})$$
$$\mathbf{let}\ q \Leftarrow [\![Q]\!]_{\Delta,x}\ \mathbf{in}\ [\lambda\Delta. q\ \langle\Delta, [\![M]\!]\rangle]$$
$$= \quad (\text{Lemma } 16)$$
$$[\![Q[x := M]]\!]_\Delta$$

4. (right)

$$[\![\mathbf{let}\ x \Leftarrow P\ \mathbf{in}\ [x]]\!]_\Delta$$
$$= \quad (\text{def } [\![-]\!]_\Delta)$$
$$\mathbf{let}\ p \Leftarrow [\![P]\!]_\Delta\ \mathbf{in}\ \mathbf{let}\ q \Leftarrow [\lambda\langle\Delta, x\rangle. x]\ \mathbf{in}\ [\lambda\Delta. q\ \langle\Delta, p\ \Delta\rangle]$$
$$= \quad (\text{left}, \beta^{\rightarrow})$$
$$\mathbf{let}\ p \Leftarrow [\![P]\!]_\Delta\ \mathbf{in}\ [\lambda\Delta. p\ \Delta]$$

(continued from previous page)
$\quad$ **let** $p \Leftarrow \llbracket P \rrbracket_\Delta$ **in** $[\lambda\Delta.\, p\, \Delta]$
$=\qquad (\eta^\rightarrow)$
$\quad$ **let** $p \Leftarrow \llbracket P \rrbracket_\Delta$ **in** $[p]$
$=\qquad$ (right)
$\quad \llbracket P \rrbracket_\Delta$

5. (assoc)

$\quad \llbracket \textbf{let}\ y \Leftarrow (\textbf{let}\ x \Leftarrow P\ \textbf{in}\ Q)\ \textbf{in}\ R \rrbracket_\Delta$
$=\qquad (\text{def } \llbracket - \rrbracket_\Delta)$
$\quad$ **let** $s \Leftarrow (\textbf{let}\ p \Leftarrow \llbracket P \rrbracket_\Delta\ \textbf{in}\ \textbf{let}\ q \Leftarrow \llbracket Q \rrbracket_{\Delta,x}\ \textbf{in}\ [\lambda\Delta.\, q\, \langle\Delta, p\, \Delta\rangle])$ **in**
$\quad$ **let** $r \Leftarrow \llbracket R \rrbracket_{\Delta,y}\ \textbf{in}\ [\lambda\Delta.\, r\, \langle\Delta, s\, \Delta\rangle]$
$=\qquad$ (assoc)
$\quad$ **let** $p \Leftarrow \llbracket P \rrbracket_\Delta$ **in**
$\quad$ **let** $q \Leftarrow \llbracket Q \rrbracket_{\Delta,x}$ **in**
$\quad$ **let** $s \Leftarrow [\lambda\Delta.\, q\, \langle\Delta, p\, \Delta\rangle]$ **in**
$\quad$ **let** $r \Leftarrow \llbracket R \rrbracket_{\Delta,y}$ **in**
$\quad\ [\lambda\Delta.\, r\, \langle\Delta, s\, \Delta\rangle]$
$=\qquad$ (left)
$\quad$ **let** $p \Leftarrow \llbracket P \rrbracket_\Delta$ **in**
$\quad$ **let** $q \Leftarrow \llbracket Q \rrbracket_{\Delta,x}$ **in**
$\quad$ **let** $r \Leftarrow \llbracket R \rrbracket_{\Delta,y}$ **in**
$\quad\ [\lambda\Delta.\, r\, \langle\Delta, q\, \langle\Delta, p\, \Delta\rangle\rangle]$
$=\qquad (\text{left}, \beta^\rightarrow)$
$\quad$ **let** $p \Leftarrow \llbracket P \rrbracket_\Delta$ **in**
$\quad$ **let** $q \Leftarrow \llbracket Q \rrbracket_{\Delta,x}$ **in**
$\quad$ **let** $r \Leftarrow \llbracket R \rrbracket_{\Delta,y}$ **in**
$\quad$ **let** $t \Leftarrow [\lambda\langle\langle\Delta, x\rangle, y\rangle.\, r\, \langle\Delta, y\rangle]$ **in**
$\quad$ **let** $s \Leftarrow [\lambda\langle\Delta, x\rangle.\, t\, \langle\langle\Delta, x\rangle, q\, \langle\Delta, x\rangle\rangle]$ **in**
$\quad\ [\lambda\Delta.\, s\, \langle\Delta, p\, \Delta\rangle]$
$=\qquad$ (assoc)
$\quad$ **let** $p \Leftarrow \llbracket P \rrbracket_\Delta$ **in**
$\quad$ **let** $s \Leftarrow (\textbf{let}\ q \Leftarrow \llbracket Q \rrbracket_{\Delta,x}\ \textbf{in}$
$\qquad\qquad \textbf{let}\ t \Leftarrow (\textbf{let}\ r \Leftarrow \llbracket R \rrbracket_{\Delta,y}\ \textbf{in}\ [\lambda\langle\langle\Delta, x\rangle, y\rangle.\, r\, \langle\Delta, y\rangle])$ **in**
$\qquad\qquad\quad [\lambda\langle\Delta, x\rangle.\, t\, \langle\langle\Delta, x\rangle, q\, \langle\Delta, x\rangle\rangle])$ **in**
$\quad\ [\lambda\Delta.\, s\, \langle\Delta, p\, \Delta\rangle]$

(continued from previous page)
$\textbf{let } p \Leftarrow [\![P]\!]_\Delta \textbf{ in}$
$\textbf{let } s \Leftarrow (\textbf{let } q \Leftarrow [\![Q]\!]_{\Delta,x} \textbf{ in}$
$\qquad\qquad \textbf{let } t \Leftarrow (\textbf{let } r \Leftarrow [\![R]\!]_{\Delta,y} \textbf{ in } [\lambda \langle\langle \Delta, x\rangle, y\rangle.\, r\, \langle \Delta, y\rangle]) \textbf{ in}$
$\qquad\qquad\qquad [\lambda\langle \Delta, x\rangle.\, t\, \langle\langle \Delta, x\rangle, q\, \langle \Delta, x\rangle\rangle]) \textbf{ in}$
$\quad [\lambda\Delta.\, s\, \langle \Delta, p\, \Delta\rangle]$

$=\qquad\quad$ (Lemma 18)

$\textbf{let } p \Leftarrow [\![P]\!]_\Delta \textbf{ in}$
$\textbf{let } s \Leftarrow (\textbf{let } q \Leftarrow [\![Q]\!]_{\Delta,x} \textbf{ in}$
$\qquad\qquad \textbf{let } r \Leftarrow [\![R]\!]_{\Delta,x,y} \textbf{ in } [\lambda\langle \Delta, x\rangle.\, r\, \langle\langle \Delta, x\rangle, q\, \langle \Delta, x\rangle\rangle]) \textbf{ in}$
$\quad [\lambda\Delta.\, s\, \langle \Delta, p\, \Delta\rangle]$

$=\qquad\quad$ (def $[\![-]\!]_\Delta$)

$[\![\textbf{let } x \Leftarrow P \textbf{ in } (\textbf{let } y \Leftarrow Q \textbf{ in } R)]\!]_\Delta$

6. (run_1)

$\qquad [\![L \bullet M]\!]_\Delta$
$=\qquad\quad$ (def $[\![-]\!]_\Delta$)
$\quad \textbf{let } h \Leftarrow [\![L]\!] \bullet \langle\rangle \textbf{ in } [\lambda\Delta.\, h\, [\![M]\!]]$
$=\qquad\quad$ (left)
$\quad \textbf{let } h \Leftarrow [\![L]\!] \bullet \langle\rangle \textbf{ in}$
$\quad \textbf{let } p \Leftarrow [\lambda\Delta.\, h] \textbf{ in}$
$\quad \textbf{let } q \Leftarrow [\lambda\langle \Delta, f\rangle.\, f\, [\![M]\!]] \textbf{ in } [\lambda\Delta.\, q\, \langle \Delta, p\, \Delta\rangle]$
$=\qquad\quad$ (assoc)
$\quad \textbf{let } p \Leftarrow \textbf{let } h \Leftarrow [\![L]\!] \bullet \langle\rangle \textbf{ in } [\lambda\Delta.\, h] \textbf{ in}$
$\quad \textbf{let } q \Leftarrow [\lambda\langle \Delta, f\rangle.\, f\, [\![M]\!]] \textbf{ in } [\lambda\Delta.\, q\, \langle \Delta, p\, \Delta\rangle]$
$=\qquad\quad$ (def $[\![-]\!]_\Delta$)
$\quad [\![\textbf{let } f \Leftarrow \textbf{run } L \textbf{ in } [f\, M]]\!]_\Delta$

7. (run_2)

$\qquad [\![\textbf{run } (\lambda^\bullet x.\, [M])]\!]_\Delta$
$=\qquad\quad$ (def $[\![-]\!]_\Delta$)
$\quad \textbf{let } h \Leftarrow (\lambda^\bullet\langle\rangle.\, [\lambda x.\, [\![M]\!]]) \bullet \langle\rangle \textbf{ in } [\lambda\Delta.\, h]$
$=\qquad\quad$ ($\beta^{\rightsquigarrow}$)
$\quad \textbf{let } h \Leftarrow [\lambda x.\, [\![M]\!]] \textbf{ in } [\lambda\Delta.\, h]$
$=\qquad\quad$ (left)
$\quad [\lambda\Delta.\, \lambda x.\, [\![M]\!]]$
$=\qquad\quad$ (def $[\![-]\!]_\Delta$)
$\quad [\![[\lambda x.\, M]]\!]_\Delta$

8. (run_3)

$\llbracket \mathbf{run}\ (\lambda^\bullet x.\, \mathbf{let}\ y \Leftarrow P\ \mathbf{in}\ Q) \rrbracket_\Delta$

$=$ (def $\llbracket - \rrbracket_\Delta$)

$\mathbf{let}\ h \Leftarrow (\lambda^\bullet \langle\rangle.\, \mathbf{let}\ p \Leftarrow \llbracket P \rrbracket_x\ \mathbf{in}\ \mathbf{let}\ q \Leftarrow \llbracket Q \rrbracket_{x,y}\ \mathbf{in}\ [\lambda x.\, q\ \langle x, p\ x\rangle]) \bullet \langle\rangle\ \mathbf{in}\ [\lambda\Delta.\, h]$

$=$ $(\beta^{\rightsquigarrow})$

$\mathbf{let}\ h \Leftarrow \mathbf{let}\ p \Leftarrow \llbracket P \rrbracket_x\ \mathbf{in}\ \mathbf{let}\ q \Leftarrow \llbracket Q \rrbracket_{x,y}\ \mathbf{in}\ [\lambda x.\, q\ \langle x, p\ x\rangle]\ \mathbf{in}\ [\lambda\Delta.\, h]$

$=$ (Lemma 18)

$\mathbf{let}\ h \Leftarrow \mathbf{let}\ p \Leftarrow \mathbf{let}\ w \Leftarrow \llbracket P \rrbracket_{\langle\rangle}\ \mathbf{in}\ [\lambda x.\, w\ \langle\rangle]\ \mathbf{in}$

         $\mathbf{let}\ q \Leftarrow \llbracket Q \rrbracket_{x,y}\ \mathbf{in}\ [\lambda x.\, q\ \langle x, p\ x\rangle]\ \mathbf{in}$

$[\lambda\Delta.\, h]$

$=$ (assoc)

$\mathbf{let}\ p \Leftarrow \llbracket P \rrbracket_{\langle\rangle}\ \mathbf{in}\ \mathbf{let}\ w \Leftarrow [\lambda x.\, p\ \langle\rangle]\ \mathbf{in}$

$\mathbf{let}\ q \Leftarrow \llbracket Q \rrbracket_{x,y}\ \mathbf{in}\ \mathbf{let}\ h \Leftarrow [\lambda x.\, q\ \langle x, w\ x\rangle]\ \mathbf{in}\ [\lambda\Delta.\, h]$

$=$ (left)

$\mathbf{let}\ p \Leftarrow \llbracket P \rrbracket_{\langle\rangle}\ \mathbf{in}\ \mathbf{let}\ q \Leftarrow \llbracket Q \rrbracket_{x,y}\ \mathbf{in}\ [\lambda\Delta.\, \lambda x.\, q\ \langle x, p\ \langle\rangle\rangle]$

$=$ (left)

$\mathbf{let}\ p \Leftarrow \llbracket P \rrbracket_{\langle\rangle}\ \mathbf{in}\ \mathbf{let}\ z \Leftarrow [\lambda\Delta.\, p\ \langle\rangle]\ \mathbf{in}\ \mathbf{let}\ q \Leftarrow \llbracket Q \rrbracket_{x,y}\ \mathbf{in}\ [\lambda\Delta.\, \lambda x.\, q\ \langle x, z\ \Delta\rangle]$

$=$ (assoc)

$\mathbf{let}\ z \Leftarrow (\mathbf{let}\ p \Leftarrow \llbracket P \rrbracket_{\langle\rangle}\ \mathbf{in}\ [\lambda\Delta.\, p\ \langle\rangle])\ \mathbf{in}\ \mathbf{let}\ q \Leftarrow \llbracket Q \rrbracket_{x,y}\ \mathbf{in}\ [\lambda\Delta.\, \lambda x.\, q\ \langle x, z\ \Delta\rangle]$

$=$ (Lemma 18)

$\mathbf{let}\ p \Leftarrow \llbracket P \rrbracket_\Delta\ \mathbf{in}\ \mathbf{let}\ q \Leftarrow \llbracket Q \rrbracket_{x,y}\ \mathbf{in}\ [\lambda\Delta.\, \lambda x.\, q\ \langle x, p\ \Delta\rangle]$

$=$ (left, $\beta^{\rightarrow}$)

$\mathbf{let}\ q \Leftarrow \llbracket Q \rrbracket_{x,y}\ \mathbf{in}$

$\mathbf{let}\ p \Leftarrow \llbracket P \rrbracket_\Delta\ \mathbf{in}$

$\mathbf{let}\ r \Leftarrow [\lambda\langle\Delta, y\rangle.\, q]\ \mathbf{in}$

$\mathbf{let}\ t \Leftarrow [\lambda\langle\langle\Delta, y\rangle, f\rangle.\, \lambda x.\, f\ \langle x, y\rangle]\ \mathbf{in}$

$\mathbf{let}\ s \Leftarrow [\lambda\langle\Delta, y\rangle.\, t\ \langle\langle\Delta, y\rangle, r\ \langle\Delta, y\rangle\rangle]\ \mathbf{in}\ [\lambda\Delta.\, s\ \langle\Delta, p\ \Delta\rangle]$

$=$ $(\beta^{\rightsquigarrow})$

$\mathbf{let}\ q \Leftarrow (\lambda^\bullet \langle\rangle.\, \llbracket Q \rrbracket_{x,y}) \bullet \langle\rangle\ \mathbf{in}$

$\mathbf{let}\ p \Leftarrow \llbracket P \rrbracket_\Delta\ \mathbf{in}$

$\mathbf{let}\ r \Leftarrow [\lambda\langle\Delta, y\rangle.\, q]\ \mathbf{in}$

$\mathbf{let}\ t \Leftarrow [\lambda\langle\langle\Delta, y\rangle, f\rangle.\, \lambda x.\, f\ \langle x, y\rangle]\ \mathbf{in}$

$\mathbf{let}\ s \Leftarrow [\lambda\langle\Delta, y\rangle.\, t\ \langle\langle\Delta, y\rangle, r\ \langle\Delta, y\rangle\rangle]\ \mathbf{in}\ [\lambda\Delta.\, s\ \langle\Delta, p\ \Delta\rangle]$

$=$ (assoc)

$\mathbf{let}\ p \Leftarrow \llbracket P \rrbracket_\Delta\ \mathbf{in}$

$\mathbf{let}\ s \Leftarrow (\mathbf{let}\ r \Leftarrow (\mathbf{let}\ q \Leftarrow (\lambda^\bullet \langle\rangle.\, \llbracket Q \rrbracket_{x,y}) \bullet \langle\rangle\ \mathbf{in}\ [\lambda\langle\Delta, y\rangle.\, q])\ \mathbf{in}$

         $\mathbf{let}\ t \Leftarrow [\lambda\langle\langle\Delta, y\rangle, f\rangle.\, \lambda x.\, f\ \langle x, y\rangle]\ \mathbf{in}\ [\lambda\langle\Delta, y\rangle.\, t\ \langle\langle\Delta, y\rangle, r\ \langle\Delta, y\rangle\rangle])\ \mathbf{in}$

$[\lambda\Delta.\, s\ \langle\Delta, p\ \Delta\rangle]$

$=$ (def $\llbracket - \rrbracket_\Delta$)

$\llbracket \mathbf{let}\ y \Leftarrow P\ \mathbf{in}\ \mathbf{let}\ f \Leftarrow \mathbf{run}\ (\lambda^\bullet \langle x, y\rangle.\, Q)\ \mathbf{in}\ [\lambda x.\, f\ \langle x, y\rangle] \rrbracket_\Delta$

A.5.3   The laws of $\mathcal{A}$ follow from the laws of $\mathcal{S}$

For each law $M = N$ or $P = Q$ of $\mathcal{A}$ we must show $\langle\!\langle P \rangle\!\rangle = \langle\!\langle Q \rangle\!\rangle$.

1. $(\beta^{\rightsquigarrow})$

$$\langle\!\langle (\lambda^{\bullet} \langle\rangle.\, Q) \bullet M \rangle\!\rangle$$

$=$ (def $\langle\!\langle - \rangle\!\rangle$)

$$\mathbf{run}\ (\lambda^{\bullet} x.\, \mathbf{let}\ y \Leftarrow \langle\!\langle Q \rangle\!\rangle\ \mathbf{in}\ [y\,x])$$

$=$ (run_3)

$$\mathbf{let}\ y \Leftarrow \langle\!\langle Q \rangle\!\rangle\ \mathbf{in}\ \mathbf{let}\ f \Leftarrow \mathbf{run}\ (\lambda^{\bullet} \langle x, y\rangle.\, [y\,x])\ \mathbf{in}\ [\lambda x.\, f\,\langle x, y\rangle]$$

$=$ (run_2)

$$\mathbf{let}\ y \Leftarrow \langle\!\langle Q \rangle\!\rangle\ \mathbf{in}\ \mathbf{let}\ f \Leftarrow [\lambda\langle x, y\rangle.\, y\,x]\ \mathbf{in}\ [\lambda x.\, f\,\langle x, y\rangle]$$

$=$ (left)

$$\mathbf{let}\ y \Leftarrow \langle\!\langle Q \rangle\!\rangle\ \mathbf{in}\ [\lambda x.\, (\lambda\langle x, y\rangle.\, y\,x)\,\langle x, y\rangle]$$

$=$ $(\beta^{\rightarrow})$

$$\mathbf{let}\ y \Leftarrow \langle\!\langle Q \rangle\!\rangle\ \mathbf{in}\ [\lambda x.\, y\,x]$$

$=$ $(\eta^{\rightarrow})$

$$\mathbf{let}\ y \Leftarrow \langle\!\langle Q \rangle\!\rangle\ \mathbf{in}\ [y]$$

$=$ (right)

$$\langle\!\langle Q \rangle\!\rangle$$

$=$ (Lemma 17)

$$\langle\!\langle Q[x := M] \rangle\!\rangle$$

2. $(\eta^{\rightsquigarrow})$

$$\langle\!\langle \lambda^{\bullet} \langle\rangle.\, L \bullet \langle\rangle \rangle\!\rangle$$

$=$ (def $\langle\!\langle - \rangle\!\rangle$)

$$\lambda^{\bullet} x.\, \mathbf{let}\ h \Leftarrow \mathbf{run}\ \langle\!\langle L \rangle\!\rangle\ \mathbf{in}\ [h\,x]$$

$=$ (run_1)

$$\lambda^{\bullet} x.\, \langle\!\langle L \rangle\!\rangle \bullet x$$

$=$ $(\eta^{\rightsquigarrow})$

$$\langle\!\langle L \rangle\!\rangle$$

3. (left)

$$\langle\!\langle \mathbf{let}\ x \Leftarrow [M]\ \mathbf{in}\ Q \rangle\!\rangle$$

$=$ (def $\langle\!\langle - \rangle\!\rangle$)

$$\mathbf{let}\ x \Leftarrow [\langle\!\langle M \rangle\!\rangle]\ \mathbf{in}\ \langle\!\langle Q \rangle\!\rangle$$

$=$ (left)

$$\langle\!\langle Q \rangle\!\rangle\,[x := \langle\!\langle M \rangle\!\rangle]$$

$=$ (Lemma 17)

$$\langle\!\langle Q[x := M] \rangle\!\rangle$$

4. (right)

$\langle\!\langle\textbf{let } \text{x} \Leftarrow \text{P } \textbf{in } [\text{x}]\rangle\!\rangle$

$=$ $\quad$ (def $\langle\!\langle-\rangle\!\rangle$)

$\textbf{let } \text{x} \Leftarrow \langle\!\langle\text{P}\rangle\!\rangle \textbf{ in } [\text{x}]$

$=$ $\quad$ (right)

$\langle\!\langle\text{P}\rangle\!\rangle$

5. (assoc)

$\langle\!\langle\textbf{let } \text{y} \Leftarrow (\textbf{let } \text{x} \Leftarrow \text{P } \textbf{in } \text{Q}) \textbf{ in } \text{R}\rangle\!\rangle$

$=$ $\quad$ (def $\langle\!\langle-\rangle\!\rangle$)

$\textbf{let } \text{y} \Leftarrow \textbf{let } \text{x} \Leftarrow \langle\!\langle\text{P}\rangle\!\rangle \textbf{ in } \langle\!\langle\text{Q}\rangle\!\rangle \textbf{ in } \langle\!\langle\text{R}\rangle\!\rangle$

$=$ $\quad$ (assoc)

$\textbf{let } \text{x} \Leftarrow \langle\!\langle\text{P}\rangle\!\rangle \textbf{ in let } \text{y} \Leftarrow \langle\!\langle\text{Q}\rangle\!\rangle \textbf{ in } \langle\!\langle\text{R}\rangle\!\rangle$

$=$ $\quad$ (def $\langle\!\langle-\rangle\!\rangle$)

$\langle\!\langle\textbf{let } \text{x} \Leftarrow \text{P } \textbf{in let } \text{y} \Leftarrow \text{Q } \textbf{in } \text{R}\rangle\!\rangle$

A.5.4 Translating $\mathcal{S}$ to $\mathcal{A}$ and back

For each term M of $\mathcal{S}$ we must show $\langle\!\langle[\![M]\!]\rangle\!\rangle = M$. For each command P of $\mathcal{S}$ we must show $\langle\!\langle[\![P]\!]_\Delta\rangle\!\rangle \bullet \Delta = P$. The proof is by mutual induction on the derivations of M and P.

1. Case $\lambda^\bullet x.\, Q$

$$
\begin{aligned}
&\langle\!\langle[\![\lambda^\bullet x.\, Q]\!]\rangle\!\rangle \\
=\quad &\text{(def } [\![-]\!]) \\
&\langle\!\langle\lambda^\bullet\langle\rangle.\, [\![Q]\!]_x\rangle\!\rangle \\
=\quad &\text{(def } [\![-]\!]) \\
&\lambda^\bullet x.\, \textbf{let } h \Leftarrow \langle\!\langle[\![Q]\!]_x\rangle\!\rangle \textbf{ in } [h\, x] \\
=\quad &\text{(induction hypothesis)} \\
&\lambda^\bullet x.\, \textbf{let } h \Leftarrow \textbf{run } (\lambda^\bullet x.\, Q) \textbf{ in } [h\, x] \\
=\quad &(run_1) \\
&\lambda^\bullet x.\, (\lambda^\bullet x.\, Q) \bullet x \\
=\quad &(\beta^{\rightsquigarrow}) \\
&\lambda^\bullet x.\, Q
\end{aligned}
$$

2. Case $L \bullet M$

$$
\begin{aligned}
&\textbf{let } d \Leftarrow \langle\!\langle[\![L \bullet M]\!]_\Delta\rangle\!\rangle \textbf{ in } [d\, \Delta] \\
=\quad &\text{(def } [\![-]\!]_\Delta) \\
&\textbf{let } d \Leftarrow \langle\!\langle\textbf{let } l \Leftarrow [\![L]\!] \bullet \langle\rangle \textbf{ in } [\lambda\Delta.\, l\, [\![M]\!]]\rangle\!\rangle \textbf{ in } [d\, \Delta] \\
=\quad &\text{(def } \langle\!\langle-\rangle\!\rangle) \\
&\textbf{let } d \Leftarrow \textbf{let } l \Leftarrow \textbf{run } \langle\!\langle[\![L]\!]\rangle\!\rangle \textbf{ in } [\lambda\Delta.\, l\, \langle\!\langle[\![M]\!]\rangle\!\rangle] \textbf{ in } [d\, \Delta] \\
=\quad &\text{(induction hypothesis)} \\
&\textbf{let } d \Leftarrow \textbf{let } l \Leftarrow \textbf{run } L \textbf{ in } [\lambda\Delta.\, l\, M] \textbf{ in } [d\, \Delta] \\
=\quad &\text{(assoc)} \\
&\textbf{let } l \Leftarrow \textbf{run } L \textbf{ in let } d \Leftarrow [\lambda\Delta.\, l\, M] \textbf{ in } [d\, \Delta] \\
=\quad &\text{(left)} \\
&\textbf{let } l \Leftarrow \textbf{run } L \textbf{ in } [l\, M] \\
=\quad &(run_1) \\
&L \bullet M
\end{aligned}
$$

3. Case $[M]$

$$
\begin{aligned}
&\textbf{let } d \Leftarrow \langle\!\langle[\![M]\!]_\Delta\rangle\!\rangle \textbf{ in } [d\, \Delta] \\
=\quad &\text{(def } [\![-]\!]_\Delta) \\
&\textbf{let } d \Leftarrow \langle\!\langle[\lambda\Delta.\, [\![M]\!]]\rangle\!\rangle \textbf{ in } [d\, \Delta] \\
=\quad &\text{(def } \langle\!\langle-\rangle\!\rangle) \\
&\textbf{let } d \Leftarrow [\lambda\Delta.\, \langle\!\langle[\![M]\!]\rangle\!\rangle] \textbf{ in } [d\, \Delta] \\
=\quad &\text{(induction hypothesis)} \\
&\textbf{let } d \Leftarrow [\lambda\Delta.\, M] \textbf{ in } [d\, \Delta] \\
=\quad &\text{(left)} \\
&[M]
\end{aligned}
$$

4. Case **let** $x \Leftarrow P$ **in** Q

$\qquad$ **let** $d \Leftarrow \langle\!\langle [\![\textbf{let } x \Leftarrow P \textbf{ in } Q]\!]_\Delta \rangle\!\rangle$ **in** $[d\,\Delta]$
$=\qquad$ (def $[\![-]\!]_\Delta$)
$\qquad$ **let** $d \Leftarrow \langle\!\langle \textbf{let } p \Leftarrow [\![P]\!]_\Delta \textbf{ in let } q \Leftarrow [\![Q]\!]_{\Delta,x} \textbf{ in } [\lambda\Delta.\, q\,\langle \Delta, p\,\Delta \rangle]\rangle\!\rangle$ **in** $[d\,\Delta]$
$=\qquad$ (def $\langle\!\langle-\rangle\!\rangle$)
$\qquad$ **let** $d \Leftarrow \textbf{let } p \Leftarrow \langle\!\langle [\![P]\!]_\Delta \rangle\!\rangle \textbf{ in let } q \Leftarrow \langle\!\langle [\![Q]\!]_{\Delta,x} \rangle\!\rangle \textbf{ in } [q\,\langle \Delta, p\,\Delta \rangle]$ **in** $[d\,\Delta]$
$=\qquad$ (assoc)
$\qquad$ **let** $p \Leftarrow \langle\!\langle [\![P]\!]_\Delta \rangle\!\rangle$ **in let** $q \Leftarrow \langle\!\langle [\![Q]\!]_{\Delta,x} \rangle\!\rangle$ **in let** $d \Leftarrow [\lambda\Delta.\, q\,\langle \Delta, p\,\Delta \rangle]$ **in** $[d\,\Delta]$
$=\qquad$ (left)
$\qquad$ **let** $p \Leftarrow \langle\!\langle [\![P]\!]_\Delta \rangle\!\rangle$ **in let** $q \Leftarrow \langle\!\langle [\![Q]\!]_{\Delta,x} \rangle\!\rangle$ **in** $[q\,\langle \Delta, p\,\Delta \rangle]$
$=\qquad$ (induction hypothesis)
$\qquad$ **let** $p \Leftarrow \textbf{run } (\lambda^\bullet \Delta.\, P)$ **in let** $q \Leftarrow \textbf{run } (\lambda^\bullet \langle \Delta, x \rangle.\, Q)$ **in** $[q\,\langle \Delta, p\,\Delta \rangle]$
$=\qquad$ (run_1)
$\qquad$ **let** $p \Leftarrow \textbf{run } (\lambda^\bullet \Delta.\, P)$ **in** $(\lambda^\bullet \langle \Delta, x \rangle.\, Q) \bullet \langle \Delta, p\,\Delta \rangle$
$=\qquad$ ($\beta^{\leadsto}$)
$\qquad$ **let** $p \Leftarrow \textbf{run } (\lambda^\bullet \Delta.\, P)$ **in** $Q[x := p\,\Delta]$
$=\qquad$ (left)
$\qquad$ **let** $p \Leftarrow \textbf{run } (\lambda^\bullet \Delta.\, P)$ **in let** $x \Leftarrow [p\,\Delta]$ **in** Q
$=\qquad$ (assoc)
$\qquad$ **let** $x \Leftarrow \textbf{let } p \Leftarrow \textbf{run } (\lambda^\bullet \Delta.\, P) \textbf{ in } [p\,\Delta]$ **in** Q
$=\qquad$ (run_1)
$\qquad$ **let** $x \Leftarrow (\lambda^\bullet \Delta.\, P) \bullet \Delta$ **in** Q
$=\qquad$ ($\beta^{\leadsto}$)
$\qquad$ **let** $x \Leftarrow P$ **in** Q

5. Case **run** L

$\qquad$ **let** $d \Leftarrow \langle\!\langle [\![\textbf{run } L]\!]_\Delta \rangle\!\rangle$ **in** $[d\,\Delta]$
$=\qquad$ (def $[\![-]\!]_\Delta$)
$\qquad$ **let** $d \Leftarrow \langle\!\langle \textbf{let } h \Leftarrow [\![L]\!] \bullet \langle\rangle \textbf{ in } [\lambda\Delta.\, h]\rangle\!\rangle$ **in** $[d\,\Delta]$
$=\qquad$ (def $\langle\!\langle-\rangle\!\rangle$)
$\qquad$ **let** $d \Leftarrow \textbf{let } h \Leftarrow \textbf{run } \langle\!\langle [\![L]\!] \rangle\!\rangle \textbf{ in } [\lambda\Delta.\, h]$ **in** $[d\,\Delta]$
$=\qquad$ (induction hypothesis)
$\qquad$ **let** $d \Leftarrow \textbf{let } h \Leftarrow \textbf{run } L \textbf{ in } [\lambda\Delta.\, h]$ **in** $[d\,\Delta]$
$=\qquad$ (assoc)
$\qquad$ **let** $h \Leftarrow \textbf{run } L$ **in let** $d \Leftarrow [\lambda\Delta.\, h]$ **in** $[d\,\Delta]$
$=\qquad$ (left)
$\qquad$ **let** $h \Leftarrow \textbf{run } L$ **in** $[h]$
$=\qquad$ (right)
$\qquad$ **run** L

A.6   Equational correspondence between $\mathcal{H}$ and $\mathcal{C}_{\textbf{app}}$

This section gives a proof of Proposition 20 (page 77). The proofs here extend the proofs of
the equational correspondence between $\mathcal{A}$ and $\mathcal{C}$ in Section A.2.

A.6.1   Extensions of Lemmas 8 and 10

We must first extend Lemmas 8 and 10 to show that the translations of substitution and weak-
ening hold for the $L \star M$ construct.

The proof of Lemma 8 for $L \star M$ is as follows:

$$
\begin{aligned}
& [\![(L \star M)[x := N]]\!]_{\Delta} \\
=\quad & \text{(def substitution)} \\
& [\![(L[x := N]) \star (M[x := N])]\!]_{\Delta} \\
=\quad & (\text{def } [\![-]\!]_{\Delta}) \\
& \texttt{arr} \,(\lambda\Delta.\, \langle [\![L[x := N]]\!], [\![M[x := N]]\!] \rangle) \ggg \texttt{app} \\
=\quad & (\leadsto_4, \text{def } \cdot) \\
& \texttt{arr} \,(\lambda\Delta.\, \langle \Delta, [\![N]\!] \rangle) \ggg \texttt{arr} \,(\lambda\langle\Delta, x\rangle.\, \langle [\![L]\!], [\![M]\!] \rangle) \ggg \texttt{app} \\
=\quad & (\text{def } [\![-]\!]_{\Delta, x}) \\
& \texttt{arr} \,(\lambda\Delta.\, \langle \Delta, [\![N]\!] \rangle) \ggg [\![L \star M]\!]_{\Delta, x}
\end{aligned}
$$

The proof of Lemma 10 for $L \star M$ is as follows:

$$
\begin{aligned}
& [\![L \star M]\!]_{\Delta'} \\
=\quad & (\text{def } [\![-]\!]_{\Delta'}) \\
& \texttt{arr} \,(\lambda\Delta'.\, \langle [\![L]\!], [\![M]\!] \rangle) \ggg \texttt{app} \\
=\quad & (\leadsto_4, \text{def } \cdot) \\
& \texttt{arr} \,(\lambda\Delta'.\, \Delta) \ggg \texttt{arr} \,(\lambda\Delta.\, \langle [\![L]\!], [\![M]\!] \rangle) \ggg \texttt{app} \\
=\quad & (\text{def } [\![-]\!]_{\Delta}) \\
& \texttt{arr} \,(\lambda\Delta'.\, \Delta) \ggg [\![L \star M]\!]_{\Delta}
\end{aligned}
$$

A.6.2 The laws of $\mathcal{H}$ follow from the laws of $\mathcal{C}_{\textbf{app}}$

For each higher-order arrows law on terms $M = N$ we must show $[\![M]\!] = [\![N]\!]$ and for each higher-order arrows law on commands $P = Q$ we must show $[\![P]\!]_\Delta = [\![Q]\!]_\Delta$.

1. $(\beta^{\mathtt{app}})$

$\qquad [\![(\lambda^\bullet x. Q) \star M]\!]_\Delta$
$=\qquad (\text{def } [\![-]\!]_\Delta)$
$\qquad \mathtt{arr}\, (\lambda\Delta. \langle [\![Q]\!]_x, [\![M]\!]\rangle) \ggg \mathtt{app}$
$=\qquad (((\lambda x. M) \times (\lambda x. N)) \cdot \mathtt{dup} = (\lambda x. \langle M, N\rangle))$
$\qquad \mathtt{arr}\, \mathtt{dup} \ggg \mathtt{arr}\, ((\lambda\Delta. [\![Q]\!]_x) \times (\lambda\Delta. [\![M]\!])) \ggg \mathtt{app}$
$=\qquad (\mathtt{swap} \cdot (f \times \mathtt{id}) \cdot \mathtt{swap} \cdot (g \times \mathtt{id}) \cdot \mathtt{dup} = g \times f)$
$\qquad \mathtt{arr}\, (\mathtt{swap} \cdot ((\lambda\Delta. [\![M]\!]) \times \mathtt{id}) \cdot \mathtt{swap} \cdot ((\lambda\Delta. [\![Q]\!]_x) \times \mathtt{id}) \cdot \mathtt{dup}) \ggg \mathtt{app}$
$=\qquad (\leadsto_4)$
$\qquad \mathtt{arr}\, \mathtt{dup} \ggg \mathtt{arr}\, ((\lambda\Delta. [\![Q]\!]_x) \times \mathtt{id}) \ggg \mathtt{arr}\, \mathtt{swap} \ggg \mathtt{arr}\, ((\lambda\Delta. [\![M]\!]) \times \mathtt{id})$
$\qquad\quad \ggg \mathtt{arr}\, \mathtt{swap} \ggg \mathtt{app}$
$=\qquad (\leadsto_5)$
$\qquad \mathtt{arr}\, \mathtt{dup} \ggg \mathtt{first}\, (\mathtt{arr}\, (\lambda\Delta. [\![Q]\!]_x)) \ggg \mathtt{arr}\, \mathtt{swap} \ggg \mathtt{first}\, (\mathtt{arr}\, (\lambda\Delta. [\![M]\!]))$
$\qquad\quad \ggg \mathtt{arr}\, \mathtt{swap} \ggg \mathtt{app}$
$=\qquad (\text{def } \&\!\&\!\&, \text{def } \mathtt{second})$
$\qquad (\mathtt{arr}\, (\lambda\Delta. [\![Q]\!]_x) \&\!\&\!\& \mathtt{arr}\, (\lambda\Delta. [\![M]\!])) \ggg \mathtt{app}$
$=\qquad (\text{Lemma } 10)$
$\qquad (\mathtt{arr}\, (\lambda\Delta. (\mathtt{arr}\, (\lambda x. \langle\Delta, x\rangle) \ggg [\![Q]\!]_{\Delta,x})) \&\!\&\!\& \mathtt{arr}\, (\lambda\Delta. [\![M]\!])) \ggg \mathtt{app}$
$=\qquad (\text{def } \&\!\&\!\&)$
$\qquad \mathtt{arr}\, \mathtt{dup} \ggg \mathtt{first}\, (\mathtt{arr}\, ((\ggg [\![Q]\!]_{\Delta,x}) \cdot ((\lambda x. \mathtt{arr}\, (\lambda y. \langle x, y\rangle)))))$
$\qquad\quad \ggg \mathtt{second}\, (\mathtt{arr}\, (\lambda\Delta. [\![M]\!])) \ggg \mathtt{app}$
$=\qquad (\leadsto_5)$
$\qquad \mathtt{arr}\, \mathtt{dup} \ggg \mathtt{arr}\, (((\ggg [\![Q]\!]_{\Delta,x}) \cdot ((\lambda x. \mathtt{arr}\, (\lambda y. \langle x, y\rangle)))) \times \mathtt{id})$
$\qquad\quad \ggg \mathtt{second}\, (\mathtt{arr}\, (\lambda\Delta. [\![M]\!])) \ggg \mathtt{app}$
$=\qquad (\mathtt{arr}\, (f \times \mathtt{id}) \ggg \mathtt{second}\, g = \mathtt{second}\, g \ggg \mathtt{arr}\, (f \times \mathtt{id}))$
$\qquad \mathtt{arr}\, \mathtt{dup} \ggg \mathtt{second}\, (\mathtt{arr}\, (\lambda\Delta. [\![M]\!]))$
$\qquad\quad \ggg \mathtt{arr}\, (((\ggg [\![Q]\!]_{\Delta,x}) \cdot ((\lambda x. \mathtt{arr}\, (\lambda y. \langle x, y\rangle)))) \times \mathtt{id}) \ggg \mathtt{app}$
$=\qquad (\leadsto_5)$
$\qquad \mathtt{arr}\, \mathtt{dup} \ggg \mathtt{second}\, (\mathtt{arr}\, (\lambda\Delta. [\![M]\!]))$
$\qquad\quad \ggg \mathtt{first}\, (\mathtt{arr}\, ((\ggg [\![Q]\!]_{\Delta,x}) \cdot ((\lambda x. \mathtt{arr}\, (\lambda y. \langle x, y\rangle))))) \ggg \mathtt{app}$
$=\qquad (\leadsto_4)$
$\qquad \mathtt{arr}\, \mathtt{dup} \ggg \mathtt{second}\, (\mathtt{arr}\, (\lambda\Delta. [\![M]\!]))$
$\qquad\quad \ggg \mathtt{first}\, (\mathtt{arr}\, ((\lambda x. \mathtt{arr}\, (\lambda y. \langle x, y\rangle)))) \ggg \mathtt{arr}\, (\ggg [\![Q]\!]_{\Delta,x}) \ggg \mathtt{app}$
$=\qquad (\leadsto_6)$
$\qquad \mathtt{arr}\, \mathtt{dup} \ggg \mathtt{second}\, (\mathtt{arr}\, (\lambda\Delta. [\![M]\!])) \ggg \mathtt{first}\, (\mathtt{arr}\, ((\lambda x. \mathtt{arr}\, (\lambda y. \langle x, y\rangle))))$
$\qquad\quad \ggg \mathtt{first}\, (\mathtt{arr}\, (\ggg [\![Q]\!]_{\Delta,x})) \ggg \mathtt{app}$
$=\qquad (\leadsto_{H3})$
$\qquad \mathtt{arr}\, \mathtt{dup} \ggg \mathtt{second}\, (\mathtt{arr}\, (\lambda\Delta. [\![M]\!])) \ggg \mathtt{first}\, (\mathtt{arr}\, ((\lambda x. \mathtt{arr}\, (\lambda y. \langle x, y\rangle))))$
$\qquad\quad \ggg \mathtt{app} \ggg [\![Q]\!]_{\Delta,x}$

(continued on next page)

(continued from previous page)

$\mathtt{arr\ dup} \ggg \mathtt{second}\,(\mathtt{arr}\,(\lambda\Delta.\,[\![M]\!])) \ggg \mathtt{first}\,(\mathtt{arr}\,((\lambda x.\,\mathtt{arr}\,(\lambda y.\,\langle x, y\rangle))))$
$\qquad \ggg \mathtt{app} \ggg [\![Q]\!]_{\Delta,x}$

$=\qquad(\leadsto_{H1})$

$\mathtt{arr\ dup} \ggg \mathtt{second}\,(\mathtt{arr}\,(\lambda\Delta.\,[\![M]\!])) \ggg \mathtt{arr\ id} \ggg [\![Q]\!]_{\Delta,x}$

$=\qquad(\leadsto_2)$

$\mathtt{arr\ dup} \ggg \mathtt{second}\,(\mathtt{arr}\,(\lambda\Delta.\,[\![M]\!])) \ggg [\![Q]\!]_{\Delta,x}$

$=\qquad(\leadsto_1, \mathtt{id} = \mathtt{id} \times \mathtt{id})$

$\mathtt{arr\ dup} \ggg \mathtt{arr}\,(\mathtt{id} \times \mathtt{id}) \ggg \mathtt{second}\,(\mathtt{arr}\,(\lambda\Delta.\,[\![M]\!])) \ggg [\![Q]\!]_{\Delta,x}$

$=\qquad(\leadsto_5)$

$\mathtt{arr\ dup} \ggg \mathtt{first}\,(\mathtt{arr\ id}) \ggg \mathtt{second}\,(\mathtt{arr}\,(\lambda\Delta.\,[\![M]\!])) \ggg [\![Q]\!]_{\Delta,x}$

$=\qquad(\mathrm{def}\ \&\!\&\!\&)$

$(\mathtt{arr\ id} \,\&\!\&\!\&\, \mathtt{arr}\,(\lambda\Delta.\,[\![M]\!])) \ggg [\![Q]\!]_{\Delta,x}$

$=\qquad(\mathrm{Lemma\ 8})$

$[\![Q[x := M]]\!]_{\Delta}$

A.6.3   The laws of $\mathcal{C}_{\mathsf{app}}$ follow from the laws of $\mathcal{H}$

For each law $M = N$ of $\mathcal{C}_{\mathsf{app}}$ we must show $\langle\!\langle M \rangle\!\rangle = \langle\!\langle N \rangle\!\rangle$.

1. $(\leadsto_{H1})$

$\langle\!\langle \mathtt{first}\,(\mathtt{arr}\,(\lambda x.\,\mathtt{arr}\,(\lambda y.\,\langle x, y\rangle))) \ggg \mathtt{app} \rangle\!\rangle$

$=\qquad(\mathrm{def}\ \langle\!\langle - \rangle\!\rangle)$

$\lambda^{\bullet} a.\,\mathbf{let}\ b \Leftarrow ((\lambda f.\,\lambda^{\bullet} z.\,\mathbf{let}\ c \Leftarrow f \bullet \mathbf{fst}\,z\ \mathbf{in}\ [\langle c, \mathbf{snd}\,z\rangle])$
$\qquad\qquad (\lambda^{\bullet} x.\,[\lambda^{\bullet} y.\,[\langle x, y\rangle]])) \bullet a\ \mathbf{in}$
$\qquad (\lambda^{\bullet} z.\,\mathbf{fst}\,z \star \mathbf{snd}\,z) \bullet b$

$=\qquad(\beta^{\rightarrow}, \beta^{\leadsto})$

$\lambda^{\bullet} a.\,\mathbf{let}\ b \Leftarrow (\mathbf{let}\ c \Leftarrow (\lambda^{\bullet} x.\,[\lambda^{\bullet} y.\,[\langle x, y\rangle]]) \bullet \mathbf{fst}\,a\ \mathbf{in}\ [\langle c, \mathbf{snd}\,a\rangle])\ \mathbf{in}$
$\qquad \mathbf{fst}\,b \star \mathbf{snd}\,b$

$=\qquad(\beta^{\leadsto}, \mathrm{assoc})$

$\lambda^{\bullet} a.\,\mathbf{let}\ c \Leftarrow [\lambda^{\bullet} y.\,[\langle \mathbf{fst}\,a, y\rangle]]\ \mathbf{in}\ \mathbf{let}\ b \Leftarrow [\langle c, \mathbf{snd}\,a\rangle]\ \mathbf{in}\ \mathbf{fst}\,b \star \mathbf{snd}\,b$

$=\qquad(\mathrm{left}, \beta_1^{\times}, \beta_2^{\times})$

$\lambda^{\bullet} a.\,(\lambda^{\bullet} y.\,[\langle \mathbf{fst}\,a, y\rangle]) \star \mathbf{snd}\,a$

$=\qquad(\beta^{\mathsf{app}})$

$\lambda^{\bullet} a.\,[\langle \mathbf{fst}\,a, \mathbf{snd}\,a\rangle]$

$=\qquad(\eta^{\times})$

$\lambda^{\bullet} a.\,[a]$

$=\qquad(\mathrm{def}\ \langle\!\langle - \rangle\!\rangle)$

$\langle\!\langle \mathtt{arr\ id} \rangle\!\rangle$

2. $(\rightsquigarrow_{H2})$

$$\langle\!\langle \text{first } (\text{arr } (L\ggg)) \ggg \text{app} \rangle\!\rangle$$
$$= \qquad (\text{def } \langle\!\langle - \rangle\!\rangle)$$
$$\lambda^{\bullet}a.\, \textbf{let } b \Leftarrow (\lambda^{\bullet}z.\, \textbf{let } x \Leftarrow (\lambda^{\bullet}f.\,[\langle\!\langle L \ggg f \rangle\!\rangle]) \bullet \textbf{fst } z \textbf{ in } [\langle x, \textbf{snd } z\rangle]) \bullet a \textbf{ in}$$
$$(\lambda^{\bullet}w.\, \textbf{fst } w \star \textbf{snd } w) \bullet b$$
$$= \qquad (\beta^{\rightsquigarrow}, \beta^{\rightsquigarrow})$$
$$\lambda^{\bullet}a.\, \textbf{let } b \Leftarrow (\textbf{let } x \Leftarrow (\lambda^{\bullet}f.\,[\langle\!\langle L \ggg f \rangle\!\rangle]) \bullet \textbf{fst } a \textbf{ in } [\langle x, \textbf{snd } a\rangle]) \textbf{ in}$$
$$\textbf{fst } b \star \textbf{snd } b$$
$$= \qquad (\text{assoc})$$
$$\lambda^{\bullet}a.\, \textbf{let } x \Leftarrow (\lambda^{\bullet}f.\,[\langle\!\langle L \ggg f \rangle\!\rangle]) \bullet \textbf{fst } a \textbf{ in}$$
$$\textbf{let } b \Leftarrow [\langle x, \textbf{snd } a\rangle] \textbf{ in}$$
$$\textbf{fst } b \star \textbf{snd } b$$
$$= \qquad (\beta^{\rightsquigarrow})$$
$$\lambda^{\bullet}a.\, \textbf{let } x \Leftarrow [\langle\!\langle L \ggg (\textbf{fst } a) \rangle\!\rangle] \textbf{ in}$$
$$\textbf{let } b \Leftarrow [\langle x, \textbf{snd } a\rangle] \textbf{ in}$$
$$\textbf{fst } b \star \textbf{snd } b$$
$$= \qquad (\text{left})$$
$$\lambda^{\bullet}a.\, \textbf{let } x \Leftarrow [\langle\!\langle L \ggg (\textbf{fst } a) \rangle\!\rangle] \textbf{ in } x \star \textbf{snd } a$$
$$= \qquad (\text{def } \langle\!\langle - \rangle\!\rangle)$$
$$\lambda^{\bullet}a.\, \textbf{let } x \Leftarrow [\lambda^{\bullet}d.\, \textbf{let } y \Leftarrow \langle\!\langle L \rangle\!\rangle \bullet d \textbf{ in } \textbf{fst } a \bullet y] \textbf{ in } x \star \textbf{snd } a$$
$$= \qquad (\eta^{\text{app}}, \beta^{\rightsquigarrow})$$
$$\lambda^{\bullet}a.\, \textbf{let } x \Leftarrow [\lambda^{\bullet}d.\, \textbf{let } y \Leftarrow \langle\!\langle L \rangle\!\rangle \bullet d \textbf{ in } \textbf{fst } a \star y] \textbf{ in } x \star \textbf{snd } a$$
$$= \qquad (\text{left}, \beta^{\text{app}})$$
$$\lambda^{\bullet}a.\, \textbf{let } y \Leftarrow \langle\!\langle L \rangle\!\rangle \bullet \textbf{snd } a \textbf{ in } \textbf{fst } a \star y$$
$$= \qquad (\text{left})$$
$$\lambda^{\bullet}a.\, \textbf{let } y \Leftarrow \langle\!\langle L \rangle\!\rangle \bullet \textbf{snd } a \textbf{ in}$$
$$\textbf{let } b \Leftarrow [\langle \textbf{fst } a, y\rangle] \textbf{ in}$$
$$\textbf{fst } b \star \textbf{snd } b$$
$$= \qquad (\text{assoc})$$
$$\lambda^{\bullet}a.\, \textbf{let } b \Leftarrow (\textbf{let } y \Leftarrow \langle\!\langle L \rangle\!\rangle \bullet \textbf{snd } a \textbf{ in } [\langle \textbf{fst } a, y\rangle]) \textbf{ in } \textbf{fst } b \star \textbf{snd } b$$
$$= \qquad (\beta^{\rightsquigarrow}, \beta^{\rightsquigarrow})$$
$$\lambda^{\bullet}a.\, \textbf{let } b \Leftarrow (\lambda^{\bullet}z.\, \textbf{let } y \Leftarrow \langle\!\langle L \rangle\!\rangle \bullet \textbf{snd } z \textbf{ in } [\langle \textbf{fst } z, y\rangle]) \bullet a \textbf{ in}$$
$$(\lambda^{\bullet}w.\, \textbf{fst } w \star \textbf{snd } w) \bullet b$$
$$= \qquad (\text{def } \langle\!\langle - \rangle\!\rangle)$$
$$\langle\!\langle \text{second } L \ggg \text{app} \rangle\!\rangle$$

3. $(\leadsto_{H3})$

$$\langle\!\langle\mathtt{first}\,(\mathtt{arr}\,(\ggg L))\ggg \mathtt{app}\rangle\!\rangle$$
$=$ (def $\langle\!\langle-\rangle\!\rangle$, $\beta^{\rightarrow}$)
$$\lambda^{\bullet}\mathtt{x}.\,\mathbf{let}\,\mathtt{y}\Leftarrow(\lambda^{\bullet}\mathtt{z}.\,\mathbf{let}\,\mathtt{w}\Leftarrow(\lambda^{\bullet}\mathtt{z}.\,[\lambda^{\bullet}\mathtt{t}.\,\mathbf{let}\,\mathtt{s}\Leftarrow\mathtt{z}\bullet\mathtt{t}\,\mathbf{in}\,\langle\!\langle\mathrm{L}\rangle\!\rangle\bullet\mathtt{s}])\bullet\mathbf{fst}\,\mathtt{z}\,\mathbf{in}$$
$$[\langle\mathtt{w},\mathbf{snd}\,\mathtt{z}\rangle])\bullet\mathtt{x}\,\mathbf{in}$$
$$(\lambda^{\bullet}\mathtt{p}.\,\mathbf{fst}\,\mathtt{p}\star\mathbf{snd}\,\mathtt{p})\bullet\mathtt{y}$$
$=$ $(\beta^{\leadsto})$
$$\lambda^{\bullet}\mathtt{x}.\,\mathbf{let}\,\mathtt{y}\Leftarrow(\mathbf{let}\,\mathtt{w}\Leftarrow[\lambda^{\bullet}\mathtt{t}.\,\mathbf{let}\,\mathtt{s}\Leftarrow\mathbf{fst}\,\mathtt{x}\bullet\mathtt{t}\,\mathbf{in}\,\langle\!\langle\mathrm{L}\rangle\!\rangle\bullet\mathtt{s}]\,\mathbf{in}\,[\langle\mathtt{w},\mathbf{snd}\,\mathtt{x}\rangle])\,\mathbf{in}$$
$$\mathbf{fst}\,\mathtt{y}\star\mathbf{snd}\,\mathtt{y}$$
$=$ (assoc)
$$\lambda^{\bullet}\mathtt{x}.\,\mathbf{let}\,\mathtt{w}\Leftarrow[\lambda^{\bullet}\mathtt{t}.\,\mathbf{let}\,\mathtt{s}\Leftarrow\mathbf{fst}\,\mathtt{x}\bullet\mathtt{t}\,\mathbf{in}\,\langle\!\langle\mathrm{L}\rangle\!\rangle\bullet\mathtt{s}]\,\mathbf{in}$$
$$\mathbf{let}\,\mathtt{y}\Leftarrow[\langle\mathtt{w},\mathbf{snd}\,\mathtt{x}\rangle]\,\mathbf{in}$$
$$\mathbf{fst}\,\mathtt{y}\star\mathbf{snd}\,\mathtt{y}$$
$=$ (left, $\beta_1^{\times}$, $\beta_2^{\times}$)
$$\lambda^{\bullet}\mathtt{x}.\,(\lambda^{\bullet}\mathtt{t}.\,\mathbf{let}\,\mathtt{s}\Leftarrow\mathbf{fst}\,\mathtt{x}\bullet\mathtt{t}\,\mathbf{in}\,\langle\!\langle\mathrm{L}\rangle\!\rangle\bullet\mathtt{s})\star\mathbf{snd}\,\mathtt{x}$$
$=$ $(\beta^{\mathtt{app}})$
$$\lambda^{\bullet}\mathtt{x}.\,\mathbf{let}\,\mathtt{s}\Leftarrow\mathbf{fst}\,\mathtt{x}\bullet\mathbf{snd}\,\mathtt{x}\,\mathbf{in}\,\langle\!\langle\mathrm{L}\rangle\!\rangle\bullet\mathtt{s}$$
$=$ $(\beta^{\leadsto})$
$$\lambda^{\bullet}\mathtt{x}.\,\mathbf{let}\,\mathtt{s}\Leftarrow(\lambda^{\bullet}\mathtt{p}.\,\mathbf{fst}\,\mathtt{p}\star\mathbf{snd}\,\mathtt{p})\bullet\mathtt{x}\,\mathbf{in}\,\langle\!\langle\mathrm{L}\rangle\!\rangle\bullet\mathtt{s}$$
$=$ (def $\langle\!\langle-\rangle\!\rangle$)
$$\langle\!\langle\mathtt{app}\ggg \mathrm{L}\rangle\!\rangle$$

A.6.4   Translating $\mathcal{H}$ to $\mathcal{C}_{\mathbf{app}}$ and back

For each term M of $\mathcal{H}$ we must show $\langle\!\langle\,[\![M]\!]\,\rangle\!\rangle = M$ and for each command P of $\mathcal{H}$ we must show $\langle\!\langle\,[\![P]\!]_\Delta\,\rangle\!\rangle\bullet\Delta = P$.

1. Case $L\star M$

$$\langle\!\langle[\![L\star M]\!]_\Delta\rangle\!\rangle$$
$=$ (def $[\![-]\!]_\Delta$)
$$\langle\!\langle\mathtt{arr}\,(\lambda\Delta.\,\langle[\![L]\!],[\![M]\!]\rangle)\ggg \mathtt{app}\rangle\!\rangle$$
$=$ (def $\langle\!\langle-\rangle\!\rangle$, $\beta^{\rightarrow}$)
$$\lambda^{\bullet}\Delta.\,\mathbf{let}\,\mathtt{y}\Leftarrow(\lambda^{\bullet}\Delta.\,[\langle\langle\!\langle[\![L]\!]\rangle\!\rangle,\langle\!\langle[\![M]\!]\rangle\!\rangle\rangle])\bullet\Delta\,\mathbf{in}\,(\lambda^{\bullet}\mathtt{p}.\,\mathbf{fst}\,\mathtt{p}\star\mathbf{snd}\,\mathtt{p})\bullet\mathtt{y}$$
$=$ $(\beta^{\leadsto})$
$$\lambda^{\bullet}\Delta.\,\mathbf{let}\,\mathtt{y}\Leftarrow[\langle\langle\!\langle[\![L]\!]\rangle\!\rangle,\langle\!\langle[\![M]\!]\rangle\!\rangle\rangle]\,\mathbf{in}\,\mathbf{fst}\,\mathtt{y}\star\mathbf{snd}\,\mathtt{y}$$
$=$ (left, $\beta_1^{\times}$, $\beta_2^{\times}$)
$$\lambda^{\bullet}\Delta.\,\langle\!\langle[\![L]\!]\rangle\!\rangle\star\langle\!\langle[\![M]\!]\rangle\!\rangle$$
$=$ (induction hypothesis)
$$\lambda^{\bullet}\Delta.\,L\star M$$

A.6.5 Translating $\mathcal{C}_{\mathbf{app}}$ to $\mathcal{H}$ and back

For each term M of $\mathcal{C}_{app}$ we must show $[\![\,\langle\!\langle M\rangle\!\rangle\,]\!] = M$.

1. Case app

$$
\begin{array}{cl}
& [\![\langle\!\langle \mathtt{app}\rangle\!\rangle]\!] \\
= & \quad (\text{def } \langle\!\langle -\rangle\!\rangle) \\
& [\![\lambda^{\bullet} \mathtt{p}.\,\mathtt{fst}\ \mathtt{p} \star \mathtt{snd}\ \mathtt{p}]\!] \\
= & \quad (\text{def } [\![-]\!]) \\
& \mathtt{arr}\ (\lambda \mathtt{p}.\,\langle \mathtt{fst}\ \mathtt{p}, \mathtt{snd}\ \mathtt{p}\rangle) \ggg \mathtt{app} \\
= & \quad (\eta^{\times}) \\
& \mathtt{arr}\ \mathtt{id} \ggg \mathtt{app} \\
= & \quad (\rightsquigarrow_{1}) \\
& \mathtt{app}
\end{array}
$$

A.7   Equational equivalence between $\mathcal{M}$ and $\mathcal{H}$

This section gives a proof of Proposition 22 (page 78).

A.7.1   The laws of $\mathcal{M}$ follow from the laws of $\mathcal{H}$

For each law $M = N$ of $\mathcal{M}$ we must show $[\![M]\!] = [\![N]\!]$.

1. (M_1)

$$[\![\texttt{return N} \ggeq \texttt{M}]\!]$$
$$= \qquad (\text{def } [\![-]\!])$$
$$\lambda^\bullet \langle\rangle.\ \textbf{let } x \Leftarrow (\lambda^\bullet \langle\rangle.\ [\![N]\!]) \star \langle\rangle \textbf{ in } ([\![M]\!]\ x) \star \langle\rangle$$
$$= \qquad (\beta^{\texttt{app}})$$
$$\lambda^\bullet \langle\rangle.\ \textbf{let } x \Leftarrow [\![N]\!] \textbf{ in } ([\![M]\!]\ x) \star \langle\rangle$$
$$= \qquad (\text{left})$$
$$\lambda^\bullet \langle\rangle.\ ([\![M]\!]\ [\![N]\!]) \star \langle\rangle$$
$$= \qquad (\eta^{\texttt{app}})$$
$$[\![M]\!]\ [\![N]\!]$$
$$= \qquad (\text{def } [\![-]\!])$$
$$[\![M\ N]\!]$$

2. (M_2)

$$[\![M \ggeq \texttt{return}]\!]$$
$$= \qquad (\text{def } [\![-]\!],\ \beta^\rightarrow)$$
$$\lambda^\bullet \langle\rangle.\ \textbf{let } x \Leftarrow [\![M]\!] \star \langle\rangle \textbf{ in } (\lambda^\bullet \langle\rangle.\ [x]) \star \langle\rangle$$
$$= \qquad (\beta^{\texttt{app}})$$
$$\lambda^\bullet \langle\rangle.\ \textbf{let } x \Leftarrow [\![M]\!] \star \langle\rangle \textbf{ in } [x]$$
$$= \qquad (\text{right})$$
$$\lambda^\bullet \langle\rangle.\ [\![M]\!] \star \langle\rangle$$
$$= \qquad (\eta^{\texttt{app}})$$
$$[\![M]\!]$$

3. (M_3)

$$[\![(\texttt{L} \ggeq \texttt{M}) \ggeq \texttt{N}]\!]$$
$$= \qquad (\text{def } [\![-]\!])$$
$$\lambda^\bullet \langle\rangle.\ \textbf{let } x \Leftarrow (\lambda^\bullet \langle\rangle.\ \textbf{let } y \Leftarrow [\![L]\!] \star \langle\rangle \textbf{ in } ([\![M]\!]\ y) \star \langle\rangle) \star \langle\rangle \textbf{ in } ([\![N]\!]\ x) \star \langle\rangle$$
$$= \qquad (\beta^{\texttt{app}})$$
$$\lambda^\bullet \langle\rangle.\ \textbf{let } x \Leftarrow \textbf{let } y \Leftarrow [\![L]\!] \star \langle\rangle \textbf{ in } ([\![M]\!]\ y) \star \langle\rangle \textbf{ in } ([\![N]\!]\ x) \star \langle\rangle$$
$$= \qquad (\text{assoc})$$
$$\lambda^\bullet \langle\rangle.\ \textbf{let } y \Leftarrow [\![L]\!] \star \langle\rangle \textbf{ in let } x \Leftarrow ([\![M]\!]\ y) \star \langle\rangle \textbf{ in } ([\![N]\!]\ x) \star \langle\rangle$$

(continued on next page)

(continued from previous page)
$\lambda^{\bullet}\langle\rangle.\ \mathbf{let}\ y \Leftarrow [\![L]\!] \star \langle\rangle\ \mathbf{in}\ \mathbf{let}\ x \Leftarrow ([\![M]\!]\ y) \star \langle\rangle\ \mathbf{in}\ ([\![N]\!]\ x) \star \langle\rangle$
$=\qquad (\beta^{\mathtt{app}})$
$\lambda^{\bullet}\langle\rangle.\ \mathbf{let}\ y \Leftarrow [\![L]\!] \star \langle\rangle\ \mathbf{in}\ (\lambda^{\bullet}\langle\rangle.\ \mathbf{let}\ x \Leftarrow ([\![M]\!]\ y) \star \langle\rangle\ \mathbf{in}\ ([\![N]\!]\ x) \star \langle\rangle) \star \langle\rangle$
$=\qquad (\text{def}\ [\![-]\!], \beta^{\rightarrow})$
$[\![L \ggg (\lambda s.\ M\ s \ggg N)]\!]$

A.7.2 The laws of $\mathcal{H}$ follow from the laws of $\mathcal{M}$

For each law $M = N$ or $P = Q$ of $\mathcal{H}$ we must show $\langle\!|M|\!\rangle = \langle\!|N|\!\rangle$ or $\langle\!|P|\!\rangle = \langle\!|Q|\!\rangle$.

1. $(\beta^{\rightsquigarrow})$

$\langle\!|(\lambda^{\bullet}x.\ Q) \bullet M|\!\rangle$
$=\qquad (\text{def}\ \langle\!|-|\!\rangle)$
$(\lambda x.\ \langle\!|Q|\!\rangle)\ \langle\!|M|\!\rangle$
$=\qquad (\beta^{\rightarrow})$
$\langle\!|Q|\!\rangle[x := \langle\!|M|\!\rangle]$
$=\qquad (\text{Lemma 21})$
$\langle\!|Q[x := M]|\!\rangle$

2. $(\eta^{\rightsquigarrow})$

$\langle\!|\lambda^{\bullet}x.\ L \bullet x|\!\rangle$
$=\qquad (\text{def}\ \langle\!|-|\!\rangle)$
$\lambda x.\ \langle\!|L|\!\rangle\ x$
$=\qquad (\eta^{\rightarrow})$
$\langle\!|L|\!\rangle$

3. (left)

$\langle\!|\mathbf{let}\ x \Leftarrow [M]\ \mathbf{in}\ Q|\!\rangle$
$=\qquad (\text{def}\ \langle\!|-|\!\rangle)$
$\mathtt{return}\ \langle\!|M|\!\rangle \ggg (\lambda x.\ \langle\!|Q|\!\rangle)$
$=\qquad (M_1, \beta^{\rightarrow})$
$\langle\!|Q|\!\rangle[x := \langle\!|M|\!\rangle]$
$=\qquad (\text{Lemma 21})$
$\langle\!|Q[x := M]|\!\rangle$

4. (right)

$$\langle\!\langle \textbf{let } x \Leftarrow P \textbf{ in } [x] \rangle\!\rangle$$
$$= \qquad (\text{def } \langle\!\langle - \rangle\!\rangle)$$
$$\langle\!\langle P \rangle\!\rangle \ggg (\lambda x.\, \texttt{return } x)$$
$$= \qquad (\eta^{\rightarrow})$$
$$\langle\!\langle P \rangle\!\rangle \ggg \texttt{return}$$
$$= \qquad (M_2)$$
$$\langle\!\langle P \rangle\!\rangle$$

5. (assoc)

$$\langle\!\langle \textbf{let } y \Leftarrow (\textbf{let } x \Leftarrow P \textbf{ in } Q) \textbf{ in } R \rangle\!\rangle$$
$$= \qquad (\text{def } \langle\!\langle - \rangle\!\rangle)$$
$$(\langle\!\langle P \rangle\!\rangle \ggg (\lambda x.\, \langle\!\langle Q \rangle\!\rangle)) \ggg (\lambda y.\, \langle\!\langle R \rangle\!\rangle)$$
$$= \qquad (M_3)$$
$$\langle\!\langle P \rangle\!\rangle \ggg (\lambda x.\, (\lambda x.\, \langle\!\langle Q \rangle\!\rangle)\, x \ggg (\lambda y.\, \langle\!\langle R \rangle\!\rangle))$$
$$= \qquad (\beta^{\rightarrow})$$
$$\langle\!\langle P \rangle\!\rangle \ggg (\lambda x.\, \langle\!\langle Q \rangle\!\rangle \ggg (\lambda y.\, \langle\!\langle R \rangle\!\rangle))$$
$$= \qquad (\text{def } \langle\!\langle - \rangle\!\rangle)$$
$$\langle\!\langle \textbf{let } x \Leftarrow P \textbf{ in } (\textbf{let } y \Leftarrow Q \textbf{ in } R) \rangle\!\rangle$$

6. $(\beta^{\texttt{app}})$

$$\langle\!\langle (\lambda^{\bullet} x.\, Q) \star M \rangle\!\rangle$$
$$= \qquad (\text{def } \langle\!\langle - \rangle\!\rangle)$$
$$(\lambda^{\bullet} x.\, \langle\!\langle Q \rangle\!\rangle)\, \langle\!\langle M \rangle\!\rangle$$
$$= \qquad (\beta^{\rightarrow})$$
$$\langle\!\langle Q \rangle\!\rangle [x := \langle\!\langle M \rangle\!\rangle]$$
$$= \qquad (\text{Lemma } 21)$$
$$\langle\!\langle Q[x := M] \rangle\!\rangle$$

A.7.3　Translating $\mathcal{M}$ to $\mathcal{H}$ and back

For each term M of $\mathcal{M}$ we must show $\langle\!\langle \llbracket M \rrbracket \rangle\!\rangle = f_A(M)$. The proof is by induction on the derivation of M.

1. Case `return`

$$f^{-1}_{A \rightarrow MA} \langle\!\langle \llbracket \texttt{return} \rrbracket \rangle\!\rangle$$
$$= \qquad (\text{def } \llbracket - \rrbracket)$$
$$f^{-1}_{A \rightarrow MA} \langle\!\langle \lambda x.\, \lambda^{\bullet}\langle\rangle.\, [x] \rangle\!\rangle$$

(continued on next page)

(continued from previous page)

$\mathtt{f}^{-1}_{A \to MA} \; \langle\!\langle \lambda\mathtt{x}.\, \lambda^{\bullet}\langle\rangle.\, [\mathtt{x}] \rangle\!\rangle$

$=\qquad (\mathrm{def}\; \langle\!\langle - \rangle\!\rangle)$

$\mathtt{f}^{-1}_{A \to MA} \; (\lambda\mathtt{x}.\, \lambda\langle\rangle.\, \mathtt{return}\, \mathtt{x})$

$=\qquad (\mathrm{def}\; \mathtt{f}^{-1}_{A \to MA})$

$(\lambda\mathtt{h}.\, \lambda\mathtt{z}.\, (\lambda\mathtt{h}.\, \mathtt{h}\, \langle\rangle \ggg (\lambda\mathtt{v}.\, \mathtt{return}\, (\mathtt{f}^{-1}_A\, (\mathtt{v})))) \, (\mathtt{h}\, (\mathtt{f}_A\, (\mathtt{z})))) \, (\lambda\mathtt{x}.\, \lambda\langle\rangle.\, \mathtt{return}\, \mathtt{x})$

$=\qquad (\beta^{\to}, \eta^{\to})$

$\lambda\mathtt{z}.\, \mathtt{return}\, (\mathtt{f}_A\, (\mathtt{z})) \ggg (\lambda\mathtt{v}.\, \mathtt{return}\, (\mathtt{f}^{-1}_A\, (\mathtt{v})))$

$=\qquad (\mathrm{M}_1, \beta^{\to})$

$\lambda\mathtt{z}.\, \mathtt{return}\, (\mathtt{f}^{-1}_A\, (\mathtt{f}_A\, (\mathtt{z})))$

$=\qquad (\mathtt{f}^{-1}_A\, (\mathtt{f}_A\, \mathtt{x}) = \mathtt{x})$

$\lambda\mathtt{z}.\, \mathtt{return}\, \mathtt{z}$

$=\qquad (\eta^{\to})$

$\mathtt{return}$

2. Case ($\ggg$)

$\mathtt{f}^{-1}_{MA \to (A \to MB) \to MB} \; \langle\!\langle [\![(\ggg)]\!] \rangle\!\rangle$

$=\qquad (\mathrm{def}\; [\![-]\!])$

$\mathtt{f}^{-1}_{MA \to (A \to MB) \to MB} \; \langle\!\langle \lambda\mathtt{a}.\, \lambda\mathtt{h}.\, \lambda^{\bullet}\langle\rangle.\, \mathbf{let}\; \mathtt{x} \Leftarrow \mathtt{a} \bullet \langle\rangle \;\mathbf{in}\; (\mathtt{h}\, \mathtt{x}) \bullet \langle\rangle \rangle\!\rangle$

$=\qquad (\mathrm{def}\; \langle\!\langle - \rangle\!\rangle)$

$\mathtt{f}^{-1}_{MA \to (A \to MB) \to MB} \; (\lambda\mathtt{a}.\, \lambda\mathtt{h}.\, \lambda\langle\rangle.\, \mathtt{a}\, \langle\rangle \ggg (\lambda\mathtt{x}.\, \mathtt{h}\, \mathtt{x}\, \langle\rangle))$

$=\qquad (\mathrm{def}\; \mathtt{f}^{-1}_{MA \to (A \to MB) \to MB})$

$(\lambda\mathtt{h}.\, (\lambda\mathtt{g}.\, ((\lambda\mathtt{l}.\, \mathtt{l}\, \langle\rangle \ggg (\mathtt{return} \cdot \mathtt{f}^{-1}_B)) \cdot \mathtt{g}$
$\qquad\qquad \cdot (\lambda\mathtt{j}.\, (\lambda\mathtt{a}.\, \lambda\langle\rangle.\, \mathtt{a} \ggg (\mathtt{return} \cdot \mathtt{f}_B)) \cdot \mathtt{j} \cdot \mathtt{f}^{-1}_A)))$
$\qquad \cdot \mathtt{h} \cdot (\lambda\mathtt{b}.\, \lambda\langle\rangle.\, \mathtt{b} \ggg (\mathtt{return} \cdot \mathtt{f}_A)))$
$\quad (\lambda\mathtt{a}.\, \lambda\mathtt{h}.\, \lambda\langle\rangle.\, \mathtt{a}\, \langle\rangle \ggg (\lambda\mathtt{x}.\, \mathtt{h}\, \mathtt{x}\, \langle\rangle))$

$=\qquad (\beta^{\to})$

$\lambda\mathtt{b}.\, \lambda\mathtt{j}.\, (((\mathtt{b} \ggg (\mathtt{return} \cdot \mathtt{f}_A)) \ggg (\lambda\mathtt{x}.\, (\mathtt{j}\, (\mathtt{f}^{-1}_A\, (\mathtt{x})) \ggg (\mathtt{return} \cdot \mathtt{f}_B))))$
$\qquad\qquad \ggg (\mathtt{return} \cdot \mathtt{f}^{-1}_B))$

$=\qquad (\mathrm{M}_3, \beta^{\to})$

$\lambda\mathtt{b}.\, \lambda\mathtt{j}.\, (\mathtt{b} \ggg (\lambda\mathtt{x}.\, \mathtt{return}\, (\mathtt{f}_A\, (\mathtt{x})) \ggg (\lambda\mathtt{x}.\, \mathtt{j}\, (\mathtt{f}^{-1}_A\, (\mathtt{x}))))$
$\qquad\qquad \ggg (\lambda\mathtt{x}.\, \mathtt{return}\, (\mathtt{f}_B\, \mathtt{x}) \ggg (\mathtt{return} \cdot \mathtt{f}^{-1}_B))$

$=\qquad (\mathrm{M}_1)$

$\lambda\mathtt{b}.\, \lambda\mathtt{j}.\, (\mathtt{b} \ggg (\lambda\mathtt{x}.\, \mathtt{return}\, (\mathtt{f}_A\, (\mathtt{x})) \ggg (\lambda\mathtt{x}.\, \mathtt{j}\, (\mathtt{f}^{-1}_A\, (\mathtt{x}))))$
$\qquad\qquad \ggg (\lambda\mathtt{x}.\, \mathtt{return}\, (\mathtt{f}^{-1}_B\, (\mathtt{f}_B\, (\mathtt{x}))))$

$=\qquad (\mathtt{f}^{-1}_B\, (\mathtt{f}_B\, \mathtt{x}) = \mathtt{x}, \eta^{\to})$

$\lambda\mathtt{b}.\, \lambda\mathtt{j}.\, (\mathtt{b} \ggg (\lambda\mathtt{x}.\, \mathtt{return}\, (\mathtt{f}_A\, (\mathtt{x})) \ggg (\lambda\mathtt{x}.\, \mathtt{j}\, (\mathtt{f}^{-1}_A\, (\mathtt{x})))) \ggg \mathtt{return}$

$=\qquad (\mathrm{M}_2)$

$\lambda\mathtt{b}.\, \lambda\mathtt{j}.\, (\mathtt{b} \ggg (\lambda\mathtt{x}.\, \mathtt{return}\, (\mathtt{f}_A\, (\mathtt{x})) \ggg (\lambda\mathtt{x}.\, \mathtt{j}\, (\mathtt{f}^{-1}_A\, (\mathtt{x})))))$

(continued from previous page)
$\lambda b. \lambda j. (b \ggg (\lambda x. \texttt{return} (f_A (x)) \ggg (\lambda x. j (f_A^{-1} (x))))))$
$=$ (M_1)
$\lambda b. \lambda j. (b \ggg (\lambda x. j (f_A^{-1} (f_A (x))))))$
$=$ $(f_A^{-1} (f_A x) = x, \eta^{\rightarrow})$
$\lambda b. \lambda j. b \ggg j$
$=$ $(\eta^{\rightarrow})$
          $(\ggg)$

A.7.4   Translating $\mathcal{H}$ to $\mathcal{M}$ and back

For each term M of $\mathcal{H}$ we must show $[\![\langle\!| M |\!\rangle]\!] = g_A(M)$. For each command P of $\mathcal{H}$ we must show $[\![\langle\!| P |\!\rangle]\!] = \lambda^\bullet \langle\rangle . . \texttt{let } z \Leftarrow P \texttt{ in } [g_A(z)]$. The proof is by mutual induction on the derivations of M and P.

1. Case $\lambda^\bullet x. Q$

     $[\![\langle\!| \lambda^\bullet x. Q |\!\rangle]\!]$
$=$ (def $\langle\!| - |\!\rangle$)
     $\lambda x. [\![\langle\!| Q |\!\rangle]\!]$
$=$ (induction hypothesis)
     $\lambda x. \lambda^\bullet \langle\rangle . \texttt{let } z \Leftarrow Q[x := g_A^{-1} (x)] \texttt{ in } [g_B (z)]$
$=$ $(\beta^{\leadsto})$
     $\lambda x. \lambda^\bullet \langle\rangle . \texttt{let } z \Leftarrow (\lambda^\bullet x. Q) \bullet (g_A^{-1} (x)) \texttt{ in } [g_B (z)]$
$=$ (def $g_{A \leadsto B}, \beta^{\rightarrow}$)
     $g_{A \leadsto B} (\lambda^\bullet x. Q)$

1. Case $L \bullet M$

     $[\![\langle\!| L \bullet M |\!\rangle]\!]$
$=$ (def $\langle\!| - |\!\rangle$)
     $[\![\langle\!| L |\!\rangle \langle\!| M |\!\rangle]\!]$
$=$ (def $[\![-]\!]$)
     $[\![\langle\!| L |\!\rangle]\!] [\![\langle\!| M |\!\rangle]\!]$
$=$ (induction hypothesis)
     $(g_{A \leadsto B} L) (g_A M)$
$=$ (def $g_{A \leadsto B}$)
     $((\lambda a. \lambda x. \lambda^\bullet \langle\rangle . \texttt{let } z \Leftarrow a \bullet g_A^{-1} x \texttt{ in } [g_B z]) L) (g_A M)$
$=$ $(\beta^{\rightarrow})$
     $\lambda^\bullet \langle\rangle . \texttt{let } z \Leftarrow L \bullet g_A^{-1} (g_A M) \texttt{ in } [g_B z]$
$=$ $(g_A^{-1} (g_A x) = x)$
     $\lambda^\bullet \langle\rangle . \texttt{let } z \Leftarrow L \bullet M \texttt{ in } [g_B z]$

2. Case $[M]$

$$\llbracket \langle\!\langle [M] \rangle\!\rangle \rrbracket$$
$=$ (def $\langle\!\langle - \rangle\!\rangle$)
$$\llbracket \mathtt{return}\ \langle\!\langle M \rangle\!\rangle \rrbracket$$
$=$ (def $\llbracket - \rrbracket$)
$$\lambda^\bullet \langle\rangle.\ \llbracket [\langle\!\langle M \rangle\!\rangle] \rrbracket$$
$=$ (induction hypothesis)
$$\lambda^\bullet \langle\rangle.\ [g_A\ M]$$
$=$ (left)
$$\lambda^\bullet \langle\rangle.\ \mathtt{let}\ z \Leftarrow [M]\ \mathtt{in}\ [g_A\ z]$$

3. Case $\mathtt{let}\ x \Leftarrow P\ \mathtt{in}\ Q$

$$\llbracket \langle\!\langle \mathtt{let}\ x \Leftarrow P\ \mathtt{in}\ Q \rangle\!\rangle \rrbracket$$
$=$ (def $\langle\!\langle - \rangle\!\rangle$)
$$\llbracket \langle\!\langle P \rangle\!\rangle \ggg (\lambda x.\ \langle\!\langle Q \rangle\!\rangle) \rrbracket$$
$=$ (def $\llbracket - \rrbracket$)
$$\lambda^\bullet \langle\rangle.\ \mathtt{let}\ x \Leftarrow \llbracket \langle\!\langle P \rangle\!\rangle \rrbracket \bullet \langle\rangle\ \mathtt{in}\ ((\lambda x.\ \llbracket \langle\!\langle Q \rangle\!\rangle \rrbracket)\ x) \bullet \langle\rangle$$
$=$ ($\beta^\rightarrow$)
$$\lambda^\bullet \langle\rangle.\ \mathtt{let}\ x \Leftarrow \llbracket \langle\!\langle P \rangle\!\rangle \rrbracket \bullet \langle\rangle\ \mathtt{in}\ \llbracket \langle\!\langle Q \rangle\!\rangle \rrbracket \bullet \langle\rangle$$
$=$ (induction hypothesis)
$$\lambda^\bullet \langle\rangle.\ \mathtt{let}\ x \Leftarrow (\lambda^\bullet \langle\rangle.\ \mathtt{let}\ z \Leftarrow P\ \mathtt{in}\ [g_A\ (z)]) \bullet \langle\rangle\ \mathtt{in}$$
$$(\lambda^\bullet \langle\rangle.\ \mathtt{let}\ z \Leftarrow Q[x := g_A^{-1}]\ \mathtt{in}\ [g_B\ (z)]) \bullet \langle\rangle$$
$=$ ($\beta^{\rightsquigarrow}$)
$$\lambda^\bullet \langle\rangle.\ \mathtt{let}\ x \Leftarrow \mathtt{let}\ z \Leftarrow P\ \mathtt{in}\ [g_A\ z]\ \mathtt{in}\ \mathtt{let}\ y \Leftarrow Q[x := g_A^{-1}]\ \mathtt{in}\ [g_B\ y]$$
$=$ (assoc)
$$\lambda^\bullet \langle\rangle.\ \mathtt{let}\ z \Leftarrow P\ \mathtt{in}\ \mathtt{let}\ x \Leftarrow [g_A\ (z)]\ \mathtt{in}\ \mathtt{let}\ y \Leftarrow Q[x := g_A^{-1}]\ \mathtt{in}\ [g_B\ y]$$
$=$ (left, $g_A^{-1}\ (g_A\ (x)) = x$)
$$\lambda^\bullet \langle\rangle.\ \mathtt{let}\ x \Leftarrow P\ \mathtt{in}\ \mathtt{let}\ y \Leftarrow Q\ \mathtt{in}\ [g_B\ (y)]$$
$=$ (assoc)
$$\lambda^\bullet \langle\rangle.\ \mathtt{let}\ y \Leftarrow \mathtt{let}\ x \Leftarrow P\ \mathtt{in}\ Q\ \mathtt{in}\ [g_B\ (y)]$$

4. Case $L \star M$

$$\llbracket \langle\!\langle L \star M \rangle\!\rangle \rrbracket$$
$=$ (def $\langle\!\langle - \rangle\!\rangle$, def $\llbracket - \rrbracket$)
$$\llbracket \langle\!\langle L \rangle\!\rangle \rrbracket \ \llbracket \langle\!\langle M \rangle\!\rangle \rrbracket$$
$=$ (induction hypothesis)
$$(g_{A \rightsquigarrow B} \ L) \ (g_A \ (M))$$
$=$ (def $g_{A \rightsquigarrow B}$, $\beta^{\rightarrow}$)
$$\lambda^{\bullet}\langle\rangle. \ \mathbf{let} \ z \Leftarrow L \bullet g_A^{-1} \ (g_A \ (M)) \ \mathbf{in} \ [g_B \ (z)]$$
$=$ ($g_A^{-1} \ (g_A \ M) = M$)
$$\lambda^{\bullet}\langle\rangle. \ \mathbf{let} \ z \Leftarrow L \bullet M \ \mathbf{in} \ [g_B \ (z)]$$
$=$ (η^{app})
$$\lambda^{\bullet}\langle\rangle. \ \mathbf{let} \ z \Leftarrow (\lambda^{\bullet}x. \ L \star x) \bullet M \ \mathbf{in} \ [g_B \ (z)]$$
$=$ ($\beta^{\rightsquigarrow}$)
$$\lambda^{\bullet}\langle\rangle. \ \mathbf{let} \ z \Leftarrow L \star M \ \mathbf{in} \ [g_B \ (z)]$$

A.8 Redundancy of $(\leadsto_{H2})$

We give below a direct proof that the $(\leadsto_{H2})$ law of $\mathcal{C}_{app}$ is redundant.

$\quad$ first $(\text{arr } (L \ggg)) \ggg$ app
$=\qquad(\leadsto_1)$
$\quad$ first $(\text{arr } (\lambda x. L \ggg \text{arr id} \ggg x)) \ggg$ app
$=\qquad(\leadsto_4, \text{snd} \cdot \text{dup} = \text{id})$
$\quad$ first $(\text{arr } (\lambda x. L \ggg \text{arr dup} \ggg \text{arr snd} \ggg x)) \ggg$ app
$=\qquad(\leadsto_{H1})$
$\quad$ first $(\text{arr } (\lambda x. L \ggg \text{arr dup} \ggg \text{first } (\text{arr } (\lambda b. \text{arr } (\lambda a. \langle b, a \rangle)))) \ggg$ app
$\qquad\qquad \ggg \text{arr snd} \ggg x)) \ggg$ app
$=\qquad(\leadsto_{H3})$
$\quad$ first $(\text{arr } (\lambda x. L \ggg \text{arr dup} \ggg \text{first } (\text{arr } (\lambda b. \text{arr } (\lambda a. \langle b, a \rangle))))$
$\qquad\qquad \ggg \text{first } (\text{arr } (\ggg(\text{arr snd} \ggg x))) \ggg \text{app})) \ggg$ app
$=\qquad(\leadsto_5)$
$\quad$ first $(\text{arr } (\lambda x. L \ggg \text{arr dup} \ggg \text{arr } ((\lambda b. \text{arr } (\lambda a. \langle b, a \rangle)) \times \text{id})$
$\qquad\qquad \ggg \text{arr } ((\ggg(\text{arr snd} \ggg x)) \times \text{id}) \ggg \text{app})) \ggg$ app
$=\qquad(\leadsto_4)$
$\quad$ first $(\text{arr } (\lambda x. L \ggg \text{arr } (\lambda w. \langle \text{arr id} \ggg x, w \rangle) \ggg \text{app})) \ggg$ app
$=\qquad(\leadsto_1)$
$\quad$ first $(\text{arr } (\lambda x. L \ggg \text{arr } (\lambda w. \langle x, w \rangle) \ggg \text{app})) \ggg$ app
$=\qquad(\leadsto_4)$
$\quad$ first $(\text{arr } (\lambda x. \text{arr dup} \ggg \text{arr fst} \ggg L \ggg \text{arr } (\lambda w. \langle x, w \rangle) \ggg \text{app})) \ggg$ app
$=\qquad(\leadsto_8)$
$\quad$ first $(\text{arr } (\lambda x. \text{arr dup} \ggg \text{first } L \ggg \text{arr fst} \ggg \text{arr } (\lambda w. \langle x, w \rangle) \ggg \text{app})) \ggg$ app
$=\qquad(\leadsto_4)$
$\quad$ first $(\text{arr } (\lambda x. \text{arr dup} \ggg \text{first } L \ggg \text{arr } (\text{id} \times \text{const } x) \ggg \text{arr swap} \ggg \text{app}))$
$\qquad \ggg$ app
$=\qquad(\leadsto_7)$
$\quad$ first $(\text{arr } (\lambda x. \text{arr dup} \ggg \text{arr } (\text{id} \times \text{const } x) \ggg \text{first } L \ggg \text{arr swap} \ggg \text{app}))$
$\qquad \ggg$ app
$=\qquad(\leadsto_4)$
$\quad$ first $(\text{arr } (\lambda x. \text{arr } (\lambda w. \langle x, w \rangle) \ggg \text{arr swap} \ggg \text{first } L \ggg \text{arr swap} \ggg \text{app})) \ggg$ app
$=\qquad(\text{def second})$
$\quad$ first $(\text{arr } (\lambda x. \text{arr } (\lambda w. \langle x, w \rangle) \ggg \text{second } L \ggg \text{app})) \ggg$ app
$=\qquad(\leadsto_6)$
$\quad$ arr $(\lambda z. \langle \text{arr } (\lambda w. \langle \text{fst } z, w \rangle) \ggg \text{second } L \ggg \text{app}, \text{snd } z \rangle) \ggg$ app
$=\qquad(\leadsto_4)$
$\quad$ arr $(\lambda z. \langle z, \text{snd } z \rangle) \ggg \text{arr } (((\lambda b. \text{arr } (\lambda a. \langle b, a \rangle)))) \times \text{id})$
$\qquad \ggg \text{arr } ((\ggg(\text{arr } (\text{fst} \times \text{id}) \ggg \text{second } L \ggg \text{app})) \times \text{id}) \ggg$ app
$=\qquad(\leadsto_5)$
$\quad$ arr $(\lambda z. \langle z, \text{snd } z \rangle) \ggg \text{first } (\text{arr } ((\lambda b. \text{arr } (\lambda a. \langle b, a \rangle)))))$
$\qquad \ggg \text{first } (\text{arr } (\ggg(\text{arr } (\text{fst} \times \text{id}) \ggg \text{second } L \ggg \text{app}))) \ggg$ app

(continued on next page)

(continued from previous page)

$\quad$ arr $(\lambda z. \langle z, snd\, z\rangle) \ggg$ first $(arr\, ((\lambda b.\, arr\, (\lambda a.\, \langle b, a\rangle)))))$
$\quad\quad \ggg$ first $(arr\, (\ggg(arr\, (fst \times id) \ggg second\, L \ggg app))) \ggg$ app
$=\quad\quad (\leadsto_4, second\, L \ggg arr\, (M \times id) = arr\, (M \times id) \ggg second\, L)$
$\quad$ arr $(\lambda z. \langle z, snd\, z\rangle) \ggg$ first $(arr\, ((\lambda b.\, arr\, (\lambda a.\, \langle b, a\rangle)))))$
$\quad\quad \ggg$ first $(arr\, (\ggg(arr\, dup \ggg arr\, (id \times snd) \ggg second\, L$
$\quad\quad\quad\quad\quad \ggg arr\, (fst \cdot fst \times id) \ggg app))) \ggg$ app
$=\quad\quad (second\, (arr\, L) = arr\, (id \times L))$
$\quad$ arr $(\lambda z. \langle z, snd\, z\rangle) \ggg$ first $(arr\, ((\lambda b.\, arr\, (\lambda a.\, \langle b, a\rangle)))))$
$\quad\quad \ggg$ first $(arr\, (\ggg(arr\, dup \ggg second\, (arr\, snd) \ggg second\, L$
$\quad\quad\quad\quad\quad \ggg arr\, (fst \cdot fst \times id) \ggg app)))$
$\quad\quad \ggg$ app
$=\quad\quad (def\, \&\!\&\!\&, \leadsto_6)$
$\quad$ arr $(\lambda z. \langle z, snd\, z\rangle) \ggg$ first $(arr\, ((\lambda b.\, arr\, (\lambda a.\, \langle b, a\rangle)))))$
$\quad\quad \ggg$ first $(arr\, (\ggg((arr\, id\, \&\!\&\!\&\, (arr\, snd \ggg L)) \ggg arr\, (fst \cdot fst \times id) \ggg app)))$
$\quad\quad \ggg$ app
$=\quad\quad (\leadsto_{H3})$
$\quad$ arr $(\lambda z. \langle z, snd\, z\rangle) \ggg$ first $(arr\, ((\lambda b.\, arr\, (\lambda a.\, \langle b, a\rangle))))) \ggg$ app
$\quad\quad \ggg ((arr\, id\, \&\!\&\!\&\, (arr\, snd \ggg L)) \ggg arr\, (fst \cdot fst \times id) \ggg app)$
$=\quad\quad (\leadsto_{H1})$
$\quad$ arr $(\lambda z. \langle z, snd\, z\rangle) \ggg arr\, id \ggg (arr\, id\, \&\!\&\!\&\, (arr\, snd \ggg L))$
$\quad\quad \ggg arr\, (fst \cdot fst \times id) \ggg$ app
$=\quad\quad (\leadsto_1)$
$\quad$ arr $(\lambda z. \langle z, snd\, z\rangle) \ggg (arr\, id\, \&\!\&\!\&\, (arr\, snd \ggg L)) \ggg arr\, (fst \cdot fst \times id) \ggg$ app
$=\quad\quad (def\, \&\!\&\!\&)$
$\quad$ arr $(\lambda z. \langle z, snd\, z\rangle) \ggg arr\, dup \ggg second\, (arr\, snd \ggg L)$
$\quad\quad \ggg arr\, (fst \cdot fst \times id) \ggg$ app
$=\quad\quad (second\, (L \ggg M) = second\, L \ggg second\, M)$
$\quad$ arr $(\lambda z. \langle z, snd\, z\rangle) \ggg arr\, dup \ggg second\, (arr\, snd) \ggg second\, L$
$\quad\quad \ggg arr\, (fst \cdot fst \times id) \ggg$ app
$=\quad\quad (second\, (arr\, L) = arr\, (id \times L))$
$\quad$ arr $dup \ggg arr\, (id \times snd) \ggg arr\, dup \ggg arr\, (id \times snd) \ggg second\, L$
$\quad\quad \ggg arr\, (fst \cdot fst \times id) \ggg$ app
$=\quad\quad (\leadsto_4)$
$\quad$ arr $(\lambda x. \langle\langle x, snd\, x\rangle, snd\, x\rangle) \ggg second\, L \ggg arr\, (fst \cdot fst \times id) \ggg$ app
$=\quad\quad (second\, L \ggg arr\, (M \times id) = arr\, (M \times id) \ggg second\, L)$
$\quad$ arr $(\lambda x. \langle\langle x, snd\, x\rangle, snd\, x\rangle) \ggg arr\, ((fst \cdot fst) \times id) \ggg second\, L \ggg$ app
$=\quad\quad (\leadsto_4)$
$\quad$ arr $id \ggg second\, L \ggg$ app
$=\quad\quad (\leadsto_1)$
$\quad$ second $L \ggg$ app

Appendix B

Formlets extras

B.1 The page construct

This appendix describes the *page* construct, which is used in combination with input-validating formlets (Section 3.6.2). The function of the page construct is to associate the context in which each formlet appears with the value representing the formlet. If the input submitted by the user is determined to be invalid, the formlet then re-renders itself together in the context, together with error messages indicating the problems wit the input.

The page construct really represents composable page *fragments*, just as formlets represent fragments of forms. As with formlets, we provide a new syntactic form based on XML literals with embedded expressions. Consider the date example of Figure 3.1. The final step in that example is to associate the formlet `travel_formlet` with the continuation `display_itinerary`:

```
handle travel_formlet display_itinerary
```

The page construct subsumes the functionality of `handle`: it associates a formlet with its continuation in the context of some larger XML value. For example, the following code associates `travel_formlet` with `display_itinerary` in the context of the enclosing `<body>` element.

```
page
  <body
      <h1>Itinerary</h1>
      {travel_formlet ⟹ display_itinerary}
  </body>
```

As we have said, the page construct is designed for use with input-validating formlets, where there are two possible outcomes when a form is submitted. Suppose that `travel_formlet`

273

```
  module type MONOID =
  sig
    type t
    val zero : t
    val (⊗) : t → t → t
  end
```

Figure B.1: The monoid interface

```
  module type PAGE =
  sig
    include MONOID
    val xml : xml  → t
    val tag : tag  → attributes  → t  → t
    val form : α VFormlets.t  → (α → t)  → t
    val render : t → xml
  end
```

Figure B.2: The page interface

has some validation procedures attached (using the `extract` function of 3.6.2). If these succeed when run on the submitted data, producing a result, then the result is passed to `display_itinerary` to compute the next page. If instead validation fails, then `travel_formlet` will be re-rendered (along with any error messages returned by validation), and sent back to the user in its enclosing context (i.e. the `<body>` element) to be edited and re-submitted.

As with formlets, the new syntactic form is defined by a straightforward desugaring into combinators that build new page values from existing page values. Here is the result of desugaring our example above:

```
Page.tag "body"
  (Page.tag "h1" (Page.xml (xml_text "Itinerary"))
  ⊗ₚ Page.form travel_formlet display_itinerary)
```

In the remainder of this appendix we define the page interface (Section B.1.1) and the associated syntactic sugar (Section B.1.2).

B.1.1 The page interface

Figure B.1 gives the `MONOID` interface; it is the OCaml analogue of the `Monoid` type class of Section 2.2.4.

Figure B.2 gives the `PAGE` interface, which extends `MONOID`. There are five constructors for pages. The first two are the monoid operations, `zero` and ⊗, which respectively construct an

```
type xml_context = xml list → xml
type recform = { recform : recform list → xml }
type form_builder = xml_context → int → recform

let tie : recform list → xml list
   = fun zs → List.map (fun z → z.recform zs) zs
```

Figure B.3: Auxiliary types used to implement pages

empty page and concatenate two pages. The third and fourth, `xml` and `tag`, lift an XML value to a page and enclose a page in an XML element. The fifth, `form`, constructs a page from a formlet and its handler function. The `render` function converts a page to XML for sending to the client.

Figure B.3 defines a number of types used in the definition of the page construct. The `xml_context` type represents multi-holed `xml` contexts. Here we represent contexts simply as functions from lists of XML values to XML values. (Appendix B.2 describes a more careful implementation in which the type of a context reflects the number of holes it contains, and in which the type does not contain inappropriate values such as non-terminating or effectful functions.) Each page contains zero or more forms; the `recform` type represents one of these. We represent each form as a function from the other forms in the group to its rendering as XML. We associate all the forms in the group with each form in this way so that when validation of a form fails all the forms in the group can be re-rendered. The `form_builder` type is a function that builds a form representation from two arguments: a multi-holed XML context and the position of the form in the group. We will store the formlets associated with a page as `form_builder` values until the page is rendered; it is only at rendering time that the full page context and the list of all the forms are available. The `tie` function renders a list of form representations by passing the whole group to each `recform` value.

Figure B.4 gives an implementation of the PAGE interface. The type `t` is a record of the three components used to represent pages: a count, `size`, of the number of embedded formlets, a `size`-holed XML context, `ctxt`, and a list of form builders, `frms`.

The monoid operations act straightforwardly on components of the record, exploiting the underlying monoidal operations at those types. At the context component, monoid multiplication constructs a new context by concatenating the contexts of the arguments, creating a new context with `l.size` +`r.size` holes, which plugs in the first `l.size` of these into the left context, and the remainder into the right.

The `tag` function lifts the tagging operation to contexts.

```
module Page : PAGE =
struct
 type t = { size : int;
            ctxt : xml_context ;
            frms : form_builder list }

  let zero = { size = 0; ctxt = const []; frms = [] }
  let (⊗) l r =
   { size = l.size + r.size;
     ctxt = (fun xs → l.ctxt (take l.size xs)
                    @ r.ctxt (drop l.size xs));
     frms = l.frms @ r.frms; }

  let tag t ats pg = { pg with ctxt = xml_tag t ats ∘ pg.ctxt }
  let xml x = { zero with ctxt = const x }

  let render pg = pg.ctxt (tie (mapi (fun m → m pg.ctxt) pg.frms))

  let mk_form contents k =
    let pickled = pickle_cont (render ∘ k) in
     <form>
       <input type="hidden" name="_k" value="{pickled}"/>
       {contents}
     </form>
  let form f h =
    let x, c = VFormlet.run f in
    let rec loop k i zs env =
     match c env with
     | Right v → h v
     | Left x → xml (k (tie (subst_at zs i
                               {recform = mk_form x ∘ loop k i})))
    in {size = 1; ctxt = List.hd;
         frms = [fun k i → {recform = mk_form x ∘ loop k i}]}
end
```

Figure B.4: The page implementation

The xml function takes an XML value and creates a page with no embedded formlets.

The render function renders a page as XML. It first builds form representations from each of the form builders in the page record by passing the page context and the position of the form builder in the list to each. (The mapi function is similar to the standard map, but passes an extra argument to the mapped function representing the position of the element in the list.) It then uses tie to produce XML renderings of the form representations, and plugs these into the context to produce the page.

Expressions

$$e ::= \cdots \mid \textbf{page } p \quad \text{(page fragment)}$$

Page quasiquotes

$$n ::= s \mid \{\, e \,\} \mid \{\, f \implies e \,\} \mid \{| \; g \; |\} \; \mid <t \; ats>n_1 \; \ldots \; n_k</t> \quad \text{node}$$
$$p ::= <t \; ats>n_1 \; \ldots \; n_k</t> \mid <\#>n_1 \; \ldots \; n_k</\#> \qquad \text{quasiquote}$$

Meta variables

$$f \quad \text{formlet} \quad g \quad \text{page-type expression} \quad s \quad \text{string}$$

Figure B.5: Page syntax.

The `mk_form` function performs part of the work that was previously the responsibility of `handle` (Section 3.5): it embeds a continuation and an rendered formlet within an HTML form. The continuation passed as argument has type $\text{env} \rightarrow \text{t}$; it is converted to a function of type $\text{env} \rightarrow \text{xml}$ by composition with the `render` function.

The `form` function associates an input-validating formlet with a handler, to construct a page. After extracting the rendering and collector of the formlet using the `run` function, it constructs a page consisting of a single formlet. Note that `List.hd` function serves as a context consisting of a single hole. The form builder is the interesting part of the page record: it is the composition of the `mk_form` function already described, and an auxiliary function, `loop`. The `loop` function encodes the validation loop for a particular formlet. It takes four arguments: a multi-holed XML context, the position of the formlet in the context, the list of all the formlets in the page, and the CGI environment. The first three of these arguments will be supplied when the page is rendered; the environment will be supplied when the form is submitted. The outcome of running `loop` depends on the outcome of the collector function, `c`. If `c` succeeds, returning a value, then the handler function of the formlet is applied to the value to construct the next page. If `c` fails, returning a re-rendering of the formlet annotated with error messages, then the re-rendered formlet replace the previous version of itself in the list and the page in which the formlet appeared is reconstructed to be sent back to the user.

B.1.2 The page syntax

We now define the page syntax and its desugaring.

Figure B.5 defines the page syntax. Besides the standard forms for XML quasiquotes given in Figure 3.12, there are two new splicing forms. The form $\{\, f \implies e \,\}$ associates the formlet

$$[\![\mathbf{page}\ \mathrm{p}]\!] = \mathrm{p}^{\mathrm{p}}$$

$$\mathrm{s}^{\mathrm{p}} = \mathtt{Page.xml}\ (\mathtt{xml\_text}\ s)$$

$$\{\,e\,\}^{\mathrm{p}} = \mathtt{Page.xml}\ [\![e]\!]$$

$$\{f \Longrightarrow e\}^{\mathrm{p}} = \mathtt{Page.form}\ [\![f]\!]\ [\![e]\!]$$

$$\{\!|\ g\ |\!\}^{\mathrm{p}} = [\![g]\!]$$

$$<t\ as>n_1 \ldots n_{\mathrm{k}}</t>^{\mathrm{p}} = \mathtt{Page.tag}\ \ t\ \ as\ \ (<\#>n_1 \ldots n_{\mathrm{k}}</\#>)^{\mathrm{p}}$$

$$<\#>n_1 \ldots n_{\mathrm{k}}</\#>^{\mathrm{p}} = n_1^{\mathrm{p}} \otimes_{\mathrm{p}} \ldots \otimes_{\mathrm{p}} n_{\mathrm{k}}^{\mathrm{p}}$$

Figure B.6: Desugaring pages.
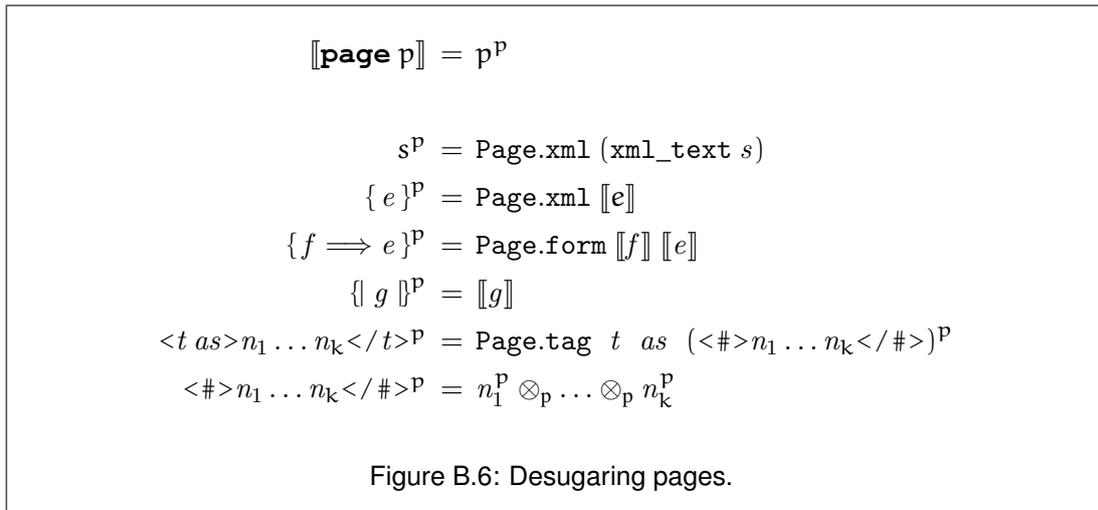
denoted by f with the handler denoted by e. The form $\{\!|\ g\ |\!\}$ indicates the splicing of the page value denoted by g.

Figure B.6 gives the desugaring algorithm for page literals. Each syntactic form translates directly into an application of a combinator in the Page module.

```
module type CONTEXT =
sig
  type item
   and t = item list

  val text : string → t
  val tag  : tag → attrs → t → t
  val hole : t

  val plug : t → t list → t

  val     xml : xml → t
  val to_xml : t → xml
end
```

Figure B.7: The untyped multi-holed context abstract type

B.2   Multi-holed contexts

We give here an implementation of formlets based on multi-holed contexts as sketched in Section 3.6.3 of Chapter 3.

There is only a small conceptual difference between the presentation of formlets in Sections 3.1–3.4 of Chapter 3 and the presentation based on multi-holed contexts described in Section 3.6.3 and this appendix. The first presentation is based on accumulating a single XML value. There are two operations: the first, `xml`, injects XML values into the XML accumulation idiom; the second, `tag`, encloses the XML accumulated within a new XML node. The second presentation is based on accumulating a sequence of XML values. There is one operation, `plug`, which inserts these values into a multi-holed XML context, creating a single XML value. The `plug` function is operationally straightforward, but some ingenuity is required to assign it a type that prevents mismatches between the number of holes in the context and the number of values to be inserted.

Figure B.7 gives the interface to the abstract type of XML contexts. It is "untyped" in the sense that the number of holes in a context is not reflected in its type. The interface is isomorphic to the interface to the XML abstract type (Figure 3.3) except for the addition of a constructor, `hole`, which constructs contexts consisting of a single hole. There is an operation, `plug`, which inserts a list of context into a context in depth-first order, creating a new context. If the number of holes in the first argument does not match the length of the list in the second then `plug` exits with an exception. There are two further operations: a total function, `xml`, which converts an `xml` value to a zero-holed context, and a partial function `to_xml`, which converts

```
 module Context : CONTEXT =
 struct
  type item = Text of string | Tag of tag × attrs × t | Hole
   and t = item list

  let tag t ats n = [Tag (t, ats, n)]
  let hole = [Hole]
  let rec to_xml xs = List.map to_xml_item xs
  and to_xml_item = function
     | Text s              → Xml.Text s
     | Tag (s, attrs, k) → Xml.Tag (s, attrs, to_xml k)
     | Hole                → failwith "Unexpected hole"
  let rec xml xs = List.concat (List.map xml_item xs)
   and xml_item = function
      | Xml.Text s → text s
      | Xml.Tag (t, ats, x) → tag t ats (xml x)
  let plug k xs =
    let rec plug = function
       | ([], xs)  → [], xs
       | (z :: zs, xs)  →
           let (k, xs) = plug_item (z, xs) in
           let (k', xs) = plug (zs, xs) in
             k @ k', xs
    and plug_item = function
        | Text s, xs → [Text s], xs
        | Tag (s, ats, k), xs → let (k, xs) = plug (k, xs)
                                    in [Tag (s, ats, k)], xs
        | Hole, [] → failwith "Too many holes"
        | Hole, x::xs → x, xs in
    match plug (k, xs) with
       | k, []  → k
       | _       → failwith "Too few holes"
 end
```

Figure B.8: An implementation of Figure B.7

a zero-holed context to XML, exiting with an exception if the argument contains holes. The types in the interfaces that follow will ensure that `plug` and `to_xml` are never passed input that causes them to raise exceptions.

Figure B.8 gives an implementation of the CONTEXT interface. We assume an implementation of the XML abstract type (Figure 3.3) in which the `item` type is implemented by a datatype with `Text` and `Tag` constructors. The implementation of CONTEXT is otherwise straightforward, so we refrain from further comment.

```
type z
type +α s
```

Figure B.9: Type-level peano numbers

```
module type NCONTEXT =
sig
 type (+μ,+ν) context
 val empty : (μ,μ) context
 val xml : xml → (μ,μ) context
 val tag : tag → attrs → (μ,ν) context → (μ,ν) context
 val hole : (μ,μ s) context
 val (++) : (κ,ν) context → (μ,κ) context → (μ,ν) context

 type (+μ,+ν,+σ,+τ) contexts
 val nil : (μ,μ,σ,σ) contexts
 val cons : (κ,ν) context → (μ,κ,σ,τ) contexts
      → (μ,ν,σ,τ s) contexts
 val append : (κ,ν,ρ,τ) contexts → (μ,κ,σ,ρ) contexts
      → (μ,ν,σ,τ) contexts
 val plug : (σ,τ) context → (μ,ν,σ,τ) contexts
      → (μ,ν) context

 val to_xml_list : (μ,μ,σ,τ) contexts → xml list
end
```

Figure B.10: The typed multi-holed context interface

Figure B.9 defines type constructors z and s (for *zero* and *successor*) which will form the basis of the type-level arithmetic used to keep track of the number of holes in what follows. Both z and s are uninhabited; we will use them as phantom types.

Figure B.10 gives the typed interface to the abstract type of XML contexts. Unlike the Context interface, here all operations are total. There are two types: context, which represents a multi-holed context, and contexts, which represents a sequence of multi-holed contexts.

A value of type (μ,ν) context, where μ and ν are instantiated to Peano numbers, represents an XML context with $μ - ν$ holes. There are five constructors for contexts: empty and xml construct contexts with $μ - μ$ (i.e. zero) holes; tag encloses an existing context in an XML element, maintaining the number of holes; hole constructs a context with a single hole (μ s − μ), and ++ concatenates contexts, adding the number of holes. It is this last operation which really exploits the type-level representation of numbers: we use the identity

```
   module NContext : NCONTEXT =
   struct
     include Context
     type (μ,ν) context = Context.t
     let empty = []
     let (++) = (@)
     type (μ,ν,σ,τ) contexts = Context.t list
     let nil = []
     let cons x xs = x :: xs
     let append = (@)
     let to_xml_list = List.map Context.to_xml
   end
```

Figure B.11: The typed multi-holed context implementation

$(\mu - \nu) = (\kappa - \nu) + (\mu - \kappa)$ to "compute" the number of holes in the output from the number of holes in the arguments without performing any actual arithmetic at the type level.

A value of type (μ, ν, σ, τ) `contexts`, where all the type parameters are instantiated to Peano numbers, represents a sequence of XML contexts $\sigma - \tau$ elements long, with $\mu - \nu$ holes in total. There are three constructors for context sequences. The first, `nil`, constructs an empty sequence: there are $\sigma - \sigma$ contexts, and a total of $\mu - \mu$ holes. The second, `cons`, adds a new context to an existing sequence: given a context with $\kappa - \nu$ holes and a sequence of length $\sigma - \tau$ with a total of $\mu - \kappa$ holes, `cons` constructs a new sequence with one additional context (making $\sigma\ s - \tau = \sigma - \tau + 1$ in all) and with $(\kappa - \nu) + (\mu - \kappa) = \mu - \nu$ holes. The third, `append`, concatenates two sequences, adding the numbers of holes and the numbers of contexts in the arguments. The last, `plug`, takes a context with $\sigma - \tau$ holes and a list of $\sigma - \tau$ contexts with a total of $\mu - \nu$ holes, and plugs the second into the first, producing a new context with $\mu - \nu$ holes. Finally, there is a safe upcast operation, `to_xml_list`, which discards types in a sequence of zero-holed contexts to produce an XML value.

Figure B.11 gives an implementation of Figure B.10. The implementation is particularly simple: the constructors for contexts are simply the constructors for values of type `Context.t`, and the constructors for context sequences are simply the standard list constructors. The `to_xml_list` operation is implemented directly in terms of the `to_xml` operation of `Context`. Note that although the implementations are the same as those of the `Context` module, the types in the interface prevent `to_xml_list` being applied to a context with holes, and prevent hole-number mismatches in the arguments of `plug`, eliminating the undesired partiality.

Figure B.12 gives an adaptation of the XML accumulation idiom of Figure 3.9 to the new setting. Computations in the idiom of Figure 3.9 accumulate a single XML value, but here

```
module NXmlWriter :
sig
 include PIDIOM
 val plug : (μ,ν) context → (μ,ν,α) t → (κ,κ s,α) t
 val run : (z,z s,α) t → xml × α
end =
struct
  open NContext
  type (μ,ν,α) t = (z,z,μ,ν) contexts × α
  let pure v = (nil, v)
  let (⊛) (xs, f) (ys, a) = (append xs ys, f a)
  let plug k (xs, v) = (cons (plug k xs), nil, v)
  let run (xs, v) = (List.hd (to_xml_list xs), v)
end
```

Figure B.12: The multi-holed XML accumulation parameterised idiom

```
module type NFORMLET =
sig
  include PIDIOM
  val plug  : (μ,ν) context → (μ,ν,α) t → (κ,κ s,α) t
  val input : (μ,μ s,string) t
  val run   : (z,z s,α) t → xml × (env → α)
end
```

Figure B.13: The multi-holed formlet interface

we accumulate a sequence of XML values and provide an operation for plugging them into a context. The `NXmlWriter` module implements the parameterised idiom interface `PIDIOM` (Figure 3.23). A computation of type $(μ,ν,α)$ `NXmlWriter.t` accumulates $μ − ν$ XML values and returns a result of type $α$. The `plug` operation takes a context with $μ − ν$ holes and a computation that accumulates the same number of values and returns a computation that accumulates a single value, formed by plugging the values into the context. As with the standard XML accumulation idiom, the `pure` and ⊛ operations correspond to the unit and multiplication of the underlying XML monoid, which is in this case the free monoid on the set of XML values. The `run` function takes a computation that accumulates a single XML value and returns the accumulated XML value and the result of the computation.

Figure B.13 gives the interface to the parameterised formlet idiom based on multi-holed contexts. As with `NXmlWriter`, a computation of type $(μ,ν,α)$ `NFORMLET.t` accumulates $μ − ν$ XML values and returns a result of type $α$. The `plug` operation is simply the `plug` of `NXmlWriter` lifted to formlets. The `input` operation corresponds to the `input` operation

```
  module ComposePI (F : PIDIOM) (G : IDIOM)
    : PIDIOM with type (+μ,+ν,+α) t = (μ,ν,α G.t) F.t


  module ComposeIP (F : IDIOM) (G : PIDIOM)
    : PIDIOM with type (+μ,+ν,+α) t = (μ,ν,α) G.t F.t
```

Figure B.14: Composition of an idiom and a parameterised idiom

```
  module NFormlet : NFormlet =
  struct
    include ComposeIP (NameGen)
             (ComposePI (NXmlWriter) (Environment))
    module N = NameGen
    module A = NXmlWriter
    module E = Environment
    let plug k f = N.pure (A.plug k) ⊛ₙ f
    let input =
     N.pure
       (fun name →
         A.plug (NContext.tag
                  "input" [("name", name)] NContext.empty)
           (A.pure (E.lookup name)))
      ⊛ₙ N.next_name
    let run v = let xml, c = A.run (N.run v) in xml, E.run c
  end
```

Figure B.15: The multi-holed formlet implementation

of the `Formlet` interface (Figure 3.5). The `run` operation extracts the XML and the collector from a formlet computation that accumulates a single XML value.

Figure B.14 gives the interface to pre- and post-composition of a parameterised idiom with an idiom. We omit the implementations, which are precisely the same as the implementation of standard idiom composition (Figure 3.10).

Figure B.15 gives the implementation of the parameterised formlet idiom of Figure B.13. As with standard formlets (Figure 3.11), formlets are formed by composing the idioms for name generation, XML accumulation and reading from an environment; here XML accumulation is implemented by a parameterised idiom, so the result is a parameterised idiom.

Figure B.16 gives the desugaring algorithm for formlets based on multi-holed contexts. Given a formlet literal **formlet** q **yields** e, the operations q^k, $q^{\#}$ and q denote respectively the XML context for q (i.e. the XML literal, but with holes in place of formlet bindings),

$$
\begin{aligned}
[\![\textbf{formlet } q \textbf{ yields } e]\!] \;&=\; \texttt{NFormlet.plug } q^{\mathrm{k}} \\
&\qquad (\texttt{NFormlet.pure} \,(\textbf{fun } q_1^{\dagger} \ldots q_{\mathrm{n}}^{\dagger} \rightarrow [\![e]\!]) \\
&\qquad \circledast \; q_1^{\#} \; \circledast \; \ldots \; \circledast \; q_{\mathrm{n}}^{\#})
\end{aligned}
$$

$$
\begin{aligned}
s^{\mathrm{k}} \;&=\; \texttt{NContext.xml (xml\_text } s) \\
\{e\}^{\mathrm{k}} \;&=\; \texttt{NContext.xml } [\![e]\!] \\
\{f \Rightarrow p\}^{\mathrm{k}} \;&=\; \texttt{NContext.hole} \\
(\texttt{<}t \; ats\texttt{>}n_1 \; \ldots \; n_{\mathrm{k}}\texttt{</}t\texttt{>})^{\mathrm{k}} \;&=\; \texttt{NContext.tag } t \; ats \; (\texttt{<\#>}n_1 \; \ldots \; n_{\mathrm{k}}\texttt{</\#>})^{\mathrm{k}} \\
(\texttt{<\#>}n_1 \; \ldots \; n_{\mathrm{k}}\texttt{</\#>})^{\mathrm{k}} \;&=\; n_1^{\mathrm{k}} \,\texttt{++}\, \ldots \,\texttt{++}\, n_{\mathrm{k}}^{\mathrm{k}}
\end{aligned}
$$

$$
\begin{aligned}
s^{\#} \;&=\; () \\
\{e\}^{\#} \;&=\; () \\
\{f \Rightarrow p\}^{\#} \;&=\; f \\
(\texttt{<}t \; ats\texttt{>}n_1 \; \ldots \; n_{\mathrm{k}}\texttt{</}t\texttt{>})^{\#} \;&=\; n_1^{\#}, \ldots, n_{\mathrm{k}}^{\#} \\
(\texttt{<\#>}n_1 \; \ldots \; n_{\mathrm{k}}\texttt{</\#>})^{\#} \;&=\; n_1^{\#}, \ldots, n_{\mathrm{k}}^{\#}
\end{aligned}
$$

$$
\begin{aligned}
s^{\dagger} \;&=\; () \\
\{e\}^{\dagger} \;&=\; () \\
\{f \Rightarrow p\}^{\dagger} \;&=\; p \\
(\texttt{<}t \; ats\texttt{>}n_1 \; \ldots \; n_{\mathrm{k}}\texttt{</}t\texttt{>})^{\dagger} \;&=\; n_1^{\dagger}, \ldots, n_{\mathrm{k}}^{\dagger} \\
(\texttt{<\#>}n_1 \; \ldots \; n_{\mathrm{k}}\texttt{</\#>})^{\dagger} \;&=\; n_1^{\dagger}, \ldots, n_{\mathrm{k}}^{\dagger}
\end{aligned}
$$

Figure B.16: Formlet desugaring based on multi-holed contexts

the sequence of formlets bound within q and the sequence of patterns to which those formlets are bound. We abuse the notation a little in the definition of $[\![-]\!]$, writing $q_i^{\dagger}$ to refer to the i-th element of the sequence $q^{\dagger}$, and similarly for $q^{\#}$. The result of the desugaring for each formlet literal is a single application of the plug function to an idiom computation in normal form (Section 2.2.5.3). We refer the reader again to Figure 3.22 of Section 3.6.3 for an example of the output.

As we have said, it is possible to combine the multi-holed context approach to producing XML with the static XHTML validation of Section 3.6.1. We refer the reader to Lindley (2008) for the details.

Bibliography

RFC 793, 1981. Transmission Control Protocol
  http://tools.ietf.org/html/rfc793.

Lift website, March 2008.
  http://liftweb.net/.

Martín Abadi. Access control in a core calculus of dependency. *Electronic Notes in Theoretical Computer Science*, 172:5–31, 2007. ISSN 1571-0661.

Andrew W. Appel. *Compiling with Continuations*. Cambridge University Press, New York, NY, USA, 2007.

Robert Atkey. What is a categorical model of arrows? In Venanzio Capretta and Conor McBride, editors, *Mathematically Structured Functional Programming*, Reykjavik University, Iceland, July 2008.

Robert Atkey. Parameterised notions of computation. *Journal of Functional Programming*, 19 (3 & 4):355–376, July 2009.

Vincent Balat. Ocsigen: typing web interaction with Objective Caml. In *ACM SIGPLAN Workshop on ML*, pages 84–94, Portland, Oregon, September 2006.

Ioannis G. Baltopoulos and Andrew D. Gordon. Secure compilation of a multi-tier web language. In *ACM SIGPLAN Workshop on Types in Language Design and Implementation*, pages 27–38, Savannah, GA, USA, January 2009. ISBN 978-1-60558-420-1.

John Billings, Peter Sewell, Mark Shinwell, and Rok Strniša. Type-safe distributed programming for OCaml. In *ACM SIGPLAN Workshop on ML*, Portland, Oregon, September 2006.

Richard Bird. A program to solve sudoku. *Journal of Functional Programming*, 16(6):671–679, July 2006.

Claus Brabrand, Anders Møller, and Michael I. Schwartzbach. Static validation of dynamically generated HTML. In *ACM SIGPLAN-SIGSOFT Workshop on Program Analysis For Software Tools and Engineering*, pages 38–45, Snowbird, Utah, USA, June 2001.

Björn Bringert and Aarne Ranta. A pattern for almost compositional functions. In *ACM SIGPLAN International Conference on Functional Programming*, Portland, Oregon, USA, September 2006.

Manuel M. T. Chakravarty, Gabriele Keller, and Simon Peyton Jones. Associated type synonyms. In *ACM SIGPLAN International Conference on Functional Programming*, pages 241–253, Tallinn, Estonia, September 2005a.

Manuel M. T. Chakravarty, Gabriele Keller, Simon Peyton Jones, and Simon Marlow. Associated types with class. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 1–13, Long Beach, California, USA, January 2005b.

James Cheney and Ralf Hinze. A lightweight implementation of generics and dynamics. In *ACM SIGPLAN Haskell Workshop*, Pittsburgh, PA, USA, October 2002.

Noam Chomsky. Three models for the description of language. *IRE Transactions on Information Theory*, (2):113–124, 1956.

Aske Simon Christensen, Anders Møller, and Michael I. Schwartzbach. Extending Java for high-level web service construction. *ACM Transactions on Programming Languages and Systems*, 25(6):814–875, 2003.

Albert Cohen and C. Herrmann. Towards a high-productivity and high-performance marshaling library for compound data. In *MetaOCaml Workshop*, Tallinn, Estonia, September 2005.

Ezra Cooper, Sam Lindley, Philip Wadler, and Jeremy Yallop. Links: web programming without tiers. In *Formal Methods for Components and Objects*, pages 266–296, Amsterdam, The Netherlands, 2006.

Ezra Cooper, Sam Lindley, Philip Wadler, and Jeremy Yallop. The essence of form abstraction. In *Asian Symposium on Programming Languages and Systems*, Bangalore, India, December 2008a.

Ezra Cooper, Sam Lindley, Philip Wadler, and Jeremy Yallop. An idiom's guide to formlets. Technical Report EDI-INF-RR-1263, School of Informatics, University of Edinburgh, 2008b.

Brian J. Corcoran, Nikhil Swamy, and Michael Hicks. Combining provenance and security policies in a web-based document management system. In *Workshop on Principles of Provenance*, Edinburgh, Scotland, UK, November 2007.

Brian J. Corcoran, Nikhil Swamy, and Michael Hicks. Cross-tier, label-based security enforcement for web applications. In *ACM SIGMOD International Conference on Management of Data*, Providence, Rhode Island, USA, June 2009.

Antony Courtney and Conal Elliott. Genuinely functional user interfaces. In *ACM SIGPLAN Haskell Workshop*, Firenze, Italy, September 2001.

Daniel de Rauglaudre. IoXML, 2002.
http://cristal.inria.fr/~ddr/IoXML/.

Derek Dreyer, Robert Harper, Manuel Chakravarti, and Gabriele Keller. Modular type classes. In *ACM SIGPLAN - SIGACT Symposium on Principles of Programming Languages*, Nice, France, January 2007.

Chris Eidhof. Formlets in Haskell, 2008.
http://blog.tupil.com/formlets-in-haskell/.

Conal Elliott. Simply efficient functional reactivity. Technical Report 2008-01, LambdaPix, April 2008. URL http://conal.net/papers/simply-reactive/.

Martin Elsman. Polymorphic equality — no tags required. In *Workshop on Types in Compilation*, volume 1473 of *Lecture Notes in Computer Science*, Kyoto, Japan, 1998.

Martin Elsman. Type-specialized serialization with sharing. In *Trends in Functional Programming*, Tallinn, Estonia, September 2005.

Martin Elsman and Ken Friis Larsen. Typing XHTML web applications in ML. In *Practical Aspects of Declarative Languages*, Dallas, TX, USA, June 2004.

Mary Fernandez, Kathleen Fisher, J. Nathan Foster, Michael Greenberg, and Yitzhak Mandelbaum. A generic programming toolkit for pads/ml: First-class upgrades for third-party developers. In *Practical Aspects of Declarative Languages*, San Francisco, CA, January 2008.

Andrzej Filinski. *Controlling Effects*. PhD thesis, School of Computer Science, Carnegie Mellon University, May 1996.

Robert Bruce Findler and Matthias Felleisen. Contracts for higher-order functions. In *ACM SIGPLAN International Conference on Functional Programming*, pages 48–59, Pittsburgh, Pennsylvania, USA, September 2002.

Jacques Garrigue. Programming with polymorphic variants. In *ACM SIGPLAN Workshop on ML*, Baltimore, Maryland, September 1998.

Jacques Garrigue. Relaxing the value restriction. In *Asian Symposium on Programming Languages and Systems*, pages 31–45, Shanghai, China, 2002.

Jeremy Gibbons. Design patterns as higher-order datatype-generic programs. In *ACM SIGPLAN Workshop on Generic Programming*, Portland, Oregon, September 2006.

Jeremy Gibbons and Bruno C. d. S. Oliveira. The essence of the iterator pattern. *Journal of Functional Programming*, 19(3&4):377–402, July 2009.

Jean-Yves Girard, Paul Taylor, and Yves Lafont. *Proofs and types*. Cambridge University Press, New York, NY, USA, 1989.

Paul Graham. Method for client-server communications through a minimal interface. US Patent 6,205,469 B1, May 1997.

Paul Graunke, Robert Bruce Findler, Shriram Krishnamurthi, and Matthias Felleisen. Automatically restructuring programs for the web. In *IEEE International Conference on Automated Software Engineering*, L'Aquila, Italy, September 2001a.

Paul T. Graunke, Shriram Krishnamurthi, Steve Van Der Hoeven, and Matthias Felleisen. Programming the web with high-level programming languages. In *European Symposium on Programming*, pages 122–136, Genova, Italy, April 2001b.

Cordelia V. Hall, Kevin Hammond, Simon L. Peyton Jones, and Philip L. Wadler. Type classes in haskell. *ACM Transactions on Programming Languages and Systems*, 18(2):109–138, 1996. ISSN 0164-0925. doi: http://doi.acm.org/10.1145/227699.227700.

David Alan Halls. *Applying Mobile Code to Distributed Systems*. PhD thesis, University of Cambridge, June 1997.

David Heinemeier Hansson. Ruby on Rails, March 2008. http://www.rubyonrails.org/.

Michael Hanus. Type-oriented construction of web user interfaces. In *ACM-SIGPLAN Symposium on Principles and Practice of Declarative Programming*, pages 27–38, Venice, Italy, July 2006.

Michael Hanus. Putting declarative programming into the web: Translating Curry to JavaScript. In *ACM-SIGPLAN Symposium on Principles and Practice of Declarative Programming*, pages 155–166, Wrocław, Poland, July 2007.

Ralf Hinze. Deriving backtracking monad transformers. In *ACM SIGPLAN International Conference on Functional Programming*, Montreal, Canada, September 2000.

Ralf Hinze. Generics for the masses. In *ACM SIGPLAN International Conference on Functional Programming*, Snowbird, Utah, September 2004.

Ralf Hinze, Johan Jeuring, and Andres Löh. Comparing approaches to generic programming in Haskell. In *Spring School on Datatype-Generic Programming*, volume 4719 of *Lecture Notes in Computer Science*, Nottingham, UK, April 2006a.

Ralf Hinze, Andres Löh, and Bruno C. d. S. Oliveira. "Scrap your boilerplate" reloaded. In *Symposium on Functional and Logic Programming*, Fuji Susono, JAPAN, April 2006b.

Haruo Hosoya and Benjamin C. Pierce. XDuce: A statically typed XML processing language. *ACM Transactions on Internet Technology*, 3(2):117–148, 2003.

Paul Hudak, Antony Courtney, Henrik Nilsson, and John Peterson. Arrows, robots, and functional reactive programming. In Johan Jeuring and Simon Peyton Jones, editors, *Advanced Functional Programming*, Oxford, UK, July 2003.

Paul Hudak, John Hughes, Simon Peyton Jones, and Philip Wadler. A history of Haskell: being lazy with class. In *ACM SIGPLAN conference on History of programming languages*, pages 12–1–12–55, San Diego, California, 2007.

John Hughes. Why functional programming matters. *Computer Journal*, 32(2), 1989.

John Hughes. The design of a pretty-printing library. In *Advanced Functional Programming*, pages 53–96, Båstad, Sweden, May 1995.

John Hughes. Generalising monads to arrows. *Science of Computer Programming*, 37:67–111, May 2000.

John Hughes. Programming with arrows. In *Advanced Functional Programming*, volume 3622 of *Lecture Notes in Computer Science*. Tartu, Estonia, August 2004.

Graham Hutton and Erik Meijer. Monadic Parsing in Haskell. *Journal of Functional Programming*, 8(4):437–444, July 1998.

Martin Jambon. json-static, 2007.
http://martin.jambon.free.fr/json-static.html.

Patrik Jansson and Johan Jeuring. Polyp – a polytypic programming language extension. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, Paris, France, January 1997.

Patrik Jansson and Johan Jeuring. Polytypic compact printing and parsing. In *European Symposium on Programming*, Amsterdam, The Netherlands, March 1999.

Thomas Johnsson. Lambda lifting: Transforming programs to recursive equations. In *Functional Programming Languages and Computer Architecture*, volume 201 of *LNCS*, Nancy, France, 1985. Springer Verlag.

Mark P. Jones. A system of constructor classes: overloading and implicit higher-order polymorphism. In *Functional Programming Languages and Computer Architecture*, pages 52–61, Copenhagen, Denmark, June 1993. ISBN 0-89791-595-X.

Mark P. Jones and John Peterson. *The Hugs 98 User Manual*, 1999.
http://cvs.haskell.org/Hugs/downloads/hugs.pdf.

Vesa Karvonen. Generics for the working ML'er. In *ACM SIGPLAN Workshop on ML*, Freiburg, Germany, October 2007.

Andrew J. Kennedy. Functional pearl: Pickler combinators. *Journal of Functional Programming*, 14(6), November 2004.

Oleg Kiselyov. Applicative translucent functors in Haskell, August 2004.
http://haskell.org/pipermail/haskell/2004-August/014463.html.

Ralf Lämmel and Simon Peyton Jones. Scrap your boilerplate: a practical design pattern for generic programming. In *ACM SIGPLAN Workshop on Types in Language Design and Implementation*, New Orleans, Louisiana, USA, January 2003.

Daan Leijen and Erik Meijer. Domain specific embedded compilers. In *Domain-Specific Languages*, pages 109–122, Austin, Texas, United States, October 1999.

Xavier Leroy. *The Objective Caml system*, November 2008.
http://caml.inria.fr/pub/docs/manual-ocaml/index.html.

Chuck Liang and Gopalan Nadathur. Tradeoffs in the intensional representation of lambda terms. In *Rewriting Techniques and Applications*, Copenhagen, Denmark, July 2002.

Sheng Liang, Paul Hudak, and Mark Jones. Monad transformers and modular interpreters. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, San Francisco, California, United States, January 1995.

Sam Lindley. Many holes in Hindley-Milner. In *ACM SIGPLAN Workshop on ML*, Victoria, British Columbia, Canada, September 2008.

Sam Lindley and Ian Stark. Reducibility and $\top\top$-lifting for computation types. In *Typed Lambda Calculi and Applications*, pages 262–277, Nara, Japan, 2005.

Sam Lindley, Philip Wadler, and Jeremy Yallop. The Arrow Calculus. Technical report, University of Edinburgh, 2008a.

Sam Lindley, Philip Wadler, and Jeremy Yallop. Idioms are oblivious, arrows are meticulous, monads are promiscuous. In Venanzio Capretta and Conor McBride, editors, *Mathematically Structured Functional Programming*, Reykjavik University, Iceland, July 2008b.

Sam Lindley, Philip Wadler, and Jeremy Yallop. The Arrow Calculus. *Journal of Functional Programming*, 20(1):51–69, January 2010.

Saunders Mac Lane. *Categories for the working mathematician*. Springer, 1998.

Jacob Matthews and Amal Ahmed. Parametric Polymorphism Through Run-Time Sealing, or, Theorems for Low, Low Prices! In *European Symposium on Programming*, March 2008.

Jacob Matthews and Robert Bruce Findler. Operational semantics for multi-language programs. *ACM Transactions on Programming Languages and Systems*, 31(3):1–44, 2009.

Conor McBride. Post to Haskell mailing list, June 2004.
http://www.haskell.org/pipermail/haskell/2004-July/014315.html.

Conor McBride and Ross Paterson. Applicative programming with effects. *Journal of Functional Programming*, 18(1):1–13, 2008.

Robin Milner, Mads Tofte, Robert Harper, and David MacQueen. *The Definition of Standard ML (Revised)*. The MIT Press, 1997.

Neil Mitchell and Stefan O'Rear. Derive, 2006.
http://www-users.cs.york.ac.uk/~ndm/derive/.

Eugenio Moggi. Computational lambda-calculus and monads. In *Symposium on Logic in Computer Science*, Pacific Grove, California, United States, June 1989.

Eugenio Moggi. Notions of computation and monads. *Information and Computation*, 93(1): 55–92, 1991.

Anders Møller and Michael I. Schwartzbach. The design space of type checkers for XML transformation languages. In *International Conference on Database Theory*, Edinbugh, Scotland, January 2005.

James H. Morris, Jr. Types are not sets. In *ACM SIGPLAN - SIGACT Symposium on Principles of Programming Languages*, pages 120–124, Boston, Massachusetts, October 1973.

Markus Mottl. Bin-prot, 2007.
http://www.janestreet.com/ocaml/.

Markus Mottl. Sexplib, 2008.
http://www.janestreet.com/ocaml/.

Martin Odersky, Philip Wadler, and Martin Wehr. A second look at overloading. In *Functional Programming Languages and Computer Architecture*, pages 135–146, La Jolla, California, United States, June 1995.

Ross Paterson. A new notation for arrows. In *ACM SIGPLAN International Conference on Functional Programming*, Firenze, Italy, September 2001.

Ross Paterson. arrows: Arrow classes and transformers, November 2008.
http://www.haskell.org/arrows/download.html, Version 0.4.1.

Simon Peyton Jones and John Hughes, editors. *Haskell 98: A Non-strict, Purely Functional Language*. Cambridge University Press, February 1999.

Simon L. Peyton Jones. *The Implementation of Functional Programming Languages*, chapter Supercombinators and Lambda-Lifting. Prentice-Hall, 1986.

PHP. PHP Hypertext Preprocessor, March 2008.
http://www.php.net/.

Benjamin C. Pierce. *Types and Programming Languages*. The MIT Press, 2002.

Andrew M. Pitts. Parametric polymorphism and operational equivalence. *Mathematical Structures in Computer Science*, 10(3):321–359, 2000.

Rinus Plasmeijer and Peter Achten. iData for the world wide web: Programming interconnected web forms. In *Symposium on Functional and Logic Programming*, Lecture Notes in Computer Science, pages 242–258, Fuji Susono, JAPAN, April 2006.

Rinus Plasmeijer, Peter Achten, and Pieter Koopman. iTasks: executable specifications of interactive work flow systems for the web. In *ACM SIGPLAN International Conference on Functional Programming*, pages 141–152, Freiburg, Germany, October 2007.

Christian Queinnec. The influence of browsers on evaluators or, continuations to program web servers. In *ACM SIGPLAN International Conference on Functional Programming*, Montreal, Canada, September 2000.

J. Rees (eds.) R. Kelsey, W. Clinger. Revised[5] report on the algorithmic language Scheme. *Higher-Order and Symbolic Computation*, 11(1), 1998.

John C. Reynolds. Definitional interpreters for higher-order programming languages. In *ACM annual conference*, Boston, Massachusetts, United States, 1972.

John C. Reynolds. Types, abstraction and parametric polymorphism. *Information Processing*, pages 513–523, 1983.

Alexey Rodriguez, Stefan Holdermans Andres Löh, and Johan Jeuring. Generic programming with fixed points for mutually recursive datatypes. In *ACM SIGPLAN International Conference on Functional Programming*, Edinburgh, Scotland, UK, September 2009.

Sergei Romanenko, Claudio Russo, and Peter Sestoft. *Moscow ML Owner's manual*, June 2000.

Claudio V. Russo. First-class structures for standard ml. *Nordic Journal of Computing*, 7(4): 348–374, 2000.

Amr Sabry and Matthias Felleisen. Reasoning about programs in continuation-passing style. *LISP and Symbolic Computation*, 6(3/4):287–358, 1993.

Martin Sandin. Tywith, 2004. http://tools.assembla.com/tywith/wiki.

Tim Sheard and Simon Peyton Jones. Template metaprogramming for Haskell. In *ACM SIGPLAN Haskell Workshop*, pages 1–16, Pittsburgh, PA, USA, October 2002.

Steve Strugnell. Creating linksCollab: an assessment of Links as a web development language. BSc thesis, University of Edinburgh, 2008.

Eijiro Sumii and Benjamin C. Pierce. A bisimulation for dynamic sealing. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, Venice, Italy, January 2004.

Nikhil Swamy, Michael Hicks, and Simon Tsang. Verified enforcement of security policies for cross-domain information flows. In *Military Communications Conference*, Orlando, Florida, October 2007.

Nikhil Swamy, Brian J. Corcoran, and Michael Hicks. Fable: A language for enforcing user-defined security policies. In *IEEE Symposium on Security and Privacy*, Oakland, CA, USA, May 2008.

S. Doaitse Swierstra and Luc Duponcheel. Deterministic, error-correcting combinator parsers. In *Advanced Functional Programming*, volume 1129 of *Lecture Notes in Computer Science*, Olympia, WA, USA, August 1996.

Don Syme. Initializing mutually referential abstract objects: The value recursion challenge. *Electronic Notes in Theoretical Computation Science*, 148(2):3–25, 2006.

Guido Tack, Leif Kornstaedt, and Gert Smolka. Generic pickling and minimization. In *ACM SIGPLAN Workshop on ML*, Tallinn, Estonia, September 2005.

The GHC Team. *The Glorious Glasgow Haskell Compilation System User's Guide, Version 6.12.1*, December 2009.
http://haskell.org/ghc/docs/6.12.1/html/users_guide/.

Peter Thiemann. A typed representation for HTML and XML documents in Haskell. *Journal of Functional Programming*, 12(4&5):435–468, July 2002.

Peter Thiemann. An embedded domain-specific language for type-safe server-side web scripting. *ACM Transactions on Internet Technology*, 5(1):1–46, 2005.

D. A. Turner. Miranda: a non-strict functional language with polymorphic types. In *Functional Programming Languages and Computer Architecture*, pages 1–16, Nancy, France, September 1985.

Juliana Kaizer Vizzotto, AndrÃl' Rauber Du Bois, and Amr Sabry. The arrow calculus as a quantum programming language. In *Workshop on Logic, Language, Information and Computation*, Tokyo, Japan, June 2009.

Philip Wadler. How to replace failure by a list of successes. In *Functional Programming Languages and Computer Architecture*, Nancy, France, September 1985.

Philip Wadler. Theorems for free! In *Functional Programming Languages and Computer Architecture*, London, United Kingdom, September 1989.

Philip Wadler. Comprehending monads. In *ACM conference on LISP and functional programming*, Nice, France, 1990.

Philip Wadler and Peter Thiemann. The marriage of effects and monads. *ACM Transactions on Computational Logic*, 4(1):1–32, 2003.

Stefan Wehr and Manuel Chakravarty. ML modules and Haskell type classes: A constructive comparison. In *Asian Symposium on Programming Languages and Systems*, Bangalore, India, December 2008.

Stephanie Weirich. Replib: A library for derivable type classes. In *ACM SIGPLAN Haskell Workshop*, Portland, Oregon, September 2006.

Noel Winstanley. DrIFT, 1997.
http://repetae.net/~john/computer/haskell/DrIFT/.

Jeremy Yallop. Practical generic programming in OCaml. In Derek Dreyer, editor, *ACM SIGPLAN Workshop on ML*, Freiburg, Germany, October 2007.

Zhe Yang. Encoding types in ML-like languages. In *ACM SIGPLAN International Conference on Functional Programming*, Baltimore, Maryland, USA, September 1998.