Technical Report

UCAM-CL-TR-309
ISSN 1476-2986

Number 309

UNIVERSITY OF CAMBRIDGE
Computer Laboratory

Strictness analysis of lazy functional programs

Peter Nicholas Benton

August 1993

15 JJ Thomson Avenue
Cambridge CB3 0FD
United Kingdom
phone +44 1223 763500

https://www.cl.cam.ac.uk/

© 1993 Peter Nicholas Benton

This technical report is based on a dissertation submitted December 1992 by the author for the degree of Doctor of Philosophy to the University of Cambridge, Pembroke College.

Technical reports published by the University of Cambridge Computer Laboratory are freely available via the Internet:

https://www.cl.cam.ac.uk/techreports/

ISSN 1476-2986

Abstract

Strictness analysis is a compile-time analysis for lazy functional languages. The information gained by a strictness analyser can be used to improve code generation for both sequential and parallel implementations of such languages.

After reviewing the syntax and semantics of a simply typed lambda calculus with constants, we describe previous work on strictness analysis. We then give a new formulation of higher order strictness analysis, called strictness logic. This is inspired by previous work on static analysis by non-standard type inference, and by work on logics of domains. We investigate some proof theoretic and semantic properties of our logic, and relate it to the conventional approach using abstract interpretation. We also consider extending the logic with disjunction.

We then describe how to extend the simply typed lambda calculus with lazy algebraic datatypes. A new construction of lattices of strictness properties of such datatypes is described. This arises from the characterisation of the solutions to the recursive domain equations associated with these types as initial algebras.

Next we consider first order (ML-style) polymorphism and show how Wadler's 'theorems for free' parametricity results may be obtained from a simple extension of the semantics of the monomorphic language. We then prove a polymorphic invariance result relating the derivable strictness properties of different substitution instances of polymorphic terms.

Preface to the Technical Report Edition

This technical report is, apart from a few very minor typographical corrections, identical to the submitted version of my thesis. Several relevant bits of work have, however, been done or come to my attention since the thesis was submitted. These include the following:

General. Jensen's thesis [Jen92a] has a very similar scope to this one, though with a slightly different perspective.

Section 2.1.2 Full Abstraction. Some very recent work by Abramsky, Jagadeesan and Malacaria at Imperial [AJM93] and by Hyland and Ong at Cambridge [HO93] leads to semantic presentations of intensionally fully abstract models for PCF.

Section 2.3.2 Type Inference for $\lambda 2$ appears recently to have been shown to be undecidable by a number of people, for example [Wel93].

Section 4.3.2 Computational Adequacy proofs for languages with general recursive types (as opposed to the merely algebraic ones considered in this thesis) can be found in various places. See, for example, [Win93] (which is also an excellent introduction to much of the background material presented here). Pitts has used the methods of [Pit92] to give a particularly neat and 'modern' proof [Pit93].

Section 4.5 Strictness Properties of Lazy Algebraic Types. A version of this work appears in [Ben93]. That paper improves slightly on the work presented here in that there is now a (rather unpleasant) syntax for the propositions associated with an arbitrary algebraic datatype and general program logic rules (expressed in terms of the new syntax). I still do not have proof rules which capture the way in which the initial algebra induction principle can be used to reason about entailment, however.

Section 6.3 Using Strictness Information. Burn and Le Métayer have also considered the problem of how to express and justify the optimisations which one wishes to make as a result of strictness analysis [BL92]. They suggest using the information to modify the translation of the source language into continuation passing style.

Finally, I should like to take this opportunity to thank my examiners Chris Hankin and Andy Pitts.

Contents

List of Figures

Chapter 1

Introduction

1.1 Functional Programming

1.1.1 Background

A program in a conventional imperative language, such as Pascal, C or FORTRAN, consists of a sequence of commands to be obeyed, each of which effects changes in the store of the computer. This view of computation is very close to the actual architecture of nearly all present-day, general-purpose computers. It is therefore relatively straightforward to compile a program written in such a language into a sequence of machine code instructions which can be executed efficiently by the machine.

There are, however, significant problems associated with programming in imperative languages, which have contributed towards what has been called 'the software crisis'. There has been rapid progress in computer hardware, so that larger and more complicated systems should be feasible. However, software technology has not kept pace. Programmers are increasingly swamped by complexity, partly because things really are more complex, but largely because they are are still using tools, languages and methodologies developed for the previous generation of systems. The practical consequences are that software is now the most expensive component of many computer systems, is frequently delivered late and usually fails to work correctly.

One approach to resolving these problems is mainly sociological: specifying how programs should be designed, laying down standards for documentation and inventing rules about how teams of people should cooperate on a project. A complementary approach is to improve the tools which are used for software development.

Much current research on improving the technologies used for program design and development falls into the broad area of formal methods. This means using mathematical techniques to reason (more or less formally) about systems. Such reasoning can provide important insights into the problem, guide the development of a solution and greatly increase confidence in the correctness of the final product. The need for formal methods is due not only to increased complexity, but also to the

increasing use of computers in so-called 'safety-critical' applications. Completely formal mathematical proofs of program correctness are usually large, complex and tedious, but this can be at least partially alleviated by the use of automated theorem proving systems. A more modest view of the place of formal methods in everyday programming is that if we develop languages which are suitable for formal proof and teach programmers about these proof principles, then the informal reasoning which they use when writing programs is much more likely to be correct.

If we wish to apply formal methods to programming, we have to start with a formal mathematical account of what a given program means. This is known as semantics. In the past, programming language definitions tended to go into great detail specifying the surface syntax, and then fall back on ambiguous English descriptions when it came to saying what that syntax actually meant. There are three main styles of semantics:

- An *operational* semantics defines the intended behaviour of programs by giving a set of rules which specify how programs are to be executed. This feels like traditional computer science.
- An *axiomatic* semantics is more directly aimed at proving properties of programs. It gives rules for proving assertions like 'if the machine is initially in a state satisfying property P, then after the execution of command C it will be in a state satisfying property Q'. This has a logical flavour.
- A *denotational* semantics gives the meaning of programs as objects in certain mathematical spaces, such as domains. This feels like traditional mathematics.

These different styles of reasoning are each good for different applications, and we obviously hope that different styles of semantics for the same language will turn out to be equivalent in some sense. In fact, the relationships between different semantics can be surprisingly complicated. This dissertation contains elements of all three approaches.

Having given a formal semantics to a programming language, one should then be able to use it to prove that a program meets its specification (expressed in some suitable formalism), or that two programs are equivalent. Unfortunately, the semantics of imperative languages can be rather intractable. In particular, the presence of side-effects means that referential transparency is lost—the simple substitution of equals for equals does not, in general, preserve the meaning of programs.

These problems become even more acute when we come to consider programming parallel machines. Whilst it is simple enough to add parallel programming constructs to an imperative language, using them efficiently and reasoning about the correctness of the resulting programs can be extremely difficult. It should be noted, however, that there has been considerable progress in the area of calculi for reasoning about the behaviour of concurrent systems.

1.1.2 Functional Languages

Functional programming languages appear to have the potential to alleviate all the difficulties mentioned above. Most of the problems seem to stem from the concept of implicit updatable state. It is the presence of state which prevents a simple statement like f(3)=f(3) from always being true in an imperative language (as the function f might both depend on and update the state). It is the state which prevents us from executing arbitrary parts of an imperative program on different processors. Functional languages do away with state, commands and looping, replacing them with side-effect-free expressions, recursion and powerful type systems.

Programming with higher-order functions and without side effects brings many benefits. Programs are clear, concise, more likely to work first time and easier to reason about. The level of abstraction is higher than in imperative languages, so programs are closer to specifications, freeing the programmer from worrying about low-level details. Functional programs are also well-suited to execution on parallel machines – so long as we preserve termination, parts of a program can be evaluated in any order, or even in parallel, without affecting the final result. The advantages of functional languages are argued in, for example, [Bac78, Hug89].

Of course, functional languages do not solve all our problems at a stroke. There are two main difficulties. The first is that whilst functional languages are good for writing certain sorts of program (compilers, theorem provers), they are less obviously appropriate for writing programs which have to manipulate state, such as real-time controllers and operating systems. Research is addressing this problem both by seeing how far the functional paradigm is applicable to this sort of application, and by considering how to re-introduce state-like ('impure') features in a controlled way. The second problem is that, because the underlying model is further away from the real machine, it is harder to compile a functional program into efficient machine code. This means that functional programs tend to run more slowly than imperative ones. There has been some work on addressing this problem by building special-purpose computers for running functional languages, but most current work is aimed at producing better compilers for conventional architectures.

The foundations of functional languages lie in the lambda calculus and combinatory logic (both of which predate computer programming). The lambda calculus is a theory of functions, which was invented by the logician Alonzo Church in the 1930s. In the pure, untyped, lambda calculus, everything is a function: functions take functions as arguments and return functions as results. This turns out to be powerful enough to code arithmetic and, in fact, to represent all functions which are computable by a Turing machine. Functional programming languages tend to be very close to their mathematical foundations, though they are usually based on a typed lambda calculus with constants. A functional program may be translated into an expression in the lambda calculus. The program is executed by applying local rewriting rules to the lambda calculus term until some normal form is reached. This process is known as reduction.

1.1.3 Strict or Lazy?

An expression in the lambda calculus will, in general, contain several redexes (subexpressions which can be reduced). The Church-Rosser theorem tells us that if a term t can be reduced to t' by one sequence of reductions, and to t'' by another, then there is a term t''' to which both t' and t'' can be reduced. This does not quite mean that the order of reductions is unimportant, however. If we choose a bad reduction strategy, we could end up reducing for ever when a different strategy would have reached a normal form (a term with no redexes).

There are two particularly natural reduction strategies. The first is called applicative-order, or innermost, reduction. This corresponds to evaluating each argument to a function before substituting the arguments for the formal parameters in the function body. Each function argument is therefore evaluated exactly once. In a programming language this is called call-by-value parameter passing. It is a natural reduction strategy, in that it is efficient and maps well onto most existing processor architectures, which have been designed with call-by-value evaluation of imperative languages in mind. The disadvantage is that it is not guaranteed to find a normal form if one exists. Languages which employ applicative-order reduction, such as ML [GMW79, MTH90] and Hope [BMS80], are called strict.

The other main reduction strategy is called normal-order, or leftmost, reduction. This corresponds to substituting arguments unevaluated into function bodies. In a programming language, this is known as call-by-name. Normal-order reduction will find a normal form if one exists, but it is potentially inefficient as an argument may end up being reduced many times, although it may also turn out not to be needed and thus avoid being reduced at all.

There is an optimisation to normal-order reduction, which is vital if one is to implement a programming language with normal-order semantics. Instead of copying the unevaluated argument into the function body, we can pass a pointer to the argument. If the argument is evaluated, then the reduced value overwrites the original expression. Any subsequent reference to the argument then picks up the reduced value for free (of course, this only preserves the result of the program because of referential transparency). This is known as lazy evaluation or call-by-need parameter passing and is characterised by the fact that each argument is either never reduced or reduced exactly once. This technique was first proposed by Wadsworth [Wad71]. Lazy languages include Ponder [Fai82, Fai85], Miranda™ [Tur85][1] and Haskell [HWA+90]. The terminology has become somewhat confused in recent years, and the word 'lazy' is now frequently used to mean either simply 'non-strict' or 'reducing to weak head normal form'. There is a small, but significant, mismatch between the traditional theory of the lambda calculus and real lazy functional languages which concerns the question of when leftmost reduction should stop. The traditional theory is based on stopping when head normal form (HNF) is reached, which roughly means that the leftmost application is the application of a variable. Lazy functional languages, by contrast, stop leftmost reduction as soon as they reach a lambda-abstraction

[1] Miranda is a trademark of Research Software Limited.

(infomally, no reduction takes place 'under a λ'). This is called weak head normal form (WHNF). The term 'lazy' is now often used to refer to the theory of the pure lambda calculus with reduction to WHNF [Abr88, Ong88]. In this work, we will be concerned with lazy functional languages in the broad sense of reducing to weak head normal form.

Lazy languages allow some novel programming techniques, such as using infinite data structures, and defining new control structures. They are generally regarded as being more 'pure' than strict languages and to be more pleasant to reason about (though this is a matter of taste). We should qualify that assertion—the applicability of familiar principles of equational reasoning makes extensional properties (*i.e.* those properties relating to the input-output behaviour of functions, typically correctness) of lazy functional programs relatively easy to prove. Intensional properties, such as time or space complexity, can be hard to reason about for normal-order evaluation, and the situation becomes significantly worse for lazy evaluation. It is not uncommon for a small, meaning-preserving, transformation to cause a large and unexpected change in efficiency. This problem is unfortunately exacerbated by the kind of work we shall be describing in this thesis, as aggressive optimisations mean that the compiled code is far from the user's original program. A good introduction to programming in lazy functional languages can be found in [BW88].

1.1.4 Implementations of Lazy Languages

The earliest implementations of lazy languages were entirely interpretive, and hence very slow indeed. These were followed by implementations which compiled code for abstract machines for graph reduction, such as the G-Machine [Aug84, Joh84, PJ87] and the Ponder Abstract Machine [Fai85]. Code for these abstract machines was then compiled into real machine code, though it still retained an interpretive character. The current state of the art is represented by closure reducers such as the Three Instruction Machine (TIM) [FW87], which are variously estimated to be between 2 and 10 times slower than good compilers for imperative languages. Strict functional languages are still slightly faster than lazy ones.

As we have already mentioned, functional programs can theoretically be executed without change on parallel machines. The hope is that the programmer will just have to make sure that the algorithm is inherently parallel (for example, some form of divide and conquer), and the same program can then be compiled to run on machines with any number of processors. Although this seems to offer 'parallelism without tears', there are still significant obstacles. One of the main problems is the generation of far too many small tasks, so that the overheads of communication reduce the benefits of parallelism. A related difficulty is that it is hard to partition tasks between the available processors in an efficient way. Several parallel implementations of functional languages have been built and appear to offer modest, but genuinely useful, performance gains [Bur88, WW87, PJCSH87, KLB89, AJ89].

One of the most important ways to reduce the performance gap between functional and imperative languages on sequential machines, and to approach the problems

of compiling for parallel machines, is to improve compilers by making use of static analysis.

1.2 Static Analysis and Optimising Compilers

Static analysis is the process of analysing the user's program at compile time to discover information which can be used to generate better code. Optimising compilers for imperative languages have been doing this sort of thing for years: detecting invariant expressions so that they can be moved out of loops, eliminating dead code and propagating constants, for example. In practice, these optimisations are often rather *ad hoc*. It is not uncommon for optimising compilers to perform unsafe transformations. Various analyses for imperative programs were justified and put into a common framework by the Cousots' seminal work on abstract interpretation [CC77, CC79].

1.2.1 Optimising Functional Programs

If we look at why lazy functional languages are still slower than their imperative counterparts (or even strict functional languages), we find that it is essentially because functional programs contain much less operational (intensional) detail (which is precisely their strength from the point of view of writing programs). Consequently, a simple-minded compilation scheme, even to a relatively efficient abstract machine code like TIM, will be excessively general. For example, it will always pass arguments unevaluated because they may not be needed. If we can discover at compile-time that a particular function always evaluates its argument, then we can compile code which passes that argument by value, thus saving the expense of building a closure. Similarly, to maintain referential transparency, the functional version of an array update operator must return a modified copy of the original array, rather than just overwriting the original. If we can be sure that, in a particular case, the original version will never be referred to again then we can compile code which performs the update in place without affecting the semantics of the program.

Clearly, the kind of information we are asking for is, in general, uncomputable. The best we can hope for from an automatic analysis system is safe approximations. For example, saying that an array might be referred to again when it actually will not be just leads to a slightly less efficient program, and is therefore safe, whereas the converse could change the result of the program, and is therefore unsafe.

There are many different analyses which have been studied in the literature. For example:

- *Strictness* analysis aims to find out how much information is definitely required about the argument to a function to produce a given amount of information about its result. The information can be used to transform call-by-name into call-by-value, to spawn tasks in a parallel machine and to allow certain source-level transformations.

- *Usage* (liveness) analysis discovers whether objects can be used in future. It can be used for compile-time garbage collection and in-place updates as well as to avoid unnecessary updates of closures that will never be referenced again.

- *Sharing* analysis is similar to usage analysis and has similar applications. It attempts to estimate how many references there will be to an object.

- *Binding-time* analysis discovers how much of the result of a function can be computed at compile time, given that we know a certain amount about its inputs. It is essential for performing partial evaluation, which is the automatic generation of specialised programs from general ones. This is a powerful technique which can, for example, produce compilers from interpreters.

- *Termination* analysis discovers if the evaluation of an expression will definitely terminate.

- *Stack usage* analysis finds a safe approximation to the number of stack locations which will be used in evaluating an expression. It is particularly useful in parallel implementations because giving each task a potentially unbounded amount of stack space is very inefficient.

- *Complexity* analysis aims to find upper or lower bounds on the number of steps a reduction will take. Its applications include deciding whether or not a computation is worth spawning as a separate task in a parallel implementation.

So there are many opportunities for optimising functional programs. Since we have claimed that functional languages have tractable semantics, we aim to use the semantics to give formal proofs that our analyses and optimisations are correct. In this work, we are concerned with the theory (and to a lesser extent, the practice) of static analysis and optimising transformations for lazy functional languages. We will focus entirely on strictness analysis, though the same ideas can be applied to other analysis problems.

1.3 Outline of Thesis

- Chapter 2 reviews some basic material which we shall use throughout the thesis. It introduces the functional language with which we shall work and gives its operational semantics. We then summarise some simple domain theory, give a denotational semantics to our language and show how this is related to the operational semantics. We then describe previous work on strictness analysis.

- Chapter 3 introduces a new formulation of strictness analysis, called *strictness logic*. This extends and gives a firm foundation for previous work on strictness analysis by non-standard type inference. We prove some semantic and

proof-theoretic results about the logic and study its relation to abstract interpretation. We also describe a more powerful logic which includes disjunction and show that a disjunctive logic proposed by Jensen is unsound.

- Chapter 4 describes the extension of our simple language to include lazy algebraic datatypes. After reviewing some standard material on the domain theoretic semantics of such types and previous work on strictness analysis of recursive types, we present a new construction of lattices of strictness properties for algebraic types. Using this construction, we show how the strictness logic may be extended to reason about a language which includes a type of lazy lists.

- Chapter 5 is concerned with the relationship between different instances of polymorphic functions in an ML-style type system. We give a simple semantics to an extension of our language including this kind of polymorphism and use this to derive the parametricity theorems which have been presented by Wadler. We then prove a polymorphic invariance theorem which relates the strictness properties which are derivable (using either abstract interpretation or our logic) of different instances of a polymorphic term.

- Finally, Chapter 6 concludes and suggests directions for further research. It includes some rather tentative suggestions about a framework for expressing and justifying the uses which can be made of strictness information in a compiler.

- Appendix A contains one of the proofs from Chapter 3 and proofs of several standard results quoted in Chapter 4.

1.4 Prerequisites

The thesis is intended to be relatively self-contained. We assume a basic knowledge of set theory and logic and some experience of a modern, typed, preferably functional, programming language. The introductory chapter is rather too terse to be used as an introduction to denotational semantics for those who have never encountered it before, but knowledge of the contents of a good undergraduate-level course should be adequate background.

Although we do not really use any category theory in this thesis, there are some uses of basic categorical terminology, which may not be familiar to all readers. Any introductory account of category theory will include definitions of all the undefined terms used here. See, for example, [BW90, AL91].

Chapter 2

Basics

This chapter reviews some background information. It introduces the syntax and semantics of the language with which we shall be working and briefly describes previous work on strictness analysis.

2.1 The Language Λ_T

This dissertation is concerned with lazy functional languages. Although there are many different lazy languages, nearly all of them are based on the simply typed lambda calculus with constants. Of course, real languages add a number of different features to this core: pattern matching, type inference, polymorphism, modules and abstract datatypes are all important aspects of modern functional languages. The typed lambda calculus is, however, the prototypical functional language and many of the extra features of real languages can be translated into it (indeed, many compilers perform such a translation at an early stage). We will therefore start by defining a very simple typed lambda calculus with constants, which we call Λ_T. This will be the basic language with which we shall work, although we shall also consider extending the language to include ML-style polymorphism, sums, recursive types and extra constructs to express strictness optimisations. Λ_T is essentially the same as Plotkin's language PCF [Plo77], except that Λ_T includes conditionals at higher type, pairs and extra language constructs instead of constants for conditionals, arithmetic and recursion. PCF has a base type for booleans, which is omitted from Λ_T.

2.1.1 Syntax

Types

Types in Λ_T are formed inductively from a single base type ι (which we shall interpret as the natural numbers) by forming products and function spaces. We shall use σ, τ and θ to range over types. It is common to add another base type for booleans, but as that makes no essential difference we have chosen not to do so. More formally,

the syntax of types is given by the following BNF grammar

$$\sigma ::= \iota \mid (\sigma \to \sigma) \mid (\sigma \times \sigma)$$

We will frequently omit parentheses in types, with the convention that $\to$ associates to the right and that $\times$ binds more tightly than $\to$. Thus $\iota \to \iota \to \iota \times \iota$ should be understood as $\iota \to (\iota \to (\iota \times \iota))$.

Terms

Terms of Λ_T are built up from typed variables and constants by application, abstraction, pairing, projections, conditionals, fixpoints and arithmetic operations. Instead of typed variables, we could have presented the language using typed abstractions or worked with typing derivations for untyped terms. We have also chosen to present the syntax with extra language constructs for conditionals and so on, rather than using constants, as this turns out to be technically slightly more convenient. We will usually use f, g, h, w, x, y, z to range over variable names, m, n for natural numbers and e, s, t, u, v for terms. The syntax is defined inductively by the rules shown in Figure 2.1, where the judgement $s :: \sigma$ is to be read 's is a well formed term of type σ'[1]. Note that typings are unique. We shall sometimes write t^τ for 't where $t :: \tau$'.

Although we have only defined one arithmetic operation, $+$, this should be regarded as generic, and we shall feel free to use multiplication, subtraction and so on in examples. Parentheses will often be omitted, with the convention that application associates to the left and binds more tightly than anything else, and that the body of an abstraction extends as far to the right as possible. The set of all terms of type σ will be written Λ_T^σ. Terms of functional type are (confusing syntax and semantics) often referred to as functions.

Free and Bound Variables and Substitution

The set $FV(t)$ of *free variables* of a term t is defined inductively as follows

$$FV(x^\sigma) = \{x^\sigma\} \qquad FV(\underline{n}) = \emptyset$$

$$FV(st) = FV(s+t) = FV((s,t)) = FV(s) \cup FV(t)$$

$$FV(\mathsf{fst}(s)) = FV(\mathsf{snd}(s)) = FV(s)$$

$$FV(\mathsf{if}\ s\ \mathsf{then}\ t_1\ \mathsf{else}\ t_2) = FV(s) \cup FV(t_1) \cup FV(t_2)$$

$$FV(\lambda x^\sigma.s) = FV(\mathsf{fix}(x^\sigma.s)) = FV(s) - \{x^\sigma\}$$

We say the variable x^σ is *bound* in $\lambda x^\sigma.t$ and in $\mathsf{fix}(x^\sigma.t)$. We identify terms which only differ in the names of bound variables. We say a term t is *closed* if $FV(t) = \emptyset$. The set of all closed terms in our language is denoted by Λ_T°. This will be combined

[1]The use of :: for typings is to avoid confusion with judgements in the strictness logic which we shall present later.

Variables	$x^\sigma :: \sigma$
Numerals	$\dfrac{n \in \mathbb{N}}{\underline{n} :: \iota}$
Application	$\dfrac{t :: \sigma \to \tau \qquad s :: \sigma}{(ts) :: \tau}$
Abstraction	$\dfrac{t :: \tau}{\lambda x^\sigma.t :: \sigma \to \tau}$
Pairs	$\dfrac{s :: \sigma \qquad t :: \tau}{(s,t) :: \sigma \times \tau}$
First projection	$\dfrac{s :: \sigma \times \tau}{\mathsf{fst}(s) :: \sigma}$
Second projection	$\dfrac{s :: \sigma \times \tau}{\mathsf{snd}(s) :: \tau}$
Conditional	$\dfrac{s :: \iota \qquad t_1 :: \tau \qquad t_2 :: \tau}{\mathsf{if}\ s\ \mathsf{then}\ t_1\ \mathsf{else}\ t_2 :: \tau}$
Recursion	$\dfrac{s :: \sigma}{\mathsf{fix}(x^\sigma.s) :: \sigma}$
Arithmetic	$\dfrac{s :: \iota \qquad t :: \iota}{s + t :: \iota}$

Figure 2.1: Syntax of Λ_T

$$\underline{n}[s/x^\sigma] = \underline{n} \qquad x^\sigma[s/x^\sigma] = s \qquad y^\tau[s/x^\sigma] = y^\tau \text{ (for } y^\tau \neq x^\sigma)$$

$$(uv)[s/x^\sigma] = (u[s/x^\sigma]v[s/x^\sigma])$$

$$\lambda y^\tau.t[s/x^\sigma] = \begin{cases} \lambda y^\tau.t & \text{if } y^\tau = x^\sigma \\ \lambda y^\tau.t[s/x^\sigma] & \text{if } y^\tau \neq x^\sigma \text{ and } y^\tau \notin FV(s) \\ \lambda z^\tau.(t[z^\tau/y^\tau])[s/x^\sigma] & (\text{where } z^\tau \text{ is new}) \text{ otherwise} \end{cases}$$

$$\mathsf{fix}(y^\tau.t)[s/x^\sigma] = \begin{cases} \mathsf{fix}(y^\tau.t) & \text{if } y^\tau = x^\sigma \\ \mathsf{fix}(y^\tau.t[s/x^\sigma]) & \text{if } y^\tau \neq x^\sigma \text{ and } y^\tau \notin FV(s) \\ \mathsf{fix}(z^\tau.(t[z^\tau/y^\tau])[s/x^\sigma]) & (\text{where } z^\tau \text{ is new}) \text{ otherwise} \end{cases}$$

$$(u, v)[s/x^\sigma] = (u[s/x^\sigma], v[s/x^\sigma])$$

$$(\mathsf{fst}(t))[s/x^\sigma] = \mathsf{fst}(t[s/x^\sigma]) \qquad (\mathsf{snd}(t))[s/x^\sigma] = \mathsf{snd}(t[s/x^\sigma])$$

$$\mathsf{if}\ t\ \mathsf{then}\ u_1\ \mathsf{else}\ u_2[s/x^\sigma] = \mathsf{if}\ t[s/x^\sigma]\ \mathsf{then}\ u_1[s/x^\sigma]\ \mathsf{else}\ u_2[s/x^\sigma]$$

$$(u + v)[s/x^\sigma] = u[s/x^\sigma] + v[s/x^\sigma]$$

Figure 2.2: Capture-Avoiding Substitution

with our earlier notation so that the set of all closed terms of type σ will be written $\Lambda_T^{\circ,\sigma}$.

If $s :: \sigma$ and $t :: \tau$, then we define $t[s/x^\sigma]$ (read 't with s substituted for free occurrences of x^σ') by induction on the structure of t as shown in Figure 2.2. This is a slightly fussy definition, since we could simply rename bound variables whenever we substitute into a binding construct, rather than just when it is absolutely necessary[2].

Lemma 2.1.1 *If $t :: \tau$ and $s :: \sigma$ then $t[s/x^\sigma] :: \tau$.*

Proof. Induction on the structure of t. □

We will also write $t[s_1/x_1^{\sigma_1}, \ldots, s_n/x_n^{\sigma_n}]$ for the *simultaneous substitution* of each s_i for free occurrences of the corresponding $x_i^{\sigma_i}$ in t. We refrain from giving a detailed definition.

[2]The whole business of variable names is an annoying technicality, and can be avoided by the use of combinators or de Bruijn indices [dB72]. Unfortunately, these are both rather difficult to read. Another possibility is to use higher-order abstract syntax, such as Martin-Löf's theory of arities and expressions (see, for example, [NPS90]), but this seems, because of its relative unfamiliarity, to create as much confusion as it removes.

2.1.2 Semantics

We now describe the intended meaning of terms in the language Λ_T. This is done by giving both an operational and a denotational semantics and showing how they are related. The operational semantics describes how terms are to be reduced whereas the denotational semantics gives the meaning of terms as mathematical objects which are more convenient for reasoning.

Operational Semantics

The operational semantics of Λ_T is given by defining an evaluation relation between closed terms of the language and *canonical* terms. Canonical terms are those terms on which our evaluator will halt; in our case this means closed terms in weak head normal form (WHNF). The canonical forms are as follows:

$$\underline{n} \qquad (s,t) \qquad \lambda x^\sigma.t$$

Note that there is one canonical form for each type constructor. We shall use c to range over canonical terms.

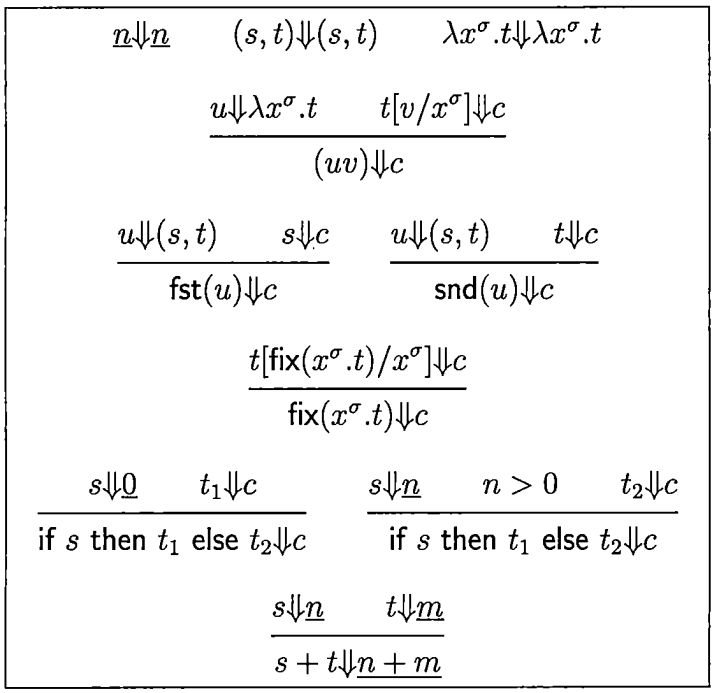
The evaluation relation $\Downarrow$ is defined by the rules shown in Figure 2.3. This is a 'big step' evaluation relation [Kah88], rather than a 'one step' rewriting relation like that used in [Plo77]. $t\Downarrow c$ should be read 't converges to c' and we shall also write $t\Downarrow$ ('t converges') for $\exists c.t\Downarrow c$ and $t\Uparrow$ ('t diverges') for $\neg(t\Downarrow)$.

This is a *call by name* (CBN) semantics, as can be seen from the rule for applications: arguments are substituted unevaluated into the bodies of functions. Note also that evaluation is deterministic—if $t\Downarrow c$ and $t\Downarrow c'$ then $c = c'$. The proof rules above are rather more than a system for deriving facts of the form $t\Downarrow c$, in that the derivations show just *how* a term is reduced. We shall sometimes use Ω^σ as an abbreviation for the term $\mathsf{fix}(x^\sigma.x^\sigma)$, which is easily seen to be a divergent term of type σ.

Elementary Domain Theory

Before presenting the denotational semantics of our language, we review some extremely basic material about ordered sets. This is mainly to fix notation, so we only introduce the minimum amount of material which we shall need in what follows. For a more comprehensive account of the theory of ordered sets, including domains, see [DP90]. For their use in denotational semantics, see [Sto77, Ten91, Plo79, Plo77].

A *preorder* P is a pair $(|P|, \sqsubseteq_P)$ where $|P|$ is a set (called the *carrier* or *underlying* set) and $\sqsubseteq_P \subseteq |P| \times |P|$ is a reflexive and transitive relation (called the *order*). Such a P is a *partial order* or *poset* if $\sqsubseteq_P$ is also antisymmetric. We will frequently abuse notation by writing P rather than $|P|$ and by omitting the subscript on the order relation where it is clear from context. Write $x \sqsupseteq y$ for $y \sqsubseteq x$. A function $f : |P| \to |Q|$ between the underlying sets of preorders is *monotone* if it preserves the order: if $x \sqsubseteq_P y$ then $f(x) \sqsubseteq_Q f(y)$.



$$\underline{n}\Downarrow\underline{n} \qquad (s,t)\Downarrow(s,t) \qquad \lambda x^\sigma.t\Downarrow\lambda x^\sigma.t$$

$$\frac{u\Downarrow\lambda x^\sigma.t \qquad t[v/x^\sigma]\Downarrow c}{(uv)\Downarrow c}$$

$$\frac{u\Downarrow(s,t) \qquad s\Downarrow c}{\mathsf{fst}(u)\Downarrow c} \qquad \frac{u\Downarrow(s,t) \qquad t\Downarrow c}{\mathsf{snd}(u)\Downarrow c}$$

$$\frac{t[\mathsf{fix}(x^\sigma.t)/x^\sigma]\Downarrow c}{\mathsf{fix}(x^\sigma.t)\Downarrow c}$$

$$\frac{s\Downarrow\underline{0} \qquad t_1\Downarrow c}{\mathsf{if}\ s\ \mathsf{then}\ t_1\ \mathsf{else}\ t_2\Downarrow c} \qquad \frac{s\Downarrow\underline{n} \qquad n>0 \qquad t_2\Downarrow c}{\mathsf{if}\ s\ \mathsf{then}\ t_1\ \mathsf{else}\ t_2\Downarrow c}$$

$$\frac{s\Downarrow\underline{n} \qquad t\Downarrow\underline{m}}{s+t\Downarrow\underline{n+m}}$$

Figure 2.3: Operational Semantics of Λ_T

If P is a poset then the *least* or *bottom* element of P, if it exists, is an element of P, written $\bot_P$, which satisfies $\forall x \in P.\, \bot_P \sqsubseteq x$. Dually, the *greatest* or *top* element, if it exists, is written $\top_P$. A poset with a least element is said to be *pointed*. A function $f : P \to Q$ between pointed posets is said to be *strict* if it preserves the bottom element (*i.e.* $f(\bot_P) = \bot_Q$).

Given $S \subseteq |P|$, the *least upper bound* or *sup* of the set S, if it exists, is an element of P, written $\bigsqcup S$, which satisfies $\forall s \in S.\, s \sqsubseteq \bigsqcup S$ and for any $p \in P$ such that $\forall s \in S.\, s \sqsubseteq p$ we have $\bigsqcup S \sqsubseteq p$. Dually, we write $\bigsqcap S$ for the *greatest lower bound* or *inf* of the set S, if it exists. We write $x \sqcup y$ for $\bigsqcup\{x, y\}$ and similarly for $x \sqcap y$. A poset P is a *lattice* if $x \sqcup y$ and $x \sqcap y$ exist for all $x, y \in P$. It is a *complete lattice* if $\bigsqcup S$ and $\bigsqcap S$ exist for all $S \subseteq P$. A subset S of a poset P is called a *chain* if $\forall x, y \in S.\, x \sqsubseteq y \vee y \sqsubseteq x$. S is an *antichain* if $\forall x, y \in S.\, (x \sqsubseteq y) \Rightarrow (x = y)$.

As a general piece of notation, when we wish to emphasize that an ω-indexed set should be thought of as a *sequence*, we will write $u = \langle u_n \rangle$ for $\{u_n \mid n \in \omega\}$. If P is a poset, then an *ω-chain* in P is a sequence $\langle x_n \rangle$ where $\forall n \in \omega.\, x_n \in P$ and $m \leq n$ implies $x_m \sqsubseteq x_n$. An *ω-complete partial order* or *ω-cpo* is a poset for which $\bigsqcup\{x_n\}$ exists for all ω-chains $\langle x_n \rangle$. If $f : P \to Q$ is a monotone map between ω-cpos which also preserves sups of ω-chains (*i.e.* $f(\bigsqcup x_n) = \bigsqcup f(x_n)$) then we say that f is *continuous*. We will refer to ω-cpos as *predomains* and write $\mathcal{P}redom$ for the category of predomains and continuous maps. Similarly, pointed ω-cpos are called *domains*, and we write $\mathcal{D}om$ for the category of domains and continuous maps. We will also write $\mathcal{D}om_S$ for the non-full subcategory of $\mathcal{D}om$ with the same objects

but only the strict maps as morphisms. Note that any set may be regarded as a predomain by equipping it with the discrete order $x \sqsubseteq y \Leftrightarrow x = y$. We shall say that a subset S of a domain D is a *subdomain* of D if $\bot_D \in S$ and for all chains $\langle x_n \rangle$ in D, if $\forall n.\, x_n \in S$ then $\bigsqcup x_n \in S$. Note that this is stronger than just requiring S to be a domain under the induced order. If S is a subdomain of D then the inclusion map $i : S \to D$ is strict and continuous.

If P is a predomain, then $O \subseteq P$ is a *Scott-open set* if $\forall x \in O, y \in P.\, x \sqsubseteq y \Rightarrow y \in O$ (we say O is *upwards closed*), and for all ω-chains $\langle x_n \rangle$ in P, $(\bigsqcup x_n) \in O \Rightarrow \exists n \in \omega . x_n \in O$ (we say O is *inaccessible by sups of ω-chains*). $C \subseteq P$ is *Scott-closed* if its complement $P \setminus C$ is Scott-open. Equivalently, C is Scott-closed if $\forall x \in C, y \in P.\, y \sqsubseteq x \Rightarrow y \in C$ (C is *downwards closed*), and for all ω-chains $\langle x_n \rangle$ in P, $\{x_n\} \subseteq C \Rightarrow (\bigsqcup x_n) \in C$ (C is *closed under sups of ω-chains*). We will also call a non-empty Scott-closed set an *ideal*. If $S \subseteq D$ then we write $\overline{S}$ for the smallest Scott-closed set containing S, which we will call the *closure* of S. The smallest downwards closed subset of D which contains S is written $\downarrow S$, and we write $\downarrow (x)$ for $\downarrow \{x\}$. Note that $\downarrow (x) = \overline{\{x\}}$.

The Scott-open subsets of a predomain D form a topology on D, called the *Scott topology*. A map $f : D \to E$ between predomains is continuous in the sense defined above iff it is continuous in the usual sense with respect to the Scott topology.

If $f : D \to D$ is a continuous map from a domain to itself, then f has a *least fixed point*, $\mathit{fix}(f)$, which is characterised by $f(\mathit{fix}(f)) = \mathit{fix}(f)$ and for any other $d \in D$ such that $f(d) = d$, $d \sqsupseteq \mathit{fix}(f)$. This value is given by $\mathit{fix}(f) = \bigsqcup_{n \in \omega} f^n(\bot_D)$. An element $d \in D$ is called a *prefixed point* of f if $f(d) \sqsubseteq d$. $\mathit{fix}(f)$ may also characterised as the least prefixed point of f.

We will say that an element d of a predomain D is *finite* (or *compact* or *isolated*) if for any chain $\langle x_n \rangle$ in D, if $d \sqsubseteq \bigsqcup x_n$ then $\exists n.\, d \sqsubseteq x_n$. We say that a predomain D is *ω-algebraic* if for all $d \in D$ there is a chain $\langle x_n \rangle$ of finite elements with $d = \bigsqcup x_n$ and, moreover, the set of finite elements of D (sometimes written D°) is countable. A subset X of a predomain D is *consistent* if it has an upper bound in D. D is *consistently complete* if every consistent subset of D has a least upper bound. All the (pre)domains with which we shall work will in fact be consistently complete ω-algebraic ω-cpos; we shall call these *Scott domains*. We will not, however, make a big issue out of what ambient category our domains should be considered to live in.

There is an alternative way of defining domains which is based on limits of directed sets, rather than simple ω-chains. A subset X of a partial order D is *directed* if it is non-empty and any pair of elements of X have an upper bound in X. A directed complete partial order (dcpo) is a partially ordered set for which every directed subset has a least upper bound. There are then definitions of continuity, open sets, finite elements and so on in terms of directed sets. It turns out that in the case of ω-algebraic posets, these definitions are equivalent to those we have given in terms of ω-chains, so which set of definitions we choose makes little difference for our purposes.

Some of the most important constructions on (pre)domains are listed below:

0 The empty set, with the empty order relation, is a predomain, though it is not a domain. It is the initial object in $\mathcal{P}redom$.

1 The one element set $\{*\}$ with the order $* \sqsubseteq *$ is a domain. It is the terminal object in $\mathcal{D}om$ and $\mathcal{P}redom$. It is both initial and terminal in $\mathcal{D}om_S$.

Product The product $D \times E$ of two predomains has the cartesian product of the two carriers as its carrier and is ordered componentwise. $D \times E$ is a domain if both D and E are, with least element $(\bot_D, \bot_E)$. The obvious projection maps $\pi_1 : D \times E \to D$ and $\pi_2 : D \times E \to E$ are continuous. This construction generalises to the product $\prod_{i \in I} D_i$ of an arbitrary set of domains. This is a categorical product in $\mathcal{D}om$ and $\mathcal{P}redom$.

Lifting The lifting $D_\bot$ of a (pre)domain D has the set $\{(0,d) \mid d \in D\} \cup \{\bot\}$ as carrier, and is ordered by $x \sqsubseteq y$ if $x = \bot$ or $x = (0,d), y = (0,d')$ and $d \sqsubseteq d'$. We will often write $[d]$ instead of $(0,d)$ for non-bottom elements of a lifted domain. The map $[\cdot] : D \to D_\bot$ is continuous, as is the map $drop : D_{\bot\bot} \to D_\bot$ which sends $\bot$ and $[\bot]$ to $\bot$ and $[[d]]$ to $[d]$. Lifting is functorial (if $f : D \to E$ then $f_\bot : D_\bot \to E_\bot$ sends $\bot$ to $\bot$ and $[d]$ to $[f(d)]$). The lift functor together with $[\cdot]$ and $drop$ (which are natural transformations) forms a monad on $\mathcal{D}om$ and on $\mathcal{P}redom$. If $f : D \to E_\bot$ in $\mathcal{P}redom$, then we shall write $lift(f) : D_\bot \to E_\bot$ for the obvious strict extension of f. A domain which arises as the lift of a set (considered as a predomain) is said to be *flat*. We will sometimes write **2** for the two-point domain $1_\bot$.

Exponential If D,E are predomains, then the set of all continuous maps from D to E, ordered pointwise ($f \sqsubseteq g \Leftrightarrow \forall x \in D. f(x) \sqsubseteq g(x)$), forms a predomain. It is a domain if E is, with least element the function which is constantly $\bot_E$. We will variously write $[D \to E]$ or E^D for this (pre)domain. The application map $app : [D \to E] \times D \to E$ given by $app(f,d) = f(d)$ is continuous. $[D \to E]$ is an exponential in both $\mathcal{D}om$ and $\mathcal{P}redom$.

Disjoint Sum If D, E are predomains, so is $D + E$, which has the set $\{(0,d) \mid d \in D\} \cup \{(1,e) \mid e \in E\}$ as carrier and is ordered by

$$\begin{aligned} x \sqsubseteq y \;\Leftrightarrow\; & (x = (0,d) \;\&\; y = (0,d') \;\&\; d \sqsubseteq d') \vee \\ & (x = (1,e) \;\&\; y = (1,e') \;\&\; e \sqsubseteq e') \end{aligned}$$

The evident injection maps $inl : D \to D + E$ and $inr : E \to D + E$ are continuous. This is a categorical coproduct in $\mathcal{P}redom$.

Separated Sum If D, E are domains then we will (with some ambiguity) write $D + E$ for the domain with the set $\{(0,d) \mid d \in D\} \cup \{(1,e) \mid e \in E\} \cup \{\bot\}$ as carrier and the order

$$\begin{aligned} x \sqsubseteq y \;\Leftrightarrow\; & (x = \bot) \vee \\ & (x = (0,d) \;\&\; y = (0,d') \;\&\; d \sqsubseteq d') \vee \\ & (x = (1,e) \;\&\; y = (1,e') \;\&\; e \sqsubseteq e') \end{aligned}$$

Again, the injection maps are continuous. The generalisation of this construction to a set of domains is written $\sum_{i \in I} D_i$ (note that $\sum_{i=1}^{n} D_i$ is not equal to $D_1 + (D_2 + \cdots + D_n) \cdots)$).

Coalesced Sum If D, E are domains then we write $D \oplus E$ for the domain with carrier $\{(0,d) \mid d \in D, d \neq \bot_D\} \cup \{(1,e) \mid e \in E, e \neq \bot_E\} \cup \{\bot\}$ and the obvious order. This is a coproduct in $\mathcal{D}om_S$.

Hoare Powerdomain If D is a predomain then we write $\mathcal{P}_H(D)$ for the predomain which has the set of all non-empty Scott-closed subsets of D as its carrier and which is ordered by inclusion. This is a domain if D is, with least element $\{\bot_D\}$. The singleton map $\{\!| \cdot |\!\} : D \to \mathcal{P}_H(D)$ which sends d to $\overline{\{d\}}$ is continuous, as is the 'big union' map $\bigcup : \mathcal{P}_H(\mathcal{P}_H(D)) \to \mathcal{P}_H(D)$. $\mathcal{P}_H(\cdot)$ is a functor (given $f : D \to E$, $\mathcal{P}_H(f) : \mathcal{P}_H(D) \to \mathcal{P}_H(E)$ sends I to $\overline{f(I)}$) and forms a monad together with the natural transformations $\{\!| \cdot |\!\}$ and $\bigcup$.

A continuous map $f : D \to E$ between domains is said to be an *embedding* if there is a continuous map $g : E \to D$ such that $g \circ f = id_D$ and $f \circ g \sqsubseteq id_E$ (with respect to the pointwise ordering of the function space which was defined above). Such a g is called a *projection*. Embeddings and projections determine each other uniquely, and we shall write f^R for the projection corresponding to the embedding f, and call this the *right adjoint* of f. Dually, we write g^L for the embedding corresponding to the projection g, and call this the *left adjoint* of g. If $f : D \to E$ is an embedding then we shall sometimes write $f : D \lhd E$. Embeddings and projections are always strict, and we write $\mathcal{D}om_E$ for the non-full subcategory of $\mathcal{D}om_S$ with domains as objects and embeddings as morphisms. Embedding-projection pairs are a special case of the notion of a Galois connection between lattices, which is itself a special case of the general notion of an adjunction between functors. In Chapter 4, we will meet a slightly different definition of projections.

Denotational Semantics

The denotational semantics of Λ_T defines the meanings of terms using domains. There are two main styles in which this can be presented. The first is the 'categorical' way: types are interpreted as objects in $\mathcal{D}om$ and terms are interpreted as morphisms from the product of the objects representing the types of their free variables. The second way is more element-oriented and uses environments. We will give the semantics in the second form, as this seems more familiar to computer scientists.

We define a type-indexed family of domains $\{D_\sigma\}$ by induction on the structure of σ as follows:

$$D_\iota = \mathbb{N}_\bot$$

$$D_{\sigma \to \tau} = [D_\sigma \to D_\tau]$$

$$D_{\sigma \times \tau} = D_\sigma \times D_\tau$$

$$[\![\underline{n}]\!]\rho = [n]$$

$$[\![x^\sigma]\!]\rho = \rho(x^\sigma)$$

$$[\![ts]\!]\rho = ([\![t]\!]\rho)([\![s]\!]\rho)$$

$$[\![\lambda x^\sigma.t]\!]\rho = \lambda d \in D_\sigma.([\![t]\!]\rho[x^\sigma \mapsto d])$$

$$[\![(s,t)]\!]\rho = ([\![s]\!]\rho, [\![t]\!]\rho)$$

$$[\![\mathsf{fst}(s)]\!]\rho = \pi_1([\![s]\!]\rho)$$

$$[\![\mathsf{snd}(s)]\!]\rho = \pi_2([\![s]\!]\rho)$$

$$[\![\mathsf{if}\ s\ \mathsf{then}\ t_1\ \mathsf{else}\ t_2]\!]\rho = \begin{cases} \bot & \text{if } [\![s]\!]\rho = \bot_\iota \\ [\![t_1]\!]\rho & \text{if } [\![s]\!]\rho = [0] \\ [\![t_2]\!]\rho & \text{otherwise} \end{cases}$$

$$[\![\mathsf{fix}(x^\sigma.s)]\!]\rho = \bigsqcup_n d_n \text{ where } d_0 = \bot_{D_\sigma}, d_{n+1} = [\![s]\!]\rho[x^\sigma \mapsto d_n]$$

$$[\![s+t]\!]\rho = \begin{cases} \bot_\iota & \text{if } [\![s]\!]\rho = \bot \text{ or } [\![t]\!]\rho = \bot \\ [m+n] & \text{if } [\![s]\!]\rho = [m] \text{ and } [\![t]\!]\rho = [n] \end{cases}$$

Figure 2.4: Denotational Semantics of Λ_T

An *environment* ρ is a type-respecting finite partial function from variable names to the disjoint union of all the D_σ; that is to say that for any x^σ, if $\rho(x^\sigma)$ is defined then it is an element of D_σ. For an environment ρ, variable x^σ and $d \in D_\sigma$, we define the environment $\rho[x^\sigma \mapsto d]$ by

$$\rho[x^\sigma \mapsto d](y^\tau) = \begin{cases} d & \text{if } y^\tau = x^\sigma \\ \rho(y^\tau) & \text{otherwise} \end{cases}$$

The meaning of a term $t :: \tau$, which we write $[\![t]\!]$, is then a partial function from environments to D_τ. $[\![t]\!]$ is defined by induction on the structure of t as shown in Figure 2.4.

It is straightforward to verify that all the above are well-defined in the sense that if $t :: \tau$ and $\rho(x^\sigma)$ is defined for all the free variables x^σ of t, then $[\![t]\!]\rho$ is defined and is indeed an element of D_τ. From now on it should be understood when we write $[\![t]\!]\rho$ that $FV(t) \subseteq dom(\rho)$.

Computational Adequacy

Having given both an operational and a denotational semantics to Λ_T, we now look at how they are related. We shall take the position that the operational semantics is primary and that we wish to be sure that the denotational semantics correctly models the relevant features of the operational semantics so that we can use it to reason about evaluation. The results which we shall present are due to Plotkin [Plo77] and to Sazonov [Saz76].

The question we immediately have to answer is: what do we mean by 'relevant' features? The most important idea is that denotationally equal terms should be, in some sense, operationally equivalent. The correct notion of operational equivalence will depend on what observations we are allowed to make of computations. The view which we shall take is that the only directly observable behaviour is that of *programs*, where a program is defined to be a closed term of type ι (*i.e.* an element of $\Lambda_T^{\circ,\iota}$). The only behaviours which a program can exhibit are to diverge or to converge to an integer value. The only way in which we can observe terms of higher type is to plug them into a complete program. In particular, this means that the only kind of 'test' which we can perform on a term of functional type is to apply it. We shall return to this issue later when we come to consider strictness-based optimisations.

We define a *context* $C[]$, somewhat informally, to be a program with a typed hole in it. For example

$$\lambda f^{\iota\to\iota}.f^{\iota\to\iota}([]^{\iota\to\iota}\underline{3})[]^{\iota\to\iota}$$

If $C[]$ is a context and t is a term of the appropriate type then $C[t]$ is the program which arises from plugging t in for all the occurrences of the hole in $C[]$. Note that this differs from the notion of substitution which we defined earlier in that here we are allowing variable capture. A further restriction for this to be well-defined is that all the free variables of t are bound in $C[t]$.

The *observational preorder* $\preceq$ is the relation on terms given by

$$t \preceq s \Leftrightarrow \forall C[].\,\forall n.\; C[t]{\Downarrow}\underline{n} \Rightarrow C[s]{\Downarrow}\underline{n}$$

This gives us an operational notion of when one term approximates another, which we need to relate to the denotational order.

The following facts about the denotational semantics are easily verified:

Lemma 2.1.2

1. *If* $\rho(x^\sigma) \sqsubseteq \rho'(x^\sigma)$ *for all* $x^\sigma \in FV(t)$ *then* $[\![t]\!]\rho \sqsubseteq [\![t]\!]\rho'$;
2. $[\![t[s/x^\sigma]]\!]\rho = [\![t]\!]\rho[x^\sigma \mapsto [\![s]\!]\rho]$;
3. *If* $[\![s]\!]\rho \sqsubseteq [\![t]\!]\rho$ *for all* ρ, *then for all* $C[]$, $[\![C[s]]\!] \sqsubseteq [\![C[t]]\!]$.

□

Now define the *observational* or *operational* meaning of a program p by

$$\mathcal{O}[\![p]\!] = \begin{cases} [n] & \text{if } p \Downarrow \underline{n} \\ \bot & \text{otherwise} \end{cases}$$

The result which we want in order to show that the two semantics agree is $\forall p \in \Lambda_T^{\circ,\iota}.\ [\![p]\!] = \mathcal{O}[\![p]\!]$. One direction of this is straightforward:

Proposition 2.1.3 *For any program p, if $p \Downarrow \underline{n}$ then $[\![p]\!] = [n]$. Equivalently, $\mathcal{O}[\![p]\!] \sqsubseteq [\![p]\!]$.*

Proof. This follows by an induction on derivations in the operational semantics to show (using Lemma 2.1.2(2)) that for all $t \in \Lambda_T^\circ$, if $t \Downarrow c$ then $[\![t]\!] = [\![c]\!]$. Then if $p \Downarrow \underline{n}$ we have $[\![p]\!] = [\![\underline{n}]\!] = [n]$ as required. □

The other direction is more difficult. A naive attempt to do induction on types or on terms fails. We need to assume a strengthened induction hypothesis to make the proof go through at higher types. We will do this by means of a *logical relation.* Logical relations are an important tool for proving properties of typed λ-terms, and we shall use them several times in this thesis. A good introduction can be found in [Mit90]. There are several closely related notions, such as computability predicates and reducibility candidates.

Define the type-indexed family $\{\mathcal{R}^\sigma\}$ of relations, where $\mathcal{R}^\sigma \subseteq D_\sigma \times \Lambda_T^{\circ,\sigma}$ as follows:

$$d\,\mathcal{R}^\iota\,t \;\Leftrightarrow\; d \sqsubseteq \mathcal{O}[\![t]\!]$$

$$d\,\mathcal{R}^{\sigma\to\tau}\,t \;\Leftrightarrow\; \forall e, s.\ e\,\mathcal{R}^\sigma\,s \;\Rightarrow\; (d\,e)\,\mathcal{R}^\tau\,(t\,s)$$

$$d\,\mathcal{R}^{\sigma\times\tau}\,t \;\Leftrightarrow\; (\pi_1(d)\,\mathcal{R}^\sigma\,\mathsf{fst}(t)) \;\&\; (\pi_2(d)\,\mathcal{R}^\tau\,\mathsf{snd}(t))$$

Lemma 2.1.4

1. *For all $t \in \Lambda_T^{\circ,\tau}$ we have $\bot_{D_\tau}\,\mathcal{R}^\tau\,t$;*
2. *If $\langle d_n \rangle$ is a chain in D_τ and for all n, $d_n\,\mathcal{R}^\tau\,t$ then $(\bigsqcup_n d_n)\,\mathcal{R}^\tau\,t$.*

Proof. Both parts are proved by simple induction on the structure of the type τ. □

Lemma 2.1.5 *For $t, t' \in \Lambda_T^{\circ,\tau}$ and $d \in D_\tau$, if $(t \Downarrow c) \Rightarrow (t' \Downarrow c)$ then $(d\mathcal{R}^\tau t) \Rightarrow (d\mathcal{R}^\tau t')$.*

Proof. This also follows by induction on types and consideration of the operational semantics. We give the case for function spaces:

- If $\tau = \sigma \to \sigma'$ then we need to show $d\,\mathcal{R}^{\sigma\to\sigma'}\,t'$ on the assumption $d\,\mathcal{R}^{\sigma\to\sigma'}\,t$. That means showing that for all d', s such that $d'\,\mathcal{R}^{\sigma}\,s$ we have $(d\,d')\,\mathcal{R}^{\sigma'}\,(t'\,s)$ on the assumption $(d\,d')\,\mathcal{R}^{\sigma'}\,(t\,s)$. This follows by induction if we can show $(t\,s)\Downarrow c' \Rightarrow (t'\,s)\Downarrow c'$; but this is immediate from the rule for application in the operational semantics and the premisses of the lemma.

□

Proposition 2.1.6 *If $t \in \Lambda^{\tau}_{T}$, ρ is an environment such that $FV(t) \subseteq dom(\rho)$ and for each $x_i^{\sigma_i} \in dom(\rho)$, $s_i \in \Lambda^{\circ,\sigma_i}_{T}$ and $(\rho(x_i^{\sigma_i}))\,\mathcal{R}^{\sigma_i}\,s_i$ then $([\![t]\!]\rho)\,\mathcal{R}^{\tau}\,(t[s_i/x_i^{\sigma_i}])$.*

Proof. Induction on the structure of t. We give a few illustrative cases:

- If $t = \underline{n}$ then $[\![t]\!]\rho = [n]$ and it is plain that $[n]\,\mathcal{R}^{\iota}\,\underline{n}$.
- If $t = \lambda x^{\sigma}.t'$ where $t' :: \tau'$ then we need to show that for any d, s such that $d\mathcal{R}^{\sigma}s$ we have $([\![\lambda x^{\sigma}.t']\!]\rho)(d)\,\mathcal{R}^{\tau'}\,((\lambda x^{\sigma}.t')[s_i/x_i^{\sigma_i}])\,s$. The first of these expressions is equal to $[\![t']\!]\rho[x^{\sigma} \mapsto d]$. Then since $t'[s_i/x_i^{\sigma_i}][s/x^{\sigma}]\Downarrow c \Rightarrow (\lambda x^{\sigma}.t'[s_i/x_i^{\sigma_i}])\,s\Downarrow c$, we can use the induction hypothesis on t' together with Lemma 2.1.5 to obtain the result.
- If $t = \mathsf{if}\ s\ \mathsf{then}\ u\ \mathsf{else}\ v$ where $u, v :: \tau$ then we consider the possible cases for $[\![s]\!]\rho$. If this is $\bot_{D_\iota}$ then $[\![t]\!]\rho = \bot_{D_\tau}$ and we are done by Lemma 2.1.4. Otherwise, assume $[\![s]\!]\rho = [0]$ so that the induction hypothesis applied to s tells us that $s[s_i/x_i^{\sigma_i}]\Downarrow\underline{0}$. Then $[\![t]\!]\rho = [\![u]\!]\rho$ so that the induction hypothesis applied to u, together with the fact that $u[s_i/x_i^{\sigma_i}]\Downarrow c \Rightarrow t[s_i/x_i^{\sigma_i}]\Downarrow c$ and Lemma 2.1.5 gives the result in this case. The argument for the case in which $[\![s]\!]\rho = [n]$ for $n > 0$ is similar.
- If $t = \mathsf{fix}(x^{\tau}.t')$ where $t' :: \tau$ then $[\![t]\!]\rho = \bigsqcup_n d_n$ where $d_0 = \bot_{D_\tau}$ and $d_{n+1} = [\![t']\!]\rho[x^{\tau} \mapsto d_n]$. Lemma 2.1.4 tells us that $d_0\,\mathcal{R}^{\tau}\,t[s_i/x_i^{\sigma_i}]$. Assume that $d_n\,\mathcal{R}^{\tau}\,t[s_i/x_i^{\sigma_i}]$. Then the induction hypothesis applied to t' tells us that

  $$([\![t']\!]\rho[x^{\tau} \mapsto d_n])\,\mathcal{R}^{\tau}\,t'[s_i/x_i^{\sigma_i}][t[s_i/x_i^{\sigma_i}]/x^{\tau}]$$

  But $[\![t']\!]\rho[x^{\tau} \mapsto d_n] = d_{n+1}$ and $t'[s_i/x_i^{\sigma_i}][t[s_i/x_i^{\sigma_i}]/x^{\tau}]\Downarrow c \Rightarrow t[s_i/x_i^{\sigma_i}]\Downarrow c$ so that by Lemma 2.1.5 we have $d_{n+1}\mathcal{R}^{\tau}t[s_i/x_i^{\sigma_i}]$ for all n by ordinary induction. Thus by Lemma 2.1.4 we are done.

□

Corollary 2.1.7 *For any program p, $\mathcal{O}[\![p]\!] = [\![p]\!]$.*

Proof. The previous proposition gives $[\![p]\!]\,\mathcal{R}^{\iota}\,p$ which means that $[\![p]\!] \sqsubseteq \mathcal{O}[\![p]\!]$. The other direction is Proposition 2.1.3. □

Corollary 2.1.8 (Computational Adequacy) *If* $\forall\rho.[\![s]\!]\rho \sqsubseteq [\![t]\!]\rho$ *then* $s \preceq t$.

Proof. By Lemma 2.1.2, for all $C[]$, $[\![C[s]]\!] \sqsubseteq [\![C[t]]\!]$. Hence by the previous result, $C[s]{\Downarrow}\underline{n} \Rightarrow C[t]{\Downarrow}\underline{n}$. □

It should be noted that for the language as we have presented it, the converse to computational adequacy, which is known as *full abstraction*, is false. There are terms s, t for which $s \preceq t$ but it is not the case that $\forall\rho.[\![s]\!]\rho \sqsubseteq [\![t]\!]\rho$. This is essentially because Λ_T is *sequential* whereas the denotational semantics contains functions which are inherently parallel (and which are not therefore the denotation of any term in the language). In terms of equality, rather than approximation, some terms can be distinguished denotationally by contexts involving these parallel functions but not by any sequential context. This means that if we base a system for reasoning about observational equality on the denotational semantics then it will be incomplete. Conversely, if we use the denotational semantics to reason that two terms are observationally different, our results may be unsound. In practice, however, we nearly always want to reason that two terms are equivalent and are prepared to live with some incompleteness. Note that if we do wish to show the observational inequivalence of two terms, this is usually easy to do using the operational semantics directly, as all we have to do is exhibit a suitable context.

The simplest way in which we can repair the failure of full abstraction is to add a parallel operator, such as parallel or, or parallel conditional, to our language. Whilst we shall not do this, it makes very little difference to any of the material in this thesis. The other approach, defining a useful semantics which *is* fully abstract for the language as we have defined it, appears extremely difficult. Despite considerable effort, the problem of obtaining a 'semantic characterization' of the[3] fully abstract model remains open. For further information on the full abstraction problem, see for example [Mil77, Plo77, BCL85, Sto88].

2.2 Strictness Analysis and Abstract Interpretation

A function $f : D \to E$ is strict if $f(\bot_D) = \bot_E$. A function of n arguments $g : D_1 \times \ldots \times D_n \to E$ is *strict in its ith argument* if $\forall d_j \in D_j$ we have

$$g(d_1, \ldots, d_{i-1}, \bot_{D_i}, d_{i+1}, \ldots, d_n) = \bot_E$$

In its simplest form, strictness analysis is the process of trying to find out which functions in a program are strict. Strictness is the denotational analogue of the operational notion of *neededness*, where we say a function g needs its ith argument if whenever an application of g is reduced, so is its ith argument. It is easy to

[3]Milner has shown that there is only one inequationally fully abstract order-extensional model for PCF [Mil77].

see that neededness implies strictness but not the other way around—the function $\lambda x^{\sigma}.\Omega^{\tau}$ is strict but does not actually evaluate its argument under the operational semantics which we have given. For most purposes, strictness is actually the more useful notion.

Strictness information has several uses. We briefly indicate some of these here:

- Transforming call-by-name parameter passing into call-by-value. We shall say a little about this in Chapter 6.
- Indicating which expressions should be *sparked* as concurrent tasks in a parallel implementation. See, for example, [PJ87, Bur87, HBPJ86, Bur91b].
- Program transformation. For example, the useful rule

  $$f(\mathsf{if}\ a\ \mathsf{then}\ b\ \mathsf{else}\ c) = \mathsf{if}\ a\ \mathsf{then}\ f(b)\ \mathsf{else}\ f(c)$$

  is only valid in general if f is strict.
- Improving the accuracy of other analyses, such as complexity analysis [Wad88].

Accurate strictness information is uncomputable (the problem is obviously equivalent to the halting problem), so if we wish to automate the process in a compiler then we shall have to accept safe approximations. This means that the analysis should indicate that a function is strict only if it definitely is.

Many program analysis problems can be approached using the framework of *abstract interpretation*. This was pioneered by the Cousots [CC77, CC79], and was first applied to strictness analysis by Mycroft [Myc80, Myc81]. The basic idea of abstract interpretation is very simple: we can find things out about a computation by running a 'simplified' version of it at compile-time. This simplified, or *abstract*, version of the program computes with (lattices of) program properties.

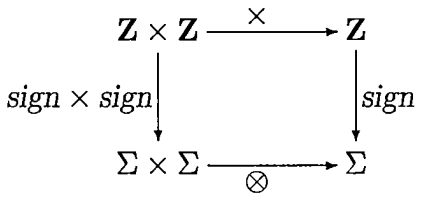
If we are interested in analysing extensional properties of programs then we can base our abstract interpretation on a denotational semantics like that which we have already seen for Λ_T. This is what we shall do for strictness analysis. If we wished to reason about intensional properties, such as complexity or sharing behaviour, then we would have to define some more detailed semantics which included this information before we could construct an abstract interpretation.

2.2.1 A Simple Example

The usual simple of example of an abstract interpretation is the so-called 'rule of signs' for simple arithmetic.

We can answer the question 'what is the sign of -314×159?' without actually performing the multiplication by using the fact that 'a negative times a positive

is a negative'. More formally, there is an abstract version, $\otimes$, of the standard multiplication operation, $\times$, which makes the following diagram commute:



where $\Sigma = \{-, 0, +\}$, $sign : \mathbf{Z} \to \Sigma$ maps an integer to its sign, and $\otimes$ is defined by the following table:

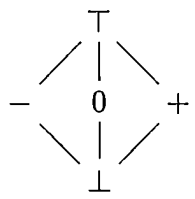
$\otimes$	$-$	0	$+$
$-$	$+$	0	$-$
0	0	0	0
$+$	$-$	0	$+$

The diagram says that $sign(n \times m) = sign(n) \otimes sign(m)$ – instead of doing the multiplication and taking the sign, we can take the two signs and do the abstract version of multiplication. In algebraic terms, this says that *sign* is a homomorphism between two algebras for a signature containing a constant for each integer and one binary operation.

Now consider extending this framework to include addition as well as multiplication. We have to add a 'don't know' element to Σ, which corresponds to the whole of $\mathbf{Z}$. We shall write $\top$ for this element. For the sake of completeness, we shall also add an element $\bot$ to correspond to the empty set of integers. The intended interpretations of the abstract points are given by the *concretisation* map $Conc : \Sigma \to \mathcal{P}(\mathbf{Z})$ defined by

$$Conc(s) = \begin{cases} \{\} & \text{if } s = \bot \\ \{0\} & \text{if } s = 0 \\ \{x \mid x > 0\} & \text{if } s = + \\ \{x \mid x < 0\} & \text{if } s = - \\ \mathbf{Z} & \text{if } s = \top \end{cases}$$

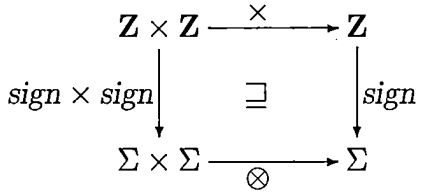
The inclusion order on $\mathcal{P}(\mathbf{Z})$ induces the following order on Σ:



The new definitions of the abstract operators $\otimes$ and $\oplus$ are given by the obvious tables. We list a few representative entries

$$\begin{array}{ll} 0 \otimes \top = 0 & - \oplus + = \top \\ - \otimes \top = \top & \top \oplus \top = \top \\ \bot \otimes + = \bot & 0 \oplus \bot = \bot \end{array}$$

The correctness of our abstract interpretation is now expressed by the following diagram:



Note that because of the approximation which we have been forced to introduce, the diagram no longer commutes exactly. Instead, the abstract value which we get by first taking signs and then performing the abstract computation is greater than or equal to (with respect to the order shown above) the abstract value we get by doing the real computation and then taking the sign. This is sometimes known as a *semi-homomorphism.*

We can also define a function $Sign : \mathcal{P}(\mathbf{Z}) \to \Sigma$ which allows us to abstract *sets* of integers. This is given by $Sign = \bigsqcup \circ \mathcal{P}(sign)$. The pair of maps *Sign* and *Conc* satisfy the following two conditions:

$$Sign \circ Conc = id_{\Sigma}$$

$$Conc \circ Sign \sqsupseteq id_{\mathcal{P}(Z)}$$

The second of these is called the *safety* conditon. Taken together, these two conditions say that the two maps form what is sometimes called an *embedding-closure pair.* This is dual to the notion of an embedding-projection pair which we defined earlier.

Other everyday examples of abstract interpretation include order of magnitude estimates, and the use of a map to plan a journey.

2.2.2 Strictness Analysis by Abstract Interpretation

Mycroft's original account of strictness analysis by abstract interpretation was restricted to first-order recursion equations over flat domains. The natural extension to higher-order functions was first described by Burn, Hankin and Abramsky in [BHA86]. This section, which is based on that paper, explains how abstract interpretation may be used to perform strictness analysis for our language Λ_T. Some of the details of the presentation are slightly different from those in *loc cit.*

We start by defining an abstract domain A_σ at each type σ

$$A_\iota = \mathbf{2}$$

$$A_{\sigma \times \tau} = A_\sigma \times A_\tau$$

$$A_{\sigma \to \tau} = [A_\sigma \to A_\tau]$$

In fact each A_σ is rather more than a domain: it is a finite (and therefore complete) lattice.

The intended relationship between the standard and abstract domains at each type is captured by the continuous maps $\alpha_\sigma : D_\sigma \to A_\sigma$, $Abs_\sigma : \mathcal{P}_H(D_\sigma) \to A_\sigma$ and $Conc_\sigma : A_\sigma \to \mathcal{P}_H(D_\sigma)$. These are defined inductively as follows

$$Abs_\sigma = \bigsqcup \circ (\mathcal{P}_H(\alpha_\sigma))$$

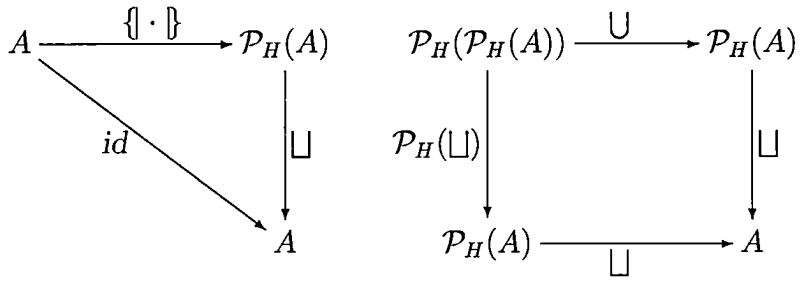
$$\alpha_\iota(\bot_{D_\iota}) = \bot_{A_\iota} \qquad \alpha_\iota([n]) = \top_{A_\iota}$$

$$\alpha_{\sigma\times\tau}(d, e) = (\alpha_\sigma(d), \alpha_\tau(e))$$

$$\alpha_{\sigma\to\tau}(f) = Abs_\tau \circ (\mathcal{P}_H(f)) \circ Conc_\sigma$$

$$Conc_\sigma(x) = \{d \in D_\sigma \mid \alpha_\sigma(d) \sqsubseteq x\}$$

Note that the definition of Abs uses $\bigsqcup : \mathcal{P}_H(A_\sigma) \to A_\sigma$. This map exists because A_σ is a complete lattice. $(A_\sigma, \bigsqcup)$ is an algebra for the $\mathcal{P}_H(\cdot)$ monad, which means that the following two diagrams commute:



The maps $Conc$ and Abs form an adjunction:

$$Conc_\sigma \circ Abs_\sigma \sqsupseteq id_{\mathcal{P}_H(D_\sigma)}$$

$$Abs_\sigma \circ Conc_\sigma \sqsubseteq id_{A_\sigma}$$

and the second inequality can in fact be strengthened to an equality.

The non-standard interpretation of Λ_T is very similar to the standard denotational semantics which we gave earlier. We define an environment for the abstract interpretation (ranged over by ρ^A) to be a finite type-respecting partial function from variables to the disjoint union of all the A_σ, and we can then define the abstract semantics $[\![t]\!]^A\rho^A \in A_\sigma$ of a term $t :: \sigma$ as shown in Figure 2.5. We shall refer to this system as the 'BHA-style abstract interpretation' in the rest of the thesis.

The abstract semantics differs from the standard semantics only in the choice of domain for the ground type and in the interpretations of numeric constants, conditionals and arithmetic operations. This can be generalised in a fairly obvious way to give the notion of an *interpretation* of our language, which is specified by a choice of base domain and by semantic equations for those language constructs which are not part of the pure simply typed lambda calculus.

The following lemma concerning the abstraction maps is easily established by induction on types, and will be useful later:

Lemma 2.2.1 *For each σ, α_σ is both strict and bottom-reflecting; that is to say $\alpha_\sigma(d) = \bot_{A_\sigma} \Leftrightarrow d = \bot_{D_\sigma}$.* □

$$[\![\underline{n}]\!]^A\rho^A = \top_{A_\iota}$$

$$[\![x^\sigma]\!]^A\rho^A = \rho^A(x^\sigma)$$

$$[\![ts]\!]^A\rho^A = ([\![t]\!]^A\rho^A)([\![s]\!]^A\rho^A)$$

$$[\![\lambda x^\sigma.t]\!]^A\rho^A = \lambda a \in A_\sigma.([\![t]\!]^A\rho^A[x^\sigma \mapsto a])$$

$$[\![(s,t)]\!]^A\rho^A = ([\![s]\!]^A\rho^A, [\![t]\!]^A\rho^A)$$

$$[\![\mathsf{fst}(s)]\!]^A\rho^A = \pi_1([\![s]\!]^A\rho^A)$$

$$[\![\mathsf{snd}(s)]\!]^A\rho^A = \pi_2([\![s]\!]^A\rho^A)$$

$$[\![\mathsf{if}\ s\ \mathsf{then}\ t_1\ \mathsf{else}\ t_2]\!]^A\rho^A = \begin{cases} \bot & \text{if } [\![s]\!]^A\rho^A = \bot_{A_\iota} \\ ([\![t_1]\!]^A\rho^A) \sqcup ([\![t_2]\!]^A\rho^A) & \text{otherwise} \end{cases}$$

$$[\![\mathsf{fix}(x^\sigma.s)]\!]^A\rho^A = \bigsqcup_n a_n \text{ where } a_0 = \bot_{A_\sigma}, a_{n+1} = [\![s]\!]^A\rho^A[x^\sigma \mapsto a_n]$$

$$[\![s+t]\!]^A\rho^A = ([\![s]\!]^A\rho^A) \sqcap ([\![t]\!]^A\rho^A)$$

Figure 2.5: BHA-style Abstract Interpretation

We shall also want the following:

Lemma 2.2.2 *For any $f \in D_{\sigma\to\tau}$ and $x \in D_\sigma$, $\alpha_{\sigma\to\tau}(f)(\alpha_\sigma(x)) \sqsupseteq \alpha_\tau(fx)$*

Proof. This uses naturality of $\{\!|\cdot|\!\}$ and the fact that $\bigsqcup \circ \{\!|\cdot|\!\}$ is the identity on A_σ. □

Since all the abstract domains are finite, we can compute the abstract denotation of a term in finite time. What we need to show is that the abstract value we get by doing this is a safe approximation to the abstraction of the standard denotation of the term. For ρ a standard environment and ρ^A an abstract environment, write $\alpha(\rho) \sqsubseteq \rho^A$ to mean that for all $x^\sigma \in dom(\rho)$, $\alpha_\sigma(\rho(x^\sigma)) \sqsubseteq \rho^A(x^\sigma)$.

Theorem 2.2.3 *If $\alpha(\rho) \sqsubseteq \rho^A$ then for any $t :: \tau$, $\alpha_\tau([\![t]\!]\rho) \sqsubseteq [\![t]\!]^A\rho^A$.*

Proof. This is proved by a fairly straightforward structural induction on terms. □

In particular, this shows that if the abstract interpretation of a term of functional type is strict, so is its standard interpretation:

Theorem 2.2.4 *If $\alpha(\rho) \sqsubseteq \rho^A$ and $f :: \sigma \to \tau$ then $[\![f]\!]^A\rho^A\bot_{A_\sigma} = \bot_{A_\tau}$ implies $[\![f]\!]\rho\bot_{D_\sigma} = \bot_{D_\tau}$.*

Proof.

$$\begin{array}{lcll}
\bot_{A_\tau} & = & ([\![f]\!]^A\rho^A)\bot_{A_\sigma} & \\
& \sqsupseteq & (\alpha_{\sigma\to\tau}([\![f]\!]\rho))\bot_{A_\sigma} & \text{by previous theorem} \\
& = & (\alpha_{\sigma\to\tau}([\![f]\!]\rho))(\alpha_\sigma(\bot_{D_\sigma})) & \text{as } \alpha_\sigma \text{ is strict} \\
& \sqsupseteq & \alpha_\tau(([\![f]\!]\rho)\bot_{D_\sigma}) & \text{by Lemma 2.2.2}
\end{array}$$

which implies that

$$[\![f]\!]\rho\bot_{D_\sigma} = \bot_{D_\tau}$$

as α_τ is bottom-reflecting. □

Example The abstract denotation of the factorial function

$$\mathsf{fact} \stackrel{\text{def}}{=} \mathsf{fix}(f^{\iota\to\iota}.\lambda x^\iota.\mathsf{if}\ x^\iota\ \mathsf{then}\ \underline{1}\ \mathsf{else}\ x^\iota * (f^{\iota\to\iota}(x^\iota - \underline{1})))$$

can be calculated as

$$[\![\mathsf{fact}]\!]^A = \bigsqcup_n a_n$$

where

$$a_0 = \bot_{A_{\iota\to\iota}}$$

and

$$\begin{array}{lcl}
a_{n+1} & = & [\![\lambda x^\iota.\mathsf{if}\ x^\iota\ \mathsf{then}\ \underline{1}\ \mathsf{else}\ x^\iota * (f^{\iota\to\iota}(x^\iota - \underline{1}))]\!]^A[f^{\iota\to\iota} \mapsto a_n] \\
& = & \lambda y \in A_\iota.\left\{\begin{array}{ll} \bot_{A_\iota} & \text{if } y = \bot_{A_\iota} \\ \top_{A_\iota} \sqcup (y \sqcap (a_n(y \sqcap \top_{A_\iota}))) & \text{otherwise} \end{array}\right. \\
& = & \lambda y \in A_\iota.y
\end{array}$$

This means that

$$[\![\mathsf{fact}]\!]^A = \lambda y \in A_\iota . y$$

so that

$$[\![\mathsf{fact}]\!]^A \bot_{A_\iota} = \bot_{A_\iota}$$

and therefore the standard denotation of the factorial function is strict.

In general, when analysing recursive definitions, we have to perform several steps of fixpoint iteration (this did not show up in the example because a_{n+1} was independent of a_n but this is not usually the case). This can be very expensive as even though all the lattices are finite, they get very large as we go up the type hierarchy. Note that to perform fixpoint iteration for higher-order functions, we have to test functions for equality at every step and this is in itself a costly operation. In fact, even for first-order functions, calculating the abstract function is complete for deterministic exponential time in the number of arguments [YH86]. Several techniques have been proposed for implementing analysers with better average case behaviour than this. See, for example, [YH86, Hun89, HH90, Hun91].

There is a different way of phrasing the correctness proof for this abstract interpretation which makes use of a logical relation between the standard and the abstract domains at each type. Instead of the concretisation map $Conc_\iota : A_\iota \to \mathcal{P}_H(D_\iota)$, we define a relation $\mathcal{A}^\iota \subseteq A_\iota \times D_\iota$ which relates $\bot_{A_\iota}$ to $\bot_{D_\iota}$ and $\top_{A_\iota}$ to everything in D_ι. This is then extended to a family $\{\mathcal{A}^\sigma\}$ of relations between the abstract and standard domains as follows:

$$f \mathcal{A}^{\sigma\to\tau} g \;\Leftrightarrow\; \forall a \in A_\sigma, d \in D_\sigma .\; a \,\mathcal{A}^\sigma\, d \;\Rightarrow\; (f\, a)\, \mathcal{A}^\tau\, (g\, d)$$

$$p \mathcal{A}^{\sigma\times\tau} q \Leftrightarrow (\pi_1(p) \mathcal{A}^\sigma \pi_1(q)) \wedge (\pi_2(p) \mathcal{A}^\tau \pi_2(q))$$

The 'basic lemma' (or 'fundamental theorem') of logical relations says that if we have a family of relations like this between two interpretations of our language, and that the extra language constructs and constants are related, then the meaning of any term in the first interpretation is related to its meaning in the second (provided the two environments are pointwise related). This allows correctness to be proved without any explicit mention of powerdomains, although, in this particular case, the broad outline of the proof is very similar to that of the one we have given. This version of the semantic basis for BHA abstract interpretation appears in [Abr90a] and [Nie86]. See also [MJ85].

2.3 Static Analysis and Type Inference

Most modern functional language implementations include *type inference*. This means that the programmer does not have to supply any explicit type information (except perhaps to disambiguate uses of overloaded built-in operators). The compiler attempts automatically to reconstruct the types of expressions in the program, and reports any type errors which it finds. In general, these languages also have polymorphic type systems based on Milner's extension [Mil78] of Hindley's system

[Hin69]. The Hindley-Milner type system (often just known as the 'ML type system') allows a single definition to be used at several different type instances. Languages with polymorphic type inference offer the programmer much of the convenience of using an untyped language whilst retaining the considerable advantages of strong typing. We shall consider polymorphism further in Chapter 5.

Effective type inference systems can never be complete with respect to an interpretation of types in a model of the untyped language for elementary computability reasons. We have, therefore, to settle for safe approximations—if the compiler reports that a term has a particular type then it definitely has that type, but some well-typed programs will be rejected.

It is clear that type inference is, at least superficially, similar to the static analysis problems which arise in optimising compilers. Both involve compile-time inference of safe, approximate information about the run-time behaviour of programs. Several people have noticed this and have presented type inference as an example of an abstract interpretation [MJ85, Kie87].

There is a slight pragmatic difference between the two sorts of analysis which is that if an inference algorithm cannot type a program then it reports an error and then stops. If an optimiser cannot discover that a program has a particular property (*e.g.* 'the argument of this array update function will not be shared') then it just behaves in some default manner (*e.g.* compiling code to copy the array). Thus analyses for optimisations are rather more like *partial* type inference for untyped languages such as LISP or Smalltalk. These languages are not statically typed and must therefore carry around and check type tags at run-time. Performance can be improved by inferring *some* type information at compile-time so that at least some of the code is free of such overheads. Indeed, the connection between static analysis and partial type inference is very close—Gomard [Gom90] presents an algorithm for partial type inference and shows how it may also be used for binding time analysis.

There are a number of reasons why we might want to investigate the link between type inference and abstract interpretation, with an emphasis on the use of type inference methods for performing program analyses. The most important of these is simply to increase our understanding, but there are several potential practical benefits. These include the possibility of getting more control over the trade-off between the complexity of an analysis and the quality of information gathered, and of developing analyses which have better average-case complexity than that of abstract interpretation. We might also want to develop analyses which can be performed at the same time as type inference, or can cope with polymorphism. Finally, we may be able to devise analysers which can obtain information which we could not get before. At present these are mostly rather vague hopes, rather than proven benefits of the type inference approach.

2.3.1 Kuo and Mishra's Strictness Type System

The use of type inference techniques for strictness analysis was first suggested by Kuo and Mishra in [KM89]. Their non-standard type system is shown in Figure 2.6.

Terms

$$t ::= x \mid \underline{n} \mid \mathsf{plus} \mid \mathsf{cond} \mid \mathsf{Y} \mid (t\,t) \mid \lambda x.t$$

Types

$$\sigma ::= \epsilon \mid \sigma \to \sigma$$
$$\epsilon ::= \emptyset \mid \Box$$

Type Inference

$$\Gamma, x : \sigma \vdash x : \sigma \qquad \Gamma \vdash \underline{n} : \Box$$

$$\Gamma \vdash \mathsf{plus} : \emptyset \to \Box \to \emptyset \qquad \Gamma \vdash \mathsf{plus} : \Box \to \emptyset \to \emptyset$$

$$\Gamma \vdash \mathsf{cond} : \emptyset \to \Box \to \Box \to \emptyset \quad \Gamma \vdash \mathsf{cond} : \Box \to \sigma \to \sigma \to \sigma$$

$$\Gamma \vdash \mathsf{Y} : (\sigma \to \sigma) \to \sigma$$

$$\frac{\Gamma \vdash t : \sigma' \to \sigma \qquad \Gamma \vdash t' : \sigma'}{\Gamma \vdash tt' : \sigma}$$

$$\frac{\Gamma, x : \sigma \vdash t : \sigma'}{\Gamma \vdash \lambda x.t : \sigma \to \sigma'}$$

$$\frac{\Gamma \vdash t : \sigma \qquad \sigma \subseteq \sigma'}{\Gamma \vdash t : \sigma'}$$

Subtyping

$$\sigma \subseteq \sigma \qquad \emptyset \subseteq \Box$$

$$\frac{\sigma \subseteq \sigma' \qquad \sigma' \subseteq \sigma''}{\sigma \subseteq \sigma''}$$

$$\frac{\sigma' \subseteq \sigma \qquad \tau \subseteq \tau'}{\sigma \to \tau \subseteq \sigma' \to \tau'}$$

$$\frac{\forall i.\sigma_i \text{ matches } \tau_i}{\sigma_1 \to \ldots \to \sigma_n \to \epsilon \subseteq \tau_1 \to \ldots \to \tau_n \to \Box}$$

Matching

$$\epsilon \text{ matches } \epsilon'$$

$$\frac{\sigma \text{ matches } \sigma' \qquad \tau \text{ matches } \tau'}{\sigma \to \tau \text{ matches } \sigma' \to \tau'}$$

Figure 2.6: Kuo and Mishra's Strictness Type System

The intention is that $\Box$ should denote 'all terms' whilst $\emptyset$ denotes 'looping terms'. For example, the set of all strict functions is represented by $\emptyset \to \emptyset$. There is clearly an inclusion between the two base types: $\emptyset \subseteq \Box$, and the type inference rules therefore have to incorporate *subtyping*. The paper is mainly concerned with obtaining an inference algorithm and draws on Fuh and Mishra's earlier work on implementing type inference in the presence of Mitchell-style subtyping [Mit84, FM88, FM89].

One of the complications in this system is that the subtyping rules differ from those considered by Mitchell in that they are *non-structural*. This means that an inclusion between two non-basic types cannot always be broken down into inclusions between their components. The usual rule for subtyping the $\to$ type constructor is

$$\frac{\sigma' \subseteq \sigma \qquad \tau \subseteq \tau'}{\sigma \to \tau \subseteq \sigma' \to \tau'}$$

which is indeed one of Kuo and Mishra's rules. Notice that this makes $\to$ antimonotonic in its first argument and monotonic in its second argument with respect to the inclusion ordering, which fits with one's intuition. This rule is not, however, sufficient to derive all the inclusions which we want. For example, we expect $\emptyset \to \emptyset \subseteq \Box \to \Box$ to hold ('every strict function maps terms to terms'), but this is not an instance of the above rule. This is the reason for the addition of their final, rather odd-looking, subtyping rule and the rules for type matching (which say when two types 'have the same shape'). Although there is no discussion of semantics or correctness, Kuo and Mishra do mention that there are other obvious but 'non-matching' inclusions, such as $\emptyset \to \emptyset \subseteq \Box$, which are not captured by their inference rules. It is therefore not really clear whether the analysis is for typed terms or untyped terms, although the paper states that the analysis is independent of any conventional type system, but that the analysis *algorithm* requires terms to be typeable (presumably in the simply typed lambda calculus sense) for it to work.

Kuo and Mishra's type system gives weaker information than the BHA-style abstract interpretation which we have previously described. This is pointed out in the paper with regard to the analysis of recursion, but in fact the system gives weaker results even for non-recursive terms. For example, consider

$$g \stackrel{\text{def}}{=} (\lambda f.\lambda x.\lambda y.\mathsf{cond}\, x\, (f \underline{1} y)\, (f y \underline{1}))\, \mathsf{plus}$$

We should like to be able to show g to be strict in its second argument, *i.e.*

$$_{/} \vdash g : \Box \to \emptyset \to \emptyset$$

but this is not derivable in the type system, although the corresponding fact *is* derivable from the abstract interpretation (modulo making trivial changes of syntax and adding type information to translate the above into Λ_T). In this example, the weakness is caused by the fact that plus has two minimal strictness types but we cannot derive the required property of g from either one of them alone. This is related to the restriction that λ-bound variables behave monomorphically in the

Hindley-Milner type system. Kuo and Mishra mention the possibility of adding intersection[4] to their system, and say:

> "We find that including boolean operators in the type language results in an extremely rich and complex language of types. This complicates several algorithms used in our strictness analysis technique. It also appears to provide a more general framework than can be used by our analysis techniques."

In fact, adding intersection types to Kuo and Mishra's system in the natural way, by adding the rules

$$\frac{\Gamma \vdash t : \sigma \qquad \Gamma \vdash t : \sigma'}{\Gamma \vdash t : \sigma \wedge \sigma'} \qquad \frac{\Gamma \vdash t : \sigma \wedge \sigma'}{\Gamma \vdash t : \sigma} \qquad \frac{\Gamma \vdash t : \sigma \wedge \sigma'}{\Gamma \vdash t : \sigma'}$$

would give an undecidable system. It is well known that for the pure simply typed lambda calculus extended with intersection types, the typeable terms are precisely the solvable terms and thus type inference is equivalent to the halting problem [CDCV81]. Adding syntactic restrictions on the use of intersection might alleviate this, however, and this is in some sense what we shall do in this thesis.

2.3.2 Other Work on Static Analysis by Type Inference

Wright has been developing an approach to neededness analysis which is based on type inference [Wri91b, Wri91a]. As we have already mentioned, neededness is an intensional property of terms which is very close to strictness. Wright's type system is based around defining different versions of the $\rightarrow$ type constructor: a function of type $\sigma \Rightarrow \tau$ takes an argument of type σ, which it needs to evaluate, to an a result of type τ. A function of type $\sigma \not\Rightarrow \tau$, on the other hand, takes an argument of type σ which it then ignores to produce its result of type τ. This is extended to include variable arrows and boolean operations on arrows. Wright has produced several different systems, some of which also incorporate intersection types. An interesting feature of Wright's work is that he uses unification over the theory of Boolean rings [MN89] in order to solve the typing constraints which arise in his inference algorithm.

A different strand of work is motivated by the Curry-Howard correspondence between types and propositions in intuitionistic logics [How80]. Several researchers have suggested that one could produce languages with more refined type systems, expressing finer resource usage properties of terms, by extending the correspondence to intuitionistic resource logics, such as suitable fragments of Girard's Linear Logic [Gir87]. This seems a promising line of research although much remains to be done

[4]We use the word intersection, rather than conjunction, to avoid possible confusion caused by the fact that the type corresponding to the logical operation of conjunction under the propositions-as-types analogy is a cartesian product, not an intersection

in devising efficient translations of ordinary typed lambda terms into these more refined languages. This process would be at the heart of any compiler based on such ideas. The pleasant technical properties of these systems also tend to evaporate somewhat when recursion and constants are added to the pure languages. See, for example, [Abr90b, Wad91, Bie92, BBHdP92].

Pierce's thesis [Pie91] also contains some remarks on how a language with an unusually powerful type system, the second order lambda calculus with bounded quantification and intersection types, can mimic various program analyses entirely within its type system. Indeed, it is amusing to note that in most domain-theoretic models of the second order lambda calculus ($\lambda 2$), the interpretation of the type $\forall\alpha.\alpha$ is precisely $\{\bot\}$ (see Chapter 5), so some strictness information is in a sense already expressible within the type system. However, as decidability of type inference for $\lambda 2$ is still an open problem, this is perhaps not a particularly good approach to obtaining practical analysis algorithms.

Chapter 3

Strictness Logic

3.1 Introduction

In the abstract interpretation approach to strictness analysis which we described in Chapter 2, the concretisation maps have the Hoare powerdomain as their target. Thus the interpretation of each abstract domain point is a non-empty Scott-closed subset of the standard domain. The similarity between strictness analysis and type inference is strengthened by the observation that subsets of this kind have also been suggested as the interpretation of types [MS82, MPS84]. In this context, non-empty Scott-closed sets are often referred to as *ideals*, and this is the nomenclature we shall adopt from now on.

The work described in this chapter arose from an attempt to improve upon, and to understand in semantic terms, Kuo and Mishra's strictness type system. One of the advantages of their system is that it makes the basic program properties (such as 'maps strict functions to strict functions') in which we are interested much more visible[1]. Although it may appear obvious, a greater emphasis on program properties, rather than particular representations of them, is long overdue. This point is also made in [Bur91a].

Obviously, some properties imply others. For example, a function which satisfies the property of being the constant bottom function also satisfies the (weaker) property of being strict. Our approach starts by identifying the basic properties at each type and then axiomatising the entailment relation between logical combinations of these properties. We then have to give a formal system for assigning properties to terms.

This idea, and other work on non-standard type systems, represents a shift from denotational analysis techniques to logical ones, so the obvious place to look for insight was the work of Abramsky and Zhang on domain logics [Abr91, Zha89]. These arise from Stone-type dualities between (topological) spaces and logics [Joh82, Vic89]—one can either view points as primary and then consider a property to

[1] Although their claim that this kind of information is not obtainable from the abstract interpretation is untrue.

be a set of points, or take properties as primary and then consider a point to be determined by the properties it satisfies.

Abramsky's work is concerned with the logic of *observable properties*: things we can observe by looking at finite bits of output. These correspond to compact open sets[2] in the Scott topology. Strictness is non-observable (intuitively, one can never observe non-termination), and the strictness properties which we shall consider correspond to closed sets. The logic presented here is nevertheless essentially a fragment of the open set logic in [Abr91], although the interpretations of propositions are very different.

It should be noted that the same system was arrived at independently by Jensen in [Jen91]. That work is complementary to this, however, in that Jensen considers the relation between conventional abstract interpretation and the logic whereas here we take as fundamental the direct relation between the logic and the standard semantics. Some of the work in this chapter and in Chapter 5 was previously reported in [Ben92b].

The plan of the rest of this chapter is as follows. Section 3.2 describes the axiomatisation of the lattice of strictness properties and shows that it is sound and complete with respect to its intended interpretation in the standard semantics. We then give, in Section 3.3, a program logic which allows us to deduce properties of terms. We show that the program logic is sound and then investigate some of its proof-theoretic properties. This leads us to reformulate the program logic to give a set of rules which are more suited to implementation. In Section 3.4 we show that our logic is equivalent in power to BHA-style abstract interpretation. Finally, in Section 3.5, we describe a more powerful strictness logic which incorporates disjunction. This is used to extend our analysis to a language with sum types. We also show the unsoundness of a disjunctive logic due to Jensen.

3.2 The Logic of Strictness Properties

We define a family of propositional theories $\{\mathcal{L}_\sigma\}$, indexed by the types of Λ_T. The theory $\mathcal{L}_\sigma$ is a pair $(L_\sigma, \leq_\sigma)$, where L_σ is a set of propositions and $\leq_\sigma \subseteq L_\sigma \times L_\sigma$ is the entailment relation. The formation and inference rules for $\mathcal{L}_\sigma$ are shown in Figure 3.1. Type superscripts on propositions will frequently be omitted. We define $\phi^\sigma = \psi^\sigma$ to mean $\phi^\sigma \leq \psi^\sigma$ and $\psi^\sigma \leq \phi^\sigma$.

At each type σ we have a domain D_σ, as defined on page 17, and a propositional theory $\mathcal{L}_\sigma$. These are related by giving an interpretation map $[\![\cdot]\!]$ which takes each proposition $\phi \in \mathcal{L}_\sigma$ to the set of elements of D_σ which satisfy it (*i.e.* its *extent*). We shall frequently write $x \models \phi$ for $x \in [\![\phi]\!]$.

[2] If D is an algebraic domain, each Scott open set is expressible as $\bigcup_{i\in I} \uparrow (b_i)$ where each b_i is a finite element of D. The *compact* open sets are those which can be so expressed with I finite.

Formation rules

$$\mathbf{t}, \mathbf{f} \in L_\sigma \qquad \frac{\phi, \psi \in L_\sigma}{\phi \wedge \psi \in L_\sigma}$$

$$\frac{\phi \in L_\sigma \qquad \psi \in L_\tau}{(\phi \to \psi) \in L_{\sigma \to \tau}} \qquad \frac{\phi \in L_\sigma \qquad \psi \in L_\tau}{(\phi \times \psi) \in L_{\sigma \times \tau}}$$

Inference rules

$$\phi \leq \phi\,[\text{refl}] \qquad \phi \leq \mathbf{t}\,[\text{t}] \qquad \mathbf{f} \leq \phi\,[\text{f}]$$

$$\phi \wedge \psi \leq \phi\,[\wedge\text{E-1}] \qquad \phi \wedge \psi \leq \psi\,[\wedge\text{E-2}]$$

$$\frac{\phi \leq \psi \qquad \psi \leq \chi}{\phi \leq \chi}\,[\text{trans}] \qquad \frac{\phi \leq \psi_1 \qquad \phi \leq \psi_2}{\phi \leq \psi_1 \wedge \psi_2}\,[\wedge\text{I}]$$

$$\frac{\phi \leq \phi' \qquad \psi \leq \psi'}{(\phi \times \psi) \leq (\phi' \times \psi')}\,[\times]$$

$$\mathbf{t}^{\sigma \times \tau} \leq \mathbf{t}^\sigma \times \mathbf{t}^\tau\,[\text{t}\times] \qquad \mathbf{f}^\sigma \times \mathbf{f}^\tau \leq \mathbf{f}^{\sigma \times \tau}\,[\text{f}\times]$$

$$(\phi \times \psi) \wedge (\phi' \times \psi') \leq (\phi \wedge \phi') \times (\psi \wedge \psi')\,[\times\wedge]$$

$$\frac{\phi' \leq \phi \qquad \psi \leq \psi'}{(\phi \to \psi) \leq (\phi' \to \psi')}\,[\to]$$

$$\mathbf{t}^{\sigma \to \tau} \leq \mathbf{t}^\sigma \to \mathbf{t}^\tau\,[\text{t}\to] \qquad \mathbf{t}^\sigma \to \mathbf{f}^\tau \leq \mathbf{f}^{\sigma \to \tau}\,[\text{f}\to]$$

$$(\phi \to \psi_1) \wedge (\phi \to \psi_2) \leq (\phi \to \psi_1 \wedge \psi_2)\,[\to\wedge]$$

Figure 3.1: Formation and Inference Rules for $\mathcal{L}_\sigma$

$$[\![\mathbf{t}^\sigma]\!] = D_\sigma$$

$$[\![\mathbf{f}^\sigma]\!] = \{\bot_{D_\sigma}\}$$

$$[\![\phi \wedge \psi]\!] = [\![\phi]\!] \cap [\![\psi]\!]$$

$$[\![\phi^\sigma \times \psi^\tau]\!] = \{(x, y) \in D_{\sigma\times\tau} \mid x \in [\![\psi^\sigma]\!] \text{ and } y \in [\![\phi^\tau]\!]\}$$

$$[\![\phi^\sigma \to \psi^\tau]\!] = \{f \in D_{\sigma\to\tau} \mid f[\![\phi^\sigma]\!] \subseteq [\![\psi^\tau]\!]\}$$

There is an somewhat unfortunate blurring of terminology going on here: the formulae of $\mathcal{L}_\sigma$ are referred to as *propositions*, but are interpreted as unary *predicates* over D_σ. A slightly different way of viewing the relationship between $\mathcal{L}_\sigma$ and D_σ is that each non-$\bot$ element of D_σ corresponds to a model of the theory $\mathcal{L}_\sigma$ in the two element meet-semilattice. Note that distinct domain points do not necessarily correspond to distinct models.

Proposition 3.2.1 *For any $\phi \in L_\sigma$, $[\![\phi]\!]$ is an ideal of D_σ.*

Proof. Induction on the structure of ϕ. For example, if $I = [\![\phi^\sigma]\!]$ and $J = [\![\psi^\tau]\!]$ are ideals, we have to show that $K = [\![\phi^\sigma \to \psi^\tau]\!]$ is too. Well, K is non-empty as it contains the constant $\bot$ function (because J contains $\bot$). K is down-closed because J is – if $f \sqsubseteq g$ and $g \in K$ then for any $d \in I$, $fd \sqsubseteq gd \in J$ and hence $fd \in J$. K also inherits closure under sups of chains from J as if $\langle f_n \rangle$ is a chain in K then for any $d \in I$, $(\bigsqcup f_n)d = \bigsqcup(f_n d) \in J$. □

Example Some examples of the sort of properties which we can express in this system:

- A function $\mathsf{f} :: \iota \to \iota$ is strict iff $[\![\mathsf{f}]\!] \in [\![\mathbf{f}^\iota \to \mathbf{f}^\iota]\!]$.
- A function $\mathsf{f} :: \iota \to \iota$ is the constant $\bot$ function iff $[\![\mathsf{f}]\!] \in [\![\mathbf{t}^\iota \to \mathbf{f}^\iota]\!]$.
- A function $\mathsf{h} :: \iota \times \iota \to \iota$ is strict in both its arguments iff $[\![\mathsf{h}]\!] \in [\![(\mathbf{f}^\iota \times \mathbf{t}^\iota \to \mathbf{f}^\iota) \wedge (\mathbf{t}^\iota \times \mathbf{f}^\iota \to \mathbf{f}^\iota)]\!]$.
- A function $\mathsf{g} :: (\iota \to \iota) \to (\iota \to \iota)$ maps strict functions to strict functions iff $[\![\mathsf{g}]\!] \in [\![(\mathbf{f}^\iota \to \mathbf{f}^\iota) \to (\mathbf{f}^\iota \to \mathbf{f}^\iota)]\!]$

Remarks 3.2.2

- Kuo and Mishra's type system roughly corresponds to the subsytem of our logic which does not contain conjunction.
- Instead of presenting the logic in terms of binary conjunction and a top element, we could have used n-ary conjunction. We shall sometimes abuse notation by using this alternative syntax. The proposition $\bigwedge_{i\in\emptyset} \phi_i^\sigma$ should be understood as being equal to $\mathbf{t}^\sigma$, though we will often treat the $\mathbf{t}$ case separately.

- If we allow n-ary conjunction, then both of the axioms $[\to \wedge]$ and $[\mathrm{t}\to]$, which describe how the type structure interacts with the logical structure, could both have been captured in the single axiom $\bigwedge_{i\in I}(\phi \to \psi_i) = (\phi \to \bigwedge_{i\in I} \psi_i)$, by allowing the indexing set to be empty.
- For any ϕ^σ, $\phi^\sigma \to \mathbf{t}^\tau = \mathbf{t}^{\sigma\to\tau}$. This corresponds to the non-structural rule in Kuo and Mishra's system.
- The rule $[\mathrm{f}\to]$ explicitly identifies $\bot$ and $\lambda x.\bot$.
- Similarly, $[\mathrm{f}\times]$ has the force of identifying $(\bot, \bot)$ with $\bot$.
- It is almost true to say that the $[\mathrm{t}\to]$ and $[\mathrm{t}\times]$ rules are η (extensionality) rules. They can be read as saying that everything in the domain associated with a functional type *is* a function, and that everything in the domain associated with a pair type *is* a pair. However, as all our interpretations are ideals, rather than arbitrary sets, these rules are actually still valid in some situations where the η rules are not (*e.g.* the semantics of a language which requires lifted products and function spaces).
- Each $\mathcal{L}_\sigma$ is consistent; that is $\mathbf{t}^\sigma \not\leq \mathbf{f}^\sigma$.
- $\phi \leq \bigwedge_{i\in I} \psi_i$ if and only if for all $i \in I$, $\phi \leq \psi_i$. Similarly, $d \models \bigwedge_{i\in I} \psi_i$ if and only if for all $i \in I$, $d \models \psi_i$.
- Every proposition in $\mathcal{L}_\iota$ is logically equivalent to either $\mathbf{t}^\iota$ or $\mathbf{f}^\iota$.
- Every proposition in $\mathcal{L}_{\sigma\to\tau}$ is logically equivalent to one of the form $\bigwedge_{i\in I}(\phi_i \to \psi_i)$.
- Every proposition in $\mathcal{L}_{\sigma\times\tau}$ is logically equivalent to one of the form $\phi^\sigma \times \psi^\tau$.
- The similarity between the basic properties of functions ($f \models \phi \to \psi$) and the Hoare triples used in axiomatic semantics of imperative languages ($\{\phi\}\, C\, \{\psi\}$) should be apparent.

Proposition 3.2.3 (Soundness) *If $\phi \leq \psi$ then $[\![\phi]\!] \subseteq [\![\psi]\!]$.*

Proof. This follows by a simple induction on the derivation of $\phi \leq \psi$. If, for example, the derivation ends in

$$\frac{\phi' \leq \phi \qquad \psi \leq \psi'}{\phi \to \psi \leq \phi' \to \psi'}$$

then given $f \models \phi \to \psi$ and $d \models \phi'$ we have $d \models \phi$ by induction hypothesis, so $fd \models \psi$ and thus by induction again, $fd \models \psi'$. Hence $f \models \phi' \to \psi'$ as required. □

Note that the proof of soundness does not really use any of the order properties of the interpretations of propositions—it is purely set-theoretic.

At first sight, it does not seem likely that the small set of rules which we have just given is going to be complete. In fact it turns out that it is because at each type we are restricting attention to a very well-behaved subset of the set of all ideals of the domain.

We write $\mathcal{LA}_\sigma$ for the Lindenbaum algebra (poset reflection) of the theory $\mathcal{L}_\sigma$ (that is to say, $\mathcal{LA}_\sigma \stackrel{\text{def}}{=} \mathcal{L}_\sigma/=_\sigma$). $\mathcal{LA}_\sigma$ is readily seen to be a finite lattice with all meets (and hence all joins as well). We shall abuse notation by writing ϕ when we really mean its equivalence class $[\phi]_=$.

Let $\alpha_\sigma : D_\sigma \to \mathcal{LA}_\sigma$ be defined by $\alpha_\sigma(x) = \bigwedge\{\phi \mid x \models \phi\}$, so α takes a domain element to the conjunction of all the propositions which it satisfies. The conjunction is finite as the lattice is. We can say some simple things about this map immediately:

Lemma 3.2.4

1. *If $x \models \phi^\sigma$ then $\alpha_\sigma(x) \leq \phi^\sigma$.*
2. *If $x \sqsubseteq y$ in D_σ then $\alpha_\sigma(x) \leq \alpha_\sigma(y)$ in $\mathcal{LA}_\sigma$.*
3. *If $\langle x_n \rangle$ is an ω-chain in D_σ then $\alpha_\sigma(\bigsqcup_n x_n) = \bigvee\{\alpha_\sigma(x_n)\}$.*

Proof. These follow from the rules for conjunction in the logic and from the fact that the interpretations of propositions are ideals. □

A sufficient (but not necessary) condition for completeness is that α be surjective. We show this by constructing an explicit map $\gamma_\sigma : \mathcal{LA}_\sigma \to D_\sigma$ such that $\alpha_\sigma \circ \gamma_\sigma = id_{\mathcal{LA}_\sigma}$.

The definition of the γ_σ proceeds by recursion on the type structure. The base case is easy, just let $\gamma_\iota(\mathbf{f}) = \bot_{D_\iota}$ and $\gamma_\iota(\mathbf{t}) = [42]$ (say). For higher types we need a couple of auxiliary definitions.

Define $h_{\sigma\times\tau} : \mathcal{L}_{\sigma\times\tau} \to (\mathcal{L}_\sigma \times \mathcal{L}_\tau)$ as follows:

$$h_{\sigma\times\tau}(\mathbf{t}^{\sigma\times\tau}) = (\mathbf{t}^\sigma, \mathbf{t}^\tau)$$

$$h_{\sigma\times\tau}(\mathbf{f}^{\sigma\times\tau}) = (\mathbf{f}^\sigma, \mathbf{f}^\tau)$$

$$h_{\sigma\times\tau}(\phi^\sigma \times \psi^\tau) = (\phi^\sigma, \psi^\tau)$$

$$\begin{aligned} h_{\sigma\times\tau}(\phi^{\sigma\times\tau} \wedge \psi^{\sigma\times\tau}) &= (\phi_1^\sigma \wedge \psi_1^\sigma, \phi_2^\tau \wedge \psi_2^\tau) \\ \text{where} \quad (\phi_1^\sigma, \phi_2^\tau) &= h_{\sigma\times\tau}(\phi^{\sigma\times\tau}) \\ \text{and} \quad (\psi_1^\sigma, \psi_2^\tau) &= h_{\sigma\times\tau}(\psi^{\sigma\times\tau}) \end{aligned}$$

Lemma 3.2.5 *If* $\phi^{\sigma\times\tau} \leq \psi^{\sigma\times\tau}$, $h_{\sigma\times\tau}(\phi^{\sigma\times\tau}) = (\phi_1^\sigma, \phi_2^\tau)$ *and* $h_{\sigma\times\tau}(\psi^{\sigma\times\tau}) = (\psi_1^\sigma, \psi_2^\tau)$ *then* $\phi_1^\sigma \leq \psi_1^\sigma$ *and* $\phi_2^\tau \leq \psi_2^\tau$.

Proof. Induction on the derivation of $\phi^{\sigma\times\tau} \leq \psi^{\sigma\times\tau}$. □

This shows that $h_{\sigma\times\tau}$ induces $\bar{h}_{\sigma\times\tau} : \mathcal{LA}_{\sigma\times\tau} \to (\mathcal{LA}_\sigma \times \mathcal{LA}_\tau)$ in the obvious way. For function spaces, we define $h_{\sigma\to\tau} : \mathcal{L}_{\sigma\to\tau} \to (\mathcal{L}_\sigma \to \mathcal{L}_\tau)$ as follows:

$$h_{\sigma\to\tau}(\mathbf{t}^{\sigma\to\tau})(\phi^\sigma) = \mathbf{t}^\tau$$

$$h_{\sigma\to\tau}(\mathbf{f}^{\sigma\to\tau})(\phi^\sigma) = \mathbf{f}^\tau$$

$$h_{\sigma\to\tau}(\phi^\sigma \to \psi^\tau)(\chi^\sigma) = \begin{cases} \psi^\tau & \text{if } \chi^\sigma \leq \phi^\sigma \\ \mathbf{t}^\tau & \text{otherwise} \end{cases}$$

$$h_{\sigma\to\tau}(\phi^{\sigma\to\tau} \wedge \psi^{\sigma\to\tau})(\chi^\sigma) = h_{\sigma\to\tau}(\phi^{\sigma\to\tau})(\chi^\sigma) \wedge h_{\sigma\to\tau}(\psi^{\sigma\to\tau})(\chi^\sigma)$$

Lemma 3.2.6 *If* $\phi_1^{\sigma\to\tau} \leq \phi_2^{\sigma\to\tau}$ *and* $\psi_1^\sigma \leq \psi_2^\sigma$ *then*

$$h_{\sigma\to\tau}(\phi_1^{\sigma\to\tau})(\psi_1^\sigma) \leq h_{\sigma\to\tau}(\phi_2^{\sigma\to\tau})(\psi_2^\sigma).$$

Proof. Induction on the derivation of $\phi_1^{\sigma\to\tau} \leq \phi_2^{\sigma\to\tau}$. □

As before, this shows that $h_{\sigma\to\tau}$ induces $\bar{h}_{\sigma\to\tau} : \mathcal{LA}_{\sigma\to\tau} \to (\mathcal{LA}_\sigma \to \mathcal{LA}_\tau)$. We can now complete the definition of γ:

$$\gamma_{\sigma\times\tau}(\phi^{\sigma\times\tau}) = (\gamma_\sigma(\phi_1^\sigma), \gamma_\tau(\phi_2^\tau)) \text{ where } (\phi_1^\sigma, \phi_2^\tau) = \bar{h}(\phi^{\sigma\times\tau})$$

$$\gamma_{\sigma\to\tau}(\phi^{\sigma\to\tau}) = \gamma_\tau \circ \bar{h}(\phi^{\sigma\to\tau}) \circ \alpha_\sigma$$

It is easy to see that γ is well-defined (in that $\gamma_\sigma(\phi^\sigma)$ is indeed an element of D_σ) and that γ is monotonic, *i.e.* that if $\phi \leq \psi$ then $\gamma(\phi) \sqsubseteq \gamma(\psi)$.

Proposition 3.2.7 $\gamma(\phi) \models \phi$.

Proof. Induction on types. The base case is trivial. For products, we may assume without loss of generality that $\phi^{\sigma\times\tau} = \psi^\sigma \times \chi^\tau$ so that $\gamma_{\sigma\times\tau}(\phi^{\sigma\times\tau}) = (\gamma_\sigma(\psi^\sigma), \gamma_\tau(\chi^\tau))$. Then by induction hypothesis $\gamma_\sigma(\psi^\sigma) \models \psi^\sigma$ and $\gamma_\tau(\chi^\tau) \models \chi^\tau$ and hence $\gamma_{\sigma\times\tau}(\phi^{\sigma\times\tau}) \models \phi^{\sigma\times\tau}$.

For function types, we may similarly assume that $\phi^{\sigma\to\tau} = \bigwedge_{i\in I} \varphi_i \to \psi_i$ so that we need to show

$$\gamma_{\sigma\to\tau}(\bigwedge_{i\in I} \varphi_i \to \psi_i) \models \bigwedge_{i\in I} \varphi_i \to \psi_i$$

which is the same as showing that for all $j \in I$

$$\gamma_{\sigma\to\tau}(\bigwedge_{i\in I} \varphi_i \to \psi_i) \models \varphi_j \to \psi_j$$

and by monotonicity of γ, it suffices to show

$$\gamma_{\sigma\to\tau}(\varphi_j \to \psi_j) \models \varphi_j \to \psi_j$$

Now if $d \models \varphi_j$, $\alpha_\sigma(d) \leq \varphi_j$ so

$$\begin{aligned}\gamma_{\sigma\to\tau}(\varphi_j \to \psi_j)(d) &= \gamma_\tau(\bar{h}(\varphi_j \to \psi_j)(\alpha_\sigma(d))) \\ &= \gamma_\tau(\psi_j)\end{aligned}$$

and by induction hypothesis $\gamma_\tau(\psi_j) \models \psi_j$, so we are done. □

Having shown that $\gamma(\phi) \models \phi$, we want to show that ϕ is the *best* proposition which $\gamma(\phi)$ satisfies. We will first need the following:

Lemma 3.2.8

1. $\bar{h}_{\sigma\to\tau}(\bigwedge_{i\in I} \phi_i \to \psi_i)(\chi) = \bigwedge\{\psi_i \mid \chi \leq \phi_i\}$.
2. *If* $\bigwedge\{\psi_i \mid \phi \leq \phi_i\} \leq \psi$ *then* $\bigwedge_{i\in I} \phi_i \to \psi_i \leq \phi \to \psi$.

Proof. The first part follows simply from the definition of $\bar{h}_{\sigma\to\tau}$. For the second part, being somewhat overcautious, if the conjunction is empty then we have $\mathbf{t} \leq \psi$ so that $\phi \to \psi = \mathbf{t}$ and we are done. Otherwise, let $I' = \{i \in I \mid \phi \leq \phi_i\}$ and note that for all $i' \in I'$, $\phi_{i'} \to \psi_{i'} \leq \phi \to \psi_{i'}$ by $[\to]$. Thus

$$\begin{aligned}\bigwedge_{i\in I} \phi_i \to \psi_i &\leq \bigwedge_{i'\in I'} \phi_{i'} \to \psi_{i'} \\ &\leq \bigwedge_{i'\in I'} \phi \to \psi_{i'} \\ &= \phi \to \bigwedge_{i'\in I'} \psi_{i'} \\ &\leq \phi \to \psi.\end{aligned}$$

□

Proposition 3.2.9 *If* $\gamma(\phi) \models \psi$ *then* $\phi \leq \psi$.

Proof. This is another induction on types. The base case is clear. For product types let $\phi^{\sigma\times\tau} = \phi_1^\sigma \times \phi_2^\tau$ and $\psi^{\sigma\times\tau} = \psi_1^\sigma \times \psi_2^\tau$. Then if $\gamma_{\sigma\times\tau}(\phi^{\sigma\times\tau}) \models \psi^{\sigma\times\tau}$ we have $\gamma_\sigma(\phi_1^\sigma) \models \psi_1^\sigma$ and $\gamma_\tau(\phi_2^\tau) \models \psi_2^\tau$ and so by induction $\phi_1^\sigma \leq \psi_1^\sigma$ and $\phi_2^\tau \leq \psi_2^\tau$ so that $\phi^{\sigma\times\tau} \leq \psi^{\sigma\times\tau}$ as required.

For function types we need to show that if

$$\gamma_{\sigma\to\tau}(\bigwedge_{i\in I} \phi_i \to \psi_i) \models \bigwedge_{j\in J} \phi_j \to \psi_j$$

then

$$\bigwedge_{i\in I} \phi_i \to \psi_i \leq \bigwedge_{j\in J} \phi_j \to \psi_j$$

which is the same as showing that for all $j \in J$, if

$$\gamma_{\sigma\to\tau}(\bigwedge_{i\in I} \phi_i \to \psi_i) \models \phi_j \to \psi_j$$

then

$$\bigwedge_{i\in I} \phi_i \to \psi_i \leq \phi_j \to \psi_j$$

Now, by Proposition 3.2.7 $\gamma_\sigma(\phi_j) \models \phi_j$ so

$$\gamma_\tau(\bar{h}_{\sigma\to\tau}(\bigwedge_{i\in I} \phi_i \to \psi_i)(\alpha_\sigma(\gamma_\sigma(\phi_j)))) \models \psi_j$$

so by the induction hypothesis in the form $\alpha \circ \gamma = id$

$$\gamma_\tau(\bar{h}_{\sigma\to\tau}(\bigwedge_{i\in I} \phi_i \to \psi_i)(\phi_j)) \models \psi_j.$$

By the induction hypothesis again, this means

$$\bar{h}_{\sigma\to\tau}(\bigwedge_{i\in I} \phi_i \to \psi_i)(\phi_j) \leq \psi_j$$

which is, by the first part of Lemma 3.2.8,

$$\bigwedge\{\psi_i \mid \phi_j \leq \phi_i\} \leq \psi_j$$

so that, by the second part of Lemma 3.2.8,

$$\bigwedge_{i\in I} \phi_i \to \psi_i \leq \phi_j \to \psi_j$$

as required. □

Corollary 3.2.10 (Completeness) *If* $[\![\phi]\!] \subseteq [\![\psi]\!]$ *then* $\phi \leq \psi$.

Proof. By Proposition 3.2.7, $\gamma(\phi) \models \phi$ so that $\gamma(\phi) \models \psi$ and therefore by Proposition 3.2.9, $\phi \leq \psi$. □

Corollary 3.2.11 (Disjunction Property) *If* $[\![\phi]\!] \subseteq [\![\psi_1]\!] \cup [\![\psi_2]\!]$ *then either* $[\![\phi]\!] \subseteq [\![\psi_1]\!]$ *or* $[\![\phi]\!] \subseteq [\![\psi_2]\!]$.

Proof. Since $\gamma(\phi) \models \phi$ we must have $\gamma(\phi) \models \psi_i$ for some $i \in \{1,2\}$. Then by Proposition 3.2.9 $\phi \leq \psi_i$ so that $[\![\phi]\!] \subseteq [\![\psi_i]\!]$ by Proposition 3.2.3. □

It is an interesting observation that in some sense we only have completeness because of the disjunction property. If we had $[\![\phi]\!] \subseteq [\![\psi_1]\!] \cup [\![\psi_2]\!]$ without $[\![\phi]\!] \subseteq [\![\psi_1]\!]$ or $[\![\phi]\!] \subseteq [\![\psi_2]\!]$ then we would certainly also have $[\![\psi_1 \to \chi \wedge \psi_2 \to \chi]\!] \subseteq [\![\phi \to \chi]\!]$ but no way of proving it without adding disjunction to the logic.

It is worth noting that the proof of Proposition 3.2.9 also shows that the converse to the second part of Lemma 3.2.8 holds. This gives a useful characterisation of the entailment relation at function types:

Lemma 3.2.12 (Entailment Decomposition)

$$\bigwedge_{i \in I}(\phi_i \to \psi_i) \leq (\phi' \to \psi') \quad \textit{if and only if} \quad \bigwedge\{\psi_i \mid \phi' \leq \phi_i\} \leq \psi'$$

Proof. The right-to-left implication is Lemma 3.2.8. For the other direction, note that the left hand side implies

$$\gamma(\bigwedge_{i \in I} \phi_i \to \psi_i) \models (\phi' \to \psi')$$

and then follow the proof of Proposition 3.2.9. □

3.3 The Program Logic

Having seen how strictness properties behave, we now give a program logic for assigning properties to terms of Λ_T. As we have already mentioned, since our system is decidable[3], it cannot be complete. This is in contrast to the situation for, say, Hoare logic for *while* programs. There all the incompleteness is in the assertion language, whereas here it is in the program logic. As it happens, the program logic *is* complete for reasoning about terms which do not contain conditionals (see [Abr90a]), but as this is not a particularly useful observation we shall not discuss it further.

The inference rules for the program logic appear in Figure 3.2. We call this system $\mathcal{PL}1$ to distinguish it from a slight variant which will be introduced later. The notation is fairly standard. A context Γ is a finite set of assumptions of the form $x^\sigma : \phi^\sigma$. The variables appearing in a context must be distinct. We write $\Gamma, x^\sigma : \phi^\sigma$ to mean the context Γ with any existing binding for x^σ removed and the binding $x^\sigma : \phi^\sigma$ added. We restrict attention to *well-formed* judgements $\Gamma \vdash s : \psi^\sigma$, in which all the free variables of s appear in Γ. It is implicit in such a judgement that $s :: \sigma$.

[3]We should really prove decidability, though we have not yet done so. It is, however, intuitively fairly obvious, and in any case it is also a corollary of the result we shall prove later on the equivalence of our logic and BHA abstract interpretation. This does not contradict our earlier remarks about the undecidability of Kuo and Mishra's system extended with conjunction. Our terms are typed, and we never form the conjunction of two properties which are associated with different types.

$$\Gamma, x^\sigma : \phi^\sigma \vdash x^\sigma : \phi^\sigma [\text{var}] \qquad \Gamma \vdash \underline{n} : \mathbf{t}^\iota [\text{nat}]$$

$$\frac{\Gamma \vdash t : \phi^\sigma \to \psi^\tau \qquad \Gamma \vdash s : \phi^\sigma}{\Gamma \vdash ts : \psi^\tau} [\text{app}]$$

$$\frac{\Gamma, x^\sigma : \phi^\sigma \vdash t : \psi^\tau}{\Gamma \vdash \lambda x^\sigma . t : \phi^\sigma \to \psi^\tau} [\text{abs}]$$

$$\frac{\Gamma \vdash s : \phi^\sigma \qquad \Gamma \vdash s : \psi^\sigma}{\Gamma \vdash s : \phi^\sigma \wedge \psi^\sigma} [\text{conj}] \qquad \frac{\Gamma \vdash s : \phi^\sigma \qquad \phi^\sigma \leq \psi^\sigma}{\Gamma \vdash s : \psi^\sigma} [\text{sub}]$$

$$\frac{\Gamma \vdash s : \phi^\sigma \qquad \Gamma \vdash t : \psi^\tau}{\Gamma \vdash (s,t) : \phi^\sigma \times \psi^\tau} [\text{pair}]$$

$$\frac{\Gamma \vdash s : \phi^\sigma \times \psi^\tau}{\Gamma \vdash \mathsf{fst}(s) : \phi^\sigma} [\text{fst}] \qquad \frac{\Gamma \vdash s : \phi^\sigma \times \psi^\tau}{\Gamma \vdash \mathsf{snd}(s) : \psi^\tau} [\text{snd}]$$

$$\frac{\Gamma \vdash s : \mathbf{f}^\iota}{\Gamma \vdash \mathsf{if}\ s\ \mathsf{then}\ t_1\ \mathsf{else}\ t_2 : \mathbf{f}^\tau} [\text{cond1}] \qquad \frac{\Gamma \vdash t_1 : \phi^\tau \qquad \Gamma \vdash t_2 : \phi^\tau}{\Gamma \vdash \mathsf{if}\ s\ \mathsf{then}\ t_1\ \mathsf{else}\ t_2 : \phi^\tau} [\text{cond2}]$$

$$\frac{\Gamma, x^\sigma : \phi^\sigma \vdash s : \phi^\sigma}{\Gamma \vdash \mathsf{fix}(x^\sigma . s) : \phi^\sigma} [\text{fix}] \qquad \frac{\Gamma \vdash s : \phi^\iota \qquad \Gamma \vdash t : \psi^\iota}{\Gamma \vdash s + t : \phi^\iota \wedge \psi^\iota} [\text{arith}]$$

Figure 3.2: The Program Logic $\mathcal{PL}1$

To be able to talk about soundness, we have to extend the notion of satisfaction to terms in context. Write $\rho \models \Gamma$ if for all x^σ in $dom(\rho)$, $\Gamma = \Gamma', x^\sigma : \phi^\sigma$ for some Γ' and $\rho(x^\sigma) \models \phi^\sigma$. Then define $\Gamma \models s : \phi^\sigma$ to mean that for all environments ρ such that $\rho \models \Gamma$, $[\![s]\!]\rho \models \phi^\sigma$.

Proposition 3.3.1 (Soundness of the program logic)

$$\textit{If } \Gamma \vdash s : \phi^\sigma \textit{ then } \Gamma \models s : \phi^\sigma$$

Proof. An easy induction on the derivation of $\Gamma \vdash s : \phi^\sigma$. We give two cases:

- If the derivation ends in an instance of the [abs] rule, so $s = \lambda x^\theta.t :: \theta \to \tau$ and $\phi^\sigma = \chi^\theta \to \psi^\tau$, then assuming $\rho \models \Gamma$ we have to show that for any $d \in D_\theta$ such that $d \models \chi^\theta$, $([\![\lambda x^\theta.t]\!]\rho)(d) \models \psi^\tau$. From the denotational semantics, $([\![\lambda x^\theta.t]\!]\rho)(d) = [\![t]\!]\rho[x^\theta \mapsto d]$. By our assumption on d we have that $\rho[x^\theta \mapsto d] \models \Gamma, x^\theta : \chi^\theta$ and hence the result follows from the induction hypothesis applied to the derivation of $\Gamma, x^\theta : \chi^\theta \vdash t : \psi^\tau$.
- If the derivation ends in [fix], so that $s = \mathsf{fix}(x^\sigma.s')$, then $[\![s]\!]\rho = \bigsqcup_n d_n$ where $d_0 = \bot_{D_\sigma}$ and $d_{n+1} = [\![s']\!]\rho[x^\sigma \mapsto d_n]$. Clearly $d_0 \models \phi^\sigma$. If $d_n \models \phi^\sigma$ then $\rho[x^\sigma \mapsto d_n] \models \Gamma, x^\sigma : \phi^\sigma$ and therefore by the induction hypothesis, $[\![s']\!]\rho[x^\sigma \mapsto d_n] \models \phi^\sigma$. So by mathematical induction $d_n \models \phi^\sigma$ for all n, and hence $(\bigsqcup_n d_n) \models \phi^\sigma$ as ideals are closed under sups of chains.

□

The proof of soundness of our logical system (which does not rely on any of the work we did in order to prove completeness of $\mathcal{L}_\sigma$ with respect to its intended interpretation) is much simpler than the corresponding proof of soundness for BHA-style abstract interpretation which we gave in Chapter 2.

Example We can use our logic to deduce that the factorial function is strict (this was shown using abstract interpretation on page 28).

$$\mathsf{fact} \stackrel{\text{def}}{=} \mathsf{fix}(f^{\iota\to\iota}.\lambda x^\iota.\mathsf{if}\ x^\iota\ \mathsf{then}\ \underline{1}\ \mathsf{else}\ x^\iota * (f^{\iota\to\iota}(x^\iota - \underline{1})))$$

$$\dfrac{\dfrac{\dfrac{\dfrac{}{\{f^{\iota\to\iota} : \mathbf{f}^\iota \to \mathbf{f}^\iota, x^\iota : \mathbf{f}^\iota\} \vdash x^\iota : \mathbf{f}^\iota}\,[\text{var}]}{\{f^{\iota\to\iota} : \mathbf{f}^\iota \to \mathbf{f}^\iota, x^\iota : \mathbf{f}^\iota\} \vdash \mathsf{if}\ x^\iota\ \mathsf{then}\ \underline{1}\ \mathsf{else}\ x^\iota * (f^{\iota\to\iota}(x^\iota - \underline{1})) : \mathbf{f}^\iota}\,[\text{cond1}]}{\{f^{\iota\to\iota} : \mathbf{f}^\iota \to \mathbf{f}^\iota\} \vdash \lambda x^\iota.\mathsf{if}\ x^\iota\ \mathsf{then}\ \underline{1}\ \mathsf{else}\ x^\iota * (f^{\iota\to\iota}(x^\iota - \underline{1})) : \mathbf{f}^\iota \to \mathbf{f}^\iota}\,[\text{abs}]}{\{\} \vdash \mathsf{fact} : \mathbf{f}^\iota \to \mathbf{f}^\iota}\,[\text{fix}]$$

Example As a second example, we show that the function g given by

$$\mathsf{g} \stackrel{\text{def}}{=} (\lambda f^{\iota\to\iota\to\iota}.\lambda x^\iota.\lambda y^\iota.\mathsf{if}\ x^\iota\ \mathsf{then}\ (f^{\iota\to\iota\to\iota}\,\underline{1}\,y^\iota)\ \mathsf{else}\ (f^{\iota\to\iota\to\iota}\,y^\iota\,\underline{1}))\,(\lambda w^\iota.\lambda z^\iota.w^\iota + z^\iota)$$

is strict in its second argument. This is essentially the same example we gave earlier of a property which could be deduced from BHA abstract interpretation, but not from Kuo and Mishra's type system.

For layout reasons, we split the derivation up into parts. Write Π for the following derivation:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\{w^\iota : \mathbf{t}^\iota, z^\iota : \mathbf{f}^\iota\} \vdash w^\iota : \mathbf{t}^\iota}\,[\text{var}] \quad \dfrac{}{\{w^\iota : \mathbf{t}^\iota, z^\iota : \mathbf{f}^\iota\} \vdash z^\iota : \mathbf{f}^\iota}\,[\text{var}]}{\{w^\iota : \mathbf{t}^\iota, z^\iota : \mathbf{f}^\iota\} \vdash w^\iota + z^\iota : \mathbf{t}^\iota \wedge \mathbf{f}^\iota}\,[\text{arith}] \qquad \mathbf{t}^\iota \wedge \mathbf{f}^\iota \leq \mathbf{f}^\iota}{\{w^\iota : \mathbf{t}^\iota, z^\iota : \mathbf{f}^\iota\} \vdash w^\iota + z^\iota : \mathbf{f}^\iota}\,[\text{sub}]}{\{w^\iota : \mathbf{t}^\iota\} \vdash \lambda z^\iota.w^\iota + z^\iota : \mathbf{f}^\iota \to \mathbf{f}^\iota}\,[\text{abs}]}{\{\} \vdash \lambda w^\iota.\lambda z^\iota.w^\iota + z^\iota : \mathbf{t}^\iota \to \mathbf{f}^\iota \to \mathbf{f}^\iota}\,[\text{abs}]$$

There is then a similar derivation Π' of $\{\} \vdash \lambda w^\iota.\lambda z^\iota.w^\iota + z^\iota : \mathbf{f}^\iota \to \mathbf{t}^\iota \to \mathbf{f}^\iota$, so that we can deduce

$$\Pi'' = \dfrac{\Pi \qquad \Pi'}{\{\} \vdash \lambda w^\iota.\lambda z^\iota.w^\iota + z^\iota : (\mathbf{t}^\iota \to \mathbf{f}^\iota \to \mathbf{f}^\iota) \wedge (\mathbf{f}^\iota \to \mathbf{t}^\iota \to \mathbf{f}^\iota)}\,[\text{conj}]$$

Now write ϕ for the proposition $(\mathbf{t}^\iota \to \mathbf{f}^\iota \to \mathbf{f}^\iota) \wedge (\mathbf{f}^\iota \to \mathbf{t}^\iota \to \mathbf{f}^\iota)$ and Γ for the context $\{f^{\iota\to\iota\to\iota} : \phi, x^\iota : \mathbf{t}^\iota, y^\iota : \mathbf{f}^\iota\}$. We have the following derivation Σ:

$$\dfrac{\dfrac{\dfrac{\dfrac{}{\Gamma \vdash f : \phi}\,[\text{var}] \qquad \phi \leq \mathbf{t}^\iota \to \mathbf{f}^\iota \to \mathbf{f}^\iota}{\Gamma \vdash f : \mathbf{t}^\iota \to \mathbf{f}^\iota \to \mathbf{f}^\iota}\,[\text{sub}] \qquad \dfrac{}{\Gamma \vdash \underline{1} : \mathbf{t}^\iota}\,[\text{nat}]}{\Gamma \vdash f\,\underline{1} : \mathbf{f}^\iota \to \mathbf{f}^\iota}\,[\text{app}] \qquad \dfrac{}{\Gamma \vdash y^\iota : \mathbf{f}^\iota}\,[\text{var}]}{\Gamma \vdash (f\,\underline{1}\,y^\iota) : \mathbf{f}^\iota}\,[\text{app}]$$

and a similar derivation Σ' of $\Gamma \vdash (f\,y^\iota\,\underline{1}) : \mathbf{f}^\iota$. Hence we can deduce

$$\dfrac{\dfrac{\dfrac{\dfrac{\Sigma \qquad \Sigma'}{\Gamma \vdash \mathsf{if}\ x^\iota\ \mathsf{then}\ (f^{\iota\to\iota\to\iota}\underline{1}\,y^\iota)\ \mathsf{else}\ (f^{\iota\to\iota\to\iota}y^\iota\,\underline{1}) : \mathbf{f}^\iota}\,[\text{cond2}]}{\{f^{\iota\to\iota\to\iota} : \phi, x^\iota : \mathbf{t}^\iota\} \vdash \lambda y^\iota.\mathsf{if}\ x^\iota\ \mathsf{then}\ (f^{\iota\to\iota\to\iota}\underline{1}\,y^\iota)\ \mathsf{else}\ (f^{\iota\to\iota\to\iota}y^\iota\,\underline{1}) : \mathbf{f}^\iota \to \mathbf{f}^\iota}\,[\text{abs}]}{\{f^{\iota\to\iota\to\iota} : \phi\} \vdash \lambda x^\iota.\lambda y^\iota.\mathsf{if}\ x^\iota\ \mathsf{then}\ (f^{\iota\to\iota\to\iota}\underline{1}\,y^\iota)\ \mathsf{else}\ (f^{\iota\to\iota\to\iota}y^\iota\,\underline{1}) : \mathbf{t}^\iota \to \mathbf{f}^\iota \to \mathbf{f}^\iota}\,[\text{abs}]}{\{\} \vdash \lambda f^{\iota\to\iota\to\iota}.\lambda x^\iota.\lambda y^\iota.\mathsf{if}\ x^\iota\ \mathsf{then}\ (f^{\iota\to\iota\to\iota}\underline{1}\,y^\iota)\ \mathsf{else}\ (f^{\iota\to\iota\to\iota}y^\iota\,\underline{1}) : \phi \to \mathbf{t}^\iota \to \mathbf{f}^\iota \to \mathbf{f}^\iota}\,[\text{abs}]$$

which, together with Π'' and an instance of [app] allows us to deduce

$$\{\} \vdash \mathsf{g} : \mathbf{t}^\iota \to \mathbf{f}^\iota \to \mathbf{f}^\iota$$

as required.

The program logic $\mathcal{PL}1$ has the awkward property of being highly non-deterministic – it is not even close to being syntax directed, either on terms or on propositions. This makes it messy to reason about and would also present difficulties in the design

of an inference algorithm. The problems mainly stem from the unrestricted use of the [sub] rule. Similar situations arise in various type systems – for example, in the Hindley-Milner system the rule for specialising polymorphic types, which is usually called [spec], may be applied anywhere in a derivation. To implement an efficient algorithm for type inference it is necessary to show that it suffices to apply [spec] at the leaves of a derivation; Kuo and Mishra prove a related result for their system.

We should like to prove a similar proposition for our program logic; namely that if a judgement is derivable, then there is a derivation in which the only uses of [sub] occur immediately below leaves. Unfortunately, this is not true with the presentation of the logic which we have given. As a trivial example, consider the derivation

$$\dfrac{\dfrac{\dfrac{}{\{x^\iota : \mathbf{t}^\iota\} \vdash x^\iota : \mathbf{t}^\iota}\,[\text{var}]}{\{\} \vdash \lambda x^\iota.x^\iota : \mathbf{t}^\iota \to \mathbf{t}^\iota}\,[\text{abs}] \qquad \mathbf{t}^\iota \to \mathbf{t}^\iota \leq \mathbf{t}^{\iota\to\iota}}{\{\} \vdash \lambda x^\iota.x^\iota : \mathbf{t}^{\iota\to\iota}}\,[\text{sub}]$$

There is no way of moving the [sub] rule up the derivation. To alleviate this problem, we reformulate the program logic to give the system $\mathcal{PL}2$, which is shown in Figure 3.3. $\mathcal{PL}2$ is somewhat less concise and natural than $\mathcal{PL}1$, and it builds [sub] into those places where it is necessary. In particular, the two new rules [absbot′] and [pairbot′] build the [f→] and [f×] rules of $\mathcal{L}_\sigma$ into the program logic. $\mathcal{PL}2$ still contains the [sub] rule, but we shall show that this is redundant. For technical convenience, $\mathcal{PL}2$ is presented in terms of n-ary, rather than binary, conjunction.

We shall sometimes decorate the turnstile in judgements with an integer to indicate which system we are working in. The first thing we need to know about the new version of the program logic is that it is equivalent to the original one.

Lemma 3.3.2 *If we consider* $\bigwedge_{i=1}^{n} \phi_i$ *to be an abbreviation for* $\phi_1 \wedge (\phi_2 \wedge \ldots \phi_n) \ldots)$ *then* $\Gamma \vdash^1 s : \phi^\sigma$ *if and only if* $\Gamma \vdash^2 s : \phi^\sigma$.

Proof. This follows because each inference rule in one system is a derivable rule in the other. For example

- The [var′] rule of $\mathcal{PL}2$ is derivable in $\mathcal{PL}1$

$$\dfrac{\dfrac{}{\Gamma, x^\sigma : \phi^\sigma \vdash x^\sigma : \phi^\sigma}\,[\text{var}] \qquad \phi^\sigma \leq \psi^\sigma}{\Gamma, x^\sigma : \phi^\sigma \vdash x^\sigma : \psi^\sigma}\,[\text{sub}]$$

- The [top′] rule of $\mathcal{PL}2$ is derivable in $\mathcal{PL}1$. This follows from a trivial induction on the structure of the term in question.

- The [absbot′] rule of $\mathcal{PL}2$ is derivable in $\mathcal{PL}1$

$$\dfrac{\dfrac{\Gamma, x^\sigma : \mathbf{t}^\sigma \vdash t : \mathbf{f}^\tau}{\Gamma \vdash \lambda x^\sigma.t : \mathbf{t}^\sigma \to \mathbf{f}^\tau}\,[\text{abs}] \qquad \mathbf{t}^\sigma \to \mathbf{f}^\tau \leq \mathbf{f}^{\sigma\to\tau}}{\Gamma \vdash \lambda x^\sigma.t : \mathbf{f}^{\sigma\to\tau}}\,[\text{sub}]$$

$$\frac{\phi^\sigma \leq \psi^\sigma}{\Gamma, x^\sigma : \phi^\sigma \vdash x^\sigma : \psi^\sigma}\,[\text{var}'] \qquad \Gamma \vdash s : \mathbf{t}^\sigma [\text{top}']$$

$$\frac{\Gamma \vdash t : \phi^\sigma \to \psi^\tau \qquad \Gamma \vdash s : \phi^\sigma}{\Gamma \vdash ts : \psi^\tau}\,[\text{app}'] \qquad \frac{\Gamma, x^\sigma : \phi^\sigma \vdash t : \psi^\tau}{\Gamma \vdash \lambda x^\sigma.t : \phi^\sigma \to \psi^\tau}\,[\text{abs}']$$

$$\frac{\Gamma, x^\sigma : \mathbf{t}^\sigma \vdash t : \mathbf{f}^\tau}{\Gamma \vdash \lambda x^\sigma.t : \mathbf{f}^{\sigma\to\tau}}\,[\text{absbot}']$$

$$\frac{\Gamma \vdash t : \phi_1^\tau \ldots \Gamma \vdash t : \phi_n^\tau}{\Gamma \vdash t : \bigwedge_{i=1}^{n} \phi_i^\tau}\,[\text{conj}'] \qquad \frac{\Gamma \vdash t : \phi^\tau \qquad \phi^\tau \leq \psi^\tau}{\Gamma \vdash t : \psi^\tau}\,[\text{sub}']$$

$$\frac{\Gamma \vdash s : \phi^\sigma \qquad \Gamma \vdash t : \psi^\tau}{\Gamma \vdash (s,t) : \phi^\sigma \times \psi^\tau}\,[\text{pair}'] \qquad \frac{\Gamma \vdash s : \mathbf{f}^\sigma \qquad \Gamma \vdash t : \mathbf{f}^\tau}{\Gamma \vdash (s,t) : \mathbf{f}^{\sigma\times\tau}}\,[\text{pairbot}']$$

$$\frac{\Gamma \vdash s : \phi^\sigma \times \psi^\tau}{\Gamma \vdash \mathsf{fst}(s) : \phi^\sigma}\,[\text{fst}'] \qquad \frac{\Gamma \vdash s : \phi^\sigma \times \psi^\tau}{\Gamma \vdash \mathsf{snd}(s) : \psi^\tau}\,[\text{snd}']$$

$$\frac{\Gamma, x^\sigma : \phi^\sigma \vdash s : \phi^\sigma \qquad \phi^\sigma \leq \psi^\sigma}{\Gamma \vdash \mathsf{fix}(x^\sigma.s) : \psi^\sigma}\,[\text{fix}']$$

$$\frac{\Gamma \vdash s : \mathbf{f}^\iota}{\Gamma \vdash \mathsf{if}\ s\ \mathsf{then}\ t_1\ \mathsf{else}\ t_2 : \phi^\tau}\,[\text{cond1}'] \qquad \frac{\Gamma \vdash t_1 : \phi^\tau \qquad \Gamma \vdash t_2 : \phi^\tau}{\Gamma \vdash \mathsf{if}\ s\ \mathsf{then}\ t_1\ \mathsf{else}\ t_2 : \phi^\tau}\,[\text{cond2}']$$

$$\frac{\Gamma \vdash s : \phi^\iota}{\Gamma \vdash s + t : \phi^\iota}\,[\text{arith1}'] \qquad \frac{\Gamma \vdash t : \phi^\iota}{\Gamma \vdash s + t : \phi^\iota}\,[\text{arith2}']$$

Figure 3.3: The Program Logic $\mathcal{PL}2$

- The [arith] rule of $\mathcal{PL}1$ is derivable in $\mathcal{PL}2$

$$\dfrac{\dfrac{\Gamma \vdash s : \phi^\iota}{\Gamma \vdash s+t : \phi^\iota}\,[\text{arith1}'] \qquad \dfrac{\Gamma \vdash t : \psi^\iota}{\Gamma \vdash s+t : \psi^\iota}\,[\text{arith2}']}{\Gamma \vdash s+t : \phi^\iota \wedge \psi^\iota}\,[\text{conj}']$$

□

Define a *normal* derivation to be a derivation in $\mathcal{PL}2$ which does not contain any use of the [sub'] rule. We shall show that any derivable judgement has a normal derivation. First we need a few lemmas.

Lemma 3.3.3 *If $s :: \sigma$ and $\mathbf{t}^\sigma \leq \phi^\sigma$ then there is a normal derivation of $\Gamma \vdash s : \phi^\sigma$.*

Proof. This can be shown by a double induction on the structure of s and the structure of ϕ^σ. If $\phi^\sigma = \mathbf{t}^\sigma$ then we are done by [top']. The case $\phi^\sigma = \mathbf{f}^\sigma$ cannot occur. If $\phi^\sigma = \bigwedge_i \psi_i^\sigma$ then for all i, $\mathbf{t}^\sigma \leq \psi_i^\sigma$ and thus by induction there is a normal proof of $\Gamma \vdash s : \psi_i^\sigma$ so we are done by [conj'].
If $\phi^\sigma = \phi_1^{\sigma_1} \to \psi_1^{\tau_1}$ then $\mathbf{t}^{\tau_1} \leq \psi_1^{\tau_1}$ by entailment decomposition. Consider the structure of s. If $s = \lambda x^{\sigma_1}.t$ then by induction there is a normal derivation of $\Gamma, x^{\sigma_1} : \phi_1^{\sigma_1} \vdash t : \psi_1^{\tau_1}$ so we are done by [abs']. If $s = t_1\, t_2$ where $t_2 :: \theta$ and $t_1 :: \theta \to \sigma_1 \to \tau_1$ then we have a normal proof of $\Gamma \vdash t_2 : \mathbf{t}^\theta$ by [top'] and, since $\mathbf{t}^{\theta \to \sigma} \leq \mathbf{t}^\theta \to \phi^\sigma$, by induction there is a normal proof of $\Gamma \vdash t_1 : \mathbf{t}^\theta \to \phi^\sigma$. Hence we are done by [app']. The remaining cases are similar. □

Lemma 3.3.4

1. *If $\Gamma \vdash^2 \underline{n} : \phi^\iota$ then $\mathbf{t}^\iota \leq \phi^\iota$.*
2. *If $\Gamma \vdash^2 x^\sigma : \phi^\sigma$ then $\Gamma = \Gamma', x^\sigma : \psi^\sigma$ for some Γ', ψ^σ such that $\psi^\sigma \leq \phi^\sigma$*

Proof. Both parts are proved by easy inductions on derivations. □

Define the *height* of a derivation to be the length of the longest path from the root to one of the leaves. Paths are not allowed to include derivations of entailments in $\mathcal{L}_\sigma$.

Lemma 3.3.5 *If there is a (normal) derivation Π of $\Gamma, x^\sigma : \phi_1^\sigma \vdash^2 t : \psi^\tau$ and $\phi_2^\sigma \leq \phi_1^\sigma$ then there is a (normal) derivation Π' of $\Gamma, x^\sigma : \phi_2^\sigma \vdash^2 t : \psi^\tau$, with the height of Π' equal to the height of Π.*

Proof. This is a simple induction on Π. Essentially, Π' is exactly the same as Π except that leaves of the form

$$\dfrac{\phi_1^\sigma \leq \chi^\sigma}{\Gamma', x^\sigma : \phi_1^\sigma \vdash^2 x^\sigma : \chi^\sigma}\,[\text{var}']$$

are replaced by

$$\frac{\dfrac{\phi_2^\sigma \leq \phi_1^\sigma \qquad \phi_1^\sigma \leq \chi^\sigma}{\phi_2^\sigma \leq \chi^\sigma}\,[\text{trans}]}{\Gamma', x^\sigma : \phi_2^\sigma \vdash^2 x^\sigma : \chi^\sigma}\,[\text{var}']$$

□

Theorem 3.3.6 (Normal Derivations) *If the judgement* $\Gamma \vdash^2 s : \phi^\sigma$ *is derivable, then it is derivable by a normal derivation.*

Proof. See Appendix A. □

Intuitively, the proof of Theorem 3.3.6 describes a procedure in which we take each occurrence of [sub′] in turn, starting with the highest, and percolate it up the derivation until it disappears, either by turning out to be unnecessary or by being absorbed into another rule. This is reminiscent of some proofs of cut elimination, and indeed one can view [sub′] as a form of cut rule.

Our normal derivations are are still not quite as 'normal' as one might like. In particular, it should be possible to move closer to a truly syntax-directed set of rules by restricting the use of [conj′] as well as that of [sub′]. For the moment, however, we leave this (and the design of an inference algorithm) for future work.

The program logic has another pleasant property. This is that the derivable strictness properties of a term are preserved by β-conversion. For conventional type systems, the preservation of typings under β-reduction is an important property (often called *subject reduction*), as it is this which ensures that "well-typed programs don't go wrong" [Mil78]. The converse, preservation of types under β-expansion, does not generally hold. A very simple example would be the term

$$u \stackrel{\text{def}}{=} (\lambda y.\lambda x.x)\,(\lambda z.z\,z)$$

which is not typeable at all in the simply-typed lambda calculus (as the subterm $z\,z$ is not typeable). After one β-reduction, u reduces to $\lambda x.x$, which is typeable.

Type systems which include conjunction do, however, usually satisfy subject expansion, and as our logic contains conjunction it is perhaps not such a surprise that it too has this property. Note that we are not claiming invariance under all expansion steps—we do not say anything about the δ rules associated with conditionals, fixpoints or arithmetic. Indeed, such a claim would clearly be false, as can be seen by consideration of the following term:

$$\lambda x^\iota.\mathsf{if}\ \underline{0}\ \mathsf{then}\ x^\iota\ \mathsf{else}\ \underline{1}$$

We are unable to deduce that this function is strict using the logic, but after applying the obvious (and sound) rewrite rule

$$\mathsf{if}\ \underline{0}\ \mathsf{then}\ t_1\ \mathsf{else}\ t_2 \rightsquigarrow t_1$$

to the body, we get $\lambda x^\iota.x^\iota$, which can be shown to be strict in the logic.

We shall need the following facts, which are both easily verified:

Lemma 3.3.7

1. *If* $x^\sigma \notin FV(t)$ *then* $\Gamma, x^\sigma : \phi^\sigma \vdash t : \psi^\tau$ *if and only if* $\Gamma \vdash t : \psi^\tau$.
2. *If* Π *is a derivation of* $\Gamma, x^\sigma : \phi^\sigma \vdash t : \psi^\tau$ *and* $y^\sigma \notin FV(t)$ *then there is a derivation* Π' *of* $\Gamma, y^\sigma : \phi^\sigma \vdash t[y^\sigma/x^\sigma] : \psi^\tau$ *which is the same height as* Π.

□

The 'if' direction of the first of these is often known as *weakening*, whilst the 'only if' direction is called *strengthening*. The second part simply asserts the entirely obvious fact that we may change variable names in terms and derivations.

Subject reduction then follows from the next lemma, which shows that our program logic has what is known as the *substitution property*.

Lemma 3.3.8 (Substitution) *If* $\Gamma, x^\sigma : \phi^\sigma \vdash t : \psi^\tau$ *and* $\Gamma \vdash s : \phi^\sigma$ *then* $\Gamma \vdash t[s/x^\sigma] : \psi^\tau$.

Proof. Induction on the height of the derivation of $\Gamma, x^\sigma : \phi^\sigma \vdash^1 t : \psi^\tau$. We consider cases according to the last rule of the derivation:

- Case [var]. In this case $t = y^\tau$. If $x^\sigma = y^\tau$ then $\phi^\sigma = \psi^\tau$ and $t[s/x^\sigma] = s$ so there is nothing to prove. If $x^\sigma \neq y^\tau$ then $t[s/x^\sigma] = y^\tau$ and the result follows from Lemma 3.3.7(1).

- Case [app]. In this case $t = t_1\, t_2$ so we have

$$\frac{\Gamma, x^\sigma : \phi^\sigma \vdash t_1 : \chi^\theta \to \psi^\tau \qquad \Gamma, x^\sigma : \phi^\sigma \vdash t_2 : \chi^\theta}{\Gamma, x^\sigma : \phi^\sigma \vdash t_1\, t_2 : \psi^\tau}\ [\text{app}]$$

  By induction, there are derivations of $\Gamma \vdash t_1[s/x^\sigma] : \chi^\theta \to \psi^\tau$ and $\Gamma \vdash t_2[s/x^\sigma] : \chi^\theta$ and so of $\Gamma \vdash (t_1[s/x^\sigma])\,(t_2[s/x^\sigma]) : \psi^\tau$. But $(t_1[s/x^\sigma])\,(t_2[s/x^\sigma]) = (t_1\, t_2)[s/x^\sigma]$ so we are done.

- Case [abs]. In this case $t = \lambda y^{\theta_1}.u :: \theta_1 \to \theta_2$. If $y^{\theta_1} = x^\sigma$ then we have

$$\frac{\Gamma, x^\sigma : \chi_1^\sigma \vdash u : \chi_2^{\theta_2}}{\Gamma, x^\sigma : \psi^\sigma \vdash \lambda x^\sigma.u : \chi_1^\sigma \to \chi_2^{\theta_2}}\ [\text{abs}]$$

  so we clearly have $\Gamma \vdash \lambda x^\sigma.u : \chi_1^\sigma \to \chi_2^{\theta_2}$ and thus we are done as $t[s/x^\sigma] = \lambda x^\sigma.u$.

  If $y^\theta \neq x^\sigma$ then the situation is

$$\frac{\Gamma, x^\sigma : \phi^\sigma, y^{\theta_1} : \chi_1^{\theta_1} \vdash u : \chi_2^{\theta_2}}{\Gamma, x^\sigma : \phi^\sigma \vdash \lambda y^{\theta_1}.u : \chi_1^{\theta_1} \to \chi_2^{\theta_2}}\ [\text{abs}]$$

  and we assume $y^{\theta_1} \notin FV(s)$ (otherwise we can apply Lemma 3.3.7(2) to rename y^{θ_1} and get us to this case). By induction $\Gamma, y^{\theta_1} : \chi_1^{\theta_1} \vdash u[s/x^\sigma] : \chi_2^{\theta_2}$ and thus $\Gamma \vdash \lambda y^{\theta_1}.(u[s/x^\sigma]) : \chi_1^{\theta_1} \to \chi_2^{\theta_2}$ and so we are done as $t[s/x^\sigma] = \lambda y^{\theta_1}.(u[s/x^\sigma])$.

- The remaining cases are similar.

□

Corollary 3.3.9 (Subject Reduction) *If* $\Gamma \vdash (\lambda x^\sigma.t)s : \phi^\tau$ *then* $\Gamma \vdash t[s/x^\sigma] : \phi^\tau$.

Proof. Assume that the derivation Π of $\Gamma \vdash^2 (\lambda x^\sigma.t)s : \phi^\tau$ is normal. It should be clear that if Π ends in [top′] then there is nothing to prove. The case where Π ends in [conj′] follows immediately given the result in the remaining possible case, which is that the derivation ends in [app′]. In that case, we must have

$$\dfrac{\dfrac{\Gamma, x^\sigma : \psi^\sigma \vdash t : \phi^\tau}{\Gamma \vdash \lambda x^\sigma.t : \psi^\sigma \to \phi^\tau}\,[\text{abs}'] \qquad \Gamma \vdash s : \psi^\sigma}{\Gamma \vdash (\lambda x^\sigma.t)s : \phi^\tau}\,[\text{app}']$$

and so by Lemma 3.3.8, there is a derivation of $\Gamma \vdash t[s/x^\sigma] : \phi^\tau$ as required. □

The converse to the previous proposition, subject expansion, follows from the following lemma, which is essentially the converse of the substitution lemma:

Lemma 3.3.10 *If* $t :: \tau$, $s :: \sigma$ *and* $\Gamma \vdash t[s/x^\sigma] : \phi^\tau$ *then there exists a* ψ^σ *such that* $\Gamma, x^\sigma : \psi^\sigma \vdash t : \phi^\tau$ *and* $\Gamma \vdash s : \psi^\sigma$.

Proof. Induction on the height of Π, a normal derivation of $\Gamma \vdash^2 t[s/x^\sigma] : \phi^\tau$. First we deal with the cases where the last rule of Π is [top′] or [conj′]:

- If the last rule is [top′] then $\Gamma, x^\sigma : \mathbf{t}^\sigma \vdash t : \mathbf{t}^\tau$ and $\Gamma \vdash s : \mathbf{t}^\sigma$, both by [top′], so we are done.
- If the last rule is [conj′] then Π looks like

$$\dfrac{\begin{matrix}\Pi_1 & & \Pi_n \\ \Gamma \vdash t[s/x^\sigma] : \phi_1^\tau & \cdots & \Gamma \vdash t[s/x^\sigma] : \phi_n^\tau\end{matrix}}{\Gamma \vdash t[s/x^\sigma] : \bigwedge_{i=1}^{n} \phi_i^\tau}\,[\text{conj}']$$

  so that by induction for all i there exists a ψ_i^σ such that $\Gamma, x^\sigma : \psi_i^\sigma \vdash t : \phi_i^\tau$ and $\Gamma \vdash s : \psi_i^\sigma$. Thus for all i we have $\Gamma, x^\sigma : \bigwedge_{j=1}^n \psi_j^\sigma \vdash t : \phi_i^\tau$ by Lemma 3.3.5 so $\Gamma, x^\sigma : \bigwedge_{i=1}^n \psi_i^\sigma \vdash t : \bigwedge_{i=1}^n \phi_i^\tau$ and $\Gamma \vdash s : \bigwedge_{i=1}^n \psi_i^\sigma$, both by [conj′], so we are done.

Otherwise, we consider cases for the structure of the term t:

- If $t = u\,v$, where $u :: \theta \to \tau$ and $v :: \theta$ then $t[s/x^\sigma] = (u[s/x^\sigma])\,(v[s/x^\sigma])$ so that we just have to deal with the case where the last rule of Π is [app′]. In this case, we must have derivations of $\Gamma \vdash u[s/x^\sigma] : \chi^\theta \to \phi^\tau$ and $\Gamma \vdash v[s/x^\sigma] : \chi^\theta$ for some χ^θ. By induction there exists a ψ_1^σ such that $\Gamma, x^\sigma : \psi_1^\sigma \vdash u : \chi^\theta \to \phi^\tau$ and $\Gamma \vdash s : \psi_1^\sigma$ and a ψ_2^σ such that $\Gamma, x^\sigma : \psi_2^\sigma \vdash v : \chi^\theta$ and $\Gamma \vdash s : \psi_2^\sigma$. So $\Gamma \vdash s : \psi_1^\sigma \wedge \psi_2^\sigma$ by [conj′] and we also have $\Gamma, x^\sigma : \psi_1^\sigma \wedge \psi_2^\sigma \vdash u : \chi^\theta \to \phi^\tau$ and $\Gamma, x^\sigma : \psi_1^\sigma \wedge \psi_2^\sigma \vdash v : \chi^\theta$, by Lemma 3.3.5. Consequently $\Gamma, x^\sigma : \psi_1^\sigma \wedge \psi_2^\sigma \vdash u\,v : \phi^\tau$ and we are done.

- If $t = \lambda y^{\theta_1}.u$ where $u :: \theta_2$ then if $y^{\theta_1} = x^\sigma$ we have $t[s/x^\sigma] = t$ so that $\Gamma \vdash t : \phi^\tau$, so clearly $\Gamma, x^\sigma : \mathbf{t}^\sigma \vdash t : \phi^\tau$ (by Lemma 3.3.7) and $\Gamma \vdash s : \mathbf{t}^\sigma$ by [top′]. So assume $y^{\theta_1} \neq x^\sigma$ and without loss of generality that $y^{\theta_1} \notin FV(s)$. This means that $t[s/x^\sigma] = \lambda y^{\theta_1}.u[s/x^\sigma]$. The two possibilities for the last rule of Π which have to be considered are [abs′] and [absbot′]. We shall just deal with [abs′] here as the case for [absbot′] is very similar. In this case, Π ends in

$$\frac{\Gamma, y^{\theta_1} : \chi_1^{\theta_1} \vdash u[s/x^\sigma] : \chi_2^{\theta_2}}{\Gamma \vdash \lambda y^{\theta_1}.u[s/x^\sigma] : \chi_1^{\theta_1} \to \chi_2^{\theta_2}}\,[\text{abs}']$$

  so that by induction we know that there exists ψ^σ such that

$$\Gamma, y^{\theta_1} : \chi_1^{\theta_1}, x^\sigma : \psi^\sigma \vdash u : \chi_2^{\theta_2} \text{ and } \Gamma, y^{\theta_1} : \chi_1^{\theta_1} \vdash s : \psi^\sigma$$

  Then $\Gamma, x^\sigma : \psi^\sigma \vdash t : \chi_1^{\theta_1} \to \chi_2^{\theta_2}$ by [abs′] and $\Gamma \vdash s : \psi^\sigma$ by Lemma 3.3.7 as $y^{\theta_1} \notin FV(s)$.

- The remaining cases are similar. When there is more than one premiss for the last rule of Π, it is necessary to use [conj′] as we did in the case of [app′] above.

□

Corollary 3.3.11 (Subject Expansion) *If* $\Gamma \vdash t[s/x^\sigma] : \phi^\tau$ *then* $\Gamma \vdash (\lambda x^\sigma.t)s : \phi^\tau$.

Proof. By Lemma 3.3.10, there exists ψ^σ such that $\Gamma, x^\sigma : \psi^\sigma \vdash t : \phi^\tau$ and $\Gamma \vdash s : \psi^\sigma$. Hence $\Gamma \vdash \lambda x^\sigma.t : \psi^\sigma \to \phi^\tau$ by [abs′] and thus $\Gamma \vdash (\lambda x^\sigma.t)s : \phi^\tau$ by [app′]. □

3.4 Strictness Logic and Abstract Interpretation

We now turn to the relationship between our strictness logic and the BHA-style abstract interpretation which we described in Chapter 2. We show that the two systems are equivalent, in terms of both the properties which they can express and the properties which can be assigned to terms.

If A and B are finite lattices with $a \in A, b \in B$ define the *step function* $[a,b] : A \to B$ by

$$[a,b](a') = \begin{cases} b & \text{if } a' \sqsubseteq a \\ \top_B & \text{otherwise} \end{cases}$$

It should be noted that the step functions which are most commonly used in domain theory step up from $\bot$, rather than up to $\top$, as ours do. $[a,b]$ is monotonic and is therefore an element of $[A \to B]$, since the finiteness assumption means that monotonicity coincides with continuity. Recall that $[A \to B]$ inherits all meets and joins from B, and is therefore itself a finite lattice.

Lemma 3.4.1 *If A, B are finite lattices and $f \in [A \to B]$ then*

$$f = \sqcap\{[a,b] \mid f \sqsubseteq [a,b]\} = \sqcap\{[a,b] \mid f(a) \sqsubseteq b\}$$

Proof. The second equality follows from the easily verified fact that $f \sqsubseteq [a,b]$ iff $f(a) \sqsubseteq b$. For the first equality, it is clear that

$$f \sqsubseteq \sqcap\{[a,b] \mid f \sqsubseteq [a,b]\}$$

so we just need the reverse inequality. For any a', $f \sqsubseteq [a', f(a')]$ so

$$\sqcap\{[a,b] \mid f \sqsubseteq [a,b]\} \sqsubseteq [a', f(a')]$$

which means

$$\sqcap\{[a,b] \mid f \sqsubseteq [a,b]\}(a') \sqsubseteq [a', f(a')](a') = f(a')$$

which is what we wanted. □

So every monotone function between finite lattices can be expressed as a meet of step functions, although this representation will not, in general, be unique. The next lemma characterises the order relation on maps in terms of this representation. It is essentially the same as the entailment decomposition result (Lemma 3.2.12) which we proved earlier:

Lemma 3.4.2

$$\sqcap_{i \in I}[a_i, b_i] \sqsubseteq \sqcap_{j \in J}[a'_j, b'_j] \quad \textit{if and only if} \quad \forall j \in J. \sqcap\{b_i \mid a'_j \sqsubseteq a_i\} \sqsubseteq b'_j$$

Proof. As before, this reduces to showing that for all $j \in J$

$$\sqcap_{i \in I}[a_i, b_i] \sqsubseteq [a'_j, b'_j] \quad \text{iff} \quad \sqcap\{b_i \mid a'_j \sqsubseteq a_i\} \sqsubseteq b'_j$$

For the left to right implication we have

$$\begin{array}{rrcl} & \sqcap_{i \in I}[a_i, b_i] & \sqsubseteq & [a'_j, b'_j] \\ \Rightarrow & (\sqcap_{i \in I}[a_i, b_i])(a'_j) & \sqsubseteq & [a'_j, b'_j](a'_j) \\ \Leftrightarrow & \sqcap\{b_i \mid a'_j \sqsubseteq a_i\} & \sqsubseteq & b'_j \end{array}$$

Whilst for the right to left direction, assume that

$$\sqcap\{b_i \mid a'_j \sqsubseteq a_i\} \sqsubseteq b'_j$$

and then for any $a \in A$, if $a \sqsubseteq a'_j$ then

$$\begin{array}{rcl}(\sqcap_{i\in I}[a_i, b_i])(a) & = & \sqcap\{b_i \mid a \sqsubseteq a_i\} \\ & \sqsubseteq & \sqcap\{b_i \mid a'_j \sqsubseteq a_i\} \\ & \sqsubseteq & b'_j \\ & = & [a'_j, b'_j](a)\end{array}$$

whilst if $a \not\sqsubseteq a'_j$

$$(\sqcap_{i\in I}[a_i, b_i])(a) \sqsubseteq \top = [a'_j, b'_j](a)$$

so that we are done in either case. □

We need to relate the propositional theory $\mathcal{L}_\sigma$ and the BHA-style abstract domain A_σ at each type σ. To do this, we define $p_\sigma : \mathcal{L}_\sigma \to A_\sigma$ as follows:

$$p_\sigma(\mathbf{t}^\sigma) = \top_{A_\sigma} \qquad p_\sigma(\mathbf{f}^\sigma) = \bot_{A_\sigma}$$

$$p_\sigma(\phi^\sigma \wedge \psi^\sigma) = p_\sigma(\phi^\sigma) \sqcap p_\sigma(\psi^\sigma)$$

$$p_{\sigma\to\tau}(\phi^\sigma \to \psi^\tau) = [p_\sigma(\phi^\sigma), p_\tau(\psi^\tau)]$$

$$p_{\sigma\times\tau}(\phi^\sigma \times \psi^\tau) = (p_\sigma(\phi^\sigma), p_\tau(\psi^\tau))$$

Lemma 3.4.3 *If $\phi^\sigma \leq \psi^\sigma$ then $p_\sigma(\phi^\sigma) \sqsubseteq p_\sigma(\psi^\sigma)$.*

Proof. An easy induction on derivations. □

This means that p_σ induces $\bar{p}_\sigma : \mathcal{LA}_\sigma \to A_\sigma$ in the usual way. Note that p and $\bar{p}$ are essentially identical to the functions h and $\bar{h}$ which we used earlier. We claim that each $\bar{p}_\sigma$ is an isomorphism, which we show by constructing its inverse. Define $q_\sigma : A_\sigma \to \mathcal{LA}_\sigma$ by:

$$q_\iota(\top_{A_\iota}) = \mathbf{t}^\iota \qquad q_\iota(\bot_{A_\iota}) = \mathbf{f}^\iota$$

$$q_{\sigma\times\tau}(a, b) = q_\sigma(a) \times q_\tau(b)$$

$$q_{\sigma\to\tau}(\sqcap_{i\in I}[a_i, b_i]) = \bigwedge_{i\in I} q_\sigma(a_i) \to q_\tau(b_i)$$

We need to check that the last clause is a good definition because of the non-uniqueness of the representation of maps in terms of step functions. This is a consequence of the following:

Lemma 3.4.4 *If $a, a' \in A_\sigma$ and $a \sqsubseteq a'$ then $q_\sigma(a) \leq q_\sigma(a')$.*

Proof. Induction on types. The case for function spaces uses Lemma 3.4.2 and Lemma 3.2.12. □

Proposition 3.4.5 *For all σ, $\bar{p}_\sigma$ is an isomorphism.*

Proof. It is trivial to show

$$\bar{p}_\sigma \circ q_\sigma = id_{A_\sigma}$$

and

$$q_\sigma \circ \bar{p}_\sigma = id_{\mathcal{LA}_\sigma}$$

for each σ by induction on types. □

Although it is not actually necessary from the point of view of showing that our strictness logic and the BHA abstract interpretation are equivalent, we would obviously hope that the intended interpretation of each proposition would coincide with the concretisation of the associated abstract domain element. The following shows that this is indeed the case:

Proposition 3.4.6 *For all ϕ^σ, $[\![\phi^\sigma]\!] = Conc_\sigma(p_\sigma(\phi^\sigma))$*

Proof. This is proved by induction on types. The only interesting case is that of function spaces (in this proof α denotes the abstraction map for the abstract interpretation, rather than the map into $\mathcal{LA}_\sigma$ which we defined earlier. Of course, we are engaged in showing that these two maps are essentially the same):

$$\begin{aligned}
&Conc_{\sigma\to\tau}(p_{\sigma\to\tau}(\bigwedge_{i\in I} \phi_i^\sigma \to \psi_i^\tau)) \\
&= Conc_{\sigma\to\tau}(\sqcap_{i\in I}[p_\sigma(\phi_i^\sigma), p_\tau(\psi_i^\tau)]) \\
&= \{f \in D_{\sigma\to\tau} \mid \alpha_{\sigma\to\tau}(f) \sqsubseteq \sqcap_{i\in I}[p_\sigma(\phi_i^\sigma), p_\tau(\psi_i^\tau)]\} \\
&= \{f \in D_{\sigma\to\tau} \mid \forall i \in I.\alpha_{\sigma\to\tau}(f) \sqsubseteq [p_\sigma(\phi_i^\sigma), p_\tau(\psi_i^\tau)]\} \\
&= \{f \in D_{\sigma\to\tau} \mid \forall i \in I.\alpha_{\sigma\to\tau}(f)(p_\sigma(\phi_i^\sigma)) \sqsubseteq p_\tau(\psi_i^\tau)\} \\
&= \{f \in D_{\sigma\to\tau} \mid \forall i \in I.(Abs_\tau \circ \mathcal{P}_H(f) \circ Conc_\sigma)(p_\sigma(\phi_i^\sigma)) \sqsubseteq p_\tau(\psi_i^\tau)\} \\
&= \{f \in D_{\sigma\to\tau} \mid \forall i \in I.(Abs_\tau \circ \mathcal{P}_H(f))([\![\phi_i^\sigma]\!]) \sqsubseteq p_\tau(\psi_i^\tau)\} \text{ by induction} \\
&= \{f \in D_{\sigma\to\tau} \mid \forall i \in I.Abs_\tau(\overline{f[\![\phi_i^\sigma]\!]}) \sqsubseteq p_\tau(\psi_i^\tau)\} \\
&= \{f \in D_{\sigma\to\tau} \mid \forall i \in I.\overline{f[\![\phi_i^\sigma]\!]} \subseteq Conc_\tau(p_\tau(\psi_i^\tau))\} \text{ as } Conc \text{ and } Abs \text{ form an adjunction} \\
&= \{f \in D_{\sigma\to\tau} \mid \forall i \in I.\overline{f[\![\phi_i^\sigma]\!]} \subseteq [\![\psi_i^\tau]\!]\} \text{ by induction} \\
&= \{f \in D_{\sigma\to\tau} \mid \forall i \in I.f[\![\phi_i^\sigma]\!] \subseteq [\![\psi_i^\tau]\!]\} \\
&= [\![\bigwedge_{i\in I} \phi_i^\sigma \to \psi_i^\tau]\!]
\end{aligned}$$

□

So the properties expressible in two systems are the same. Now we show that the program logic and the abstract semantic equations are equally powerful. If $\Gamma = \{x_i^{\sigma_i} : \phi_i^{\sigma_i}\}$ is a strictness context, define the abstract environment $p\Gamma$ by $p\Gamma(x_i^{\sigma_i}) = p_{\sigma_i}(\phi_i^{\sigma_i})$.

Proposition 3.4.7 *If $t :: \tau$ and $\Gamma \vdash t : \phi^\tau$ then $[\![t]\!]^A(p\Gamma) \sqsubseteq p_\tau(\phi^\tau)$.*

Proof. Induction on the derivation Π of $\Gamma \vdash^1 t : \phi^\tau$. We give a few representative cases:

- If the last rule of Π is [app] then $t = u\,v$ and we have (omitting type subscripts and superscripts) derivations of $\Gamma \vdash u : \psi \to \phi$ and $\Gamma \vdash v : \psi$. Thus by induction, $[\![u]\!]^A(p\Gamma) \sqsubseteq p(\psi \to \phi) = [p(\psi), p(\phi)]$ and $[\![v]\!]^A(p\Gamma) \sqsubseteq p(\psi)$. Hence $[\![u\,v]\!]^A(p\Gamma) = ([\![u]\!]^A(p\Gamma))([\![v]\!]^A(p\Gamma)) \sqsubseteq p(\phi)$ as required.

- If the last rule is [abs], so $t = \lambda x^\sigma.u$ and $\phi = \chi \to \psi$, then we have a derivation of $\Gamma, x^\sigma : \chi \vdash u : \psi$ and so by induction $[\![u]\!]^A(p\Gamma[x^\sigma \mapsto p\chi]) \sqsubseteq p\psi$. Thus $[\![\lambda x^\sigma.u]\!]^A(p\Gamma) = \lambda a \in A_\sigma.[\![u]\!]^A(p\Gamma[x^\sigma \mapsto a]) \sqsubseteq [p(\chi), p(\psi)] = p(\phi)$ and we are done.

- If the last rule is [conj], so $\phi = \chi \wedge \psi$, then we have derivations of $\Gamma \vdash t : \chi$ and $\Gamma \vdash t : \psi$ so that $[\![t]\!]^A(p\Gamma) \sqsubseteq p(\chi)$ and $[\![t]\!]^A(p\Gamma) \sqsubseteq p(\psi)$. Thus $[\![t]\!]^A(p\Gamma) \sqsubseteq p(\chi) \sqcap p(\psi) = p(\phi)$ and we are done.

- If the last rule is [fix], so $t = \mathsf{fix}(x^\tau.u)$, then we have a derivation of $\Gamma, x^\tau : \phi \vdash u : \phi$ so by induction $[\![u]\!]^A(p\Gamma[x^\tau \mapsto p(\phi)]) \sqsubseteq p(\phi)$. This means that $p(\phi)$ is a prefixed point of the map which takes $a \in A_\tau$ to $[\![u]\!]^A(p\Gamma[x^\tau \mapsto a])$. The result then follows as $[\![t]\!](p\Gamma)$ is the *least* prefixed point of that map.

□

Proposition 3.4.8 *If $t :: \tau$ and $[\![t]\!]^A(p\Gamma) \sqsubseteq p_\tau(\phi^\tau)$ then $\Gamma \vdash^2 t : \phi^\tau$.*

Proof. This direction is proved by induction on the structure of t.

- If $t = x^\tau$ then $[\![t]\!]^A(p\Gamma) = (p\Gamma)(x^\tau)$ so $\Gamma = \Gamma', x^\tau : \psi^\tau$ where $p(\psi^\tau) \sqsubseteq p(\phi^\tau)$ but this means $\psi^\tau \leq \phi^\tau$ as p reflects order, so we are done by [var′].

- If $t = u\,v$ where $v :: \sigma$, then write f for $[\![u]\!]^A(p\Gamma) \in [A_\sigma \to A_\tau]$ and a for $[\![v]\!]^A(p\Gamma) \in A_\sigma$. Now pick ψ^σ such that $a = p(\psi)$, which we can do as p is surjective. By induction, $\Gamma \vdash v : \psi$. As $f(p(\psi)) \sqsubseteq p(\phi)$ we have $f \sqsubseteq [p(\psi), p(\phi)] = p(\psi \to \phi)$ and thus by induction $\Gamma \vdash u : \psi \to \phi$. Then by [app′], $\Gamma \vdash u\,v : \phi$ as required.

- If $t = \lambda x^\sigma.u$ then without loss of generality we can assume $\phi = \bigwedge_{i\in I} \psi_i \to \chi_i$ so we have $[\![t]\!]^A(p\Gamma) \sqsubseteq \sqcap_{i\in I}[p(\psi_i), p(\chi_i)]$. This means that for all $i \in I$, $[\![u]\!]^A(p\Gamma[x^\sigma \mapsto p(\psi_i)]) \sqsubseteq p(\chi_i)$ and thus by induction $\Gamma, x^\sigma : \psi_i \vdash u : \chi_i$ so that $\Gamma \vdash t : \psi_i \to \chi_i$ by [abs'] for each i. Then by [conj'] we have $\Gamma \vdash t : \phi$ and we are done.

- If $t = \mathsf{fix}(x^\tau.u)$ then let $a = [\![t]\!]^A(p\Gamma)$ and $p(\psi) = a$. Then as a is a fixpoint, we know that $[\![u]\!]^A(p\Gamma[x^\tau \mapsto a]) = a$ and so by induction $\Gamma, x^\tau : \psi \vdash u : \psi$ and since $a \sqsubseteq p(\phi)$ implies $\psi \leq \phi$, we have $\Gamma \vdash t : \phi$ by [fix'].

- If $t = (u, v)$ then $[\![t]\!]^A(p\Gamma) = (a, b)$ where $a = [\![u]\!]^A(p\Gamma)$ and $b = [\![v]\!]^A(p\Gamma)$. We can assume $\phi = \psi \times \chi$, so we have $(a, b) \sqsubseteq (p(\psi), p(\chi))$ and hence $a \sqsubseteq p(\psi)$ so that by induction $\Gamma \vdash u : \psi$ and similarly for v and χ. Then by [pair'] we have $\Gamma \vdash t : \phi$ as required.

□

The previous pair of propositions establish the equivalence of the logic and the abstract interpretation. This was the main result of [Jen91]. Whilst the differences between Jensen's formal system and ours are entirely negligible, there is a slight difference in the way in which this equivalence is shown, and this deserves some further comment.

Jensen's work gives an interpretation of each proposition in $\mathcal{L}_\sigma$ as an ideal of the *abstract* domain A_σ, where here the word 'ideal' is meant in the slightly more common lattice-theoretical sense (that is, a non-empty subset which is down closed and closed under binary joins). He thus shows an isomorphism between $\mathcal{LA}_\sigma$ and $\mathsf{Idl}(A_\sigma)$, the lattice of ideals of the abstract domain[4]. Equivalently, there is an isomorphism between A_σ and $(\mathsf{Fil}(\mathcal{LA}_\sigma))^{\mathrm{op}}$, the lattice of *filters* of $\mathcal{LA}_\sigma$ ordered by reverse inclusion (the definition of a filter is dual to that of an ideal *viz.* a non-empty, upwards closed subset which is closed under binary meets).

So Jensen takes the abstract domain as given, and then presents *that* in logical form, in the spirit of the work of Abramsky or Zhang [Abr91, Zha89]. However, because the abstract domains are actually finite lattices, every ideal is the downwards closure of a single element (*i.e.* every ideal is *principal*). This means that $\mathsf{Idl}(A_\sigma) \cong A_\sigma$, which explains why the slightly more direct relation between $\mathcal{LA}_\sigma$ and A_σ which we have used is equivalent to that used by Jensen.

3.5 Disjunctive Strictness Logic

There are some rather simple terms for which the logic which we have described gives unsatisfactory results. A typical example would be the function g given by

$$\mathsf{g} \stackrel{\mathrm{def}}{=} \lambda x.\lambda y.\mathsf{plus}\,(\mathsf{if}\ x\ \mathsf{then}\ (\underline{1}, y)\ \mathsf{else}\ (y, \underline{1}))$$

[4] Although the published proof incorrectly claims that $\bigwedge_i \phi_i \to \psi_i \leq \bigwedge_j \phi'_j \to \psi'_j$ iff $\forall j.\exists i.\phi'_j \leq \phi_i \,\&\, \psi_i \leq \psi'_j$.

where

$$\mathsf{plus} \stackrel{\text{def}}{=} \lambda p.\mathsf{fst}(p) + \mathsf{snd}(p)$$

It is easy to deduce that $\mathbf{g}$ is strict in x, but we cannot deduce that it is also strict in y, although this is clearly the case. The problem arises from the way in which the strictness properties of the two arms of the conditional are joined together when we try to find a ψ such that

$$\{x : \mathbf{t}, y : \mathbf{f}\} \vdash \mathsf{if}\ x\ \mathsf{then}\ (\underline{1}, y)\ \mathsf{else}\ (y, \underline{1}) : \psi$$

Intuitively, we can see that, in such a context, the conditional expression will either satisfy $\phi_1 = \mathbf{f} \times \mathbf{t}$ or $\phi_2 = \mathbf{t} \times \mathbf{f}$ and that whichever of these alternatives turns out to be the case, the application of plus will satisfy $\mathbf{f}$. In the logic we have given, since it is impossible to tell which ϕ_i will be satisfied by the conditional, the best we can deduce is that it will certainly satisfy $\mathbf{t} \times \mathbf{t}$; but this is too weak to allow the required strictness property of $\mathbf{g}$ to be deduced.

The obvious way to strengthen the logic to cope with this kind of example is to add disjunction, with the intended interpretation of $\phi \vee \psi$ being the union of the interpretations of ϕ and ψ. We should then be able to deduce

$$\{x : \mathbf{t}, y : \mathbf{f}\} \vdash \mathsf{if}\ x\ \mathsf{then}\ (\underline{1}, y)\ \mathsf{else}\ (y, \underline{1}) : (\mathbf{t} \times \mathbf{f}) \vee (\mathbf{f} \times \mathbf{t})$$

and then, provided we can also show

$$\vdash \mathsf{plus} : (\mathbf{t} \times \mathbf{f}) \vee (\mathbf{f} \times \mathbf{t}) \to \mathbf{f}$$

we will be able to deduce that $\mathbf{g}$ is strict in y. For first-order abstract interpretation, this kind of treatment of pairs has been called a *dependent attribute* or *relational* method, by contrast with the *independent attribute* (*i.e.* a property of pairs is a pair of properties) treatment which we described earlier [JM81].

There is a second reason for wanting to add disjunction, which is really a generalisation of the one above. This is to be able to obtain a sensible treatment of strictness properties of sum types. Since we have not discussed these at all so far, we make a brief digression to introduce the language Λ_{T+}, which is Λ_T extended with sums.

3.5.1 The Language Λ_{T+}

Syntax

The types of Λ_{T+} are the same as those for Λ_T, extended with an extra clause for sums:

$$\sigma ::= \iota \mid (\sigma \to \sigma) \mid (\sigma \times \sigma) \mid (\sigma + \sigma)$$

The term-forming rules for Λ_T (Figure 2.1) are extended by the following rules:

$$\frac{t :: \sigma}{\mathsf{inl}_{\sigma+\tau}(t) :: \sigma + \tau} \qquad \frac{t :: \tau}{\mathsf{inr}_{\sigma+\tau}(t) :: \sigma + \tau}$$

$$\frac{t :: \sigma + \tau \qquad u :: \theta \qquad v :: \theta}{\mathsf{case}\, t\, \mathsf{of}\, \mathsf{inl}(x^\sigma) \Rightarrow u \mid \mathsf{inr}(y^\tau) \Rightarrow v :: \theta}$$

where the variables x^σ and y^τ are bound in $\mathsf{case}\, t\, \mathsf{of}\, \mathsf{inl}(x^\sigma) \Rightarrow u \mid \mathsf{inr}(y^\tau) \Rightarrow v$. The definition of free variables is extended by the following clauses:

$$FV(\mathsf{inl}_{\sigma+\tau}(t)) = FV(\mathsf{inr}_{\sigma+\tau}(t)) = FV(t)$$

$$FV(\mathsf{case}\, t\, \mathsf{of}\, \mathsf{inl}(x^\sigma) \Rightarrow u \mid \mathsf{inr}(y^\tau) \Rightarrow v) = FV(t) \cup (FV(u) - \{x^\sigma\}) \cup (FV(v) - \{y^\tau\})$$

and the definition of substitution (Figure 2.2) is extended by

$$\mathsf{inl}_{\tau_1+\tau_2}(t)[s/x^\sigma] = \mathsf{inl}_{\tau_1+\tau_2}(t[s/x^\sigma]) \qquad \mathsf{inr}_{\tau_1+\tau_2}(t)[s/x^\sigma] = \mathsf{inr}_{\tau_1+\tau_2}(t[s/x^\sigma])$$

and (rather unpleasantly)

$$\begin{aligned}&(\mathsf{case}\, t\, \mathsf{of}\, \mathsf{inl}(y_1^{\tau_1}) \Rightarrow u_1 \mid \mathsf{inr}(y_2^{\tau_2}) \Rightarrow u_2)[s/x^\sigma] \\ &\quad = \mathsf{case}\, t[s/x^\sigma]\, \mathsf{of}\, \mathsf{inl}(z_1^{\tau_1}) \Rightarrow v_1 \mid \mathsf{inr}(z_2^{\tau_2}) \Rightarrow v_2\end{aligned}$$

where, for $i \in \{1, 2\}$,

$$z_i^{\tau_i} = \begin{cases} y_i^{\tau_i} & \text{if } y_i^{\tau_i} = x^\sigma \text{ or } y_i^{\tau_i} \notin FV(s) \\ w_i^{\tau_i} \text{ (a fresh variable)} & \text{otherwise} \end{cases}$$

and

$$v_i = \begin{cases} u_i & \text{if } y_i^{\tau_i} = x^\sigma \\ u_i[s/x^\sigma] & \text{if } y_i^{\tau_i} \notin FV(s) \\ u_i[w_i^{\tau_i}/y_i^{\tau_i}][s/x^\sigma] & \text{otherwise} \end{cases}$$

Operational Semantics

The canonical forms of type $\sigma + \tau$ are $\mathsf{inl}_{\sigma+\tau}(t)$ and $\mathsf{inr}_{\sigma+\tau}(t)$. The evaluation rules of the operational semantics (Figure 2.3) are augmented by

$$\mathsf{inl}_{\sigma+\tau}(t) \Downarrow \mathsf{inl}_{\sigma+\tau}(t) \qquad \mathsf{inr}_{\sigma+\tau}(t) \Downarrow \mathsf{inr}_{\sigma+\tau}(t)$$

$$\frac{t \Downarrow \mathsf{inl}_{\sigma+\tau}(t') \qquad u[t'/x^\sigma] \Downarrow c}{\mathsf{case}\, t\, \mathsf{of}\, \mathsf{inl}(x^\sigma) \Rightarrow u \mid \mathsf{inr}(y^\tau) \Rightarrow v \Downarrow c}$$

$$\frac{t \Downarrow \mathsf{inr}_{\sigma+\tau}(t') \qquad v[t'/y^\tau] \Downarrow c}{\mathsf{case}\, t\, \mathsf{of}\, \mathsf{inl}(x^\sigma) \Rightarrow u \mid \mathsf{inr}(y^\tau) \Rightarrow v \Downarrow c}$$

Denotational Semantics

The domain $D_{\sigma+\tau}$ associated with the type $\sigma + \tau$ is $D_\sigma + D_\tau$. Recall that this is the *separated* sum of the domains D_σ and D_τ, and that it comes equipped with the two continuous maps $inl : D_\sigma \to D_\sigma + D_\tau$ and $inr : D_\tau \to D_\sigma + D_\tau$ given

by $inl(d) = (0, d)$ and $inr(e) = (1, e)$. The semantic equations in Figure 2.4 are extended by

$$[\![\mathsf{inl}_{\sigma+\tau}(t)]\!]\rho = inl([\![t]\!]\rho)$$

$$[\![\mathsf{inr}_{\sigma+\tau}(t)]\!]\rho = inr([\![t]\!]\rho)$$

$$[\![\mathsf{case}\, t\, \mathsf{of}\, \mathsf{inl}(x^\sigma) \Rightarrow u \mid \mathsf{inr}(y^\tau) \Rightarrow v]\!]\rho = \begin{cases} \bot & \text{if } [\![t]\!]\rho = \bot_{D_{\sigma+\tau}} \\ [\![u]\!]\rho[x^\sigma \mapsto d] & \text{if } [\![t]\!]\rho = inl(d) \\ [\![v]\!]\rho[y^\tau \mapsto e] & \text{if } [\![t]\!]\rho = inr(e) \end{cases}$$

Computational Adequacy

The results we gave for Λ_T concerning substitution, the denotational semantics and contexts all go through for Λ_{T+}. The proof of computational adequacy also goes through exactly as before, if we extend the definition of the logical relation $\{\mathcal{R}^\sigma\}$ by

$$\begin{aligned} d\,\mathcal{R}^{\sigma+\tau}\,t \;\Leftrightarrow\; & d = \bot \\ \text{or}\;\; & d = inl(d'),\ t{\Downarrow}\mathsf{inl}_{\sigma+\tau}(t') \text{ and } d'\mathcal{R}^\sigma t' \\ \text{or}\;\; & d = inr(d''),\ t{\Downarrow}\mathsf{inr}_{\sigma+\tau}(t'') \text{ and } d''\mathcal{R}^\tau t'' \end{aligned}$$

This concludes the summary of the syntax and semantics of Λ_{T+}, and we now return to strictness analysis.

3.5.2 The Logic of Disjunctive Strictness Properties

To extend strictness logic to Λ_{T+}, we first have to decide what the propositional theory $\mathcal{L}_{\sigma+\tau}$ should contain. That is, which ideals of $D_{\sigma+\tau}$ do we wish to reason about? Clearly, $\mathcal{L}_{\sigma+\tau}$ will contain $\mathbf{t}$ and $\mathbf{f}$, and if $\phi \in \mathcal{L}_\sigma$ then it seems natural to have $\mathsf{inl}_{\sigma+\tau}(\phi) \in \mathcal{L}_{\sigma+\tau}$, with $[\![\mathsf{inl}_{\sigma+\tau}(\phi)]\!] = \overline{inl([\![\phi]\!])}$, and similarly for $\psi \in \mathcal{L}_\tau$.

If we try to axiomatise these properties within the conjunctive framework of our first logic, there are two problems. The first is that we lose the disjunction property, as $[\![\mathbf{t}^{\sigma+\tau}]\!] = [\![\mathsf{inl}_{\sigma+\tau}(\mathbf{t}^\sigma)]\!] \cup [\![\mathsf{inr}_{\sigma+\tau}(\mathbf{t}^\tau)]\!]$. This means, as we remarked after Corollary 3.2.11, that our axiomatisation of the properties of function spaces cannot be complete. The second problem is that the resulting inference system is very weak. This is because, in practice, it is only rarely possible to deduce at compile time which summand a term of type $\sigma + \tau$ lies in. Consequently, the best deducible property of such a term is often simply $\mathbf{t}^{\sigma+\tau}$, so we gain very little useful deductive power compared to the simple system which just has $\mathbf{t}$ and $\mathbf{f}$ in $\mathcal{L}_{\sigma+\tau}$.

So, both in order to strengthen our deductive system, and to have any hope of a good treatment of sum types, we are naturally led to add disjunction to the logic of strictness properties. The formation rules for the family $\{\mathcal{L}_\sigma^\vee\}$ of propositional theories, indexed by the types of Λ_{T+}, is defined in Figure 3.4. The intended interpretations

$$\frac{}{\mathbf{t} \in L^{\vee}_{\sigma}} \qquad \frac{}{\mathbf{f} \in L^{\vee}_{\sigma}}$$

$$\frac{\phi \in L^{\vee}_{\sigma} \qquad \psi \in L^{\vee}_{\sigma}}{\phi \wedge \psi \in L^{\vee}_{\sigma}} \qquad \frac{\phi \in L^{\vee}_{\sigma} \qquad \psi \in L^{\vee}_{\sigma}}{\phi \vee \psi \in L^{\vee}_{\sigma}}$$

$$\frac{\phi \in L^{\vee}_{\sigma} \qquad \psi \in L^{\vee}_{\tau}}{\phi \to \psi \in L^{\vee}_{\sigma \to \tau}} \qquad \frac{\phi \in L^{\vee}_{\sigma} \qquad \psi \in L^{\vee}_{\tau}}{\phi \times \psi \in L^{\vee}_{\sigma \times \tau}}$$

$$\frac{\phi \in L^{\vee}_{\sigma}}{\mathsf{inl}_{\sigma+\tau}(\phi) \in L^{\vee}_{\sigma+\tau}} \qquad \frac{\psi \in L^{\vee}_{\tau}}{\mathsf{inr}_{\sigma+\tau}(\psi) \in L^{\vee}_{\sigma+\tau}}$$

Figure 3.4: Formation Rules for $\mathcal{L}^{\vee}_{\sigma}$

of these propositions are

$$[\![\mathbf{t}^{\sigma}]\!] = D_{\sigma}$$
$$[\![\mathbf{f}^{\sigma}]\!] = \{\bot_{D_{\sigma}}\}$$
$$[\![\phi \wedge \psi]\!] = [\![\phi]\!] \cap [\![\psi]\!]$$
$$[\![\phi \vee \psi]\!] = [\![\phi]\!] \cup [\![\psi]\!]$$
$$[\![\phi^{\sigma} \times \psi^{\tau}]\!] = \{(x, y) \in D_{\sigma \times \tau} \mid x \in [\![\psi^{\sigma}]\!] \text{ and } y \in [\![\phi^{\tau}]\!]\}$$
$$[\![\phi^{\sigma} \to \psi^{\tau}]\!] = \{f \in D_{\sigma \to \tau} \mid f[\![\phi^{\sigma}]\!] \subseteq [\![\psi^{\tau}]\!]\}$$
$$[\![\mathsf{inl}_{\sigma+\tau}(\phi)]\!] = \{inl(d) \mid d \in [\![\phi]\!]\} \cup \{\bot_{D_{\sigma+\tau}}\}$$
$$[\![\mathsf{inr}_{\sigma+\tau}(\psi)]\!] = \{inr(e) \mid e \in [\![\psi]\!]\} \cup \{\bot_{D_{\sigma+\tau}}\}$$

Lemma 3.5.1 *For all $\phi^{\sigma} \in \mathcal{L}^{\vee}_{\sigma}$, $[\![\phi^{\sigma}]\!]$ is an ideal of D_{σ}.*

Proof. Trivial. The fact that the union of two ideals is an ideal may be verified directly or seen as a consequence of the fact that ideals are closed sets. □

The logical inference rules, which are the same at all types, are shown in Figure 3.5, whilst the type-specific rules are in Figure 3.6. The [Irr] rule for function spaces makes use of an auxiliary *irreducibility* predicate on propositions, the inference rules for which are shown in Figure 3.7. This is intended to pick out those propositions whose interpretations are ideals which are not properly expressible as the join of two smaller ideals. Such a proposition has an interpretation which is the downwards closure of a single point[5].

[5]Technically, the property of a topological space that every irreducible closed set is the closure of a unique point is known as *sobriety*.

$$\phi \leq \phi\,[\text{refl}] \qquad \phi \leq \mathbf{t}\,[\text{t}] \qquad \mathbf{f} \leq \phi\,[\text{f}]$$

$$\frac{\phi \leq \psi \qquad \psi \leq \chi}{\phi \leq \chi}\,[\text{trans}]$$

$$\phi \wedge \psi \leq \phi\,[\wedge\text{E-1}] \qquad \phi \wedge \psi \leq \psi\,[\wedge\text{E-2}]$$

$$\phi \leq \phi \vee \psi\,[\vee\text{I-1}] \qquad \psi \leq \phi \vee \psi\,[\vee\text{I-2}]$$

$$\frac{\phi \leq \psi_1 \qquad \phi \leq \psi_2}{\phi \leq \psi_1 \wedge \psi_2}\,[\wedge\text{I}] \qquad \frac{\phi_1 \leq \psi \qquad \phi_2 \leq \psi}{\phi_1 \vee \psi_2 \leq \psi}\,[\vee\text{E}]$$

$$\phi \wedge (\psi \vee \chi) \leq (\phi \wedge \psi) \vee (\phi \wedge \chi)\,[\text{dist}]$$

Figure 3.5: Logical Rules for $\mathcal{L}^{\vee}_{\sigma}$

Proposition 3.5.2 (Soundness)

1. *If* $\mathsf{Irr}(\phi^\sigma)$ *then* $\exists d \in D_\sigma$ *such that* $[\![\phi^\sigma]\!] = \downarrow (d)$.
2. *If* $\phi^\sigma \leq \psi^\sigma$ *then* $[\![\phi^\sigma]\!] \subseteq [\![\psi^\sigma]\!]$.

Proof. The two parts are proved simultaneously by induction on derivations. We consider cases according to the last rule used:

- Case [Irrf]. Then $[\![\mathbf{f}^\sigma]\!] = \{\perp_{D_\sigma}\} = \downarrow (\perp_{D_\sigma})$.
- Case [Irr$\times$]. We have $\mathsf{Irr}(\phi^\sigma)$ and $\mathsf{Irr}(\psi^\tau)$ and so by induction $\exists d \in D_\sigma$, $e \in D_\tau$ such that $[\![\phi^\sigma]\!] = \downarrow (d)$ and $[\![\psi^\tau]\!] = \downarrow (e)$. Hence $[\![\phi^\sigma \times \psi^\tau]\!] = [\![\phi^\sigma]\!] \times [\![\psi^\tau]\!] = \downarrow (d, e)$.
- Case [Irr=]. We have $\mathsf{Irr}(\phi^\sigma)$ and $\phi^\sigma = \psi^\sigma$. By induction $\exists d \in D_\sigma . [\![\phi^\sigma]\!] = \downarrow (d)$ and $[\![\phi^\sigma]\!] = [\![\psi^\sigma]\!]$ so we are done.
- Case [Irr$\rightarrow$]. By induction $\exists e \in D_\tau . [\![\psi^\tau]\!] = \downarrow (e)$ and so

$$[\![\mathbf{t}^\sigma \to \psi^\tau]\!] = \{f : D_\sigma \to D_\tau \mid \forall d \in D_\sigma . f(d) \in [\![\psi^\tau]\!]\} = \downarrow (\lambda d \in D_\sigma . e).$$

- Case [Irr+1]. By induction $\exists d \in d_\sigma . [\![\phi^\sigma]\!] = \downarrow (d)$. Then

$$[\![\mathsf{inl}_{\sigma+\tau}(\phi^\sigma)]\!] = \downarrow (inl(d)).$$

- All the logical rules are sound for trivial set-theoretic reasons.

Function Spaces

$$\mathbf{t} \leq \mathbf{t} \to \mathbf{t}\,[\mathrm{t}\to] \qquad \mathbf{t} \to \mathbf{f} \leq \mathbf{f}\,[\mathrm{f}\to]$$

$$\frac{\phi' \leq \phi \qquad \psi \leq \psi'}{\phi \to \psi \leq \phi' \to \psi'}\,[\to]$$

$$(\phi \to \psi_1) \wedge (\phi \to \psi_2) \leq \phi \to \psi_1 \wedge \psi_2\,[\to \wedge]$$

$$(\phi_1 \to \psi) \wedge (\phi_2 \to \psi) \leq \phi_1 \vee \phi_2 \to \psi\,[\to \vee]$$

$$\frac{\mathsf{Irr}(\phi)}{\phi \to \psi_1 \vee \psi_2 \leq (\phi \to \psi_1) \vee (\phi \to \psi_2)}\,[\mathrm{Irr}]$$

Products

$$\mathbf{t} \leq \mathbf{t} \times \mathbf{t}\,[\mathrm{t}\times] \qquad \mathbf{f} \times \mathbf{f} \leq \mathbf{f}\,[\mathrm{f}\times]$$

$$\frac{\phi \leq \phi' \qquad \psi \leq \psi'}{\phi \times \psi \leq \phi' \times \psi'}\,[\times]$$

$$(\phi \times \psi) \wedge (\phi' \times \psi') \leq (\phi \wedge \phi') \times (\psi \wedge \psi')\,[\times \wedge]$$

$$\phi \times (\psi_1 \vee \psi_2) \leq (\phi \times \psi_1) \vee (\phi \times \psi_2)\,[\times \vee 1]$$

$$(\phi_1 \vee \phi_2) \times \psi \leq (\phi_1 \times \psi) \vee (\phi_2 \times \psi)\,[\times \vee 2]$$

Sums

$$\mathbf{t} \leq \mathsf{inl}(\mathbf{t}) \vee \mathsf{inr}(\mathbf{t})\,[\mathrm{t}+] \qquad \mathsf{inl}(\phi) \wedge \mathsf{inr}(\psi) \leq \mathbf{f}\,[\mathrm{f}+]$$

$$\frac{\phi \leq \phi'}{\mathsf{inl}(\phi) \leq \mathsf{inl}(\phi')}\,[\mathrm{inl}] \qquad \frac{\psi \leq \psi'}{\mathsf{inr}(\psi) \leq \mathsf{inr}(\psi')}\,[\mathrm{inr}]$$

$$\mathsf{inl}(\phi) \wedge \mathsf{inl}(\psi) \leq \mathsf{inl}(\phi \wedge \psi)\,[+ \wedge 1] \qquad \mathsf{inr}(\phi) \wedge \mathsf{inr}(\psi) \leq \mathsf{inr}(\phi \wedge \psi)\,[+ \wedge 2]$$

$$\mathsf{inl}(\phi \vee \psi) \leq \mathsf{inl}(\phi) \vee \mathsf{inl}(\psi)\,[+ \vee 1] \qquad \mathsf{inr}(\phi \vee \psi) \leq \mathsf{inr}(\phi) \vee \mathsf{inr}(\psi)\,[+ \vee 2]$$

Figure 3.6: Type-specific Rules for $\mathcal{L}_\sigma^\vee$

$$\mathsf{Irr}(\mathbf{f})\ [\text{Irrf}] \qquad \frac{\mathsf{Irr}(\phi) \qquad \psi = \phi}{\mathsf{Irr}(\psi)}\ [\text{Irr}=]$$

$$\frac{\mathsf{Irr}(\phi) \qquad \mathsf{Irr}(\psi)}{\mathsf{Irr}(\phi \times \psi)}\ [\text{Irr}\times] \qquad \frac{\mathsf{Irr}(\psi)}{\mathsf{Irr}(\mathbf{t} \to \psi)}\ [\text{Irr}\to]$$

$$\frac{\mathsf{Irr}(\phi)}{\mathsf{Irr}(\mathsf{inl}(\phi))}\ [\text{Irr}+1] \qquad \frac{\mathsf{Irr}(\psi)}{\mathsf{Irr}(\mathsf{inr}(\psi))}\ [\text{Irr}+2]$$

Figure 3.7: Irreducible Propositions

- Case [Irr]. We have

$$\frac{\mathsf{Irr}(\phi^\sigma)}{\phi^\sigma \to \psi_1^\tau \vee \psi_2^\tau \leq (\phi^\sigma \to \psi_1^\tau) \vee (\phi^\sigma \to \psi_2^\tau)}\ [\text{Irr}]$$

  By induction, $\exists d \in D_\sigma.[\![\phi^\sigma]\!] = \downarrow(d)$. Then if $f \models \phi^\sigma \to \psi_1^\tau \vee \psi_2^\tau$, since $d \models \phi^\sigma$ we must have $f(d) \models \psi_1^\tau \vee \psi_2^\tau$ so that $f(d) \models \psi_i^\tau$ for some $i \in \{1, 2\}$. But then for any $d' \in D_\sigma$ such that $d' \models \phi^\sigma$, $d' \sqsubseteq d$ so that $f(d') \sqsubseteq f(d)$ by monotonicity and therefore $f(d') \models \psi_i^\tau$. This means $f \models \phi^\sigma \to \phi_i^\tau$ for some i, so we are done.

- The remaining type-specific rules are easily verified.

□

Corollary 3.5.3 *For all types σ of Λ_T, $\mathcal{L}_\sigma^\vee$ is a conservative extension of $\mathcal{L}_\sigma$.*

Proof. This is an immediate consequence of Proposition 3.5.2 and Corollary 3.2.10, since if $\phi, \psi \in \mathcal{L}_\sigma$ then $\phi \leq \psi$ is provable in $\mathcal{L}_\sigma^\vee$ implies that $[\![\phi]\!] \subseteq [\![\psi]\!]$ which implies that $\phi \leq \psi$ is provable in $\mathcal{L}_\sigma$ by completeness. □

Remarks 3.5.4

- The other distributive law:

$$(\phi \vee \psi) \wedge (\phi \vee \chi) \leq \phi \vee (\psi \wedge \chi)$$

  is a derivable rule in $\mathcal{L}_\sigma^\vee$.

- Each $\mathcal{LA}_\sigma^\vee$ is a finite distributive lattice.

- Obviously, we could have used n-ary conjunctions and disjunctions. However, in contrast to the situation for conjunction, the nullary form of the $[\to \vee]$ rule, *viz.*

$$\mathbf{t}^{\sigma\to\tau} \leq \mathbf{f}^{\sigma} \to \phi^{\tau}$$

  is *not* a rule of our logic. It would be valid for strict function spaces.

- Any $\phi \in \mathcal{L}_{\iota}^{\vee}$ is logically equivalent to $\mathbf{t}^{\iota}$ or $\mathbf{f}^{\iota}$.

- Any $\phi^{\sigma\times\tau}$ is logically equivalent to a proposition of the form $\bigvee_{i\in I}(\phi_i^{\sigma} \times \psi_i^{\sigma})$.

- Any proposition in $\mathcal{L}_{\sigma\to\tau}^{\vee}$ is logically equivalent to one of the form

$$\bigvee_{i\in I}\bigwedge_{j\in J_i}(\phi_{ij}^{\sigma} \to \psi_{ij}^{\tau})$$

- Any proposition in $\mathcal{L}_{\sigma+\tau}^{\vee}$ is logically equivalent either to $\mathbf{f}^{\sigma+\tau}$ or to a proposition of the form $\mathsf{inl}_{\sigma+\tau}(\phi^{\sigma}) \vee \mathsf{inr}_{\sigma+\tau}(\psi^{\tau})$.

I do not know whether the following is true or not:

Conjecture 3.5.5 (Completeness)

1. *If* $\exists d \in D_{\sigma}$ *such that* $[\![\phi^{\sigma}]\!] = {\downarrow}(d)$ *then* $\mathsf{Irr}(\phi^{\sigma})$.
2. *If* $[\![\phi^{\sigma}]\!] \subseteq [\![\psi^{\sigma}]\!]$ *then* $\phi^{\sigma} \leq \psi^{\sigma}$.

□

It would be nice if we could approach the question of completeness by first proving that every proposition could be put into a well-behaved normal form, in which disjunction only appeared at the outermost level. Unfortunately, this is not the case. The reason for this is that there are many propositions which are neither irreducible nor expressible as the disjunction of irreducible propositions ($\mathbf{t}^{\iota}$ is one such proposition).

Note also that if Conjecture 3.5.5 is true, it must depend on a fairly delicate property of the domains and ideals we have chosen. This is that the extent of each proposition which is neither irreducible nor expressible as a join of irreducibles is 'infinitely wide'. More formally (although this may not be quite the right statement of the condition), for each proposition ϕ^{σ} which satisfies the conditions above, there must be no finite antichain $A \subseteq D_{\sigma}$ such that $[\![\phi^{\sigma}]\!] = \bigcup_{a\in A} {\downarrow}(a)$.

To see the reason why we cannot have completeness without some property like this, consider what would happen if we were to add a type of booleans to our language, treated in the same way as the existing type of natural numbers. The domain associated with the boolean type would be the three point domain $\{true, false\}_{\perp}$ and $\mathcal{LA}_{\mathrm{bool}}^{\vee}$ would just contain $\mathbf{t}$ and $\mathbf{f}$. Note that $\mathbf{t}^{\mathrm{bool}}$ is certainly not irreducible,

but has what we might call a 'width' of 2. Now write σ for $\iota + (\iota + \iota)$. We have the following:

$$\begin{aligned}[\![\mathbf{t}^{\mathrm{bool}} \to \mathsf{inl}_\sigma(\mathbf{t}) \vee \mathsf{inr}_\sigma(\mathsf{inl}_{\iota+\iota}(\mathbf{t})) \vee \mathsf{inr}_\sigma(\mathsf{inr}_{\iota+\iota}(\mathbf{t}))]\!] \subseteq \\ [\![\quad (\mathbf{t}^{\mathrm{bool}} \to \mathsf{inl}_\sigma(\mathbf{t}) \vee \mathsf{inr}_\sigma(\mathsf{inl}_{\iota+\iota}(\mathbf{t}))) \\ \vee \quad (\mathbf{t}^{\mathrm{bool}} \to \mathsf{inl}_\sigma(\mathbf{t}) \vee \mathsf{inr}_\sigma(\mathsf{inr}_{\iota+\iota}(\mathbf{t}))) \\ \vee \quad (\mathbf{t}^{\mathrm{bool}} \to \mathsf{inr}_\sigma(\mathsf{inl}_{\iota+\iota}(\mathbf{t})) \vee \mathsf{inr}_\sigma(\mathsf{inr}_{\iota+\iota}(\mathbf{t}))) \\ ]\!]\end{aligned}$$

But there is no way to deduce the corresponding fact in the logic. The attempt to fix this by adding the rule

$$\mathbf{t}^{\mathrm{bool}} \to \phi_1 \vee \phi_2 \vee \phi_3 \leq (\mathbf{t}^{\mathrm{bool}} \to \phi_1 \vee \phi_2) \vee (\mathbf{t}^{\mathrm{bool}} \to \phi_1 \vee \phi_3) \vee (\mathbf{t}^{\mathrm{bool}} \to \phi_2 \vee \phi_3)$$

still leads to an incomplete system because there are higher type propositions which have larger finite widths (*e.g.* $\mathbf{t}^{\mathrm{bool}\to\mathrm{bool}}$ has width 4). Even if there were any way of calculating these widths automatically, the resulting inference system would be extremely unpleasant.

We can obtain a better treatment of booleans by adding two more basic irreducible propositions to $\mathcal{L}^{\vee}_{\mathrm{bool}}$, namely $\mathsf{inl}_{\mathrm{bool}}()$ and $\mathsf{inr}_{\mathrm{bool}}()$ with $[\![\mathsf{inl}_{\mathrm{bool}}()]\!] = \{\mathrm{true}, \bot\}$ and $[\![\mathsf{inr}_{\mathrm{bool}}()]\!] = \{\mathrm{false}, \bot\}$. However, the property logic is still not complete, as we are unable to show, for example, that

$$(\mathsf{inl}_{\mathrm{bool}}() \to \mathsf{inl}_{\mathrm{bool}}()) \wedge (\mathsf{inr}_{\mathrm{bool}}() \to \mathsf{inr}_{\mathrm{bool}}())$$

is irreducible. This could be remedied by adding another rule to the irreducibility predicate, such as

$$\frac{\forall i \in I.\mathsf{Irr}(\phi_i)\ \&\ \mathsf{Irr}(\psi_i) \qquad \mathbf{t} \leq \bigwedge_{i\in I} \phi_i}{\mathsf{Irr}(\textstyle\bigwedge_{i\in I}(\phi_i \to \psi_i))}$$

But, for the moment, we must leave further investigation of these matters for future work.

It should be noted that the way in which the lattices $\mathcal{LA}^{\vee}_{\sigma}$ are constructed does not fit the standard notion of how abstract domains should be built up. Usually, the abstract domain associated with a type $\sigma \diamond \tau$, where $\diamond$ is some type constructor, is $A_\sigma \hat{\diamond} A_\tau$ where A_σ, A_τ are the abstract domains associated with σ and τ and $\hat{\diamond}$ is some operation on abstract domains. The situation is more complicated for the disjunctive logic, as $\mathcal{LA}^{\vee}_{\sigma\diamond\tau}$ is not obtainable purely from the two lattices $\mathcal{LA}^{\vee}_{\sigma}$ and $\mathcal{LA}^{\vee}_{\tau}$ – it is also necessary to know which elements of the component lattices are to be regarded as irreducible. To give a concrete example, consider adding a unit type with D_{unit} defined to be the two-point domain $\{*\}_\bot$. The only sensible way to treat this in our logic is to have $\mathcal{LA}^{\vee}_{\mathrm{unit}}$ just containing $\mathbf{t}$ and $\mathbf{f}$. This means that $\mathcal{LA}^{\vee}_{\mathrm{unit}}$ is isomorphic to $\mathcal{LA}^{\vee}_{\iota}$. However, $\mathbf{t}^{\mathrm{unit}}$ is irreducible, whereas $\mathbf{t}^{\iota}$ is not. This means that $\mathcal{LA}^{\vee}_{\mathrm{unit}\to\iota+\iota}$ is *not* isomorphic to $\mathcal{LA}^{\vee}_{\iota\to\iota+\iota}$.

Interestingly, the disjunctive strictness properties are expressive enough for us to be able to see where full abstraction fails for our language. It is the case that for any *closed* term t of type $\iota \times \iota \to \iota$, if

$$[\![t]\!] \models \mathbf{f} \times \mathbf{f} \to \mathbf{f}$$

then

$$[\![t]\!] \models (\mathbf{t} \times \mathbf{f} \to \mathbf{f}) \vee (\mathbf{f} \times \mathbf{t} \to \mathbf{f})$$

Intuitively, because t is sequential, if it diverges when both its arguments do then either it always diverges, or it always evaluates a particular one of its arguments first and will thus diverge if that argument does, irrespective of whether the other does or not. The axiom

$$\mathbf{f} \times \mathbf{f} \to \mathbf{f} \;\leq\; (\mathbf{t} \times \mathbf{f} \to \mathbf{f}) \vee (\mathbf{f} \times \mathbf{t} \to \mathbf{f})$$

is not, however, valid in our denotational semantics. Nor would it be operationally sound to add it to our logic, because of the restriction to closed terms above. This indicates that, for a sequential language, even if we had a complete axiomatisation of the inclusion order on our disjunctive collection of ideals it would still be in some sense incomplete for reasoning about the entailment relation between the operational properties which those ideals are intended to denote.

3.5.3 The Disjunctive Program Logic

Despite all these complications, we do still have a sound logic for reasoning about disjunctive strictness properties, and we can give an associated program logic which allows such properties to be assigned to terms of Λ_{T+}. This is shown in Figure 3.8.

Proposition 3.5.6 (Soundness of $\mathcal{PL}\vee$) *If* $\Gamma \vdash^{\vee} s : \phi^{\sigma}$ *then* $\Gamma \models s : \phi^{\sigma}$.

Proof. This is proved by a fairly straightforward induction on the derivation of $\Gamma \vdash s : \phi^{\sigma}$. Most of the cases are the same as for the proof of Proposition 3.3.1. Some of the new cases are:

- If the derivation ends in [sum1], so we have

$$\frac{\Gamma \vdash s : \phi^{\sigma}}{\Gamma \vdash \mathsf{inl}_{\sigma+\tau}(s) : \mathsf{inl}_{\sigma+\tau}(\phi^{\sigma})}\,[\text{sum1}]$$

  then for any environment ρ such that $\rho \models \Gamma$, we have $[\![s]\!]\rho \models \phi^{\sigma}$ by induction. Then $[\![\mathsf{inl}_{\sigma+\tau}(s)]\!]\rho = \mathit{inl}([\![s]\!]\rho)$ which clearly satisfies $\mathsf{inl}_{\sigma+\tau}(\phi^{\sigma})$.

- If the last rule is [case2], we have

$$\frac{\Gamma \vdash t : \mathsf{inl}_{\sigma+\tau}(\phi^{\sigma}) \qquad \Gamma, x^{\sigma} : \phi^{\sigma} \vdash u : \psi^{\theta}}{\Gamma \vdash \mathsf{case}\, t\, \mathsf{of}\, \mathsf{inl}(x^{\sigma}) \Rightarrow u \mid \mathsf{inr}(y^{\tau}) \Rightarrow v : \psi^{\theta}}$$

  For any ρ such that $\rho \models \Gamma$, we know by induction that $[\![t]\!]\rho \models \mathsf{inl}_{\sigma+\tau}(\phi^{\sigma})$. This means that either

$$\Gamma \vdash \underline{n} : \mathbf{t}^\iota \ [\text{nat}] \qquad \Gamma, x^\sigma : \phi^\sigma \vdash x^\sigma : \phi^\sigma \ [\text{var}]$$

$$\frac{\Gamma, x^\sigma : \phi^\sigma \vdash t : \psi^\tau}{\Gamma \vdash \lambda x^\sigma.t : \phi^\sigma \to \psi^\tau}\ [\text{abs}] \qquad \frac{\Gamma \vdash t : \phi^\sigma \to \psi^\tau \qquad \Gamma \vdash s : \phi^\sigma}{\Gamma \vdash (t\,s) : \psi^\tau}\ [\text{app}]$$

$$\frac{\Gamma \vdash s : \phi^\sigma \qquad \Gamma \vdash t : \psi^\tau}{\Gamma \vdash (s,t) : \phi^\sigma \times \psi^\tau}\ [\text{pair}] \qquad \frac{\Gamma \vdash s : \phi^\sigma \times \psi^\tau}{\Gamma \vdash \mathsf{fst}(s) : \phi^\sigma}\ [\text{fst}]$$

$$\frac{\Gamma \vdash s : \phi^\sigma \times \psi^\tau}{\Gamma \vdash \mathsf{snd}(s) : \psi^\tau}\ [\text{snd}] \qquad \frac{\Gamma \vdash s : \phi^\sigma}{\Gamma \vdash \mathsf{inl}_{\sigma+\tau}(s) : \mathsf{inl}_{\sigma+\tau}(\phi^\sigma)}\ [\text{sum1}]$$

$$\frac{\Gamma \vdash t : \psi^\tau}{\Gamma \vdash \mathsf{inr}_{\sigma+\tau}(t) : \mathsf{inr}_{\sigma+\tau}(\psi^\tau)}\ [\text{sum2}] \qquad \frac{\Gamma \vdash s : \phi^\sigma \qquad \phi^\sigma \leq \psi^\sigma}{\Gamma \vdash s : \psi^\sigma}\ [\text{sub}]$$

$$\frac{\Gamma \vdash t : \mathbf{f}^{\sigma+\tau}}{\Gamma \vdash \mathsf{case}\,t\,\mathsf{of}\,\mathsf{inl}(x^\sigma) \Rightarrow u \mid \mathsf{inr}(y^\tau) \Rightarrow v : \mathbf{f}^\theta}\ [\text{case1}]$$

$$\frac{\Gamma \vdash t : \mathsf{inl}_{\sigma+\tau}(\phi^\sigma) \qquad \Gamma, x^\sigma : \phi^\sigma \vdash u : \psi^\theta}{\Gamma \vdash \mathsf{case}\,t\,\mathsf{of}\,\mathsf{inl}(x^\sigma) \Rightarrow u \mid \mathsf{inr}(y^\tau) \Rightarrow v : \psi^\theta}\ [\text{case2}]$$

$$\frac{\Gamma \vdash t : \mathsf{inr}_{\sigma+\tau}(\phi^\tau) \qquad \Gamma, y^\tau : \phi^\tau \vdash v : \psi^\theta}{\Gamma \vdash \mathsf{case}\,t\,\mathsf{of}\,\mathsf{inl}(x^\sigma) \Rightarrow u \mid \mathsf{inr}(y^\tau) \Rightarrow v : \psi^\theta}\ [\text{case3}]$$

$$\frac{\Gamma \vdash s : \phi^\sigma \qquad \Gamma \vdash s : \psi^\sigma}{\Gamma \vdash s : \phi^\sigma \wedge \psi^\sigma}\ [\text{conj}] \qquad \frac{\Gamma, x^\sigma : \phi^\sigma \vdash t : \chi^\tau \qquad \Gamma, x^\sigma : \psi^\sigma \vdash t : \chi^\tau}{\Gamma, x^\sigma : \phi^\sigma \vee \psi^\sigma \vdash t : \chi^\tau}\ [\text{disj}]$$

$$\frac{\Gamma \vdash s : \mathbf{f}^\iota}{\Gamma \vdash \mathsf{if}\ s\ \mathsf{then}\ t_1\ \mathsf{else}\ t_2 : \mathbf{f}^\tau}\ [\text{cond1}] \qquad \frac{\Gamma \vdash t_1 : \phi^\tau \qquad \Gamma \vdash t_2 : \phi^\tau}{\Gamma \vdash \mathsf{if}\ s\ \mathsf{then}\ t_1\ \mathsf{else}\ t_2 : \phi^\tau}\ [\text{cond2}]$$

$$\frac{\Gamma, x^\sigma : \phi^\sigma \vdash s : \phi^\sigma}{\Gamma \vdash \mathsf{fix}(x^\sigma.s) : \phi^\sigma}\ [\text{fix}] \qquad \frac{\Gamma \vdash s : \phi^\iota \qquad \Gamma \vdash t : \psi^\iota}{\Gamma \vdash s + t : \phi^\iota \wedge \psi^\iota}\ [\text{arith}]$$

Figure 3.8: The Program Logic $\mathcal{PL}\vee$

(a) $[\![t]\!]\rho = \bot_{D_{\sigma+\tau}}$, in which case

$$[\![\mathsf{case}\, t\, \mathsf{of}\, \mathsf{inl}(x^\sigma) \Rightarrow u \mid \mathsf{inr}(y^\tau) \Rightarrow v]\!]\rho = \bot_{D_\theta}$$

and we are done since $\bot_{D_\theta} \models \psi^\theta$, or

(b) $[\![t]\!]\rho = inl(d)$, where $d \models \phi^\sigma$. In this case,

$$[\![\mathsf{case}\, t\, \mathsf{of}\, \mathsf{inl}(x^\sigma) \Rightarrow u \mid \mathsf{inr}(y^\tau) \Rightarrow v]\!]\rho = [\![u]\!]\rho[x^\sigma \mapsto d]$$

and since $\rho[x^\sigma \mapsto d] \models \Gamma, x^\sigma : \phi^\sigma$ we know $[\![u]\!]\rho[x^\sigma \mapsto d] \models \psi^\theta$ by induction.

- If the last rule is [disj], so the situation is

$$\frac{\Gamma, x^\sigma : \phi_1^\sigma \vdash t : \psi^\tau \qquad \Gamma, x^\sigma : \phi_2^\sigma \vdash t : \psi^\tau}{\Gamma, x^\sigma : \phi_1^\sigma \vee \phi_2^\sigma \vdash t : \psi^\tau}\,[\mathrm{disj}]$$

then given ρ such that $\rho \models \Gamma, x^\sigma : \phi_1^\sigma \vee \phi_2^\sigma$, it is clear that for some $i \in \{1, 2\}$ $\rho \models \Gamma, x^\sigma : \phi_i^\sigma$. Thus $[\![t]\!]\rho \models \psi^\tau$ by induction applied to the i^{th} premiss of the last rule.

□

Example We can now deduce that the function g, which we gave earlier as an example of a term for which our conjunctive logic gave unsatisfactory results, is strict in its second argument. Recall that g was defined as follows:

$$\mathsf{g} \stackrel{\text{def}}{=} \lambda x.\lambda y.\mathsf{plus}\,(\mathsf{if}\ x\ \mathsf{then}\ (\underline{1}, y)\ \mathsf{else}\ (y, \underline{1}))$$

where

$$\mathsf{plus} \stackrel{\text{def}}{=} \lambda p.\mathsf{fst}(p) + \mathsf{snd}(p)$$

Write Γ for the context $\{x : \mathbf{t}, y : \mathbf{f}\}$, and Π for the following derivation:

$$\dfrac{\dfrac{\dfrac{\dfrac{}{\Gamma, p : \mathbf{t} \times \mathbf{f} \vdash p : \mathbf{t} \times \mathbf{f}}\,[\mathrm{var}]}{\Gamma, p : \mathbf{t} \times \mathbf{f} \vdash \mathsf{fst}(p) : \mathbf{t}}\,[\mathrm{fst}] \qquad \dfrac{\dfrac{}{\Gamma, p : \mathbf{t} \times \mathbf{f} \vdash p : \mathbf{t} \times \mathbf{f}}\,[\mathrm{var}]}{\Gamma, p : \mathbf{t} \times \mathbf{f} \vdash \mathsf{snd}(p) : \mathbf{f}}\,[\mathrm{snd}]}{\Gamma, p : \mathbf{t} \times \mathbf{f} \vdash \mathsf{fst}(p) + \mathsf{snd}(p) : \mathbf{t} \wedge \mathbf{f}}\,[\mathrm{arith}] \qquad \mathbf{t} \wedge \mathbf{f} \leq \mathbf{f}}{\Gamma, p : \mathbf{t} \times \mathbf{f} \vdash \mathsf{fst}(p) + \mathsf{snd}(p) : \mathbf{f}}\,[\mathrm{sub}]$$

There is a similar derivation Π' of

$$\Gamma, p : \mathbf{f} \times \mathbf{t} \vdash \mathsf{fst}(p) + \mathsf{snd}(p) : \mathbf{f}$$

so that we can form Π'':

$$\dfrac{\dfrac{\Pi \qquad \Pi'}{\Gamma, p : (\mathbf{t} \times \mathbf{f}) \vee (\mathbf{f} \times \mathbf{t}) \vdash \mathsf{fst}(p) + \mathsf{snd}(p) : \mathbf{f}}\,[\mathrm{disj}]}{\Gamma \vdash \mathsf{plus} : (\mathbf{t} \times \mathbf{f}) \vee (\mathbf{f} \times \mathbf{t}) \to \mathbf{f}}\,[\mathrm{abs}]$$

We can also form the derivation Σ:

$$\dfrac{\dfrac{\dfrac{}{\Gamma \vdash \underline{1} : \mathbf{t}}\,[\text{nat}] \qquad \dfrac{}{\Gamma \vdash y : \mathbf{f}}\,[\text{var}]}{\Gamma \vdash (\underline{1}, y) : \mathbf{t} \times \mathbf{f}}\,[\text{pair}] \qquad \mathbf{t} \times \mathbf{f} \leq (\mathbf{t} \times \mathbf{f}) \vee (\mathbf{f} \times \mathbf{t})}{\Gamma \vdash (\underline{1}, y) : (\mathbf{t} \times \mathbf{f}) \vee (\mathbf{f} \times \mathbf{t})}\,[\text{sub}]$$

and there is a similar derivation Σ' of

$$\Gamma \vdash (y, \underline{1}) : (\mathbf{t} \times \mathbf{f}) \vee (\mathbf{f} \times \mathbf{t})$$

so that we can form

$$\dfrac{\Pi'' \qquad \dfrac{\dfrac{\dfrac{\dfrac{\Sigma \qquad \Sigma'}{\Gamma \vdash \mathsf{if}\ x\ \mathsf{then}\ (\underline{1}, y)\ \mathsf{else}\ (y, \underline{1}) : (\mathbf{t} \times \mathbf{f}) \vee (\mathbf{f} \times \mathbf{t})}\,[\text{cond2}]}{\Gamma \vdash \mathsf{plus}\,(\mathsf{if}\ x\ \mathsf{then}\ (\underline{1}, y)\ \mathsf{else}\ (y, \underline{1})) : \mathbf{f}}\,[\text{app}]}{\{x : \mathbf{t}\} \vdash \lambda y.\mathsf{plus}\,(\mathsf{if}\ x\ \mathsf{then}\ (\underline{1}, y)\ \mathsf{else}\ (y, \underline{1})) : \mathbf{f} \to \mathbf{f}}\,[\text{abs}]}{\{\} \vdash \mathsf{g} : \mathbf{t} \to \mathbf{f} \to \mathbf{f}}\,[\text{abs}]$$

as required.

3.5.4 Related Work

Nielson, in [Nie85], describes a first-order abstract interpretation which uses a tensor product of abstract domains to abstract product types. His work is restricted to atomically generated lattices of properties. Burn, in [Bur91a], considers extending this idea to higher-order strictness analysis and discovers this to be problematic.

Jensen has also looked at extending his strictness logic by adding disjunction [Jen92b]. He presents a disjunctive logic for a language with base types, pairs and function spaces, and relates this to a model using downwards-closed subsets of domains from an 'abstract interpretation' which is based on tensor products and linear function spaces. However, neither the logic nor the non-standard semantics is related to the standard semantics. Unfortunately, they are both unsound.

The main difference between Jensen's logic and ours is in the treatment of irreducibles. Jensen's system essentially treats as irreducible all those propositions which come from the conjunctive logic we described earlier. For example, he treats $\mathbf{t}^\sigma$ as irreducible for all σ. Whilst the interpretation of these propositions may be irreducible in the lattice consisting of those ideals which we have chosen to reason about (cf. Corollary 3.2.11), they are not irreducible in the lattice $\mathcal{P}_H(D_\sigma)$ of *all* ideals (though Jensen does not give any interpretation of propositions as program properties). As a simple example to show how this makes the logic unsound, consider the term (omitting type annotations)

$$\mathsf{g} \stackrel{\text{def}}{=} \lambda z.(\lambda f.\mathsf{snd}(f\ \underline{1}\ z) + \mathsf{fst}(f\ \underline{0}\ z))\ \mathsf{h}$$

where

$$\mathsf{h} \stackrel{\text{def}}{=} \lambda x.\lambda y.\mathsf{if}\ x\ \mathsf{then}\ (\underline{1}, y)\ \mathsf{else}\ (y, \underline{1})$$

Now, for any Γ, we can certainly deduce

$$\Gamma, x : \mathbf{t}, y : \mathbf{f} \vdash \mathsf{if}\ x\ \mathsf{then}\ (\underline{1}, y)\ \mathsf{else}\ (y, \underline{1}) : (\mathbf{t} \times \mathbf{f}) \vee (\mathbf{f} \times \mathbf{t})$$

and hence

$$\Gamma, x : \mathbf{t} \vdash \lambda y.\mathsf{if}\ x\ \mathsf{then}\ (\underline{1}, y)\ \mathsf{else}\ (y, \underline{1}) : \mathbf{f} \to (\mathbf{t} \times \mathbf{f}) \vee (\mathbf{f} \times \mathbf{t})$$

Then as $\mathsf{Irr}(\mathbf{f})$, we can correctly deduce that

$$\Gamma, x : \mathbf{t} \vdash \lambda y.\mathsf{if}\ x\ \mathsf{then}\ (\underline{1}, y)\ \mathsf{else}\ (y, \underline{1}) : (\mathbf{f} \to \mathbf{t} \times \mathbf{f}) \vee (\mathbf{f} \to \mathbf{f} \times \mathbf{t})$$

So that

$$\Gamma \vdash \mathsf{h} : \mathbf{t} \to (\mathbf{f} \to \mathbf{t} \times \mathbf{f}) \vee (\mathbf{f} \to \mathbf{f} \times \mathbf{t})$$

If we wrongly have $\mathsf{Irr}(\mathbf{t})$ then we can then derive

$$\Gamma \vdash \mathsf{h} : (\mathbf{t} \to \mathbf{f} \to \mathbf{t} \times \mathbf{f}) \vee (\mathbf{t} \to \mathbf{f} \to \mathbf{f} \times \mathbf{t})$$

Now consider the subterm $(\lambda f.\mathsf{snd}(f\,\underline{1}\,z) + \mathsf{fst}(f\,\underline{0}\,z))$ of g. We can deduce

$$\{z : \mathbf{f}, f : \mathbf{t} \to \mathbf{f} \to \mathbf{t} \times \mathbf{f}\} \vdash \mathsf{snd}(f\,\underline{1}\,z) : \mathbf{f}$$

and hence that

$$\{z : \mathbf{f}, f : \mathbf{t} \to \mathbf{f} \to \mathbf{t} \times \mathbf{f}\} \vdash \mathsf{snd}(f\,\underline{1}\,z) + \mathsf{fst}(f\,\underline{0}\,z) : \mathbf{f}$$

Similarly

$$\{z : \mathbf{f}, f : \mathbf{t} \to \mathbf{f} \to \mathbf{f} \times \mathbf{t}\} \vdash \mathsf{snd}(f\,\underline{1}\,z) + \mathsf{fst}(f\,\underline{0}\,z) : \mathbf{f}$$

And hence, by [disj],

$$\{z : \mathbf{f}, f : (\mathbf{t} \to \mathbf{f} \to \mathbf{t} \times \mathbf{f}) \vee (\mathbf{t} \to \mathbf{f} \to \mathbf{f} \times \mathbf{t})\} \vdash \mathsf{snd}(f\,\underline{1}\,z) + \mathsf{fst}(f\,\underline{0}\,z) : \mathbf{f}$$

So that

$$\{z : \mathbf{f}\} \vdash \lambda f.\mathsf{snd}(f\,\underline{1}\,z) + \mathsf{fst}(f\,\underline{0}\,z) : (\mathbf{t} \to \mathbf{f} \to \mathbf{t} \times \mathbf{f}) \vee (\mathbf{t} \to \mathbf{f} \to \mathbf{f} \times \mathbf{t}) \to \mathbf{f}$$

Then by [app]

$$\{z : \mathbf{f}\} \vdash (\lambda f.\mathsf{snd}(f\,\underline{1}\,z) + \mathsf{fst}(f\,\underline{0}z))\,\mathsf{h} : \mathbf{f}$$

so

$$\{\} \vdash \mathsf{g} : \mathbf{f} \to \mathbf{f}$$

which means that g is strict. This is untrue, as g is easily seen to be the constant 2 function. The only error is the step which assumes that $\mathbf{t}'$ is irreducible.

Chapter 4

Algebraic Datatypes

4.1 Introduction

The language Λ_T as described so far lacks an important feature of all real functional languages—it has no support for any kind of recursive datatype. Moreover, such types cannot easily be translated into Λ_T (in contrast to the situation for the pure untyped lambda calculus or, in certain cases, any of the higher order typed lambda calculi). This is a serious shortcoming, as recursive types, especially lists, are a central part of functional programming.

Even before we start thinking about the extra complications introduced by polymorphism, there are roughly three levels of complexity of recursive type definitions. The terminology is far from fixed, and there are various subtleties concerning the difference between strict and lazy languages, but, at least informally, we will distinguish between

1. Algebraic types. These are also known as sum-of-products types or polynomial types. This class includes lists, binary trees, syntax trees, and so on. They are characterised by the fact that the body of the type definition is made up of sums and products of simple types, previously defined datatypes and recursive occurrences of the name of the type being defined. A typical example would be (in MLish syntax) the type of lists of functions from integers to integers:

   $$\mathsf{datatype\ flist} = \mathsf{Nil} \mid \mathsf{Cons\ of\ (int}\rightarrow\mathsf{int)}\times\mathsf{flist;}$$

   Note that this definition includes an arrow, but is still algebraic since the type being defined does not occur within the scope of the arrow.

2. Inductive types. These extend the algebraic types by allowing the name of the type being defined to occur positively within function spaces. We refrain from giving a detailed definition of positive and negative occurrences, but for non-nested arrow types this means that the type being defined may occur on the right hand end of an arrow, but not on the left. A typical example would

be the following type of infinitely branching trees, with integer labels on the nodes:

$$\mathsf{datatype\ itree} = \mathsf{Empty} \mid \mathsf{Node\ of\ int}\times(\mathsf{int}\to\mathsf{itree});$$

3. Recursive types. These allow the body of the definition to be an arbitrary type expression involving the type being defined. This extra power is not used very often in practice. A trivial (but genuine) example of its use is the following datatype which occurs in an interpreter, written in ML, for an untyped version of Λ_T. The datatype for expressions includes a clause for arithmetic operations, which are implemented by ML functions:

$$\mathsf{datatype\ exp} = \mathsf{N\ of\ int} \mid \ldots \mid \mathsf{Arith\ of\ exp}\times\mathsf{exp}\to\mathsf{exp};$$

It should be noted that from the point of view of implementations, there is no particular difference between the three classes. Furthermore, the techniques for solving recursive domain equations which we shall describe in the next section will allow us to give a denotational semantics to arbitrary recursive types. The real distinction comes when we try to find reasoning principles. Whilst algebraic and inductive types are fairly well understood, the subject of reasoning principles for more general classes of recursive types is still an active area of research. See, for example, [Pit92].

The rest of this chapter is organised as follows. Section 4.2 contains some basic material on the solution of recursive domain equations. The account given here follows that of [Plo79]. A more general categorical treatment may be found in [SP82] and presentations using predomains and partial functions in [Ten91] and [Plo85].

Section 4.3 uses this material to give the semantics of an extension of Λ_T with lazy algebraic types. Section 4.4 then gives a brief overview of existing work on strictness analysis in the presence of recursive types, including abstract interpretation and projection-based approaches. Finally, in Section 4.5, we suggest a new construction of lattices of strictness properties of algebraic types. This is motivated by the categorical view of the solutions of the domain equations associated with these types as initial algebras.

4.2 Recursive Domain Equations

If we wish to give a denotational semantics to a language which includes recursive types, then we need to find appropriate domains for modelling such types. A domain D, suitable for giving the denotations of terms of type natlist, where the definition of natlist is

$$\mathsf{natlist} = \mathsf{Nil} \mid \mathsf{Cons\ of\ nat}\times\mathsf{natlist};$$

should satisfy the domain equation

$$D \cong 1 + N \times D$$

where N is the domain associated with the type **nat**, and for a lazy language, 1 is the one-point domain, $+$ is the separated sum operation and $\times$ is cartesian product. Such a D will contain elements which model finite, partial and infinite lists of natural numbers[1]. Since the domain constructing operations are functorial, we can rephrase the problem as that of finding a fixed point of the functor $F : \mathcal{D}om \to \mathcal{D}om$ given by $F(D) = 1 + N \times D$. The solution should also be the *least* fixed point in some sense. The basic approach is to solve recursive domain equations in a way analogous to the way in which we solve the equations arising from recursive definitions of domain *elements* by taking limits of chains in ω-cpos. Thus we want to treat the class of ω-cpos as a large ω-cpo. To do this, we need something analogous to the order relation of a ω-cpo, that is a notion of when one domain approximates another. The appropriate notion is that D approximates E when there is an embedding $\rho : D \lhd E$, and we shall therefore solve domain equations in the category $\mathcal{D}om_E$ of domains and embeddings. Working in $\mathcal{D}om_E$ also solves a problem which we have not yet mentioned, which is that the domain equations arising from arbitrary recursive types do not always give rise to endofunctors on $\mathcal{D}om$, because $\to$ is contravariant in its first argument. Mixed variance functors on $\mathcal{D}om$, however, give rise to covariant ones on $\mathcal{D}om_E$, so this complication can be avoided.

Note that the material which we shall present on the solution of recursive domain equations is slightly more general than we will need later, as we are only really interested in algebraic datatypes. However, it seems best to present the technique for arbitrary recursive types and then show that the solutions for the algebraic ones are particularly well-behaved. Although we ultimately wish to think of the solutions which we construct as living in $\mathcal{D}om$, it will turn out that the category in which they have the properties which we shall use is $\mathcal{D}om_S$. Thus we shall regard the basic domain forming operations as functors on $\mathcal{D}om_S$, rather than on $\mathcal{D}om$.

A functor $T : (\mathcal{D}om_S^{op})^m \times (\mathcal{D}om_S)^n \to \mathcal{D}om_S$ is *locally monotonic* if whenever we have $f_i, f_i' : D_i' \to D_i$ and $g_j, g_j' : E_j \to E_j'$ such that $f_i \sqsubseteq f_i'$ and $g_j \sqsubseteq g_j'$ for $0 \leq i \leq m$, $0 \leq j \leq n$ then

$$T(f_0, \ldots, f_{m-1}, g_0, \ldots, g_{n-1}) \sqsubseteq T(f_0', \ldots, f_{m-1}', g_0', \ldots, g_{n-1}')$$

All of our domain forming operations $\Sigma, \Pi : \mathcal{D}om_S{}^n \to \mathcal{D}om_S$, $\to : \mathcal{D}om_S^{op} \times \mathcal{D}om_S \to \mathcal{D}om_S$ are locally monotonic, as are $P_j : \mathcal{D}om_S{}^n \to \mathcal{D}om_S$ (the j^{th} projection functor) and $K_D : \mathcal{D}om_S{}^n \to \mathcal{D}om_S$ (the constant D functor).

Lemma 4.2.1 *If $T : (\mathcal{D}om_S^{op})^m \times (\mathcal{D}om_S)^n \to \mathcal{D}om_S$ is locally monotonic, then it gives rise to a covariant functor $T^E : \mathcal{D}om_E{}^{m+n} \to \mathcal{D}om_E$ given by*

$$T^E(D_0, \ldots, D_{m-1}, E_0, \ldots, E_{n-1}) = T(D_0, \ldots, D_{m-1}, E_0, \ldots, E_{n-1})$$

and for $f_i : D_i \lhd D_i'$, $g_j : E_j \lhd E_j'$

$$T^E(f_0, \ldots, f_{m-1}, g_0, \ldots, g_{n-1}) = T(f_0^R, \ldots, f_{m-1}^R, g_0, \ldots, g_{n-1})$$

□

[1] If we were working with a strict language, then the domain theoretic interpretations of $+$ and $\times$ would be varied.

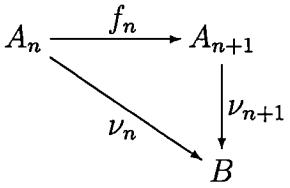
Corresponding to the idea of an ω-chain in a cpo, we have the notion of an ω-chain, or ω-diagram in $\mathcal{D}om_E$.

If $\mathcal{C}$ is a category, then an ω-diagram $\Delta = \langle A_n, f_n \rangle$ is a sequence of pairs of objects and morphisms of $\mathcal{C}$ such that $f_n : A_n \to A_{n+1}$:

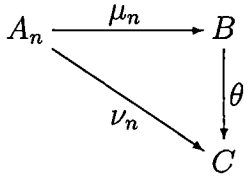
$$A_0 \xrightarrow{f_0} A_1 \xrightarrow{f_1} A_2 \xrightarrow{f_2} \cdots$$

Given such a Δ and m, n with $m \leq n$, we write $f_{mn} : A_m \to A_n$ for the map $f_{n-1} \circ \cdots \circ f_m$. We will also write Δ^- for $\langle A_{n+1}, f_{n+1} \rangle$, that is, the diagram Δ without its first element.

A *cocone* from the diagram Δ consists of an object B of $\mathcal{C}$ (called the *vertex*) and a sequence ν of morphisms $\nu_n : A_n \to B$ such that all the triangles



commute. If $\mu : \Delta \to B$ and $\nu : \Delta \to C$ are cocones then a *cocone morphism* from μ to ν is a map $\theta : B \to C$ in $\mathcal{C}$ such that all the triangles



commute. With this definition of morphism, the collection of cocones over Δ forms a category. An initial object of this category is called a *universal* (or *initial* or *colimiting*) cocone. If $\mu : \Delta \to B$ is initial, then we write $B = \varinjlim \Delta$ and call θ a *mediating* morphism. If $\mu : \Delta \to B$ is a cocone, then so is $\mu^- : \Delta^- \to B$ where $\mu^- = \langle \mu_{n+1} \rangle$. If μ is universal, then so is μ^-.

Lemma 4.2.2 *If $\Delta = \langle D_m, f_m \rangle$ is an ω-diagram in $\mathcal{D}om_E$, and $\rho : \Delta \to D$ is a cocone such that $\bigsqcup_n \rho_n \circ \rho_n^R = id_D$ then ρ is universal.*

Proof. See Appendix A. □

Lemma 4.2.3 *For any ω-diagram $\Delta = \langle D_m, f_m \rangle$ in $\mathcal{D}om_E$, there exists a cocone $\rho : \Delta \to D$ such that $\bigsqcup \rho_n \circ \rho_n^R = id_D$.*

Proof. See Appendix A. □

The previous two lemmas together imply that every ω-diagram in $\mathcal{D}om_E$ has a colimit (*i.e.* that $\mathcal{D}om_E$ is *ω-cocomplete*). Furthermore, the converse to Lemma 4.2.2 holds:

Lemma 4.2.4 *If $\rho' : \Delta \to D'$ is universal in $\mathcal{D}om_E$, then $\bigsqcup_n \rho'_n \circ \rho'^R_n = id_{D'}$.*

Proof. See Appendix A. □

Corresponding to a continuous function between ω-cpos, we have the notion of an ω-continuous functor, one which preserves colimits of all ω-diagrams. More precisely, we shall say that a functor $F : \mathcal{D}om_E \to \mathcal{D}om_E$ is *ω-continuous* if whenever $\rho : \Delta \to D$ is universal, so is $F(\rho) : F(\Delta) \to F(D)$. This means that $F(\varinjlim \Delta) \cong \varinjlim F(\Delta)$ where if $\Delta = \langle D_m, f_m \rangle$ then

$$F(\Delta) \stackrel{\text{def}}{=} \langle F(D_m), F(f_m) \rangle$$

The definition of ω-continuity extends in fairly obvious way to functors of several arguments. It is also straightforward to verify that ω-continuity is preserved by composition of functors.

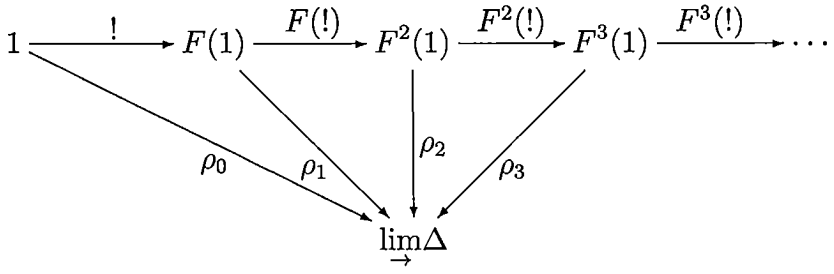
We say a functor $T : (\mathcal{D}om_S^{op})^m \times \mathcal{D}om_S{}^n \to \mathcal{D}om_S$ is *locally continuous* if it is locally monotonic and whenever we have an increasing chain of strict functions $\langle f_l^{(i)} \rangle_{l \in \omega}$ with $f_l^{(i)} : D'_i \to D_i$ for each $0 \leq i \leq m$ and a chain $\langle g_l^{(j)} \rangle_{l \in \omega}$ with $g_l^{(j)} : E_j \to E'_j$ for each $0 \leq j \leq n$ then

$$\begin{aligned} &\bigsqcup_{l \in \omega} T(f_l^{(0)}, \ldots, f_l^{(m-1)}, g_l^{(0)}, \ldots, g_l^{(n-1)}) \\ &= T(\bigsqcup_l f_l^{(0)}, \ldots, \bigsqcup_l f_l^{(m-1)}, \bigsqcup_l g_l^{(0)}, \ldots, \bigsqcup_l g_l^{(n-1)}) \end{aligned}$$

All of our domain constructors $+, \times, \to, K_D, P_i$ are locally continuous.

Lemma 4.2.5 *If $T : (\mathcal{D}om_S^{op})^m \times \mathcal{D}om_S{}^n \to \mathcal{D}om_S$ is locally continuous then $T^E : \mathcal{D}om_E{}^{m+n} \to \mathcal{D}om_E$ is ω-continuous.* □

We are now in a position to solve the domain equations arising from recursive type definitions. For simplicity, we shall restrict ourselves to the case of non-mutually recursive type definitions. Such a type definition gives rise to an ω-continuous functor $F : \mathcal{D}om_E \to \mathcal{D}om_E$. For example, the functor associated with the type declaration for lists of natural numbers which we gave earlier is $F = +^E(K_1^E, \times^E(K_N^E, Id^E))$. We then form the ω-diagram $\Delta = \langle D_m, f_m \rangle$ in $\mathcal{D}om_E$ by $D_0 = 1$, $D_{m+1} = F(D_m)$, $f_0 =!$ (the unique embedding $! : 1 \lhd F(1)$) and $f_{m+1} = F(f_m)$; and construct the universal cocone:



Note that this does indeed construct a solution to the recursive domain equation as

$$\begin{aligned} F(\varinjlim \Delta) &\cong \varinjlim (F\Delta) \text{ as } F \text{ is continuous} \\ &= \varinjlim \Delta^- \\ &= \varinjlim \Delta \end{aligned}$$

We will usually write Fix_F for $\varinjlim \Delta$ and η_F for the isomorphism from $F(\mathsf{Fix}_F)$ to Fix_F. η_F is given by $\bigsqcup \rho_{m+1} \circ (F\rho_m)^R$, and its inverse is $\bigsqcup F\rho_m \circ \rho_m^R$. We shall return to the question of the sense in which these solutions are initial in Section 4.5.

4.3 Extending Λ_T with Algebraic Datatypes

We now explain how we can use the results of the previous section to give a denotational semantics to Λ_T extended with lazy algebraic datatypes. We restrict ourselves to algebraic types to keep things simple, and because we are currently unable to say anything sensible about strictness analysis for more general recursive types. To simplify matters further, we shall only describe how to add a single algebraic datatype a (in sum-of-products form[2]) to the language (so we are really defining a parameterised collection of languages). The language arising from adding the algebraic type a to Λ_T will be denoted by $\Lambda_{T,a}$.

4.3.1 Syntax

Types

The type declaration for the type a has the form

$$\begin{aligned} a \;=\; & C_1 \text{ of } \delta_{1,0} \times \ldots \times \delta_{1,m_1-1} \\ \mid\; & C_2 \text{ of } \delta_{2,0} \times \ldots \times \delta_{2,m_2-1} \\ \vdots\; & \\ \mid\; & C_n \text{ of } \delta_{n,0} \times \ldots \times \delta_{n,m_n-1}; \end{aligned}$$

Note that, just in the above, '|' is part of the syntax of the declaration, not a bit of meta-syntax of BNF notation. The type expressions δ are given by the following BNF grammar:

$$\delta ::= \kappa \mid a$$

$$\kappa ::= \iota \mid (\kappa \to \kappa) \mid (\kappa \times \kappa) \;.$$

[2]Earlier, we claimed that sum-of-products was synonymous with algebraic. This is not quite true unless our domain-theoretic interpretations of the sum and product type constructors satisfy the distributive law $A \times (B + C) \cong (A \times B) + (A \times C)$, which ours do not. We shall simply ignore this fact, and continue to use the term 'algebraic'.

Finally, the types of $\Lambda_{T,a}$, ranged over by σ,τ and θ, are given by

$$\sigma ::= \iota \mid (\sigma \to \sigma) \mid (\sigma \times \sigma) \mid a$$

The tokens C_i, $1 \leq i \leq n$, are the *constructors* of the type a. They are assumed to be distinct. Note that whilst m_i may be 0, n is always at least 1.

Terms

The term-forming rules of Λ_T are augmented with the n rules of the form

$$\frac{t_0 :: \delta_{i,0} \quad \cdots \quad t_{m_i-1} :: \delta_{i,m_i-1}}{C_i(t_0, \ldots, t_{m_i-1}) :: a}$$

and

$$\frac{t :: a \qquad u_1 :: \theta \quad \cdots \quad u_n :: \theta}{\begin{array}{lll} \mathsf{acase}\, t & \mathsf{of} & C_1(x_{1,0}, \ldots, x_{1,m_1-1}) \Rightarrow u_1 \\ & \mid & C_2(x_{2,0}, \ldots, x_{2,m_2-1}) \Rightarrow u_2 \\ & \vdots & \\ & \mid & C_n(x_{n,0}, \ldots, x_{n,m_n-1}) \Rightarrow u_n \end{array} :: \theta}$$

Where we have omitted the obvious type superscripts on the $x_{i,j}$, which are bound within the respective u_i. We should really write, for example, $x_{n,m_n-1}^{\delta_{n,m_n-1}}$. We also omit the formal definition of substitution for the new constructs.

4.3.2 Semantics

Operational Semantics

The canonical terms of type a are all those of the form

$$C_i(t_0, \ldots, t_{m_i-1})$$

The evaluation relation of Figure 2.3 is augmented with the $2n$ new rules

$$C_i(t_0, \ldots, t_{m_i-1}) \Downarrow C_i(t_0, \ldots, t_{m_i-1})$$

and

$$\frac{t \Downarrow C_i(t_0, \ldots, t_{m_i-1}) \qquad u_i[t_0/x_{i,0}, \ldots, t_{m_i-1}/x_{i,m_i-1}] \Downarrow c}{\begin{array}{lll} \mathsf{acase}\, t & \mathsf{of} & C_1(x_{1,0}, \ldots, x_{1,m_1-1}) \Rightarrow u_1 \\ & \mid & C_2(x_{2,0}, \ldots, x_{2,m_2-1}) \Rightarrow u_2 \\ & \vdots & \\ & \mid & C_n(x_{n,0}, \ldots, x_{n,m_n-1}) \Rightarrow u_n \end{array} \Downarrow c}$$

Denotational Semantics

The domain-theoretic interpretation of types is just as for Λ_T, with the extra clause

$$D_a = \mathsf{Fix}_{T^E}$$

where

$$T(X) = \sum_{i=1}^{n} \prod_{j=0}^{m_i-1} D_{i,j}(X)$$

and

$$D_{i,j}(X) = \begin{cases} D_\kappa & \text{if } \delta_{i,j} = \kappa \\ X & \text{if } \delta_{i,j} = a \end{cases}$$

The semantic equations of Figure 2.4 are extended by

$$[\![C_i(t_0, \ldots, t_{m_i-1})]\!]\rho \;=\; \eta_{T^E}(in_i([\![t_0]\!]\rho, \ldots, [\![t_{m_i-1}]\!]\rho))$$

and

$$[\![\mathsf{acase}\ t\ \mathsf{of}\ \ldots \Rightarrow u_n]\!]\rho$$

$$= \begin{cases} \bot_{D_\theta} & \text{if } [\![t]\!]\rho = \bot_{D_a} \\ [\![u_1]\!]\rho[x_{1,0} \mapsto d_0, \ldots, x_{1,m_1-1} \mapsto d_{m_1-1}] & \text{if } [\![t]\!]\rho = \eta_{T^E}(in_1(d_0, \ldots, d_{m_1-1})) \\ \vdots & \\ [\![u_n]\!]\rho[x_{n,0} \mapsto d_0, \ldots, x_{n,m_n-1} \mapsto d_{m_n-1}] & \text{if } [\![t]\!]\rho = \eta_{T^E}(in_n(d_0, \ldots, d_{m_n-1})) \end{cases}$$

Computational Adequacy

To show that our previous adequacy result still holds for $\Lambda_{T,a}$ we need to extend the logical relation $\mathcal{R}$ which was defined on page 20 to the type a. Given a domain D and a relation $R \subseteq D \times \Lambda^{\circ,a}_{T,a}$, define the relation $T(R) \subseteq T(D) \times \Lambda^{\circ,a}_{T,a}$ by

$$\begin{aligned} d\,T(R)\,t \quad \text{iff} \quad & d = \bot \\ \text{or} \quad & d = in_i(e_0, \ldots, e_{m_i-1}),\ t{\Downarrow}C_i(v_0, \ldots, v_{m_i-1}) \text{ and} \\ & \forall j.\, e_j P_j v_j \text{ where } P_j = \begin{cases} \mathcal{R}^\kappa & \text{if } \delta_{ij} = \kappa \\ R & \text{if } \delta_{ij} = a \end{cases} \end{aligned}$$

Now let the cocone used in the construction of $D_a = \mathsf{Fix}_{T^E}$ be $\nu : \Delta \to \mathsf{Fix}_{T^E}$ where $\Delta = \langle D_k, f_k \rangle$ and define for each $k \in \omega$ the relation $\mathcal{R}^a_k \subseteq D_k \times \Lambda^{\circ,a}_{T,a}$ by

$$\bot\, \mathcal{R}^a_0\, t\ \forall t \in \Lambda^{\circ,a}_{T,a}$$

$$\mathcal{R}^a_{k+1} = T(\mathcal{R}^a_k)$$

Finally, define $\mathcal{R}^a \subseteq D_a \times \Lambda^{\circ,a}_{T,a}$ by

$$d\,\mathcal{R}^a\,t \text{ iff } \forall k \in \omega.\, \nu^R_k(d)\,\mathcal{R}^a_k\,t$$

It is straightforward to verify that Lemma 2.1.4 and Lemma 2.1.5 both still hold with this definition. We shall also need the following two lemmas:

Lemma 4.3.1 *If $d \in T(\mathsf{Fix}_{T^E})$, $t \in \Lambda^{\circ,a}_{T,a}$ and $d\,T(\mathcal{R}^a)\,t$ then $\eta(d)\,\mathcal{R}^a\,t$.*

Proof. If $d = \bot$ then the result is immediate. Otherwise, $d = in_i(e_0, \ldots, e_{m_i-1})$, $t \Downarrow C_i(v_0, \ldots, v_{m_i-1})$ and $\forall j.\, e_j\,\mathcal{R}^{\delta_{ij}}\,v_j$. To obtain the result, it suffices to show that $(\nu^R_{k+1} \circ \eta)(d)\,\mathcal{R}^a_{k+1}\,t$ for all k. But $\nu^R_{k+1} \circ \eta = (T\nu_k)^R = T(\nu^R_k)$ and $T(\nu^R_k)(d) = in_i(e'_0, \ldots, e'_{m_i-1})$ where

$$e'_j = \begin{cases} e_j & \text{if } \delta_{ij} = \kappa \\ \nu^R_k(e_j) & \text{if } \delta_{ij} = a \end{cases}$$

and since for any j such that $\delta_{ij} = a$ we know that $e_j\,\mathcal{R}^a\,v_j$ we have that $e'_j\,\mathcal{R}^a_k\,v_j$ and hence $T(\nu^R_k)(d)\,T(\mathcal{R}^a_k)\,t$ and we are done. □

Lemma 4.3.2 *If $d \in \mathsf{Fix}_{T^E}$, $T \in \Lambda^{\circ,a}_{T,a}$ and $d\,\mathcal{R}^a\,t$ then $\eta^{-1}(d)\,T(\mathcal{R}^a)\,t$.*

Proof. Again, if $d = \bot$ then the result is immediate. Otherwise

$$d = \eta(in_i(e_0, \ldots, e_{m_i-1})).$$

We know that $\nu^R_{k+1}(d)\,\mathcal{R}^a_{k+1}\,t$ for all k, and this means that $((T\nu_k)^R \circ \eta^{-1})(d)\,T(\mathcal{R}^a_k)\,t$. This is equivalent to $T(\nu^R_k)(in_i(e_0, \ldots, e_{m_i-1}))\,T(\mathcal{R}^a_k)\,t$. But

$$T(\nu^R_k)(in_i(e_0, \ldots, e_{m_i-1})) \;=\; in_i(e'_0, \ldots, e'_{m_i-1})$$

where

$$e'_j \;=\; \begin{cases} e_j & \text{if } \delta_{ij} = \kappa \\ \nu^R_k(e_j) & \text{if } \delta_{ij} = a \end{cases}$$

and thus $t \Downarrow C_i(v_0, \ldots, v_{m_i-1})$ such that for all j

$$e'_j\,P_j\,v_j \text{ where } P_j = \begin{cases} \mathcal{R}^\kappa & \text{if } \delta_{ij} = \kappa \\ \mathcal{R}^a_k & \text{if } \delta_{ij} = a \end{cases}$$

and this means that $e_j\,\mathcal{R}^{\delta_{ij}}\,v_j$ so the result follows. □

We can now extend the proof of Proposition 2.1.6 to cover the new language constructs associated with the type a:

Proposition 4.3.3 *If $t \in \Lambda^\tau_{T,a}$, ρ is an environment such that $FV(t) \subseteq dom(\rho)$ and for each $x^{\sigma_l}_l \in dom(\rho)$, $s_l \in \Lambda^{\circ,\sigma_l}_{T,a}$ and $\rho(x^{\sigma_l}_l)\mathcal{R}^{\sigma_l}s_l$ then $[\![t]\!]\rho\mathcal{R}^\tau t[s_l/x^{\sigma_l}_l]$.*

Proof. There are two new cases:

- If $t = C_i(v_0, \ldots, v_{m_i-1})$ then

  $$[\![t]\!]\rho = \eta(in_i([\![v_0]\!]\rho, \ldots, [\![v_{m_i-1}]\!]\rho))$$

  and

  $$t[s_l/x^{\sigma_l}_l] = C_i(v_0[s_l/x^{\sigma_l}_l], \ldots, v_{m_i-1}[s_l/x^{\sigma_l}_l])$$

By induction, $[\![v_j]\!]\rho\,\mathcal{R}^{\delta_{ij}}\,v_j[s_l/x_l^{\sigma_l}]$ so that

$$\eta^{-1}([\![t]\!]\rho)\,T(\mathcal{R}^a)\,t[s_l/x_l^{\sigma_l}]$$

and thus by Lemma 4.3.1

$$[\![t]\!]\rho\,\mathcal{R}^a\,t[s_l/x_l^{\sigma_l}]$$

as required.

- If

$$t = \mathsf{acase}\ v\ \mathsf{of}\ C_1(y_{1,0},\ldots,y_{1,m_1-1}) \Rightarrow u_1 \mid \cdots \mid C_n(y_{n,0},\ldots,y_{n,m_n-1}) \Rightarrow u_n$$

then if $[\![v]\!]\rho = \bot$, $[\![t]\!]\rho = \bot$ and the result is immediate. Otherwise,

$$[\![v]\!]\rho = \eta(in_i(e_0,\ldots,e_{m_i-1}))$$

for some i and by induction we know that

$$[\![v]\!]\rho\,\mathcal{R}^a\,v[s_l/x_l^{\sigma_l}]$$

By Lemma 4.3.2, this means

$$in_i(e_0,\ldots,e_{m_i-1})\,T(\mathcal{R}^a)\,v[s_l/x_l^{\sigma_l}]$$

so that $v{\Downarrow}C_i(v_0,\ldots,v_{m_i-1})$ where for all j, $e_j\,\mathcal{R}^{\delta_{ij}}\,v_j$. Hence we can apply the induction hypothosis to u_i to deduce that

$$[\![u_i]\!]\rho[y_{i,0}\mapsto e_0,\ldots,y_{i,m_i-1}\mapsto e_{m_i-1}]\ \mathcal{R}^\tau\ u_i[s_l/x_l^{\sigma_l}][v_j/y_j]$$

which means

$$[\![t]\!]\rho\ \mathcal{R}^\tau\ u_i[s_l/x_l^{\sigma_l}][v_j/y_j]$$

But $(u_i[s_l/x_l^{\sigma_l}][v_j/y_j]){\Downarrow}c \Rightarrow t[s_l/x_l^{\sigma_l}]{\Downarrow}c$ so that

$$[\![t]\!]\rho\ \mathcal{R}^\tau\ t[s_l/x_l^{\sigma_l}]$$

as required.

□

4.4 Previous Work on Strictness Analysis and Recursive Types

4.4.1 Projection Analysis

When we come to look at strictness analysis of first order functions over structured datatypes, such as the algebraic types in $\Lambda_{T,a}$ (or even just pairs), there is a much

richer collection of 'natural' strictness properties than is the case for functions over flat domains. For example, in the language $\Lambda_{T,nlist}$, which is Λ_T augmented with the algebraic type *nlist*, defined by

$$nlist = \mathsf{Nil} \mid \mathsf{Cons\ of}\ \iota \times nlist$$

we can define the functions **length** and **sumlist**, both of which have the type $nlist \to \iota$, by

$$\mathsf{length} \stackrel{\text{def}}{=} \mathsf{fix}(f.\lambda l.\mathsf{nlistcase}\ l\ \mathsf{of}\ \mathsf{Nil} \Rightarrow \underline{0} \mid \mathsf{Cons}(x, xs) \Rightarrow \underline{1} + (f\ xs))$$

and

$$\mathsf{sumlist} \stackrel{\text{def}}{=} \mathsf{fix}(f.\lambda l.\mathsf{nlistcase}\ l\ \mathsf{of}\ \mathsf{Nil} \Rightarrow \underline{0} \mid \mathsf{Cons}(x, xs) \Rightarrow x + (f\ xs))$$

Both of these functions are clearly strict, but this is not particularly interesting. What we really want to know is that **length** needs to evaluate the entire structure of its argument (the *spine* of the list) but does not evaluate any of the list elements whilst **sumlist** evaluates both the whole structure and all the elements. For functions which also return structured results, the amount of evaluation peformed on the argument can depend on that demanded of the result. Consider, for example, the function

$$\mathsf{mapinc} \stackrel{\text{def}}{=} \mathsf{fix}(f.\lambda l.\mathsf{nlistcase}\ l\ \mathsf{of}\ \mathsf{Nil} \Rightarrow \mathsf{Nil} \mid \mathsf{Cons}(x, xs) \Rightarrow \mathsf{Cons}(x + \underline{1}, f\ xs))$$

of type $nlist \to nlist$. **mapinc** will evaluate exactly as much of its argument as is demanded of its result. This means that l will just have its structure evaluated in the evaluation of

$$\mathsf{length}\ (\mathsf{mapinc}\ l)$$

but will also have all its elements evaluated during the evaluation of

$$\mathsf{sumlist}\ (\mathsf{mapinc}\ l)$$

Thus we can view functions as taking a level of demand on their results to a level of demand on their arguments[3].

In [WH87], Wadler and Hughes chose domain-theoretic *projections* to capture the idea of a level of demand. In this context, a projection on a domain D is a continuous function $\alpha : D \to D$ such that $\alpha \circ \alpha = \alpha$ (α is *idempotent*) and $\alpha \sqsubseteq id_D$. Projections in this sense are in 1-1 correspondence with projections as we defined them earlier, in connection with embedding-projection pairs. Given a projection in our earlier sense $\nu : D \to E$, we know that ν has a unique left adjoint $\nu^L : E \to D$ which satisfies $\nu^L \circ \nu \sqsubseteq id_D$ and $\nu \circ \nu^L = id_E$. But then $(\nu^L \circ \nu) \circ (\nu^L \circ \nu) = \nu^L \circ id_E \circ \nu = \nu^L \circ \nu$ so $\nu^L \circ \nu : D \to D$ is a projection in the new sense. Conversely, given a projection $\alpha : D \to D$ as above, then the set $I = \{\alpha(d) \mid d \in D\}$ is a subdomain of D, so the inclusion map $i : I \to D$ is continuous and α gives rise to $\alpha' : D \to I$. Then $i \circ \alpha' = \alpha \sqsubseteq id_D$ and $\alpha' \circ i = id_I$ as $\alpha \circ \alpha = \alpha$.

[3]Note that this is not something special about projection analysis—the ideal properties which we have previously discussed can also been seen this way.

Projection analysis looks for properties of the form

$$\beta \circ f \circ \alpha = \beta \circ f \tag{4.1}$$

where $f : D \to E$ is a continuous map and $\alpha : D \to D$ and $\beta : E \to E$ are projections. If f satisfies (4.1) then we say 'f is α-strict in a β-strict context'. Projection analysis has been applied both to strictness analysis [WH87] and to binding-time analysis [Lau89]. An example of a projection which is useful in strictness analysis is $H : D_{nlist} \to D_{nlist}$, which is given by

$$H(d) = \begin{cases} \bot & \text{if } d = \bot \\ \eta(in_1(*)) & \text{if } d = \eta(in_1(*)) \\ \bot & \text{if } d = \eta(in_2(\bot, xs)) \\ \eta(in_2(x, H(xs))) & \text{if } d = \eta(in_2(x, xs)) \text{ and } x \neq \bot \end{cases}$$

We can rephrase this definition in a slightly more readable, but less formal, way by mixing syntax and semantics:

$$\begin{array}{lcl} H(\bot) & = & \bot \\ H(\mathsf{Nil}) & = & \mathsf{Nil} \\ H(\mathsf{Cons}(\bot, xs)) & = & \bot \\ H(\mathsf{Cons}(x, xs)) & = & \mathsf{Cons}(x, H(xs)) \text{ if } x \neq \bot \end{array}$$

If $f : nlist \to D$ satisfies $f = f \circ H$ then we say f is *head-strict*. Head-strictness is a property which does not seem to be expressible in any ideal-based analysis. Note that the use we can make of head-strictness in a compiler is slightly more subtle than it might at first appear. What head-strictness of f tells us is that the argument of any application of f may be replaced by H applied to the argument. This does not, however, mean that we can safely *evaluate* H applied to the argument before calling f, as can be seen by considering the function

$$\lambda x^{nlist}.\underline{3}$$

which is easily seen to be head-strict, but for which it is unsafe to perform any evaluation of the argument. If f is head-strict, then what we can deduce is that every Cons cell in the argument *which contributes to the result of* f can safely have its head evaluated too. Formalising the uses to which projection information can be put has turned out to be rather awkward; see [Bur90b].

The set of properties which can be expressed in the form given in Equation 4.1 above does not include the most simple $f \bot = \bot$ kind of strictness. To deal with this, Wadler and Hughes lift all their domains, adding a new bottom element which they call 'abort' (and write as a lightning bolt). They then analyse the lifted versions of functions. Burn has established a connection between ideal properties and the properties expressible using a restricted class of projections (called *smash projections*) over lifted domains [Bur90a].

Hunt has generalised projection analysis to higher-order functions by making use of an abstract interpretation in which the points of the abstract domains represent

partial equivalence relations (pers) on (rather than subsets of) the standard domains [Hun90, Hun91].

Projection analysis (or Hunt's per analysis) requires a finite lattice of projections (or pers) to be chosen at each type. There is still no completely satisfactory way of picking such a lattice for arbitrary recursive, or even merely algebraic, types. Hunt gives a construction of a lattice of pers for any algebraic type in [Hun91], but this leads to a lattice which omits some of the more useful points.

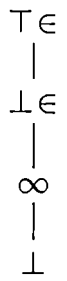
4.4.2 Ideal-based Analyses

To extend an ideal-based analysis, such as our strictness logic, to a language which includes some form of recursive type we first have to decide which ideals we wish to reason about. This is a rather difficult problem. The collection of ideals which we pick for a recursive type should have the following properties:

1. It should be finite.
2. It should not be too big. This is because a large collection of properties will lead to an impractically slow analysis system.
3. It should contain as many 'useful' (in terms of optimisation-enabling) properties as possible. This plainly pulls against the previous requirement.
4. It should also contain sufficient points to enable us to deduce useful properties of real programs. Even if, for example, we were only interested in the optimisations which can be performed as a result of simple strictness, an analysis for Λ_T which only made use of the two-point lattice at every type would be extremely weak.
5. It should be closed under intersection, so that each domain element has a 'best' abstraction.
6. There should be some procedure by which the lattice of properties is derived from the type declaration.
7. The compiler should be able automatically to calculate a representation of the lattice of properties from the type declaration.
8. The ordering on the representation should be sound with respect to the inclusion ordering on the interpretations of representatives, and as complete as is practical.

There is currently no technique for constructing lattices of ideals which meets all the above criteria (though we sketch a promising approach to the problem in the next section). For particular types, there have been various more or less *ad hoc*

suggestions. The best-known of these is Wadler's four-point abstract domain for lazy lists of elements of a flat domain [Wad87]:



The abstract point $\bot$ denotes the singleton ideal containing the $\bot$ element of the list domain. ∞ denotes all infinite lists and lists with an undefined tail. $\bot\in$ contains in addition all the finite lists which contain a $\bot$ element and $\top\in$ denotes all lists. This collection of properties seems to work fairly well in practice[4], although it is not clear how to generalise it to lists of elements from non-flat domains or to, say, trees of elements from a flat domain. Wadler suggests that since the lattice given above is the double lifting of the two-point lattice which is used to abstract elements of a flat domain, the abstraction of lists of elements of D_σ should be $(A_{\sigma\bot})_\bot$. An even simpler approach is just to use the same lattice for all algebraic types—[Bur91b] abstracts lists and trees with elements of any type using this lattice. Whilst this ignores all the structure of the elements, and the structure of the datatype, it may well be that for a practical implementation any more sophisticated approach leads to lattices which are impractically large. [Bur91b] also gives three other abstract domains for lists, which we shall mention later.

Ernoult and Mycroft [EM91] have suggested an alternative construction of lattices of ideals for lists, which is based on the idea that we should choose those properties which 'treat all the elements the same'. They call the properties which they pick *uniform ideals*. There is, however, no formal definition of what constitutes a uniform ideal, so it is not clear how to generalise the idea to other datatypes. Hughes and Ferguson have devised a (rather complicated) construction which yields a finite set of subsets for any lazy algebraic type [HF89].

4.5 A New Construction

In this section we present some preliminary ideas about a different approach to the construction of a suitable lattice of ideals of D_a, where a is a lazy algebraic type. To begin with, notice that it seems futile to attempt to define a 'good' set of ideals in some external way; that is in a way which is independent of the syntactic definition of a. This is because, for example, whatever finite lattice we choose to abstract *nlist*, we shall surely choose a larger lattice as the abstraction of the isomorphic type

$$nlist' = \mathsf{Nil}' \mid \mathsf{Cons}' \text{ of } \iota \times nlist$$

[4]Though it gives poor information if programs are written using head, tail and isnil, rather than listcase.

In any case, a good set of ideals will also reflect the way in which values of type a are *used*, and this is not captured in the order structure of D_a alone.

Note also that a Scott-closed subset of a domain D is precisely the kernel $f^{-1}(\bot)$ of a continuous map $f \to \mathcal{O}$, where $\mathcal{O}$ is the two-point domain (sometimes called *Sierpinski space*). A non-empty Scott-closed set, one of our ideals, is the kernel of a *strict* map into $\mathcal{O}$. Thus we can rephrase the problem of finding a good set of ideals of D_a as that of finding a good set of strict maps $f : D_a \to \mathcal{O}$. Our construction is based on the observation that a natural collection of such maps arises by considering the sense in which D_a is the initial solution to the recursive domain equation associated with the type a.

Lemma 4.5.1 *If $\Delta = \langle D_m, f_m \rangle$ is an ω-diagram in $\mathcal{D}om_E$ and $\rho : \Delta \to D$ is a universal cocone in $\mathcal{D}om_E$, then ρ is a universal cocone in $\mathcal{D}om_S$ as well.*

Proof. See Appendix A. □

Recall that if $F : \mathcal{C} \to \mathcal{C}$ is an endofunctor on a category $\mathcal{C}$, then an *F-algebra* consists of a pair (A, α) where A is an object of $\mathcal{C}$ and $\alpha : FA \to A$. The collection of F-algebras are the objects of a category $F-\mathcal{A}lg$ in which a morphism from (A, α) to (B, β) is a morphism $f : A \to B$ in $\mathcal{C}$ such that

$$\begin{array}{ccc} FA & \xrightarrow{Ff} & FB \\ {\scriptstyle\alpha}\downarrow & & \downarrow{\scriptstyle\beta} \\ A & \xrightarrow[f]{} & B \end{array}$$

commutes. Such an f is called an *F-homomorphism.*

Now it is the case that for an arbitrary ω-continuous functor $F : \mathcal{D}om_E \to \mathcal{D}om_E$, the pair (Fix_F, η_F) is the initial F-algebra (see [Plo79] for the details). If, however, F arises as T^E where $T : \mathcal{D}om_S \to \mathcal{D}om_S$ is a locally continuous functor, as is the case for the functors associated with algebraic types, then we can say something rather more useful:

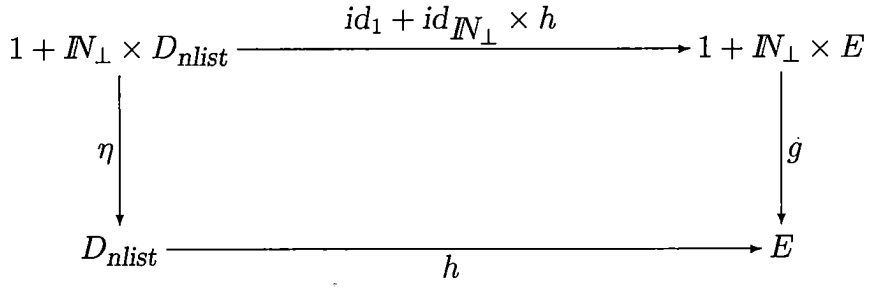
Proposition 4.5.2 *If $T : \mathcal{D}om_S \to \mathcal{D}om_S$ is a locally continuous functor then the pair $(\mathsf{Fix}_{T^E}, \eta_{T^E})$ is the initial T-algebra*

Proof. See Appendix A. □

From now on, if $T : \mathcal{D}om_S \to \mathcal{D}om_S$ is a locally continuous functor, we shall blur the distinction between T and T^E whenever it seems convenient.

Homomorphisms from the initial T-algebra capture a kind of primitive recursion over our lazy datatypes. These maps are very familiar to functional programmers, and it is worth seeing what the construction gives in a simple case:

Example In the case of lazy lists of natural numbers, Proposition 4.5.2 says that for any domain E and any strict map $g : 1 + \mathbb{N}_\perp \times E \to E$ there is a unique strict $h : D_{nlist} \to E$ such that



commutes. This means, if $E = D_\sigma$ and $g = [\![\mathsf{g}]\!]$ that h is the denotation of $\mathsf{reduce'\ g}$ where

$$\mathsf{reduce'} :: ((1 + \iota \times \sigma) \to \sigma) \to \mathit{nlist} \to \sigma$$

is given by

$$\begin{aligned}\mathsf{reduce'} \stackrel{\text{def}}{=} \mathsf{fix}(f.\lambda g.\lambda l.\mathsf{nlistcase}\ l\ \ \mathsf{of}\ \ & \mathsf{Nil} \Rightarrow g(\mathsf{inl}()) \\ \mid\ & \mathsf{Cons}(x, xs) \Rightarrow g(\mathsf{inr}(x, f\ xs)))\end{aligned}$$

Actually, we have cheated slightly above, in that we have not described any variant of Λ_T which includes a type 1 which is interpreted as the one-point domain, and we really want g to range over strict functions only. By using the domain isomorphisms $(A + B) \to_\perp C \cong (A \to C) \times (B \to C)$, $(1 \to A) \cong A$ and $(A \times B) \to C \cong A \to (B \to C)$ (currying), we can obtain the more familiar definition which is used by functional programmers:

$$\mathsf{reduce} :: (\iota \to \sigma \to \sigma) \to \sigma \to \mathit{nlist} \to \sigma$$

$$\begin{aligned}\mathsf{reduce} \stackrel{\text{def}}{=} \mathsf{fix}(f.\lambda g.\lambda a.\lambda l.\mathsf{nlistcase}\ l\ \ \mathsf{of}\ \ & \mathsf{Nil} \Rightarrow a \\ \mid\ & \mathsf{Cons}(x, xs) \Rightarrow g\,x\,(f\,g\,a\,xs))\end{aligned}$$

The reduce function is also known as fold. It is an extremely useful function, and many other common list-manipulating functions, such as sumlist, length, map, reverse and append can be defined in terms of it. If we write $(g\ n\ s)$ as $n \odot s$, and l is a finite list $[n_1, \ldots, n_k]$, then the result of (reduce g a l) is $n_1 \odot (n_2 \odot \cdots n_k \odot a) \cdots)$. Functions like reduce are sometimes known as *iterators*, and they also arise naturally from the encodings of algebraic types in higher-order polymorphic lambda calculi, such as $\lambda 2$ [GLT89].

We shall present our construction of strictness property lattices for algebraic types in several stages. The first step is to generalise the idea of going from the domain of lists of natural numbers to the domain of lists of strictness properties of natural numbers. This latter domain is, of course, still infinite and therefore unsuitable for our purposes (it contains points which represent properties such as 'has a $\perp$ in every prime-numbered position'). We shall then cut this domain down to a small finite set of properties by using the associated iterator to define a small set of maps from it into $\mathcal{O}$.

Given the type declaration for an algebraic type a, we defined D_a as Fix_T, where $T : \mathcal{D}om_S \to \mathcal{D}om_S$ was a locally continuous functor derived from the type declaration. We now introduce the functor $\widehat{T} : \mathcal{D}om_S \to \mathcal{D}om_S$ which is defined in terms of abstract, rather than standard, domains[5].

$$\widehat{T}(X) = \sum_{i=1}^{n} \prod_{j=0}^{m_i - 1} \widehat{D}_{i,j}(X)$$

where

$$\widehat{D}_{i,j}(X) = \begin{cases} A_\kappa & \text{if } \delta_{i,j} = \kappa \\ X & \text{if } \delta_{i,j} = a \end{cases}$$

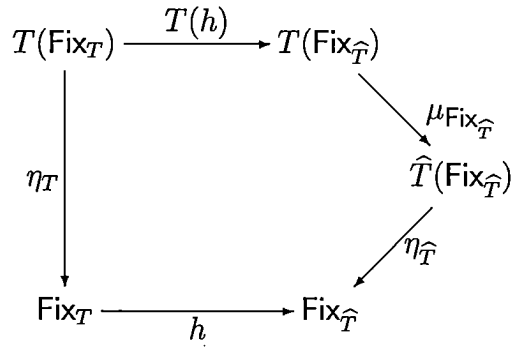
The pair $(\mathsf{Fix}_{\widehat{T}}, \eta_{\widehat{T}})$ captures the construction implicit in the intuitive idea of the domain of lists of strictness properties. We now wish to give a formal definition of the obvious abstraction map from Fix_T to $\mathsf{Fix}_{\widehat{T}}$. In the case of lists of natural numbers, this is what one thinks of as $map(\alpha_\iota)$, where $\alpha_\iota : D_\iota \to A_\iota$ is the abstraction map we used earlier. Firstly, note that for any domain X, there is a strict map $\mu_X : T(X) \to \widehat{T}(X)$ given by

$$\mu_X = \sum_{i=1}^{n} \prod_{j=0}^{m_i - 1} f_{i,j}(X)$$

where $f_{i,j}(X) : D_{i,j}(X) \to \widehat{D}_{i,j}(X)$ is the strict map given by

$$f_{i,j}(X) = \begin{cases} \alpha_\kappa & \text{if } \delta_{i,j} = \kappa \\ id_X & \text{if } \delta_{i,j} = a \end{cases}$$

Now $\eta_{\widehat{T}} \circ \mu_{\mathsf{Fix}_{\widehat{T}}}$ makes $\mathsf{Fix}_{\widehat{T}}$ into a T-algebra, so by Proposition 4.5.2, there is a unique h such that the following diagram in $\mathcal{D}om_S$ commutes:



[5]We shall use the term 'abstract domain' instead of 'lattice of strictness properties', though this should not imply any commitment to the use of classical abstract interpretation techniques to perform the analysis. Note also that when we write A_τ for the abstract domain associated with the type τ, we are being somewhat ambiguous. This can be taken to be either the sublattice of $\mathcal{P}_H(D_\tau)$ consisting of those ideals in which we are interested, or as the approximate *representation* of that lattice which is part of whatever formal system we use to reason about these properties. In the presence of a completeness result like Corollary 3.2.10, these will be equivalent; but this is not always the case, as we saw in the case of the disjunctive logic.

and this is the map we want.

The second step in the construction uses the fact that, by Proposition 4.5.2 again, $(\mathsf{Fix}_{\widehat{T}}, \eta_{\widehat{T}})$ is the initial $\widehat{T}$-algebra. This means that, given any strict map $g : \widehat{T}(\mathcal{O}) \to \mathcal{O}$, there is a unique strict map $g^* : \mathsf{Fix}_{\widehat{T}} \to \mathcal{O}$ such that

$$\begin{array}{ccc} \widehat{T}(\mathsf{Fix}_{\widehat{T}}) & \xrightarrow{\widehat{T}(g^*)} & \widehat{T}(\mathcal{O}) \\ \downarrow{\scriptstyle \eta_{\widehat{T}}} & & \downarrow{\scriptstyle g} \\ \mathsf{Fix}_{\widehat{T}} & \xrightarrow[g^*]{} & \mathcal{O} \end{array}$$

commutes. Because $\widehat{T}(\mathcal{O})$ is constructed from finite sums and products of finite domains, it will itself always be a finite domain. Thus there will only be a finite number of maps $g : \widehat{T}(\mathcal{O}) \to \mathcal{O}$. Each one of these gives rise to an ideal of Fix_T, namely

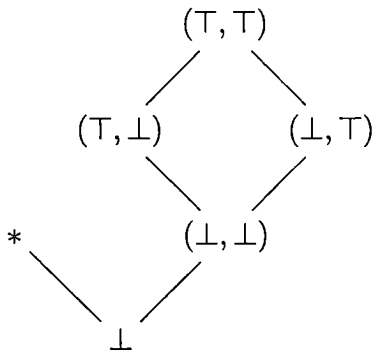
$$K_g = (g^* \circ h)^{-1}(\bot)$$

and it is the collection of all these K_g which we propose as a good set of basic strictness properties of the type a. We shall also find it helpful to define $J_g \subseteq \mathsf{Fix}_{\widehat{T}}$ to be $(g^*)^{-1}(\bot)$.

Let us see how this works out in the familiar case of lists of natural numbers. The abstract domain A_ι is the two-point domain, which we will write as $\mathbf{2}$. This is, of course, isomorphic to $\mathcal{O}$, but this is just coincidental. The domain of lazy lists of elements of $\mathbf{2}$ (*i.e.* $\mathsf{Fix}_{\widehat{T}}$) will be written $L\mathbf{2}$. The previous diagram thus specialises to the following:

$$\begin{array}{ccc} 1 + \mathbf{2} \times L\mathbf{2} & \xrightarrow{id_1 + id_{\mathbf{2}} \times g^*} & 1 + \mathbf{2} \times \mathcal{O} \\ \downarrow{\scriptstyle \eta} & & \downarrow{\scriptstyle g} \\ L\mathbf{2} & \xrightarrow[g^*]{} & \mathcal{O} \end{array}$$

The domain $1 + \mathbf{2} \times \mathcal{O}$ looks like this:



and there are twelve strict monotone maps g from this domain into $\mathcal{O}$. We can

n	$g_n(inl(*))$	$g_n(inr(x,y))$	K_n
1	$\bot$	$\bot$	All lists.
2	$\top$	$\bot$	All non-empty lists.
3	$\bot$	x	The empty list. Non-empty lists with $\bot$ as the first element.
4	$\top$	x	Non-empty lists with $\bot$ as the first element.
5	$\bot$	y	All lists.
6	$\top$	y	All infinite and partial lists.
7	$\bot$	$x \sqcap y$	All lists.
8	$\top$	$x \sqcap y$	Infinite and partial lists. Lists containing at least one $\bot$ element.
9	$\bot$	$x \sqcup y$	The empty list. Finite, infinite and partial lists all of whose elements are $\bot$.
10	$\top$	$x \sqcup y$	Partial and infinite lists all of whose elements are $\bot$.
11	$\bot$	$\top$	The empty list.
12	$\top$	$\top$	Just the completely undefined list $\bot$.

Figure 4.1: Basic Strictness Properties of *nlist*.

describe each of these maps by giving a pair, the first component of which is the image of the point $inl(*)$ and the second component of which is the image of the point $inr(x,y)$ for $x \in \mathbf{2}$, $y \in \mathcal{O}$. A little calculation then gives the informal interpretation of each K_g as shown in Figure 4.1 (we write K_n for K_{g_n}).

This collection of properties includes Wadler's four points: his $\bot$, ∞, $\bot \in$ and $\top \in$ are our K_{12}, K_6, K_8 and K_1 respectively. Burn's A^{hs} domain [Bur91b] consists of our K_{12}, K_4 and K_1. Note also that our twelve maps only give rise to ten distinct properties, as K_1, K_5 and K_7 are all the whole of the domain D_{nlist}.

These ten properties are all intuitively compelling: this seems to be the 'right' construction. These ten points alone, however, do not quite satisfy all the criteria we laid down earlier for a good collection of properties. This is because the collection is not closed under intersection. The intersection of K_8 and K_9 is the set of non-empty lists all of whose elements are $\bot$, and this is not one of our properties. This is not a serious problem—we just have to add intersections in explicitly, exactly as we did in the strictness logic. We might also want to add unions, if we were trying to extend the disjunctive strictness logic to algebraic types.

By unfolding more, that is, by solving the domain equation with T^n instead of T, we can in principle get an infinite sequence of larger and larger lattices of strictness properties. This would, however, be rather unwieldy, and it is not clear that the extra points would be particularly useful.

Another of our criteria for a good collection of properties was that we should be able to calculate an entailment relation on a set of representations of the properties

which was sound and fairly complete with respect to the real inclusion ordering on the properties. It is clear that the representation of the set of properties produced by our construction will be based on some representation for the maps g_n. For a single simple type like *nlist*, it is straightforward (though very tedious) to calculate by hand what all the basic properties and all their intersections are, and then to work out what the inclusion ordering is and what proof rules or abstract semantic equations should be given to each of the constructors and destructors associated with the type. Even for hand calculation, however, we should like some general reasoning principles to help with this task, and ideally we should like the process to be mechanizable in a compiler so that abstract domains can be synthesized for arbitrary user-defined algebraic types. Note that we already have one simple principle for deducing inclusions between properties, namely that if $g \sqsubseteq g'$ then $g^* \sqsubseteq g'^*$ and hence $K_{g'} \subseteq K_g$. This is not, however, sufficient to deduce all the inclusions which we should like. In the case of *nlist*, for example, we want to know that $K_1 \subseteq K_5$, but it is not the case that $g_5 \sqsubseteq g_1$.

Fortunately, initiality offers a solution to this problem too. We can make use of a general induction principle for initial T-algebras which is due to Lehmann, Smyth and Plotkin [LS81, Plo79]. This principle actually captures the intuitive reasoning which was used to deduce the informal interpretations of each of the K_n given in Figure 4.1. We shall state the principle in a rather general form and then see how this specialises to our construction, and to our favourite example in particular.

Proposition 4.5.3 (Initial T-Algebra Induction Principle) *If $T : \mathcal{C} \to \mathcal{C}$ is a functor, (A, α) is the initial T-algebra and $m : B \to A$ is a mono in $\mathcal{C}$ which extends to a T-homomorphism $(B, \beta) \to (A, \alpha)$, then m is an isomorphism.*

Proof. We have

$$\begin{array}{ccc} TB & \xrightarrow{Tm} & TA \\ {\scriptstyle\beta}\downarrow & & \downarrow{\scriptstyle\alpha} \\ B & \xrightarrow[m]{} & A \end{array}$$

for m mono and for some β. By initiality of (A, α) there is a unique h such that

$$\begin{array}{ccc} TA & \xrightarrow{Th} & TB \\ {\scriptstyle\alpha}\downarrow & & \downarrow{\scriptstyle\beta} \\ A & \xrightarrow[h]{} & B \end{array}$$

and thus $m \circ h : A \to A$ is a T-homomorphism, which must be id_A by initiality. Then $m \circ (h \circ m) = m = m \circ id_B$ so that $h \circ m = id_B$ as m is mono. Hence m is an isomorphism with inverse h. □

How does this help? Well, it leads immediately to

Corollary 4.5.4 *If $T : \mathcal{D}om_S \to \mathcal{D}om_S$ is a functor, (D, α) is the initial T-algebra and P is a subdomain of D with inclusion map $i : P \to D$ then the following induction principle is valid:*

$$\frac{\forall x \in TP.(\alpha \circ Ti)(x) \in P}{\forall y \in D.y \in P}$$

Proof. If the premiss of the proof rule holds, then we can define $\beta : TP \to P$ by $\beta(x) = (\alpha \circ Ti)(x)$ and the diagram

$$\begin{array}{ccc} TP & \xrightarrow{Ti} & TD \\ {\scriptstyle\beta}\downarrow & & \downarrow{\scriptstyle\alpha} \\ P & \xrightarrow[i]{} & D \end{array}$$

commutes, so that by Proposition 4.5.3 i is an isomorphism and hence P is the whole of D. □

And using this, we can get a simple condition on the maps g which is sufficient for J_g to be the whole of $\mathsf{Fix}_{\widehat{T}}$ and hence for K_g to be the whole of Fix_T.

Remark 4.5.5 A more categorical definition of J_g would have been by the pullback square

$$\begin{array}{ccc} J_g & \xrightarrow{j} & \mathsf{Fix}_{\widehat{T}} \\ {\scriptstyle !}\downarrow & \lrcorner & \downarrow{\scriptstyle g^*} \\ 1 & \xrightarrow[!]{} & \mathcal{O} \end{array}$$

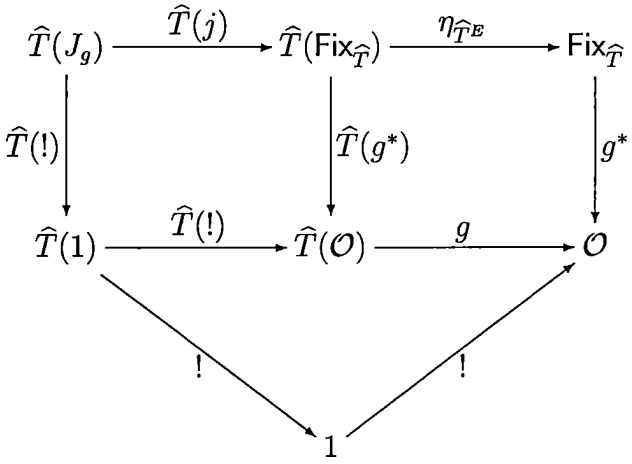
where j is the inclusion map. Note that J_g is a subdomain of $\mathsf{Fix}_{\widehat{T}}$, and not just a subset.

Proposition 4.5.6 *If the following diagram commutes:*

$$\begin{array}{ccc} \widehat{T}(1) & \xrightarrow{\widehat{T}(!)} & \widehat{T}(\mathcal{O}) \\ {\scriptstyle !}\downarrow & & \downarrow{\scriptstyle g} \\ 1 & \xrightarrow[!]{} & \mathcal{O} \end{array}$$

then J_g is the whole of $\mathsf{Fix}_{\widehat{T}}$.

Proof. Consider the following:



The square on the left commutes because it is $\widehat{T}$ applied to the pullback square in the remark above. The square on the right commutes because it is the definition of g^*, and the large triangle on the bottom commutes by assumption. Hence the outside path commutes, and since the composite $! \circ \widehat{T}(!) : \widehat{T}(J_g) \to 1$ on the left must be equal to $! : \widehat{T}(J_g) \to 1$, we have that

$$\forall x \in \widehat{T}(J_g).\, (\eta_{\widehat{T}E} \circ \widehat{T}j)(x) \in J_g$$

so that by Corollary 4.5.4 we are done. □

Checking the condition of Proposition 4.5.6 is a simple, finite computation which could be incorporated into a mechanizable formal system for reasoning about strictness properties. Let us see how it works out in the case of *nlist*. In this case, the condition on g is that the following pair of equations hold:

$$g(\mathit{inl}(*)) = \bot$$

$$\forall x \in \mathbf{2}.\, g(\mathit{inr}(x, \bot)) = \bot$$

and by monotonicity, the second of these reduces to checking

$$g(\mathit{inr}(\top, \bot)) = \bot$$

(Note that this simplification would apply in the analysis of lists of any type, since all our abstract domains have a top element.) Now, looking at Figure 4.1, we can see that these two conditions pick out g_1, g_5 and g_7, which are precisely those g_n for which K_n is the whole of the domain.

But we need more than a rule for determining when some K_g is the whole of Fix_T. We also want a rule for deducing more general inclusions between intersections of the K_g. In the case of *nlist*, for example, we should like some way of deducing that $K_{11} \cap K_{10} \subseteq K_{12}$ and that $K_8 \cap K_3 \subseteq K_4$. The initial algebra induction principle can give us this too.

If A and B are ideals of a domain D then the set

$$A \Rightarrow B \stackrel{\text{def}}{=} \{d \in D \mid d \in A \Rightarrow d \in B\}$$

is a subdomain of D, since it clearly contains $\perp_D$ and if $\langle d_n \rangle$ is a chain in $A \Rightarrow B$ then if $\bigsqcup d_n \in A$ we must have $d_n \in A$ for all n by down-closure of A. Hence $d_n \in B$ for all n and so $\bigsqcup d_n \in B$ as B is closed under sups of chains. Obviously, if $A \Rightarrow B$ is the whole of D then $A \subseteq B$.

Since the intersection of a set of ideals is an ideal, this means by Corollary 4.5.4 that to show that $J_{m_1} \cap \cdots \cap J_{m_k} \subseteq J_n$ (and hence that $K_{m_1} \cap \cdots \cap K_{m_k} \subseteq K_n$), it suffices to show

$$\forall l \in \widehat{T}(J_{m_1} \cap \cdots \cap J_{m_k} \Rightarrow J_n).\ (\eta_{\widehat{T}} \circ \widehat{T}i)(l) \in J_{m_1} \cap \cdots \cap J_{m_k} \Rightarrow J_n$$

where $i : (J_{m_1} \cap \cdots \cap J_{m_k} \Rightarrow J_n) \to \mathsf{Fix}_{\widehat{T}}$ is the inclusion map. I do not yet have a restatement of this condition in such a pleasant form as the condition of Proposition 4.5.6, but it is relatively easy to unwind in the special case of *nlist* to obtain

Proposition 4.5.7 *If the following two conditions hold*

1. *If* $\forall i.\ g_{m_i}(inl(*)) = \perp$ *then* $g_n(inl(*)) = \perp$
2. *For all* $b_1, \ldots, b_k, b \in \mathcal{O}$ *such that if* $\forall i.\ b_i = \perp$ *then* $b = \perp$ *we have that* $\forall x \in \mathbf{2}$ *if* $\forall i.\ g_{m_i}(inr(x, b_i)) = \perp$ *then* $g_n(inr(x, b)) = \perp$

then $K_{m_1} \cap \cdots \cap K_{m_k} \subseteq K_n$. □

Interestingly, this does not subsume Proposition 4.5.6, since Proposition 4.5.7 cannot be used to deduce $K_1 \subseteq K_7$. This is because, in the case of a g^* which is actually the constant $\perp$ function, the quantification in the second condition of Proposition 4.5.7 includes some irrelevant sets of values for the b_i. The conditions of Proposition 4.5.7 are finitely checkable, and it can be seen that this will remain true of the corresponding conditions for any algebraic type.

Using Propositions 4.5.6 and 4.5.7, we discover that only two more points need to be added the properties shown in Figure 4.1 to obtain a set which is closed under intersection. These are $K_4 \cap K_6$, with the informal interpretation of 'partial and infinite lists with $\perp$ as the first element', and $K_4 \cap K_9$ which corresponds to 'non-empty lists all of whose elements are $\perp$'. For *nlist*, we thus get the lattice of properties shown in Figure 4.2. This figure was deduced with the aid of a short computer program to check the conditions of Proposition 4.5.7.

Burn's A^{rp} abstract domain for lists [Bur91b] contains our K_1, K_4, K_6, $K_4 \cap K_6$, K_8 and K_{12}. His A^{hts} domain contains one point in addition to these, which we do not have. This corresponds to non-empty lists with a $\perp$ head and a tail which satisfies our K_8, and is perhaps harder to justify intuitively than the other points. This point

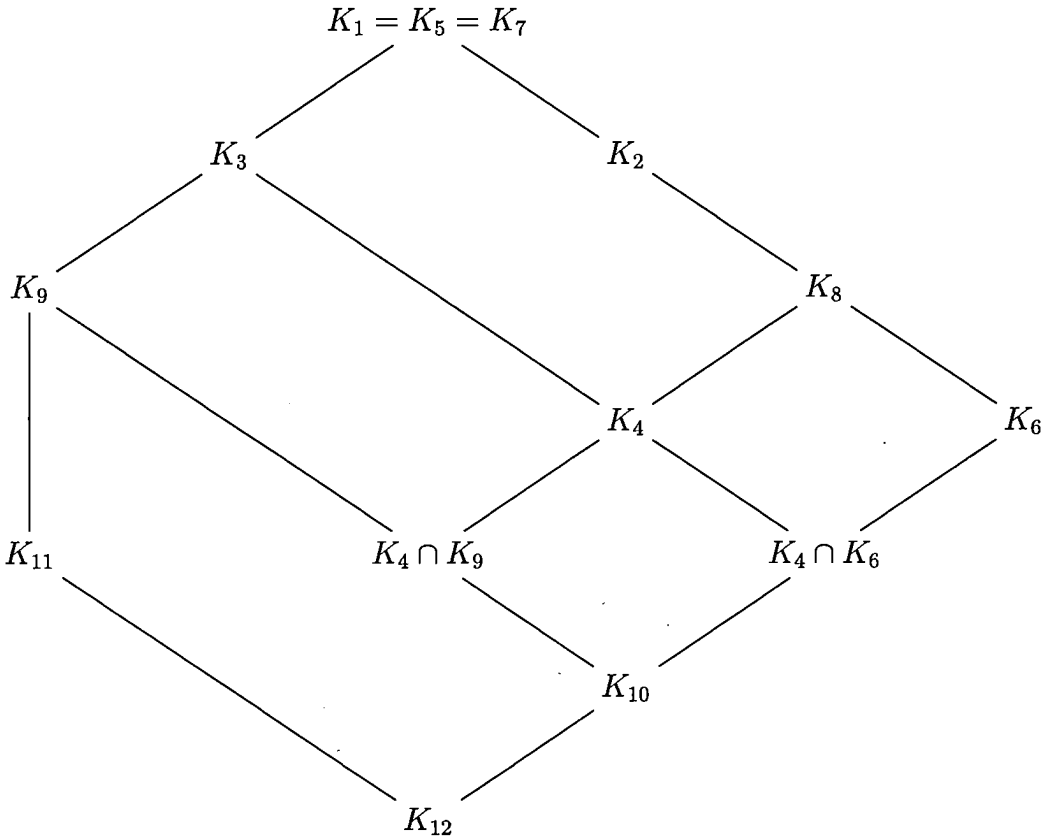


Figure 4.2: The Lattice of Strictness Properties of *nlist*

$$\Gamma \vdash \mathsf{Nil} : K_{11}$$

$$\frac{\Gamma \vdash u : \mathbf{f} \qquad g(\bot, \top) = \bot}{\Gamma \vdash \mathsf{Cons}(u, l) : K_g}\ [\text{Cons1}]$$

$$\frac{\Gamma \vdash u : \mathbf{f} \qquad \Gamma \vdash l : K_g \qquad g(\bot, \bot) = \bot}{\Gamma \vdash \mathsf{Cons}(u, l) : K_g}\ [\text{Cons2}]$$

$$\frac{\Gamma \vdash l : K_g \qquad g(\top, \bot) = \bot}{\Gamma \vdash \mathsf{Cons}(u, l) : K_g}\ [\text{Cons3}] \qquad \frac{g(\top, \top) = \bot}{\Gamma \vdash \mathsf{Cons}(u, l) : K_g}\ [\text{Cons4}]$$

Figure 4.3: Proof Rules for Constructors of *nlist*

would, however, appear (along with a large number of others) if we were to apply our construction using the functor T^2, rather than T, in the way mentioned earlier.

Clearly, much further work remains to be done on strictness analysis of algebraic types. In particular, we should look at

1. How the construction proposed here works for examples other than *nlist*;
2. Obtaining a good syntactic representation for the K_g in the general case;
3. Synthesising proof rules which express the entailment relation generated by Proposition 4.5.6 and a more general version of Proposition 4.5.7;
4. Synthesising program logic rules for the constructors and destructors associated with algebraic types.

For the moment we simply give, without any formal justification, the slightly informal proof rules for the type *nlist* which are shown in Figures 4.3 and 4.4. These, together with the entailment relation on $\mathcal{L}_{nlist}$ which is implicit in Figure 4.2, extend the program logic $\mathcal{PL}1$ to the language $\Lambda_{T,nlist}$.

The proof rules for constructors are quite pleasant—the four rules for Cons can be used to derive all of the 24 individual proof rules, such as

$$\frac{\Gamma \vdash u : \mathbf{f} \qquad \Gamma \vdash l : K_9}{\Gamma \vdash \mathsf{Cons}(u, l) : K_4 \cap K_9}$$

which one might be tempted to write down. The pattern of these four rules should generalise to constructors of other algebraic types. The rules for the destructor nlistcase appear slightly more arbitrary. There is a pattern, but it seems harder to abstract. Note in particular the disjunctive character of the [nlistcase8] rule, and the fact that the rules for $K_4 \cap K_6$ and $K_4 \cap K_9$ are not derivable from the other rules. Given our previous remarks about sums, it may well be that extending the disjunctive logic to algebraic types gives a more natural set of rules. This extension

$$\frac{\Gamma \vdash u : \phi \qquad \Gamma, x : \mathbf{t}, xs : K_1 \vdash v : \phi \qquad \Gamma \vdash l : K_1}{\Gamma \vdash \mathsf{nlistcase}\ l\ \mathsf{of\ Nil} \Rightarrow u \mid \mathsf{Cons}(x, xs) \Rightarrow v\ : \phi}\ [\text{nlistcase1}]$$

$$\frac{\Gamma, x : \mathbf{t}, xs : K_1 \vdash v : \phi \qquad \Gamma \vdash l : K_2}{\Gamma \vdash \mathsf{nlistcase}\ l\ \mathsf{of\ Nil} \Rightarrow u \mid \mathsf{Cons}(x, xs) \Rightarrow v\ : \phi}\ [\text{nlistcase2}]$$

$$\frac{\Gamma \vdash u : \phi \qquad \Gamma, x : \mathbf{f}, xs : K_1 \vdash v : \phi \qquad \Gamma \vdash l : K_3}{\Gamma \vdash \mathsf{nlistcase}\ l\ \mathsf{of\ Nil} \Rightarrow u \mid \mathsf{Cons}(x, xs) \Rightarrow v\ : \phi}\ [\text{nlistcase3}]$$

$$\frac{\Gamma, x : \mathbf{f}, xs : K_1 \vdash v : \phi \qquad \Gamma \vdash l : K_4}{\Gamma \vdash \mathsf{nlistcase}\ l\ \mathsf{of\ Nil} \Rightarrow u \mid \mathsf{Cons}(x, xs) \Rightarrow v\ : \phi}\ [\text{nlistcase4}]$$

$$\frac{\Gamma, x : \mathbf{t}, xs : K_6 \vdash v : \phi \qquad \Gamma \vdash l : K_6}{\Gamma \vdash \mathsf{nlistcase}\ l\ \mathsf{of\ Nil} \Rightarrow u \mid \mathsf{Cons}(x, xs) \Rightarrow v\ : \phi}\ [\text{nlistcase6}]$$

$$\frac{\Gamma, x : \mathbf{t}, xs : K_8 \vdash v : \phi \qquad \Gamma, x : \mathbf{f}, xs : K_1 \vdash v : \phi \qquad \Gamma \vdash l : K_8}{\Gamma \vdash \mathsf{nlistcase}\ l\ \mathsf{of\ Nil} \Rightarrow u \mid \mathsf{Cons}(x, xs) \Rightarrow v\ : \phi}\ [\text{nlistcase8}]$$

$$\frac{\Gamma \vdash u : \phi \qquad \Gamma, x : \mathbf{f}, xs : K_9 \vdash v : \phi \qquad \Gamma \vdash l : K_9}{\Gamma \vdash \mathsf{nlistcase}\ l\ \mathsf{of\ Nil} \Rightarrow u \mid \mathsf{Cons}(x, xs) \Rightarrow v\ : \phi}\ [\text{nlistcase9}]$$

$$\frac{\Gamma, x : \mathbf{f}, xs : K_{10} \vdash v : \phi \qquad \Gamma \vdash l : K_{10}}{\Gamma \vdash \mathsf{nlistcase}\ l\ \mathsf{of\ Nil} \Rightarrow u \mid \mathsf{Cons}(x, xs) \Rightarrow v\ : \phi}\ [\text{nlistcase10}]$$

$$\frac{\Gamma \vdash u : \phi \qquad \Gamma \vdash l : K_{11}}{\Gamma \vdash \mathsf{nlistcase}\ l\ \mathsf{of\ Nil} \Rightarrow u \mid \mathsf{Cons}(x, xs) \Rightarrow v\ : \phi}\ [\text{nlistcase11}]$$

$$\frac{\Gamma \vdash l : K_{12}}{\Gamma \vdash \mathsf{nlistcase}\ l\ \mathsf{of\ Nil} \Rightarrow u \mid \mathsf{Cons}(x, xs) \Rightarrow v\ : \mathbf{f}}\ [\text{nlistcase12}]$$

$$\frac{\Gamma, x : \mathbf{f}, xs : K_9 \vdash v : \phi \qquad \Gamma \vdash l : K_4 \cap K_9}{\Gamma \vdash \mathsf{nlistcase}\ l\ \mathsf{of\ Nil} \Rightarrow u \mid \mathsf{Cons}(x, xs) \Rightarrow v\ : \phi}\ [\text{nlistcase49}]$$

$$\frac{\Gamma, x : \mathbf{f}, xs : K_6 \vdash v : \phi \qquad \Gamma \vdash l : K_4 \cap K_6}{\Gamma \vdash \mathsf{nlistcase}\ l\ \mathsf{of\ Nil} \Rightarrow u \mid \mathsf{Cons}(x, xs) \Rightarrow v\ : \phi}\ [\text{nlistcase46}]$$

Figure 4.4: Proof Rules for nlistcase

should be straightforward, as the idea behind Proposition 4.5.7 can also be applied to reasoning about inclusions between finite unions of properties. The rules we have given are, however, powerful enough to derive many useful strictness properties:

Examples

- Given the definition

$$\mathsf{ones} \stackrel{\text{def}}{=} \mathsf{fix}(l.\mathsf{Cons}(\underline{1}, l))$$

we can derive the judgement

$$\vdash \mathsf{ones} : K_6$$

In other words, **ones** is an infinite or partial list.

- Given our previous definition of **length**, the judgement

$$\vdash \mathsf{length} : K_6 \to \mathbf{f}$$

is derivable. This corresponds to the fact that **length** evaluates the entire spine of its argument.

- Similarly, for **sumlist** we can derive

$$\vdash \mathsf{sumlist} : K_8 \to \mathbf{f}$$

which means that $(\mathsf{sumlist}\, l)$ will diverge if l is infinite, partial or contains any divergent elements. This corresponds to the fact that **sumlist** evaluates all the structure and all the elements of its argument.

- Given the following set of definitions:

$$\begin{array}{rcl}
\mathsf{from} & \stackrel{\text{def}}{=} & \lambda n.\mathsf{fix}(l.\mathsf{Cons}(n, (\mathsf{mapinc}\, l))) \\
\mathsf{pick} & \stackrel{\text{def}}{=} & \mathsf{fix}(f.\lambda m.\lambda l. \\
 & & \quad \mathsf{nlistcase}\ l\ \ \mathsf{of}\ \ \mathsf{Nil} \Rightarrow \Omega \\
 & & \quad\quad\quad\quad \mid\ \ \mathsf{Cons}(x, xs) \Rightarrow \mathsf{if}\ m\ \mathsf{then}\ x\ \mathsf{else}\ f\ (x - \underline{1})\ xs) \\
\mathsf{sillyplus} & \stackrel{\text{def}}{=} & \lambda m.\lambda n.\mathsf{pick}\, m\ (\mathsf{from}\, n)
\end{array}$$

we can deduce

$$\vdash \mathsf{sillyplus} : \mathbf{t} \to \mathbf{f} \to \mathbf{f}$$

i.e. that **sillyplus** is strict in n.

- Given the following definition:

$$\begin{array}{rcl}
\mathsf{map} \stackrel{\text{def}}{=} \mathsf{fix}(f.\lambda h.\lambda l.\mathsf{nlistcase}\ l\ \ \mathsf{of}\ \ \mathsf{Nil} \Rightarrow \mathsf{Nil} \\
\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad \mid\ \ \mathsf{Cons}(x, xs) \Rightarrow \mathsf{Cons}(h\, x, f\, h\, xs))
\end{array}$$

we can derive

$$\vdash \mathsf{map} : \mathbf{f}^{\iota \to \iota} \to K_2 \to K_4 \cap K_9$$

which says that in a $K_4 \cap K_9$-strict context, $\mathsf{map}\, h\, l$ is strict in h if l is non-empty.

Chapter 5

Parametricity, Free Theorems and Polymorphic Invariance

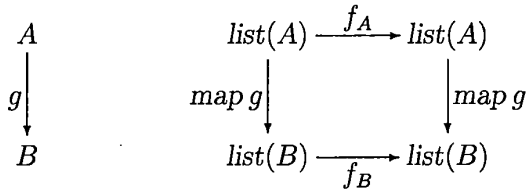
5.1 Introduction

Most modern functional languages have type systems which are more powerful than that of $\Lambda_{T,a}$, in that they incorporate some form of polymorphism. Strachey [Str67] distinguished between two kinds of polymorphism in programming languages. The first is *ad hoc polymorhism*, or *overloading*, in which the same expression may denote different operations at different types (for example, the use of the symbol + to represent both integer and real addition). The second is *parametric polymorphism*, in which an expression denotes 'the same thing' uniformly at different types (for example, a polymorphic identity function).

This chapter is concerned with parametric polymorphism, and the question of what is really meant by 'the same thing' above; that is, with semantic relationships between the different instances of a term with a polymorphic type. Results which show in what sense a polymorphic term behaves uniformly at different types are known as *parametricity* results. Parametricity has been (and continues to be) widely studied, mainly in the framework of models for $\lambda 2$ and other higher-order typed lambda calculi [Rey83, BFSS90].

In [Wad89], Wadler suggested that parametricity is a source of theorems which are useful in proving properties of programs. For each polymorphic type σ, there is a theorem which is satisfied by all terms of type σ. A typical example is is the type $\sigma = \forall\alpha.list(\alpha) \to list(\alpha)$. Intuitively (and ignoring the possibility of non-termination), any function f of type σ cannot 'look at' any of the elements of its argument, since it will work on lists of any type. f can therefore only be a rearranging function, such as the function which reverses its argument, or which deletes every element in an even-numbered position. This means that if we map a (strict) function g over a list and then apply f to the result, we get the same as if we had applied f to the original list and then mapped g over the result. In a slightly

informal diagram:



Categorically we can view $list(-)$ as a functor on $\mathcal{D}om$, with the action on morphisms given by *map*. Were it not for the restriction that g be strict, this diagram would then say that f is a natural transformation $list(-) \dot{\to} list(-)$.

Wadler justified these 'free theorems' by appealing to a model of $\lambda 2$ [BMM90]. Most functional languages do not have type systems which are this sophisticated. They tend to use (extensions of) the Hindley-Milner type system [Mil78], and it seems excessive to model this relatively simple system using a full-blown model of $\lambda 2$. In Section 5.2 we show how to extend the syntax of $\Lambda_{T,a}$ to include an explicitly-typed variant of Hindley-Milner ('ML-style') polymorphism and show how this may be modelled by a modest extension of the domain-theoretic semantics which we have already given for the monomorphic language. We then show how Wadler's results may be obtained from this semantics by a logical relations argument. Whilst the connection between logical relations and parametricity is well-known, our treatment of the let construct and of parameterised lazy algebraic datatypes seems to be new. Note that this is far from being 'state of the art' work on parametricity—we are merely showing how something simple may be done simply.

The material in Section 5.2 is something of a diversion from the main topic of this dissertation, which is strictness analysis. It arose, however, from the work described in Section 5.3, which concerns the relationship between the derivable strictness properties (in the conjunctive logic of Chapter 3 or the equivalent BHA abstract interpretation) of different instances of terms with polymorphic types. As well as being an interesting theoretical question, this is potentially very important for implementations, as we should like to be able to analyse polymorphic functions without analysing all their monomorphic instances separately.

The relationship between the strictness properties of different instances of polymorphic functions was first studied by Abramsky in [Abr85]. A complication in that work was that the characterisation of strictness was denotational (semantic) whereas that of polymorphism came from the syntactic rules for type assignment. To reconcile these two viewpoints, Abramsky had to go via an operational semantics (which is inherently syntactic) for the abstract functions. By proving a computational adequacy result for the BHA-style abstract semantics with respect to this operational semantics, and showing that evaluation in the operational semantics was unaffected by the erasure of type information, he was able to show that if the abstract interpretation of one instance of a function is strict, then all instances are. This property of the abstract interpretation is an example of Abramsky's notion of *polymorphic invariance*.

Hughes has approached the problem semantically in [Hug88], but the techniques used there are restricted to first-order functions. More recently, a better understanding of the semantics of polymorphism in terms of *relators* [MS92] allowed Abramsky and Jensen to give a much neater semantic proof of Abramsky's orginal invariance result [AJ91].

The main result of Section 5.3 is a polymorphic invariance theorem. This states, roughly, that a property ϕ can be deduced of a particular instance of a polymorphic term t if and only if it can be deduced of all instances of t for which the property ϕ 'makes sense'. This improves on previous results in that it shows polymorphic invariance of higher-order strictness properties, rather than just a first-order property (simple strictness) of higher-order functions.

5.2 ML-style Polymorphism and Free Theorems

One of the main reasons for the popularity of the Hindley-Milner type system in functional languages is that, whilst it is powerful enough to offer the programmer significant advantages over a monomorphic type system, it is simple enough for there to be an practical type inference algorithm[1]. We shall not discuss the inference algorithm, which has been very well described elsewhere (see [Mil78, DM82, Car87], for example); instead we add explicit polymorphic types to $\Lambda_{T,a}$ and assert that there is a straightforward translation of typing derivations for untyped terms into this extended language $\Lambda_{P,a}$.

5.2.1 The Language $\Lambda_{P,a}$

Syntax

Types We assume a single parameterised algebraic type definition, which has the general form:

$$\begin{aligned} a(\xi) \;&=\; C_1 \text{ of } \delta_{1,0} \times \cdots \times \delta_{1,m_1-1} \\ &\;\;\vdots \\ &\mid\; C_n \text{ of } \delta_{n,0} \times \cdots \times \delta_{n,m_n-1}; \end{aligned}$$

where the type expressions δ are given by the grammar

$$\delta \;::=\; a(\xi) \mid \kappa$$

$$\kappa \;::=\; \iota \mid \kappa \times \kappa \mid \kappa \to \kappa \mid \xi$$

[1]The worst-case complexity of the inference algorithm is actually exponential in the size of the program, but this does not appear to cause any serious problems in practice [KMM91].

The variable ξ is not a proper type variable, it is merely a placeholder in the declaration of the parameterised type. Note also that we do not allow non-uniform recursion in type definitions—ML allows declarations such as

$$\mathsf{datatype\ 'a\ silly = A \mid B\ of\ ('a \times 'a)\ silly;}$$

although values of such types are rather hard to use. We do not allow such declarations. Although the datatype declaration is only parameterised on a single type ξ, this restriction is made only for notational convenience in what follows.

We are given a set of type variables, which are ranged over with α and β. The types of $\Lambda_{P,a}$ are then defined by

$$\sigma ::= \iota \mid \alpha \mid \sigma \to \sigma \mid \sigma \times \sigma \mid a(\sigma)$$

and for each δ and type σ we define the type $\delta(\sigma)$ in the obvious way, *viz*

$$a(\xi)(\sigma) = a(\sigma) \qquad \iota(\sigma) = \iota$$

$$(\kappa_1 \times \kappa_2)(\sigma) = \kappa_1(\sigma) \times \kappa_2(\sigma) \qquad (\kappa_1 \to \kappa_2)(\sigma) = \kappa_1(\sigma) \to \kappa_2(\sigma)$$

$$\xi(\sigma) = \sigma$$

A *substitution* S is defined to be a map from type variables to types. As usual, this extends to a map from types to types which is also denoted by S. If σ is a type, then we write $TV(\sigma)$ for the set of type variables occuring in σ.

Terms Although $\Lambda_{P,a}$ has explicitly typed variables, we shall need some restrictions on term formation which are best expressed by defining the well-formed terms relative to a context. A context Θ is a finite set of typed variables $\{x_0^{\sigma_0}, \ldots, x_{k-1}^{\sigma_{k-1}}\}$ and we write Θ, x^σ for $\Theta \cup \{x^\sigma\}$. The rules for term formation are shown in Figure 5.1, where the judgement $\Theta \vdash s :: \sigma$ is read as 'in context Θ, s is a well-formed term of type σ'.

Note that this syntax makes the *use* of polymorphism in a let statement very explicit by binding distinct variables to distinct type instances of the polymorphic term. Some 'trick' of this general form seems necessary to make our rather simple-minded approach to semantics work. This corresponds to the way in which the type inference algorithm makes fresh copies of generic type variables (those which do not occur in the context) for each occurrence of a let-bound identifier. It is also well-known that the ML type system can be presented using a rule for let which substitutes copies of the *term* being bound for each occurrence of the bound variable in the body. Our approach can be seen as somewhere between copying the entire term and just generating new copies of type variables.

The side condition on the rule for forming let expressions is perhaps stronger than is absolutely necessary since we could allow substitution for type variables which only appear in the context in the types of variables which will subsequently become bound by a let, rather than a λ or a fix construct. The rule given has, however, the advantage of simplicity. A term which is well-formed in some context has the same unique type in each context in which it is well-formed. We will sometimes just write $t :: \tau$ for $\Theta \vdash t :: \tau$ when Θ is either clear or unimportant.

$$\Theta, x^\sigma \vdash x^\sigma :: \sigma \qquad \Theta \vdash \underline{n} :: \iota$$

$$\frac{\Theta, x^\sigma \vdash t :: \tau}{\Theta \vdash \lambda x^\sigma.t :: \sigma \to \tau} \qquad \frac{\Theta \vdash t :: \sigma \to \tau \qquad \Theta \vdash s :: \sigma}{\Theta \vdash (t\, s) :: \tau}$$

$$\frac{\Theta \vdash s :: \sigma \qquad \Theta \vdash t :: \tau}{\Theta \vdash (s,t) :: \sigma \times \tau} \qquad \frac{\Theta \vdash s :: \iota \qquad \Theta \vdash t :: \iota}{\Theta \vdash s + t :: \iota}$$

$$\frac{\Theta \vdash s :: \sigma \times \tau}{\Theta \vdash \mathsf{fst}(s) :: \sigma} \qquad \frac{\Theta \vdash s :: \sigma \times \tau}{\Theta \vdash \mathsf{snd}(s) :: \tau}$$

$$\frac{\Theta \vdash s :: \iota \qquad \Theta \vdash t_1 :: \tau \qquad \Theta \vdash t_2 :: \tau}{\Theta \vdash \mathsf{if}\ s\ \mathsf{then}\ t_1\ \mathsf{else}\ t_2 :: \tau} \qquad \frac{\Theta, x^\sigma \vdash s :: \sigma}{\Theta \vdash \mathsf{fix}(x^\sigma.s) :: \sigma}$$

$$\frac{\Theta \vdash s :: \sigma \qquad \Theta, x_1^{S_1\sigma}, \ldots, x_k^{S_k\sigma} \vdash t :: \tau}{\Theta \vdash \mathsf{let}\ x_1^{S_1\sigma}, \ldots, x_k^{S_k\sigma} = s\ \mathsf{in}\ t :: \tau} \quad \text{provided } S_i(\alpha) = \alpha \text{ for all } i \text{ and } \alpha \in \bigcup_{x^\sigma \in \Theta} TV(\sigma)$$

$$\frac{\Theta \vdash s_0 :: \delta_{i,0}(\sigma) \quad \cdots \quad \Theta \vdash s_{m_i-1} :: \delta_{i,m_i-1}(\sigma)}{\Theta \vdash C_i^\sigma(s_0, \ldots, s_{m_i-1}) :: a(\sigma)}$$

$$\frac{\begin{array}{l}\Theta \vdash s :: a(\sigma) \\ \Theta, x_{1,0}^{\delta_{1,0}(\sigma)}, \ldots, x_{1,m_1-1}^{\delta_{1,m_1-1}(\sigma)} \vdash t_1 :: \tau \qquad \cdots \qquad \Theta, x_{n,0}^{\delta_{n,0}(\sigma)}, \ldots, x_{n,m_n-1}^{\delta_{n,m_n-1}(\sigma)} \vdash t_n :: \tau\end{array}}{\begin{array}{l}\Theta \vdash \mathsf{acase}^\sigma\ s\ \ \mathsf{of}\ \ C_1(x_{1,0}, \ldots, x_{1,m_1-1}) \Rightarrow t_1 \\ \qquad\qquad \vdots \qquad\qquad\qquad\qquad\qquad\qquad :: \tau \\ \qquad\qquad \mid\ \ C_n(x_{n,0}, \ldots, x_{n,m_n-1}) \Rightarrow t_n\end{array}}$$

Figure 5.1: Syntax of $\Lambda_{P,a}$

Examples

- The following term is well-formed in the empty context:

$$\mathsf{let}\ f_1^{(\alpha\to\alpha)\to(\alpha\to\alpha)}, f_2^{\alpha\to\alpha} = \lambda x^{\beta}.x^{\beta}\ \mathsf{in}\ (f_1^{(\alpha\to\alpha)\to(\alpha\to\alpha)}\ f_2^{\alpha\to\alpha}) :: \alpha\to\alpha$$

- The following is *not* well-formed, as it violates the side condition on the rule for let:

$$\lambda x^{\alpha}.\mathsf{let}\ y^{\iota} = x^{\alpha}\ \mathsf{in}\ y^{\iota} + \underline{3} :: \alpha\to\iota$$

- In the language $\Lambda_{P,list}$, for which the algebraic type declaration is

$$list(\xi)\ =\ \mathsf{Nil} \mid \mathsf{Cons\ of}\ \xi\times list(\xi);$$

the following is a well-formed term of type ι in the empty context:

$$\begin{array}{l}\mathsf{let}\ l_1^{list(\iota)\to\iota}, l_2^{list(\iota\times\iota)\to\iota} = \mathsf{fix}(f^{list(\alpha)\to\iota}, \lambda y^{list(\alpha)}.\\ \qquad \mathsf{listcase}^{\alpha}\ y^{list(\alpha)}\ \mathsf{of}\ \ \mathsf{Nil}\Rightarrow\underline{0}\\ \qquad\qquad\qquad\qquad \mid\ \ \mathsf{Cons}(x^{\alpha}, xs^{list(\alpha)})\Rightarrow\underline{1}+(f^{list(\alpha)\to\iota}\ xs^{list(\alpha)}))\\ \mathsf{in}\\ \quad (l_1^{list(\iota)\to\iota}\ \mathsf{Cons}^{\iota}(\underline{2},\mathsf{Nil}^{\iota})) + (l_2^{list(\iota\times\iota)\to\iota}\ \mathsf{Cons}^{\iota\times\iota}((\underline{3},\underline{4}),\mathsf{Nil}^{\iota\times\iota}))\end{array}$$

Denotational Semantics

Define a *world* w to be a function from type variables to domains. The denotational semantics of $\Lambda_{P,a}$ will be parameterised on a choice of world. It may be helpful to think of a type variable as representing an unspecified base type; a world then specifies a particular interpretation for each of these base types.

For a world w and a type σ, the domain D_σ^w is given by

$$D_\iota^w = \mathbb{N}_\perp \qquad\qquad D_\alpha^w = w(\alpha)$$

$$D_{\sigma\to\tau}^w = [D_\sigma^w \to D_\tau^w] \qquad\qquad D_{\sigma\times\tau}^w = D_\sigma^w \times D_\tau^w$$

$$D_{a(\sigma)}^w = \mathsf{Fix}_T$$

where the functor $T : \mathcal{D}om_S \to \mathcal{D}om_S$ is defined as

$$T(X) = \sum_{i=1}^{n}\prod_{j=0}^{m_i-1} D_{i,j}(X)$$

and

$$D_{i,j}(X) = \begin{cases} D_{\kappa(\sigma)}^w & \text{if } \delta_{i,j} = \kappa \\ X & \text{if } \delta_{i,j} = a(\xi)\end{cases}$$

An w-environment ρ is a finite, type-respecting, partial function from variable names to the disjoint union of all the D_σ^w (so $\forall x^\sigma \in dom(\rho).\ \rho(x^\sigma) \in D_\sigma^w$). Given such a ρ,

$$[\![x^\sigma]\!]^w\rho = \rho(x^\sigma) \qquad [\![\underline{n}]\!]^w\rho = [n]$$

$$[\![s+t]\!]^w\rho = \begin{cases} \bot & \text{if } [\![s]\!]^w\rho = \bot \text{ or } [\![t]\!]^w\rho = \bot \\ [m+n] & \text{if } [\![s]\!]^w\rho = [m] \text{ and } [\![t]\!]^w\rho = [n] \end{cases}$$

$$[\![\lambda x^\sigma.t]\!]^w\rho = \lambda d \in D^w_\sigma.[\![t]\!]^w\rho[x^\sigma \mapsto d] \qquad [\![t\,s]\!]^w\rho = ([\![t]\!]^w\rho)\,([\![s]\!]^w\rho)$$

$$[\![(t,s)]\!]^w\rho = ([\![t]\!]^w\rho,\ [\![s]\!]^w\rho)$$

$$[\![\mathsf{fst}(s)]\!]^w\rho = \pi_1([\![s]\!]^w\rho) \qquad [\![\mathsf{snd}(s)]\!]^w\rho = \pi_2([\![s]\!]^w\rho)$$

$$[\![\mathsf{if}\ s\ \mathsf{then}\ t_1\ \mathsf{else}\ t_2]\!]^w\rho = \begin{cases} \bot & \text{if } [\![s]\!]^w\rho = \bot \\ [\![t_1]\!]^w\rho & \text{if } [\![s]\!]^w\rho = [0] \\ [\![t_2]\!]^w\rho & \text{otherwise} \end{cases}$$

$$[\![\mathsf{fix}(x^\sigma.s)]\!]^w\rho = \bigsqcup_k d_k \quad \text{where } d_0 = \bot,\ d_{k+1} = [\![s]\!]^w\rho[x^\sigma \mapsto d_k]$$

$$[\![\mathsf{let}\ x_1^{S_1\sigma}, \ldots, x_k^{S_k\sigma} = s\ \mathsf{in}\ t]\!]^w\rho = [\![t]\!]^w\rho[x_1^{S_1\sigma} \mapsto [\![s]\!]^{S_1 w}\rho, \ldots, x_k^{S_k\sigma} \mapsto [\![s]\!]^{S_k w}\rho]$$

$$[\![C_i^\sigma(t_0, \ldots, t_{m_i-1})]\!]^w\rho = \eta(in_i([\![t_0]\!]^w\rho, \ldots, [\![t_{m_i-1}]\!]^w\rho))$$

$$[\![\mathsf{acase}^\sigma\ s\ \mathsf{of}\ C_1(x_{1,0}, \ldots, x_{1,m_1-1}) \Rightarrow t_1 \mid \cdots \Rightarrow t_n]\!]^w\rho$$
$$= \begin{cases} \bot & \text{if } [\![s]\!]^w\rho = \bot \\ [\![t_i]\!]^w\rho[x_{i,0} \mapsto d_0, \ldots, x_{i,m_i-1} \mapsto d_{m_i-1}] & \text{if } [\![s]\!]^w\rho = \eta(in_i(d_0, \ldots, d_{m_i-1})) \end{cases}$$

Figure 5.2: Denotational Semantics of $\Lambda_{P,a}$

the extended environment $\rho[x^\sigma \mapsto d]$ for $d \in D^w_\sigma$ is defined as before, and we write $\Theta(\rho)$ for $dom(\rho)$ regarded as a context.

If w is a world and S is a substitution, define the world Sw by $(Sw)(\alpha) = D^w_{S\alpha}$. Then $D^{Sw}_\sigma = D^w_{S\sigma}$ for any type σ. Note that with this definition of the action of a substitution on a world, composition of substitutions gets reversed:

$$D^{S_1(S_2 w)}_\sigma = D^{S_2 w}_{S_1\sigma} = D^w_{S_2(S_1\sigma)} = D^w_{(S_2 \circ S_1)\sigma} = D^{(S_2 \circ S_1)w}_\sigma$$

For a world w, w-environment ρ and term t such that $\Theta(\rho) \vdash t :: \tau$, we define $[\![t]\!]^w\rho \in D^w_\tau$ by induction on the structure of t as shown in Figure 5.2. We omit the detailed verification that this is well-defined, but note that the side-condition on the formation rule for **let** ensures that the right-hand side of the corresponding semantic equation is well-defined, by making sure that the w-environment ρ is also a $S_i w$-environment for each i. Note that neither the syntax nor the semantics of $\Lambda_{P,a}$ makes any explicit mention of universally quantified types such as $\forall\alpha.\,\alpha \to \alpha$.

A related treatment of this kind of polymorphism may be found in recent work of

Phoa and Fourman on the categorical semantics of core ML [PF92, Pho92]. Given a cartesian closed category $\mathcal{C}$ in which the monomorphic language may be interpreted, they model the polymorphic language in the polynomial category $\mathcal{C}[X_1, \ldots, X_n]$, which is roughly the category obtained by adding n indeterminate objects to $\mathcal{C}$, in a manner analagous to the construction of the ring $R[X_1, \ldots, X_n]$ of polynomials in n variables over a ring R.

It should also be pointed out that we have not given any operational semantics to the polymorphic language here. Doing this, and then proving computational adequacy of the denotational semantics with respect to the operational semantics should not be very difficult.

5.2.2 Free Theorems

Having given the denotational semantics, we now turn to the relationship between the interpretations of a term at different worlds. For two worlds w and w', we assume that we are given a family $\{g_\alpha\}$ of strict maps, indexed by the type variables, where $g_\alpha : w(\alpha) \to w'(\alpha)$. From these maps, we shall then define a logical relation $\{\mathcal{P}^\sigma\}$ where $\mathcal{P}^\sigma \subseteq D^w_\sigma \times D^{w'}_\sigma$, and show that the interpretation of a term at world w is related to its interpretation at world w'. This will allow us to prove the kind of parametricity results which were mentioned in the introduction.

The definition of $\mathcal{P}^\sigma$ is by induction on the structure of the type σ. For non-algebraic types, the definition is as follows:

$$d\,\mathcal{P}^\iota\,d' \;\Leftrightarrow\; d = d'$$

$$d\,\mathcal{P}^\alpha\,d' \;\Leftrightarrow\; g_\alpha(d) = d'$$

$$f\,\mathcal{P}^{\sigma\to\tau}\,f' \;\Leftrightarrow\; \forall d, d'.\, d\,\mathcal{P}^\sigma\,d' \Rightarrow (f\,d)\,\mathcal{P}^\tau\,(f'\,d')$$

$$p\,\mathcal{P}^{\sigma\times\tau}\,p' \;\Leftrightarrow\; \pi_1(p)\,\mathcal{P}^\sigma\,\pi_1(p') \text{ and } \pi_2(p)\,\mathcal{P}^\tau\,\pi_2(p')$$

For algebraic types, we proceed in a way which is similar to that by which we proved computational adequacy for $\Lambda_{T,a}$ in Chapter 4. Let the functors $T, T' : \mathcal{D}om_S \to \mathcal{D}om_S$ be those associated with the construction of $D^w_{a(\sigma)}$ and $D^{w'}_{a(\sigma)}$ via the cocones $\nu : \Delta \to \mathsf{Fix}_T$ and $\nu' : \Delta' \to \mathsf{Fix}_{T'}$ where $\Delta = \langle D_k, f_k\rangle$ and $\Delta' = \langle D'_k, f'_k\rangle$.

Given a relation $R \subseteq E \times E'$ between two domains, define the relation $(T, T')(R) \subseteq T(E) \times T'(E')$ by

$$\begin{aligned} d\,(T,T')(R)\,d' \;\Leftrightarrow\;& d = \bot \text{ and } d' = \bot \\ \text{or}\;& d = in_i(d_0, \ldots, d_{m_i-1}),\; d' = in_i(d'_0, \ldots, d'_{m_i-1}) \text{ and} \\ & \forall j.\, d_j\,P_j\,d'_j \end{aligned}$$

where

$$P_j = \begin{cases} \mathcal{P}^{\kappa(\sigma)} & \text{if } \delta_{ij} = \kappa \\ R & \text{if } \delta_{ij} = a(\xi) \end{cases}$$

Now define, for $k \in \omega$, the relation $\mathcal{P}_k^{a(\sigma)} \subseteq D_k \times D'_k$ by

$$\mathcal{P}_0^{a(\sigma)} = \{(\bot, \bot)\}$$

$$\mathcal{P}_{k+1}^{a(\sigma)} = (T, T')(\mathcal{P}_k^{a(\sigma)})$$

and then define $\mathcal{P}^{a(\sigma)} \subseteq D^w_{a(\sigma)} \times D^{w'}_{a(\sigma)}$ by

$$d\, \mathcal{P}^{a(\sigma)}\, d' \;\Leftrightarrow\; \forall k.\, (\nu_k^R\, d)\, \mathcal{P}_k^{a(\sigma)}\, (\nu_k'^R\, d')$$

Lemma 5.2.1 *For all types σ, the relation $\mathcal{P}^\sigma$ is strict and ω-complete. That is to say*

1. $\bot_{D^w_\sigma}\, \mathcal{P}^\sigma\, \bot_{D^{w'}_\sigma}$ *and*
2. *If $\langle d_n \rangle$ and $\langle d'_n \rangle$ are ω-chains in D^w_σ and $D^{w'}_\sigma$ respectively, such that for all n, $d_n\, \mathcal{P}^\sigma\, d'_n$, then $\bigsqcup d_n\, \mathcal{P}^\sigma \bigsqcup d'_n$.*

Proof. Straightforward induction on types. The case for α uses the fact that the map g_α is strict. The case for $a(\sigma)$ uses the fact that all the approximating domains D_k and D'_k are finite, so the ω-completeness of the approximating relations is immediate. □

Lemma 5.2.2 *If $d \in D^w_{a(\sigma)}$, $d' \in D^{w'}_{a(\sigma)}$ and $d\, \mathcal{P}^{a(\sigma)}\, d'$ then*

$$\eta_T^{-1}(d)\ (T, T')(\mathcal{P}^{a(\sigma)})\ \eta_{T'}^{-1}(d').$$

Proof. We know that for all $k \in \omega$

$$(\nu_{k+1}^R\, d)\ \mathcal{P}_{k+1}^{a(\sigma)}\ (\nu_{k+1}'^R\, d')$$

which means

$$(\nu_{k+1}^R\, d)\ (T, T')(\mathcal{P}_k^{a(\sigma)})\ (\nu_{k+1}'^R\, d')$$

But since $\nu_{k+1}^R = T(\nu_k^R) \circ \eta_T^{-1}$, and similarly for $\nu_{k+1}'^R$, this means

$$(T(\nu_k^R) \circ \eta_T^{-1})(d)\ (T, T')(\mathcal{P}_k^{a(\sigma)})\ (T'(\nu_k'^R) \circ \eta_{T'}^{-1})(d')$$

so that either $(T(\nu_k^R) \circ \eta_T^{-1})(d)$ and $(T'(\nu_k'^R) \circ \eta_{T'}^{-1})(d')$ are both $\bot$, in which case $\eta_T^{-1}(d)$ and $\eta_{T'}^{-1}(d')$ are too and the result is immediate; or

$$(T(\nu_k^R) \circ \eta_T^{-1})(d) = in_i(x_0, \ldots, x_{m_i - 1})$$

$$(T'(\nu_k'^R) \circ \eta_{T'}^{-1})(d') = in_i(x'_0, \ldots, x'_{m_i - 1})$$

for some i, and for all j, $x_j\, P_j\, x'_j$ where

$$P_j = \begin{cases} \mathcal{P}^{\kappa(\sigma)} & \text{if } \delta_{ij} = \kappa \\ \mathcal{P}_k^{a(\sigma)} & \text{if } \delta_{ij} = a(\xi) \end{cases}$$

And since this holds for all k, we have

$$\eta_T^{-1}(d) = in_i(y_0, \ldots, y_{m_i-1})$$

$$\eta_{T'}^{-1}(d') = in_i(y'_0, \ldots, y'_{m_i-1})$$

where for all j, $y_j \, \mathcal{P}^{\delta_{ij}(\sigma)} \, y'_j$ and we are done. □

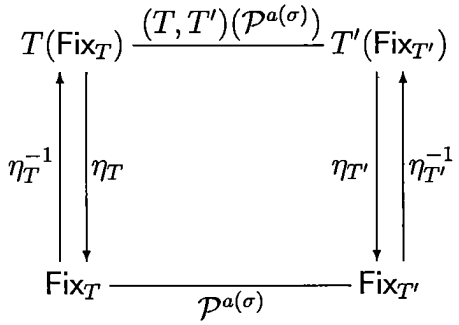
A very similar argument (in the other direction) yields the converse:

Lemma 5.2.3 *If $d \in T(D^w_{a(\sigma)})$, $d' \in T'(D^{w'}_{a(\sigma)})$ and $d \, (T,T')(\mathcal{P}^{a(\sigma)}) \, d'$ then*

$$\eta_T(d) \; \mathcal{P}^{a(\sigma)} \; \eta_{T'}(d').$$

□

The previous two lemmas may be summarised in the following picture:



In fact, $\mathcal{P}^{a(\sigma)}$ is the *unique* strict and ω-complete relation which has this property. Whilst we do not immediately need to know this in order to obtain our parametricity theorem, it will turn out to be useful in interpreting the 'free theorems' which arise from the main result.

Lemma 5.2.4 *If $R \subseteq \mathsf{Fix}_T \times \mathsf{Fix}_{T'}$ is a strict and ω-complete relation such that*

$$d \, R \, d' \quad \Leftrightarrow \quad (\eta_T^{-1}(d)) \, (T,T')(R) \, (\eta_{T'}^{-1}(d'))$$

then $R = \mathcal{P}^{a(\sigma)}$.

Proof. Assume that R satisfies the conditions above. First we show by induction that for all k, if $d \, R \, d'$ then $(\nu_k^R d) \; \mathcal{P}_k^{a(\sigma)} \; (\nu_k'^R d')$. From this we can deduce that $R \subseteq \mathcal{P}^{a(\sigma)}$. The case $k = 0$ is trivial. For the induction step, if $d \, R \, d'$ then we have

$$(\eta_T^{-1}(d)) \, (T,T')(R) \, (\eta_{T'}^{-1}(d'))$$

by assumption. So either $\eta_T^{-1}(d)$ and $\eta_{T'}^{-1}(d')$ are both $\perp$, in which case so are d and d' and the result follows as ν_{k+1}^R and $\nu_{k+1}'^R$ are strict maps and $\mathcal{P}_{k+1}^{a(\sigma)}$ is a strict relation; or

$$\eta_T^{-1}(d) = in_i(d_0, \ldots, d_{m_i-1})$$

$$\eta_{T'}^{-1}(d') = in_i(d'_0, \ldots, d'_{m_i-1})$$

and $\forall j.\, d_j\ P_j\ d'_j$ where

$$P_j = \begin{cases} \mathcal{P}^{\kappa(\sigma)} & \text{if } \delta_{ij} = \kappa \\ R & \text{if } \delta_{ij} = a(\xi) \end{cases}$$

By induction, this means

$$(T(\nu_k^R) \circ \eta_T^{-1})(d) = in_i(e_0, \ldots, e_{m_i-1})$$

$$(T'(\nu_k'^R) \circ \eta_{T'}^{-1})(d') = in_i(e'_0, \ldots, e'_{m_i-1})$$

such that $\forall j.\, e_j\ Q_j\ e'_j$ where

$$Q_j = \begin{cases} \mathcal{P}^{\kappa(\sigma)} & \text{if } \delta_{ij} = \kappa \\ \mathcal{P}_k^{a(\sigma)} & \text{if } \delta_{ij} = a(\xi) \end{cases}$$

So that

$$(T(\nu_k^R) \circ \eta_T^{-1})(d)\ (T, T')\mathcal{P}_k^{a(\sigma)}\ (T'(\nu_k'^R) \circ \eta_{T'}^{-1})(d')$$

which is the same as

$$\nu_{k+1}^R(d)\ \mathcal{P}_{k+1}^{a(\sigma)}\ \nu_{k+1}'^R(d')$$

and we are done.

For the other direction, define $R_k \subseteq D_k \times D'_k$ by

$$d\ R_k\ d' \quad \Leftrightarrow \quad (\nu_k\, d)\ R\ (\nu'_k\, d')$$

We first prove by induction that for all k, if $d\, \mathcal{P}_k^{a(\sigma)}\, d'$ then $d\, R_k\, d'$. The base case is easy, as both ν_0 and ν'_0 are strict maps and R was assumed to be a strict relation. Now assume that $d\, \mathcal{P}_{k+1}^{a(\sigma)}\, d'$. This means that $d\, (T, T')\mathcal{P}_k^{a(\sigma)}\, d'$, so that either both d and d' are $\perp$, in which case the result is immediate; or

$$d = in_i(d_0, \ldots, d_{m_i-1})$$

$$d' = in_i(d'_0, \ldots, d'_{m_i-1})$$

and $\forall j.\, d_j\ P_j\ d'_j$ where

$$P_j = \begin{cases} \mathcal{P}^{\kappa(\sigma)} & \text{if } \delta_{ij} = \kappa \\ \mathcal{P}_k^{a(\sigma)} & \text{if } \delta_{ij} = a(\xi) \end{cases}$$

and by induction this means that $\forall j.\, d_j\ Q_j\ d'j$ where

$$Q_j = \begin{cases} \mathcal{P}^{\kappa(\sigma)} & \text{if } \delta_{ij} = \kappa \\ R_k & \text{if } \delta_{ij} = a(\xi) \end{cases}$$

This means

$$(T(\nu_k)\, d)\ (T, T')R\ (T'(\nu'_k)\, d')$$

and thus, by the assumption on R, that

$$(\eta_T \circ T(\nu_k))(d)\ R\ (\eta_{T'} \circ T'(\nu'_k))(d')$$

which is

$$\nu_{k+1}(d)\ R\ \nu'_{k+1}(d')$$

as required.

Now, by the definition of $\mathcal{P}^{a(\sigma)}$ and the above, we know that if $d\ \mathcal{P}^{a(\sigma)}\ d'$ then $\forall k.\ (\nu_k^R\ d)\ R_k\ (\nu_k'^R\ d')$ and hence that

$$(\nu_k \circ \nu_k^R)(d)\ R\ (\nu'_k \circ \nu_k'^R)(d')$$

But as $d = \bigsqcup_k(\nu_k \circ \nu_k^R)(d)$, $d' = \bigsqcup_k(\nu'_k \circ \nu_k'^R)(d')$, and R was assumed to be ω-complete, we have $d\,R\,d'$. Thus $\mathcal{P}^{a(\sigma)} \subseteq R$ and we are done. □

The previous proofs about relations on algebraic types are all slightly unsatisfactory: they are all much the same, and they all rely on a rather crude and detailed examination of the action of the functors T and T'. It would be more pleasant if things were done making more use of universal properties, rather than explicit calculations with elements. This should be possible by treating the domain constructors as functors over some category of relations. Leaving this aside, however, we are now in a position to prove the main result of this section:

Theorem 5.2.5 *Let w, w' be two worlds, $\{g_\alpha\}$ a family of strict maps which induces the logical relation $\{\mathcal{P}^\sigma\}$ as above, $\Theta \vdash t :: \tau$, S be a substitution, ρ an Sw-environment, ρ' an Sw'-environment with $dom(\rho) = dom(\rho') = \Theta$ such that $\forall x^\sigma \in \Theta.\ \rho(x^\sigma)\ \mathcal{P}^{S\sigma}\ \rho'(x^\sigma)$. Then*

$$(\llbracket t \rrbracket^{Sw}\rho)\ \mathcal{P}^{S\tau}\ (\llbracket t \rrbracket^{Sw'}\rho').$$

Proof. Induction on the structure of t. We sketch a few of the interesting cases:

- If $t = x^\tau$ then $\llbracket t \rrbracket^{Sw}\rho = \rho(x^\tau)$ and $\llbracket t \rrbracket^{Sw'}\rho' = \rho'(x^\tau)$ and we have $\rho(x^\tau)\mathcal{P}^{S\tau}\rho'(x^\tau)$ by assumption.
- If $t = \lambda x^\sigma.u :: \sigma \to \tau$ then if $d \in D_\sigma^{Sw}$, $d' \in D_\sigma^{Sw'}$ and $d\mathcal{P}^{S\sigma}d'$ then by induction

  $$(\llbracket u \rrbracket^{Sw}\rho[x^\sigma \mapsto d])\ \mathcal{P}^{S\tau}\ (\llbracket u \rrbracket^{Sw'}\rho'[x^\sigma \mapsto d'])$$

  so that

  $$(\llbracket t \rrbracket^{Sw}\rho)\ \mathcal{P}^{S\sigma\to S\tau}\ (\llbracket t \rrbracket^{Sw'}\rho')$$

  and as $(S\sigma \to S\tau) = S(\sigma \to \tau)$ we are done.
- If $t = u\,v$ with $v :: \tau$ and $u :: \sigma \to \tau$ then by induction we have

  $$(\llbracket u \rrbracket^{Sw}\rho)\ \mathcal{P}^{S(\sigma\to\tau)}\ (\llbracket u \rrbracket^{Sw'}\rho')$$

  and

  $$(\llbracket v \rrbracket^{Sw}\rho)\ \mathcal{P}^{S\sigma}\ (\llbracket v \rrbracket^{Sw'}\rho')$$

  so that

  $$(\llbracket u \rrbracket^{Sw}\rho)\,(\llbracket v \rrbracket^{Sw}\rho)\ \mathcal{P}^{S\tau}\ (\llbracket u \rrbracket^{Sw'}\rho')\,(\llbracket v \rrbracket^{Sw'}\rho')$$

  as required.

- If $t = \mathsf{let}\ x_1^{S_1\sigma}, \ldots, x_k^{S_k\sigma} = u\ \mathsf{in}\ v$, where $\Theta \vdash u :: \sigma$ and $\Theta, x_1^{S_1\sigma}, \ldots, x_k^{S_k\sigma} \vdash v :: \tau$ then we know that for all i, S_i is the identity on any type variable occuring in Θ so that ρ is an $(S \circ S_i)w$-environment and ρ' is an $(S \circ S_i)w'$-environment and for any $y^\theta \in \Theta$, $\rho(y^\theta)\ \mathcal{P}^{(S\circ S_i)\theta}\ \rho'(y^\theta)$. This means that we can apply the induction hypothesis to u and deduce that

$$([\![u]\!]^{(S\circ S_i)w}\rho)\ \mathcal{P}^{(S\circ S_i)\sigma}\ ([\![u]\!]^{(S\circ S_i)w'}\rho')$$

for each i. Therefore we can apply the induction hypothesis to v to get

$$\begin{array}{l}([\![v]\!]^{Sw}\rho[x_1^{S_1\sigma} \mapsto [\![u]\!]^{S_1(Sw)}\rho, \ldots, x_k^{S_k\sigma} \mapsto [\![u]\!]^{S_k(Sw)}\rho])\ \mathcal{P}^{S\tau} \\ \quad ([\![v]\!]^{Sw'}\rho'[x_1^{S_1\sigma} \mapsto [\![u]\!]^{S_1(Sw')}\rho', \ldots, x_k^{S_k\sigma} \mapsto [\![u]\!]^{S_k(Sw')}\rho'])\end{array}$$

as required.

- If $t = C_i^\sigma(u_0, \ldots, u_{m_i-1}) :: a(\sigma)$ then for all j, $u_j :: \delta_{ij}(\sigma)$ and by induction we have

$$([\![u_j]\!]^{Sw}\rho)\ \mathcal{P}^{S(\delta_{ij}(\sigma))}\ ([\![u_j]\!]^{Sw'}\rho')$$

Thus, if T, T' are the functors associated with the construction of $D_{a(\sigma)}^{Sw}$ and $D_{a(\sigma)}^{Sw'}$ respectively,

$$\begin{array}{l}(in_i([\![u_0]\!]^{Sw}\rho, \ldots, [\![u_{m_i-1}]\!]^{Sw}\rho))\ (T, T')(\mathcal{P}^{S(a(\sigma))}) \\ \quad (in_i([\![u_0]\!]^{Sw'}\rho', \ldots, [\![u_{m_i-1}]\!]^{Sw'}\rho'))\end{array}$$

and thus by Lemma 5.2.3

$$\begin{array}{l}(\eta_T(in_i([\![u_0]\!]^{Sw}\rho, \ldots, [\![u_{m_i-1}]\!]^{Sw}\rho)))\ \mathcal{P}^{S(a(\sigma))} \\ \quad (\eta_{T'}(in_i([\![u_0]\!]^{Sw'}\rho', \ldots, [\![u_{m_i-1}]\!]^{Sw'}\rho')))\end{array}$$

and we are done.

- If

$$\begin{array}{rcl} t = \mathsf{acase}^\sigma\ u\ \ \mathsf{of} & C_1(x_{1,0}^{\delta_{1,0}(\sigma)}, \ldots, x_{1,m_1-1}^{\delta_{1,m_1-1}(\sigma)}) & \Rightarrow v_1 \\ \vdots & & \\ \mid & C_n(x_{n,0}^{\delta_{n,0}(\sigma)}, \ldots, x_{n,m_n-1}^{\delta_{n,m_n-1}(\sigma)}) & \Rightarrow v_n \end{array}$$

where $\forall i.\ \Theta, x_{i,0}^{\delta_{i,0}(\sigma)}, \ldots, x_{i,m_i-1}^{\delta_{i,m_i-1}(\sigma)} \vdash v_i :: \tau$ then by induction

$$([\![u]\!]^{Sw}\rho)\ \mathcal{P}^{S(a(\sigma))}\ ([\![u]\!]^{Sw'}\rho')$$

so that, by Lemma 5.2.2,

$$(\eta_T^{-1}([\![u]\!]^{Sw}\rho))\ (T, T')(\mathcal{P}^{S(a(\sigma))})\ (\eta_{T'}^{-1}([\![u]\!]^{Sw'}\rho'))$$

This means that either

$$\eta_T^{-1}([\![u]\!]^{Sw}\rho) = \bot \quad \text{and} \quad \eta_{T'}^{-1}([\![u]\!]^{Sw'}\rho') = \bot$$

in which case $[\![t]\!]^{Sw}\rho = \bot$ and $[\![t]\!]^{Sw'}\rho' = \bot$ so that the result follows as $\mathcal{P}^{S\tau}$ is strict; or

$$\eta_T^{-1}([\![u]\!]^{Sw}\rho) = in_i(d_0, \ldots, d_{m_i-1})$$
$$\eta_{T'}^{-1}([\![u]\!]^{Sw'}\rho') = in_i(d'_0, \ldots, d'_{m_i-1})$$

where $\forall j,\ d_j \in D^{Sw}_{\delta_{ij}(\sigma)}$ and $d'_j \in D^{Sw'}_{\delta_{ij}(\sigma)}$ satisfy $d_j\ \mathcal{P}^{S(\delta_{ij}(\sigma))}\ d'_j$. In this case we can apply the induction hypothesis to v_i to deduce

$$([\![v_i]\!]^{Sw}\rho[x_{1,0}^{\delta_{1,0}(\sigma)} \mapsto d_0, \ldots, x_{1,m_1-1}^{\delta_{1,m_1-1}(\sigma)} \mapsto d_{m_1-1}])\ \mathcal{P}^{S\tau}$$
$$([\![v_i]\!]^{Sw'}\rho'[x_{1,0}^{\delta_{1,0}(\sigma)} \mapsto d'_0, \ldots, x_{1,m_1-1}^{\delta_{1,m_1-1}(\sigma)} \mapsto d'_{m_1-1}])$$

since the two extended environments satisfy the conditions of the theorem, and we are done.

- If $t = \mathsf{fix}(x^\sigma.u)$ then $[\![t]\!]^{Sw}\rho = \bigsqcup d_n$ where $d_0 = \bot_{D^{Sw}_\sigma}$ and $d_{n+1} = [\![u]\!]^{Sw}\rho[x^\sigma \mapsto d_n]$. Similarly, $[\![t]\!]^{Sw'}\rho' = \bigsqcup d'_n$ where $d'_0 = \bot_{D^{Sw'}_\sigma}$ and $d_{n+1} = [\![u]\!]^{Sw'}\rho'[x^\sigma \mapsto d'_n]$. Since $\mathcal{P}^{S\sigma}$ is strict, this means that $d_0\ \mathcal{P}^{S\sigma}\ d'_0$. If $d_n\ \mathcal{P}^{S\sigma}\ d'_n$ then the induction hypothesis applied to u tells us that

$$([\![u]\!]^{Sw}\rho[x^\sigma \mapsto d_n])\ \mathcal{P}^{S\sigma}\ ([\![u]\!]^{Sw'}\rho'[x^\sigma \mapsto d'_n])$$

and hence that $d_n\ \mathcal{P}^{S\sigma}\ d'_n$ for all n by mathematical induction. Then, as $\mathcal{P}^{S\sigma}$ is ω-complete, we have

$$(\bigsqcup d_n)\ \mathcal{P}^{S\sigma}\ (\bigsqcup d'_n)$$

as required.

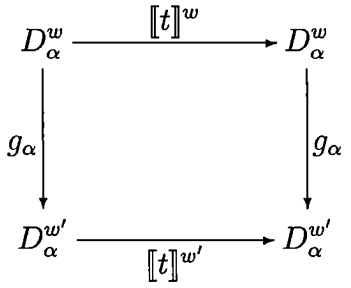
□

Example If $\emptyset \vdash t : \alpha$ then by taking $w' = w$, $g_\alpha : w(\alpha) \to w(\alpha)$ to be the constant $\bot$ function (which is certainly strict) and S to be the identity, we have that

$$g([\![t]\!]^w) = [\![t]\!]^w$$

which means that $[\![t]\!]^w = \bot$ for any world w. If we informally add the implicit quantification over type variables which do not appear in the context, any term with type $\forall\alpha.\,\alpha$ is denotationally equal to $\bot$.

Example If $\emptyset \vdash t : \alpha \to \alpha$ then for any worlds w, w' and strict $g_\alpha : w(\alpha) \to w'(\alpha)$, the following commutes:



By taking $w = w'$ and g_α to be the constant $\bot$ function, this implies that $[\![t]\!]^w$ must be strict for all w. Thus if we pick w such that $w(\alpha) = \mathcal{O}$ (the two-point domain), then $[\![t]\!]^w$ must be either the constant $\bot$ function, or the identity. If it is the constant $\bot$ function, then so is $[\![t]\!]^{w'}$ for all worlds w', as can be verified by taking the map $g : w'(\alpha) \to \mathcal{O}$ to map $\bot$ to $\bot$ and everything else to $\top$. If, on the other hand, $[\![t]\!]^w$ is the identity then so is $[\![t]\!]^{w'}$ for all w'. This follows by contradiction: if $[\![t]\!]^{w'}$ is not the identity, then there exists a $d \in w'(\alpha)$ with $d \neq \bot$ and $[\![t]\!]^{w'}(d) \neq d$. Then define $g : \mathcal{O} \to w'(\alpha)$ by $g(\bot) = \bot$ and $g(\top) = d$, and the appropriate version of the square above fails to commute. Hence there does not exist such a d.

So any term of type $\forall\alpha.\, \alpha \to \alpha$ is uniformly either the identity or the constant $\bot$ function.

Example In $\Lambda_{P,list}$, if $\emptyset \vdash t :: list(\alpha) \to list(\alpha)$ then given worlds w,w' and a strict $g_\alpha : w(\alpha) \to w'(\alpha)$ then we have, writing f and f' for $[\![t]\!]^w$ and $[\![t]\!]^{w'}$ respectively, that

$$\forall l \in D^w_{list(\alpha)}, l' \in D^{w'}_{list(\alpha)}.\ l\, \mathcal{P}^{list(\alpha)}\, l' \ \Rightarrow\ (f\, l)\, \mathcal{P}^{list(\alpha)}\, (f'\, l')$$

But what does $\mathcal{P}^{list(\alpha)}$ really mean? Intuitively, we would expect that two lists are related if and only if they are the same length and their elements are pointwise related, and this is indeed the case. One way to make this more formal is to define the strict function $map\,(g_\alpha) : D^w_{list(\alpha)} \to D^{w'}_{list(\alpha)}$ much as we defined the function which maps the abstraction function down a list of elements of $\mathbb{N}_\bot$ in Chapter 4. It is then easy to check that the relation $R \subseteq D^w_{list(\alpha)} \times D^{w'}_{list(\alpha)}$ which is defined by

$$l\, R\, l' \ \Leftrightarrow\ map\,(g_\alpha)(l) = l'$$

is strict, ω-complete and satisfies

$$l\, R\, l' \ \Leftrightarrow\ \eta_T^{-1}(l)\ (T,T')(R)\ \eta_{T'}^{-1}(l')$$

and thus by Lemma 5.2.4, $\mathcal{P}^{list(\alpha)} = R$. This means that we can rephrase our free theorem as

$$\forall l \in D^w_{list(\alpha)}, l' \in D^{w'}_{list(\alpha)}.\ map\,(g_\alpha)(l) = l' \ \Rightarrow\ map\,(g_\alpha)(f(l)) = f'(l')$$

or

$$\begin{array}{ccc} D^w_{list(\alpha)} & \xrightarrow{\ f\ } & D^w_{list(\alpha)} \\ \Big\downarrow{\scriptstyle map\,(g_\alpha)} & & \Big\downarrow{\scriptstyle map\,(g_\alpha)} \\ D^{w'}_{list(\alpha)} & \xrightarrow[\ f'\ ]{} & D^{w'}_{list(\alpha)} \end{array}$$

which was the diagram we gave in the introduction. This argument could be tidied up somewhat, as we have not directly used the fact that the construction of $D^w_{list(\alpha)}$ is functorial in D^w_α. Were we to do so, the morphism $map\,(g_\alpha)$ could be introduced in a much less *ad hoc* way than it is here.

Remark 5.2.6 The restriction that all the maps g_α be strict gives us another application for strictness analysis. If we wished to use parametricity results in proving or transforming programs then a strictness analyser would be an important part of any automated proof system or proof assistant.

5.3 Polymorphic Invariance

This section considers the relationship between the derivable strictness properties of different instances of terms with polymorphic types. We use a technique similar to that of the previous section to prove a polymorphic invariance theorem for BHA abstract interpretation, which we then carry over to the conjunctive strictness logic.

It should be noted that there are really two notions of polymorphic invariance—we can either show that strictness *properties* are polymorphic invariants (*e.g.* if one instance of a function is strict, all instances are), or we can show that our *analysis method* detects a property at all types if it detects it at one. It is the second form which we consider here, and which Abramsky's work addresses [Abr85]. He observes that this is the more important notion from the point of view of an implementation, as it is this which can be used to ensure that an analysis does not lose information by analysing only simple instances.

In fact, the situation is not as simple as the above might suggest. For example, the identity function specialised to arguments of type $\iota \to \iota$ has the property of taking strict functions to strict functions, but this cannot be a property of the version of the identity which is specialised to arguments of type ι. What we actually show is that a strictness property holds at a particular instance of a term iff it holds at all instances for which that property makes sense (in a sense to be made precise).

The invariance theorem which we shall prove was originally obtained for the conjunctive strictness logic by purely syntactic (proof theoretic) means (a sketch of the proof may be found in [Ben92b]). The details of this proof are, however, extremely fiddly. It is much easier to prove in the context of BHA abstract interpretation; although the statement of the result is slightly clearer in terms of the logic. The result is proved for Λ_T extended with type variables—we do not consider the let construct (or algebraic datatypes). This is partly for simplicity, but also because our result implies that there is more than one way in which one could choose to treat let in an analysis (though one could take the view that the formal system should be independent of this, and that the use of the invariance theorem is a matter of implementation).

We now extend the definition of Λ_T types to include type variables, which are ranged over with α, β:

$$\sigma ::= \alpha \mid \iota \mid \sigma \to \sigma \mid \sigma \times \sigma$$

Terms are formed exactly as before (Figure 2.1)—we merely allow the type superscripts of variables to include type variables.

As in the preceding section, define a *substitution* to be a function from type variables to types, and extend this to a function from types to types. To make the translation

of results between the abstract interpretation and the logic easier, we shall also need to define the action of substitutions on terms. If $t :: \sigma$ is a term and S is a substitution, then $S(t) :: S(\sigma)$ is defined in a fairly obvious way, by defining

$$S(x^\tau) = x^{S\tau} \qquad S(\lambda x^\tau.u) = \lambda x^{S\tau}.S(u) \qquad S(u\,v) = (S(u))\,(S(v))$$

and so on. If this definition is understood naively, it can introduce unwanted aliasing between variables. For example, if $S : \alpha \mapsto \iota$ then the action of S on the term $\lambda x^\alpha.\lambda x^\iota.x^\alpha$ completely changes its meaning. We shall gloss over this by assuming that appropriate renaming is done to prevent any such clashes. If $S = S'' \circ S'$ then we shall say that $S'(t)$ is a *simpler* instance of the term t than $S(t)$, or that $S(t)$ is a *more complicated* instance than $S'(t)$.

Rather than parameterise the abstract interpretation on a choice of world, we shall give a fixed interpretation of type variables, and relate the denotations of t and $S(t)$, for a substitution S. The BHA abstract interpretation of Section 2.2.2 is therefore extended to the new version of the language by defining $A_\alpha = \mathbf{2}$ for any type variable α. The semantic equations of Figure 2.5 remain unchanged.

Define the maps $\epsilon_\sigma : A_\sigma \to A_{S\sigma}$ and $\kappa_\sigma : A_{S\sigma} \to A_\sigma$ inductively as follows:

$$\epsilon_\iota = id_{A_\iota} \qquad \kappa_\iota = id_{A_\iota}$$

$$\epsilon_\alpha(\top_{A_\alpha}) = \top_{A_{S\alpha}} \qquad \epsilon_\alpha(\bot_{A_\alpha}) = \bot_{A_{S\alpha}}$$

$$\kappa_\alpha(\bot_{A_{S\alpha}}) = \bot_{A_\alpha} \qquad \kappa_\alpha(x) = \top_{A_\alpha}\ (x \neq \bot)$$

$$\epsilon_{\sigma\times\tau} = \epsilon_\sigma \times \epsilon_\tau \qquad \kappa_{\sigma\times\tau} = \kappa_\sigma \times \kappa_\tau$$

$$\epsilon_{\sigma\to\tau}(f) = \epsilon_\tau \circ f \circ \kappa_\sigma \qquad \kappa_{\sigma\to\tau}(g) = \kappa_\tau \circ g \circ \epsilon_\sigma$$

Proposition 5.3.1 *For all σ, the maps ϵ_σ and κ_σ are strict and monotone (and thus continuous). Moreover, they satisfy*

$$\kappa_\sigma \circ \epsilon_\sigma = id_{A_\sigma}$$

$$\epsilon_\sigma \circ \kappa_\sigma \sqsupseteq id_{A_{S\sigma}}$$

Proof. Induction on types. □

This says κ_σ is the left adjoint to ϵ_σ. Hence, by the adjoint functor theorem, ϵ_σ preserves meets and κ_σ preserves joins.

Now define the logical relation $\{\mathcal{S}^\sigma\}$, where $\mathcal{S}^\sigma \subseteq A_\sigma \times A_{S\sigma}$ by

$$d\,\mathcal{S}^\iota\,e \;\Leftrightarrow\; d = e$$

$$d\,\mathcal{S}^\alpha\,e \;\Leftrightarrow\; d = \kappa_\alpha(e)$$

$$f\,\mathcal{S}^{\sigma\to\tau}\,g \;\Leftrightarrow\; \forall d, e.\, d\,\mathcal{S}^\sigma\,e \Rightarrow (f\,d)\,\mathcal{S}^\tau\,(g\,e)$$

$$(d, d')\,\mathcal{S}^{\sigma\times\tau}\,(e, e') \;\Leftrightarrow\; d\,\mathcal{S}^\sigma\,e \text{ and } d'\,\mathcal{S}^\tau\,e'$$

Proposition 5.3.2 *For all σ, $\mathcal{S}^\sigma$ satisfies the following:*

(a) *For any $d \in A_\sigma$, $d\,\mathcal{S}^\sigma\,\epsilon_\sigma(d)$;*

(b) *For any $d \in A_\sigma$ and $e \in A_{S\sigma}$, if $d\,\mathcal{S}^\sigma\,e$ then $d = \kappa_\sigma(e)$.*

Proof. The two parts are proved simultaneously by induction on types. The base cases are immediate from the definitions. The case for products is straightforward. For function spaces, we have:

(a) We wish to show $f\,\mathcal{S}^{\sigma\to\tau}\,\epsilon_{\sigma\to\tau}(f)$. By induction (b), if $d\,\mathcal{S}^\sigma\,e$ then $d = \kappa_\sigma(e)$. Thus

$$\epsilon_{\sigma\to\tau}(f)(e) = (\epsilon_\tau \circ f \circ \kappa_\sigma)(e) = \epsilon_\tau(f(d))$$

and by induction (a), we have $(f\,d)\,\mathcal{S}^\tau\,\epsilon_\tau(f\,d)$ so that

$$(f\,d)\,\mathcal{S}^\tau\,\epsilon_{\sigma\to\tau}(f)(e)$$

and we are done.

(b) Assume $f\,\mathcal{S}^{\sigma\to\tau}\,g$. We know by induction (a) that for all $d \in A_\sigma$, $d\,\mathcal{S}^\sigma\,\epsilon_\sigma(d)$. Thus

$$(f\,d)\,\mathcal{S}^\tau\,g(\epsilon_\sigma(d)).$$

This means

$$\begin{aligned}(f\,d) &= (\kappa_\tau \circ g \circ \epsilon_\sigma)(d) \quad \text{by induction (b)}\\ &= \kappa_{\sigma\to\tau}(g)(d) \quad \text{by definition}\end{aligned}$$

as required.

□

Note that part (a) of this result, together with the fact that ϵ_σ is strict, implies that for all σ, $\mathcal{S}^\sigma$ is a strict relation. We will also need to know that it respects joins:

Lemma 5.3.3 *For all σ, if $d\,\mathcal{S}^\sigma\,e$ and $d'\,\mathcal{S}^\sigma\,e'$ then $(d \sqcup d')\,\mathcal{S}^\sigma\,(e \sqcup e')$.*

Proof. This is a simple induction on types. We give the case for function spaces: Assume $f\,\mathcal{S}^{\sigma\to\tau}\,g$ and $f'\,\mathcal{S}^{\sigma\to\tau}\,g'$. Then if $d\,\mathcal{S}^\sigma\,e$ we have

$$(f\,d)\,\mathcal{S}^\tau\,(g\,e) \quad \text{and} \quad ((f'\,d)\,\mathcal{S}^\tau\,(g'\,e)$$

so that

$$((f\,d) \sqcup (f'\,d))\;\mathcal{S}^\tau\;((g\,e) \sqcup (g'\,e))$$

by induction. But this is

$$((f \sqcup f')(d))\;\mathcal{S}^\tau\;((g \sqcup g')(e))$$

as joins in the function space are inherited from those of the codomain. □

Theorem 5.3.4 *If $t :: \tau$ and ρ_1^A,ρ_2^A are abstract environments such that $FV(t) \subseteq dom(\rho_1^A)$ and for all $x^\sigma \in dom(\rho_1^A)$, $x^{S\sigma} \in dom(\rho_2^A)$ and $\rho_1^A(x^\sigma)\ \mathcal{S}^\sigma\ \rho_2^A(x^{S\sigma})$ then*

$$[\![t]\!]^A\rho_1^A\ \mathcal{S}^\tau\ [\![S(t)]\!]^A\rho_2^A.$$

Proof. This is an induction on the structure of t, much like our previous logical relations proofs:

- If $t = \underline{n}$ then $[\![t]\!]^A\rho_1^A = [\![S(t)]\!]^A\rho_2^A = \top_{A_\iota}$. So we are done by the definition of $\mathcal{S}^\iota$.
- If $t = x^\sigma$ then there is nothing to prove by the assumptions on ρ_1^A and ρ_2^A.
- If $t = u\,v$ where $u :: \sigma \to \tau$ and $v :: \sigma$, then by induction

  $$[\![u]\!]^A\rho_1^A\ \mathcal{S}^{\sigma\to\tau}\ [\![S(u)]\!]^A\rho_2^A$$

  $$[\![v]\!]^A\rho_1^A\ \mathcal{S}^\sigma\ [\![S(v)]\!]^A\rho_2^A$$

  So that

  $$(([\![u]\!]^A\rho_1^A)\,([\![v]\!]^A\rho_1^A))\ \mathcal{S}^\tau\ (([\![S(u)]\!]^A\rho_2^A)\,([\![S(v)]\!]^A\rho_2^A))$$

  which is

  $$[\![u\,v]\!]^A\rho_1^A\ \mathcal{S}^\tau\ [\![S(u\,v)]\!]^A\rho_2^A$$

  as required.
- If $t = \lambda x^\sigma.u$ where $u :: \tau$ then by induction, if $d\ \mathcal{S}^\sigma\ e$ then

  $$([\![u]\!]^A\rho_1^A[x^\sigma \mapsto d])\ \mathcal{S}^\tau\ ([\![S(u)]\!]^A\rho_2^A[x^{S\sigma} \mapsto e])$$

  This means

  $$([\![\lambda x^\sigma.u]\!]^A\rho_1^A)(d)\ \mathcal{S}^\tau\ ([\![S(\lambda x^\sigma.u)]\!]^A\rho_2^A)(e)$$

  so we are done.
- If $t = \mathsf{if}\ s\ \mathsf{then}\ t_1\ \mathsf{else}\ t_2$ where $t_1, t_2 :: \tau$ and $s :: \iota$ then by induction and the definition of $\mathcal{S}^\iota$ we have $[\![s]\!]^A\rho_1^A = [\![S(s)]\!]^A\rho_2^A$. There are then two subcases to consider. If $[\![s]\!]^A\rho_1^A = \bot_{A_\iota}$ then

  $$[\![t]\!]^A\rho_1^A = \bot_{A_\tau} \quad \text{and} \quad [\![S(t)]\!]^A\rho_2^A = \bot_{A_{S\tau}}.$$

  Then as $\mathcal{S}^\tau$ is strict, we are done. If $[\![s]\!]^A\rho_1^A = \top_{A_\iota}$ then

  $$[\![t]\!]^A\rho_1^A = ([\![t_1]\!]^A\rho_1^A) \sqcup ([\![t_2]\!]^A\rho_1^A)$$

  $$[\![S(t)]\!]^A\rho_2^A = ([\![S(t_1)]\!]^A\rho_2^A) \sqcup ([\![t_2]\!]^A\rho_2^A)$$

  so that by the induction hypothesis applied to t_1 and t_2 and Lemma 5.3.3 we have the result.

- If $t = u_1 + u_2$ where $u_1, u_2 :: \iota$, then for $i \in \{1, 2\}$

$$[\![u_i]\!]^A \rho_1^A = [\![S(u_i)]\!]^A \rho_2^A$$

and the result follows immediately.

- If $t = \mathsf{fix}(x^\sigma.u)$ where $u :: \sigma$ then

$$[\![t]\!]^A \rho_1^A = \bigsqcup_{k \in \omega} d_k \quad \text{where } d_0 = \bot_{A_\sigma},\ d_{k+1} = [\![t]\!]^A \rho_1^A[x^\sigma \mapsto d_k]$$

$$[\![S(t)]\!]^A \rho_2^A = \bigsqcup_{k \in \omega} e_k \quad \text{where } e_0 = \bot_{A_{S\sigma}},\ e_{k+1} = [\![S(t)]\!]^A \rho_2^A[x^{S\sigma} \mapsto e_k]$$

Then as $\mathcal{S}^\sigma$ is strict, $d_0\ \mathcal{S}^\sigma\ e_0$. By induction applied to u, if $d_k\ \mathcal{S}^\sigma\ e_k$ then $d_{k+1}\ \mathcal{S}^\sigma\ e_{k+1}$. Thus for all k we have $d_k\ \mathcal{S}^\sigma\ e_k$. Now, as A_σ and $A_{S\sigma}$ are both finite, $\exists m.\ \bigsqcup_k d_k = d_m$ and $\exists m'.\ \bigsqcup_k e_k = e_{m'}$. So if $n = \max(m, m')$

$$[\![t]\!]^A \rho_1^A = d_n \quad \text{and} \quad [\![S(t)]\!]^A \rho_2^A = e_n$$

And as $d_n\ \mathcal{S}^\sigma\ e_n$ we are done.

- The cases for pairing and projections are similar.

□

Corollary 5.3.5 (Polymorphic Invariance for Abstract Interpretation) *If $t :: \tau$ and ρ_1^A, ρ_2^A are abstract environments such that $FV(t) \subseteq dom(\rho_1^A)$ and for all $x^\sigma \in dom(\rho_1^A)$, $x^{S\sigma} \in dom(\rho_2^A)$ and $\rho_2^A(x^{S\sigma}) = \epsilon_\sigma(\rho_1^A(x^\sigma))$ then for any $e \in A_\tau$*

$$([\![S(t)]\!]^A \rho_2^A) \sqsubseteq \epsilon_\tau(e) \ \Leftrightarrow\ ([\![t]\!]^A \rho_1^A) \sqsubseteq e$$

Proof. By Lemma 5.3.2(a), the two environments satisfy the conditions of Theorem 5.3.4 so that

$$([\![t]\!]^A \rho_1^A)\ \mathcal{S}^\tau\ ([\![S(t)]\!]^A \rho_2^A)$$

So by Lemma 5.3.2(b),

$$([\![t]\!]^A \rho_1^A) \ = \ \kappa_\tau([\![S(t)]\!]^A \rho_2^A)$$

and the result follows by Proposition 5.3.1. □

It is instructive to see what this means in terms of the conjunctive strictness logic. The logic is extended to Λ_T with type variables by defining $\mathcal{L}_\alpha$ to be the theory which just contains $\mathbf{t}^\alpha$ and $\mathbf{f}^\alpha$ as basic propositions, so $\mathcal{L}_\alpha \cong \mathcal{L}_\iota$. The program logic remains unchanged. We omit the easy verification that the correspondence between the logic and the abstract interpretation which was shown in Section 3.4 extends to a correspondence between this new version of the logic and the new abstract interpretation.

We now wish to define a function which will embed the propositional theory associated with a type σ into the theory of a more complicated instance of σ. The intuitive idea is that any property at type σ can be regarded as making sense at type $S\sigma$, but not the other way around. For example, for a term t of type $\alpha \to \alpha$, the property of being strict (satisfying $\mathbf{f} \to \mathbf{f}$) makes sense for all instances of t, whereas the property of mapping strict functions to strict functions only makes sense for instances which have types of the form $(\sigma \to \sigma) \to (\sigma \to \sigma)$.

Define $\mathcal{E}_\sigma : \mathcal{L}_\sigma \to \mathcal{L}_{S\sigma}$ as follows:

$$\mathcal{E}_\sigma(\mathbf{t}^\sigma) = \mathbf{t}^{S\sigma} \qquad \mathcal{E}_\sigma(\mathbf{f}^\sigma) = \mathbf{f}^{S\sigma}$$

$$\mathcal{E}_\sigma(\phi \wedge \psi) = \mathcal{E}_\sigma(\phi) \wedge \mathcal{E}_\sigma(\psi)$$

$$\mathcal{E}_{\sigma\to\tau}(\phi^\sigma \to \psi^\tau) = \mathcal{E}_\sigma(\phi^\sigma) \to \mathcal{E}_\tau(\psi^\tau)$$

$$\mathcal{E}_{\sigma\times\tau}(\phi^\sigma \times \psi^\tau) = \mathcal{E}_\sigma(\phi^\sigma) \times \mathcal{E}_\tau(\psi^\tau)$$

If we wish to emphasize which substitution S is intended, then we shall write $\mathcal{E}^S_\sigma$ for the map defined above.

It is easy to see that if $\phi^\sigma \leq \psi^\sigma$ then $\mathcal{E}_\sigma(\phi^\sigma) \leq \mathcal{E}_\sigma(\psi^\sigma)$. This means that $\mathcal{E}_\sigma$ induces $\bar{\mathcal{E}}_\sigma : \mathcal{LA}_\sigma \to \mathcal{LA}_{S\sigma}$. Now recall from Section 3.4 that there is an isomorphism $\bar{p}_\sigma : \mathcal{LA}_\sigma \to A_\sigma$ with inverse q_σ. A simple calculation, which we omit, gives the following:

Lemma 5.3.6 *For all σ, the following commutes:*

$$\begin{array}{ccc} \mathcal{LA}_\sigma & \xrightarrow{\bar{\mathcal{E}}_\sigma} & \mathcal{LA}_{S\sigma} \\ \downarrow{\scriptstyle \bar{p}_\sigma} & & \downarrow{\scriptstyle \bar{p}_{S\sigma}} \\ A_\sigma & \xrightarrow[\epsilon_\sigma]{} & A_{S\sigma} \end{array}$$

□

This means that ϵ_σ is essentially the same as $\bar{\mathcal{E}}_\sigma$, but defined in terms of the abstract interpretation, rather than the logic. Corollary 5.3.5 can therefore be restated in terms of the logic. If $\Gamma = \{x_i^{\sigma_i} : \phi_i^{\sigma_i}\}$ is a strictness context, then define $S\Gamma = \{x_i^{S(\sigma_i)} : \mathcal{E}_{\sigma_i}(\phi_i^{\sigma_i})\}$.

Proposition 5.3.7 (Polymorphic Invariance for Strictness Logic) *For any Γ and $t :: \tau$*

$$\Gamma \vdash t : \phi^\tau \;\Leftrightarrow\; S\Gamma \vdash S(t) : \mathcal{E}_\tau(\phi^\tau).$$

Proof. Recall from Section 3.4 that, given a strictness context $\Gamma = \{x_i^{\sigma_i} : \phi_i^{\sigma_i}\}$, the abstract environment $p\Gamma$ was defined by $(p\Gamma)(x_i^{\sigma_i}) = \bar{p}_{\sigma_i}(\phi_i^{\sigma_i})$. By Lemma 5.3.6, for any Γ, the abstract environments $p\Gamma$ and $p(S\Gamma)$ satisfy the conditions of Corollary 5.3.5:

$$\begin{aligned}\epsilon_{\sigma_i}((p\Gamma)(x_i^{\sigma_i})) &= \epsilon_{\sigma_i}(\bar{p}_{\sigma_i}(\phi_i^{\sigma_i})) \\ &= \bar{p}_{S\sigma}(\mathcal{E}_\sigma(\phi_i^{\sigma_i})) \\ &= (p(S\Gamma))(x^{S\sigma})\end{aligned}$$

So that

$$\begin{aligned}\Gamma \vdash t : \phi^\tau &\Leftrightarrow [\![t]\!]^A(p\Gamma) \sqsubseteq \bar{p}_\tau(\phi^\tau) \text{ by Propositions 3.4.7 and 3.4.8} \\ &\Leftrightarrow [\![S(t)]\!]^A(p(S\Gamma)) \sqsubseteq \epsilon_\tau(\bar{p}_\tau(\phi^\tau)) \text{ by Corollary 5.3.5} \\ &\Leftrightarrow [\![S(t)]\!]^A(p(S\Gamma)) \sqsubseteq \bar{p}_{S\tau}(\mathcal{E}_\tau(\phi_\tau)) \text{ by Lemma 5.3.6} \\ &\Leftrightarrow S\Gamma \vdash S(t) : \mathcal{E}_\tau(\phi^\tau) \text{ by Propositions 3.4.7 and 3.4.8}\end{aligned}$$

□

The left-to-right implication of Proposition 5.3.7 can be seen as a soundness result for the logic extended with type variables. It says that anything we can derive about a term containing some type variables can be derived at all its (monomorphic) instances and, by the earlier soundness result, is therefore semantically valid at all those instances.

It is the right-to-left implication, however, which is particularly interesting. This says that if we can show that a complicated instance of a term has a property which makes sense at a simpler instance, then we can derive that property at the simpler instance. To put it another way, if we wish to know if a complicated instance $S(t) :: S\tau$ of a term satisfies a particular property $\phi^{S\tau}$, it suffices to analyse the simplest instance $S'(t)$ of t for which ϕ makes sense, that is for which there exists a $\psi^{S'\tau}$ such that $\mathcal{E}^{S''}_{S'\tau}(\psi^{S'\tau}) = \phi^{S\tau}$ where S'' is the substitution such that $S = S'' \circ S'$.

Pleasant though this result is, it is not the strongest result we could have hoped for. One might imagine that the only properties which we can ever show a complicated instance of a polymorphic term to have are one which make sense at the simplest instance. More formally, this would mean:

$$\Gamma \vdash S(t) : \phi^{S\tau} \quad \Rightarrow \quad \exists \Gamma', \psi^\tau \text{ st. } \Gamma \leq S\Gamma', \mathcal{E}_\tau(\psi^\tau) \leq \phi^{S\tau} \text{ and } \Gamma' \vdash t : \psi^\tau$$

But this is clearly untrue, as our original example of the polymorphic identity function shows. At type $\alpha \to \alpha$, the identity satisfies the properties of being strict ($\mathbf{f}^\alpha \to \mathbf{f}^\alpha$) and of being a function ($\mathbf{t}^\alpha \to \mathbf{t}^\alpha$). At type $(\beta \to \beta) \to (\beta \to \beta)$, it satisfies the additional property of taking strict functions to strict functions and it is easy to see that neither of the first two properties entails this.

A slightly more plausible hope is that there is some more sophisticated way in which knowing all the (derivable) strictness properties of the simplest instance of a function would enable us to deduce all the (derivable) properties of any instance.

By understanding polymorphic functions as natural transformations, Hughes has shown [Hug88] (in terms of abstract interpretation) that this is indeed the case for first-order functions. For higher-order functions, however, it is unfortunately false. Consider the following two functions of type $(\alpha \to \alpha) \to (\alpha \to \alpha)$:

$$\mathsf{once} \stackrel{\text{def}}{=} \lambda f^{\alpha\to\alpha}.\lambda x^{\alpha}.f^{\alpha\to\alpha}\, x^{\alpha}$$

$$\mathsf{twice} \stackrel{\text{def}}{=} \lambda f^{\alpha\to\alpha}.\lambda x^{\alpha}.f^{\alpha\to\alpha}\,(f^{\alpha\to\alpha}\, x^{\alpha})$$

At type $(\alpha \to \alpha) \to (\alpha \to \alpha)$ (or, equivalently, $(\iota \to \iota) \to (\iota \to \iota)$), these two functions have exactly the same derivable strictness properties. This is not the case at all instances. At type

$$((\beta \to \beta) \to (\beta \to \beta)) \to ((\beta \to \beta) \to (\beta \to \beta))$$

the function twice can be shown to satisfy

$$\begin{array}{l}(((\mathbf{t}^{\beta} \to \mathbf{t}^{\beta}) \to (\mathbf{f}^{\beta} \to \mathbf{f}^{\beta})) \wedge ((\mathbf{f}^{\beta} \to \mathbf{f}^{\beta}) \to (\mathbf{t}^{\beta} \to \mathbf{f}^{\beta})))\\ \quad \to ((\mathbf{t}^{\beta} \to \mathbf{t}^{\beta}) \to (\mathbf{t}^{\beta} \to \mathbf{f}^{\beta}))\end{array}$$

whereas once cannot. Thus there is no way of deriving all the strictness properties of more complicated instances from those of the simplest (or else we would be able to deduce the above property of once, which would be unsound).

So to get the best information we *do* have to analyse some more complicated instances of polymorphic functions. Even if we were only directly interested in simple strictness, and we have shown that we can get this by analysing at the simplest instance, to do that analysis for a function f, we may have to analyse a higher order instance of some function g, which is called by f. What our result tells us is that we only have to analyse the simplest instance of g for which the property we are looking for makes sense, not the instance at which g is actually called. This is a considerable improvement over the approach of [Abr85], which suggests completely reanalysing each instance of g.

Unfortunately, when analysing a term, we may not always know exactly what property we are looking for. This is because of the [app] rule, which has a formula in the premisses which does not appear in the conclusion. This means that some kind of proof search is necessary, and it is far from clear how we should use our polymorphic invariance result to make this more efficient. The most simple-minded approach is to derive only those properties which make sense at all instances. Whilst this will lose information about more complicated instances, it is sound and each polymorphic function need only be analysed once.

A more accurate approach is to analyse each instance separately, but to use information which has already been gathered about other instances to speed up that analysis. Hunt observes, for example, that the frontiers technique for solving the fixpoint equations arising from abstract interpretation can be speeded up if we start out knowing a safe upper bound on the result [Hun91]. If we analyse the simplest

instance of a function, then our invariance result means that this gives just such an upper bound on the abstract interpretation of any more complicated instance.

Finally, we could analyse each instance separately and only make use of the invariance result in cases where we do happen to know for which property we are searching. For example, if $f :: \sigma \to \tau$ is a polymorphic function and we are analysing an application of its $S(\sigma \to \tau)$ instance to a term u of type $S(\sigma)$ to see if it satisfies $\mathbf{f}^{S\tau}$, then the situation is

$$\frac{\Gamma \vdash f^{S(\sigma\to\tau)} : \phi^{S\sigma} \to \mathbf{f}^{S\tau} \qquad \Gamma \vdash u^{S\sigma} : \phi^{S\sigma}}{\Gamma \vdash (f\,u) : \mathbf{f}^{S\tau}}\ [\text{app}]$$

It may happen that a complete analysis of u reveals that the best (smallest) property $\phi^{S\sigma}$ which it can be shown to satisfy is one which makes sense at a simpler instance $S'\sigma$ of σ than $S\sigma$ (note that we are not assuming that u itself has a polymorphic type), so that $\phi^{S\sigma} = \mathcal{E}^{S''}_{S'\sigma}(\psi^{S'\sigma})$, where $S = S'' \circ S'$. In this case, we may safely analyse the instance of f at type $S'(\sigma \to \tau)$ to see if it can be shown to satisfy $\psi^{S'\sigma} \to \mathbf{f}^{S'\tau}$ without losing any information.

It seems likely that the polymorphic invariance theorem will extend to the treatment of lazy algebraic types which was presented in Chapter 4. It is natural to ask whether a similar result can be proved for the disjunctive strictness logic. Our previous remark that the Lindenbaum algebras of the disjunctive theories are not built up in a compositional way indicates that an invariance result for the disjunctive logic would probably be much more complicated and harder to obtain than in the conjunctive case. A weaker (more incomplete) version of the disjunctive logic, which did not include irreducibility, might be more tractable.

Chapter 6

Conclusions and Further Work

In this chapter we summarise the work which has been presented in this thesis and outline the prospects for further work which arise from it. We also sketch an approach to an important problem which we have so far said little about, namely that of formalising what use can be made in a compiler of the results of strictness analysis.

6.1 Summary

In Chapter 2 a simple lazy functional language was introduced. We gave it an operational and a denotational semantics and proved a computational adequacy result relating the two. We then introduced the problem of strictness analysis and reviewed previous work on strictness analysis by abstract interpretation and non-standard type inference.

In Chapter 3 we introduced a new formulation of higher-order strictness analysis, called strictness logic. We showed that the axiomatisation of conjunctive strictness properties is sound and complete with respect to its intended interpretation in the denotational semantics. An associated program logic was then given for assigning properties to terms and this was shown to be sound. This provides a semantic basis for Kuo and Mishra's strictness type system and also extends it.

The program logic was then reformulated slightly to give a system in which any derivable judgement has a normal derivation, in which the subsumption rule is used only at the leaves. This helps in reasoning about the logic and is a first step towards devising an inference algorithm. It was then shown that the derivable strictness properties of a term are invariant under β-conversion, and that the logic is equivalent in power to the traditional approach using abstract interpretation.

We showed how the logical approach to strictness analysis leads naturally to a more powerful logic which incorporates disjunction. This logic does not appear to correspond to an abstract interpretation in the traditional sense. The disjunctive logic was shown to be sound, and we used it to extend our analysis to a language which

inludes sum types. We then discussed the problem of completeness in axiomatising the entailment relation on disjunctive strictness properties and showed that the disjunctive logic given by Jensen in [Jen92b] is unsound.

In Chapter 4 we considered extending our language with lazy algebraic datatypes. We reviewed the denotational semantics of such types and previous work on strictness analysis of programs which use them. A new construction of lattices of strictness properties of lazy algebraic types was then presented. This makes use of the fact that the solutions to the recursive domain equations associated with these types are initial algebras. A major advantage of our construction over previous suggestions is that it is applicable to any algebraic datatype.

We showed how the initial algebra induction principle can be used to reason about the inclusion ordering on the properties which are picked out by our technique. The new construction was used to extend the conjunctive strictness logic to a language including a type of lazy lists of natural numbers.

In Chapter 5 we proved two parametricity results concerning languages with first-order ('ML-style') polymorphism. A simple denotational semantics was given to a language which includes Milner's **let** construct and parameterised lazy algebraic datatypes. This semantics was then used to derive the parametricity results presented by Wadler in [Wad89], which were originally justified by appealing to a model of the full second-order polymorphic lambda calculus.

We then used similar techniques to prove a polymorphic invariance theorem for abstract interpretation, which we also translated into a result about the conjunctive strictness logic. This theorem relates the derivable strictness properties of different instances of polymorphic terms and improves on previous results in the area.

6.2 Further Work

There are many loose ends and prospects for further research arising from the work presented here. This section gives some of them, in roughly the order in which the topics arose in the thesis.

6.2.1 Implementations

A major gap in the work we have done is that we have not attempted to devise an inference algorithm for any of the versions of the strictness logic. A first step towards this would be to improve the result about normal derivations, so as to control the use of conjunction as well as that of subsumption. In practice we probably do not want to define a 'principal types' algorithm, as that would correspond to computing the whole of the abstract function, which can be extremely expensive. One of the reasons which we gave at the end of Chapter 2 for studying type-like techniques for program analysis was to get better control over the trade-off between the complexity and the accuracy of an analysis. Whilst we have not really justified this claim, there is reason to hope that, by separating out the individual properties, the logic will lead

to inference systems which are able to deduce useful information without analysing every function completely.

A related observation is that it is probably futile to attempt to speed up analyses by working with very small lattices (I am thinking particularly of algebraic types): this will make much useful information unobtainable and in any case it is almost impossible to prevent the lattices associated with higher-order types from becoming enormous. A much more promising approach is to investigate the use of heuristics in guiding a possibly incomplete proof search, and the logical formulation seems well-suited to this. The Cousots' work on the use of widening and narrowing operators in abstract interpretation should be relevant here—they use heuristics to change the size of lattices dynamically during fixpoint iteration.

It would also be interesting to investigate the use of equational or order-sorted unification in obtaining inference algorithms. We have already mentioned Wright's use of unification over the theory of Boolean rings, though as our logic is not classical (we certainly do not want to attempt to add negation) this is probably not the theory which we should use.

6.2.2 Disjunctive Strictness Logic

The material on the disjunctive logic is really just a first step. Much remains to be done on investigating the proof-theory and semantics of the system, though I suspect that pursuing the question of completeness further would not be particularly profitable.

6.2.3 Foundations

Throughout this work we have been very specific. We have considered one particular analysis problem in the context of a particular concrete model. There is a more general picture: we have a cartesian closed category with types as objects and terms as morphisms, and a lattice of distinguished subobjects sitting over each object. Morphisms in the base then induce maps between the appropriate lattices (one might want to rephrase things in terms of inverse images of open sets, rather than direct images of closed sets as we have done). Variations on this kind of structure have been very well studied in categorical logic, which could well lead to further insights into program analysis.

It might also be interesting to explore further the connections between the logical approach to program analysis and domain logics. Whilst domain logics were one of the inspirations for the work presented here, we have not really said very much about them. It may be possible to present the standard semantics in a logical form too, so that everything is done using logic. The properties in which we are interested do not, however, appear to be expressible in terms of the compact open sets which have been used in previous work on presenting domain theory in logical form.

6.2.4 Algebraic Datatypes

The approach which we have suggested to strictness analysis of algebraic datatypes seems promising, but we should certainly consider more examples. We also need to address the problems of finding good syntactic representations of the properties and of synthesizing proof rules, both for the entailment relation (using the initial algebra induction principle) and for the program logic.

6.2.5 Parametricity

It would be pleasant to tidy up the proofs concerning algebraic types which we used to obtain the parametricity result for $\Lambda_{P,a}$. As we have mentioned, this should be possible by solving domain equations in a category of ω-complete relations (see [SP82]). Given such a setting, one might be able to extend the result to a larger class of recursive types. It would also be interesting to look at parametricity in the framework of Fourman and Phoa's more general categorical models for first-order polymorphism.

The polymorphic invariance theorem for strictness analysis should be extended to algebraic datatypes. We should also look further at the question of how best to exploit the result in analysing real polymorphic programs.

6.3 Using Strictness Information

A large area of further work concerns formalising the uses which can be made of strictness information. This is a surprisingly neglected topic: in contrast to the formal nature of much work on strictness analysis, most accounts of the associated optimising transformations are rather unconvincing. Here we sketch some very preliminary ideas about a possible approach to the problem, though much remains to be done before we can make any use of the framework we propose.

We will concentrate on sequential implementations and the question of when we can safely replace call-by-name with call-by-value. The usual argument is that $\beta\delta$ reduction in our language is Church-Rosser, so that if two evaluation strategies both terminate then they must reach the *same* normal form. Consequently, we can change the default evaluation strategy precisely when we can show that we will not change the termination properties of the program. Informally, the reason that call-by-name and call-by-value strategies have different termination behaviour is that call-by-value may diverge when evaluating a function argument which call-by-name would never have evaluated. So if, for a particular application, we can show that if the argument diverges then the application would have diverged anyway, then we know that we will not change the termination behaviour of the whole program by passing that argument by value.

There are, however, at least two ways in which the approach outlined above can become rather clumsy. The first is that it cannot describe the complementary opti-

misations which many real compilers perform as a result of strictness analysis. These involve changing the compiled code to reflect the fact that at certain points in the program we know that a function argument will *already* have been evaluated. The second difficulty is that when we come to consider the strictness properties of functions which operate on structured datatypes, such as lists, the notion of reduction strategy is too messy. In dealing with the optimisations associated with some of the information which can be gathered by projection analysis, for instance, we can find ourselves having to consider environments containing terms paired with reduction strategies. In any case, making the connection between a denotational semantics, which models a particular evaluation strategy, and possible changes in evaluation strategy is not straightforward.

Probably the most developed approach to using strictness information is that of Burn. He works with *evaluation transformers* which are intended to show how a function maps a particular level of demand on its result into levels of demand on its arguments, paying particular attention to the optimisation of functions which manipulate lists [Bur90b, Bur91b]. The proofs still require a certain amount of 'hand waving' about reduction strategies, however.

Here we propose that, instead of reasoning about reduction strategies, we translate the source language into a target language, which has a fixed reduction strategy. Optimisations should then be expressed as changes in the translation, rather than in evaluation order. This does of course have the appeal of being closer to actual practice, since we ultimately intend our optimisations to be realised as changes in the translation of the source language to machine code. It should be noted that it is possible to have both run-time and compile-time choice of code versions. Burn's work considers both of these, though here we shall restrict attention to compile-time optimisations.

There are several candidates for the target language. The most pleasant situation would be if we could have the source and target languages being the same. Then our optimisations would just be source to source transformations. Unfortunately, our call-by-name source language is not expressive enough to describe the optimisations which we wish to make. At the other end of the scale, we might consider using the kind of intermediate language which is used in the back end of compilers, such as G-machine code, or even three-address code. Nielson and Nielson have considered the use of first-order strictness information in changing the compilation rules for a version of the categorical abstract machine [NN90]. This, however, is too low-level for correctness proofs to be comfortable. What we really want of a target language is that it has just enough extra structure to express optimisations, but not so much that denotational reasoning becomes difficult.

It would be possible to make the target language the call-by-value lambda calculus. There is a well-known translation of call-by-name into call-by-value which uses higher order functions to simulate closures. This is an attractive choice, but has the disadvantage of being somewhat complicated, essentially because there are two levels of evaluation: the underlying evaluation strategy of the strict language and the 'simulated' call-by-name evaluation on top.

Another plausible choice for the target language is Moggi's computational lambda calculus [Mog89]. This is a typed metalogic which makes explicit the distinction between computations and values, which appears to be exactly the kind of extra refinement we need to describe strictness-based optimisations. The problem with the computational lambda calculus for our purposes, however, is that it is not operational enough. Even though it is possible to define an evaluation relation between terms of computation type and terms of value type, this is only evaluation up to provable equality in the computational lambda calculus. What we really want from our intermediate language is to be able to see in exactly what order reductions take place and to show that the changed translation of a program reaches the same normal form as the original, rather than just to know that the original translation and the optimised translation are denotationally equal in some system. We shall therefore define a language which bears some resemblance to the computational lambda calculus but which has a much more operational flavour.

Our general approach is similar to that taken by researchers working on using resource-conscious lambda calculi to improve compilation of functional languages. The languages which arise from applying the Curry-Howard correspondence to resource-conscious intuitionistic logics do not in themselves give analysis systems for reasoning about programs written in a more conventional typed lambda calculus such as Λ_T. Instead, they are intermediate languages with refined type systems to make resource usage more explicit. The aim is to use static analysis to improve the default translation of a language such as Λ_T into these refined languages. (Interestingly, the computational lambda calculus can be seen as arising via the Curry-Howard correspondence from an intuitionistic modal logic [Ben92a, BBdP92].)

The intermediate language arises from the very simple idea of extending the source language Λ_T (or $\Lambda_{T,a}$) with a strict let construct. (This should not be confused with Milner's let construct for polymorphism which we used in Chapter 5.) The intended operational behaviour of let $x \Leftarrow e$ in e' is that e is evaluated to WHNF, the result is bound to x and then e' is evaluated.

This will cause difficulties if we attempt a naive extension of the denotational semantics which we have given for Λ_T. The problem is that the semantics does not really model reduction to WHNF, as it identifies $\lambda x.\bot$, a function which when applied always diverges, with $\bot$, a divergent computation. That did not cause any problems when we gave the semantics of Λ_T because the only operational test which could be performed on a term of functional type was to apply it to an argument. However, the new let construct allows us to test functions for termination without applying them, and this makes the distinction significant: let $f \Leftarrow \lambda x.\Omega$ in 3 will return 3, whereas let $f \Leftarrow \Omega$ in 3 will fail to terminate. This is, of course, the distinction which is made in the lazy lambda calculus, but not in the classical theory of the lambda calculus [Abr88, Ong88]. A similar argument applies to pairs since previously the only test which could be performed on a pair was to project one component. For this reason, the semantics of the intermediate language has to use lifted products and function spaces.

The intermediate language, which we call Λ_{op}, has a type system which enforces a

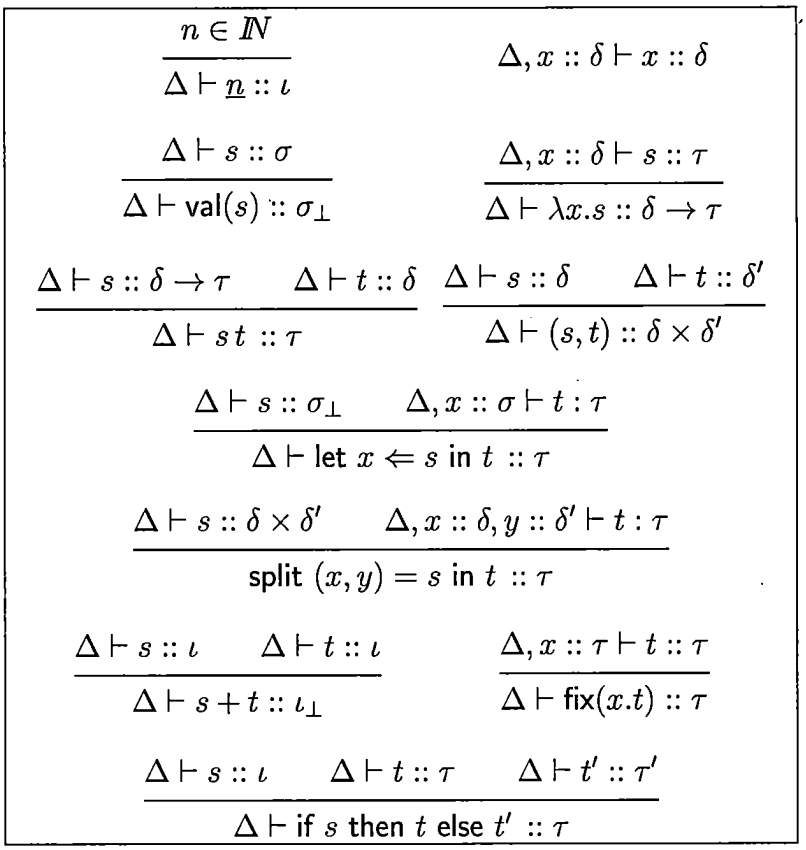


Figure 6.1: Syntax of Λ_{op}

separation between computations and values, though not in quite the same sense as in the computational lambda calculus. We take the rather alarmingly literal view that values are terms in canonical form whereas computations are non-canonical terms. A closely related system has been proposed by Peyton Jones and Launchbury [PJL91]. Their type system separates boxed values from unboxed values, and they also suggest that it could be useful in expressing strictness-based optimisations.

The types of Λ_{op} are divided into classes:

$$\begin{array}{lcll} \sigma & ::= & \iota \mid \delta \to \tau \mid \delta \times \delta & \text{[Value Types]} \\ \tau & ::= & \sigma_\perp & \text{[Computation Types]} \\ \delta & ::= & \sigma \mid \tau & \text{[Types]} \end{array}$$

The syntax of Λ_{op} is shown in Figure 6.1. It is presented using typing derivations, rather than typed variables. A type context Δ is a finite set of assumptions of the form $x :: \delta$ and we write $\Delta, x :: \delta$ for Δ with any existing assumption for x removed and the assumption $x :: \delta$ added.

Note that all terms of value type are in canonical form. The $\mathsf{val}(\cdot)$ construct allows a value to be treated as a computation, whilst the let construct allows a computation to be evaluated to a value within the context of another computation. We choose

$$\mathsf{val}(t)\Downarrow t \qquad \underline{n}+\underline{m}\Downarrow\underline{m+n}$$

$$\frac{t[s/x]\Downarrow c}{(\lambda x.t)\,s\Downarrow c} \qquad \frac{s\Downarrow c \qquad t[c/x]\Downarrow c'}{\mathsf{let}\ x \Leftarrow s\ \mathsf{in}\ t\Downarrow c'}$$

$$\frac{t\Downarrow c}{\mathsf{if}\ \underline{0}\ \mathsf{then}\ t\ \mathsf{else}\ t'\Downarrow c} \qquad \frac{t'\Downarrow c}{\mathsf{if}\ \underline{n+1}\ \mathsf{then}\ t\ \mathsf{else}\ t\Downarrow c}$$

$$\frac{t[u/x,v/y]\Downarrow c}{\mathsf{split}\ (x,y)=(u,v)\ \mathsf{in}\ t\Downarrow c} \qquad \frac{t[\mathsf{fix}(x.t)/x]\Downarrow c}{\mathsf{fix}(x.t)\Downarrow c}$$

Figure 6.2: Operational Semantics of Λ_{op}

to present the destructor for pairs using **split**, rather than projections as we should otherwise have to have two versions of each projection according to whether or not the type of the relevant factor in the product was a computation type. Note also that both the type of the body of a **let** construct and the result type of functions are constrained to be computation types. One might wish to change this.

The operational semantics of Λ_{op} is given by an evaluation relation $\Downarrow$ which relates closed terms of computation type $\sigma_\perp$ to closed terms of value type σ. This is shown in Figure 6.2. Note that there are no clauses of the form $c\Downarrow c$ for c canonical, as there were in the operational semantics of Λ_T. This is not just a notational trick—it corresponds to the fact that terms of value types are known to be in WHNF and therefore need not be evaluated before they can be used.

Λ_{op} can be given an adequate denotational semantics in $\mathcal{P}redom$. The predomains associated with types are

$$D_\iota = I\!N \text{ (the set of natural numbers with the discrete order)}$$

$$D_{\delta\to\tau} = [D_\delta \to D_\tau] \qquad D_{\delta\times\delta'} = D_\delta \times D_{\delta'}$$

$$D_{\sigma_\perp} = (D_\sigma)_\perp$$

We omit the details of the denotational semantics, which should be fairly obvious, but note that the $\mathsf{val}(\cdot)$ construct is interpreted by lifting and that the **let** construct is essentially interpreted by Kleisli composition for the lift monad.

There are then translations of both call-by-name and call-by-value simply-typed lambda calculi with constants into Λ_{op}. These translations are essentially the same as those given by Moggi in [Mog89] (see also [Cro92, Wad90]). The call-by-name translation maps a term t of type θ to a Λ_{op} term $[\![t]\!]^n$ of type $[\![\theta]\!]^n$, where

$$[\![\iota]\!]^n = \iota_\perp \qquad [\![\theta\times\theta']\!]^n = ([\![\theta]\!]^n \times [\![\theta']\!]^n)_\perp$$

$$[\![\theta\to\theta']\!]^n = ([\![\theta]\!]^n \to [\![\theta']\!]^n)_\perp$$

$$[\![\underline{n}]\!]^n = \mathsf{val}(\underline{n}) \qquad [\![x]\!]^n = x$$

$$[\![s\,t]\!]^n = \mathsf{let}\ f \Leftarrow [\![s]\!]^n\ \mathsf{in}\ f\,[\![t]\!]^n \quad (\text{where } f \text{ is fresh})$$

$$[\![(s,t)]\!]^n = \mathsf{val}([\![s]\!]^n, [\![t]\!]^n) \qquad [\![\lambda x.u]\!]^n = \mathsf{val}(\lambda x.[\![u]\!]^n)$$

$$[\![\mathsf{fst}(u)]\!]^n = \mathsf{let}\ p \Leftarrow [\![u]\!]^n\ \mathsf{in\ split}\ (x,y) = p\ \mathsf{in}\ x \quad (x,y,p \text{ fresh})$$

$$[\![\mathsf{snd}(u)]\!]^n = \mathsf{let}\ p \Leftarrow [\![u]\!]^n\ \mathsf{in\ split}\ (x,y) = p\ \mathsf{in}\ y \quad (x,y,p \text{ fresh})$$

$$[\![u+v]\!]^n = \mathsf{let}\ x \Leftarrow [\![u]\!]^n\ \mathsf{in\ let}\ y \Leftarrow [\![v]\!]^n\ \mathsf{in}\ x+y \quad (x,y \text{ fresh})$$

$$[\![\mathsf{if}\ s\ \mathsf{then}\ t\ \mathsf{else}\ t']\!]^n = \mathsf{let}\ x \Leftarrow [\![s]\!]^n\ \mathsf{in\ if}\ x\ \mathsf{then}\ [\![t]\!]^n\ \mathsf{else}\ [\![t']\!]^n \quad (x \text{ fresh})$$

$$[\![\mathsf{fix}(x.u)]\!]^n = \mathsf{fix}(x.[\![u]\!]^n)$$

Figure 6.3: Translating Call-By-Name into Λ_{op}

The call-by-name translation is shown in Figure 6.3.

The translation of call-by-value, of which we omit the details, maps a term of type θ into a Λ_{op} term of type $[\![\theta]\!]^v_\perp$, where $[\![\theta]\!]^v$ is defined by

$$[\![\iota]\!]^v = \iota \qquad [\![\theta \times \theta']\!]^v = [\![\theta]\!]^v \times [\![\theta']\!]^v$$

$$[\![\theta \to \theta']\!]^v = [\![\theta]\!]^v \to [\![\theta']\!]^v_\perp$$

The terms resulting from the call-by-name translation are rather large, but this is because all the implicit details about evaluation order have been made explicit. The translation can be improved somewhat by making use of equations which hold between terms of Λ_{op}. Two of the most useful are:

$$\mathsf{let}\ x \Leftarrow \mathsf{val}(c)\ \mathsf{in}\ t = t[c/x]$$

$$\mathsf{let}\ x \Leftarrow (\mathsf{let}\ y \Leftarrow u\ \mathsf{in}\ v)\ \mathsf{in}\ w = \mathsf{let}\ y \Leftarrow u\ \mathsf{in}\ (\mathsf{let}\ x \Leftarrow v\ \mathsf{in}\ w)$$

(where the second rule has the side-condition that $y \notin FV(w)$).

We then hope to be able to justify using strictness information to change this translation. For example, the default translation of

$$\mathsf{double} \stackrel{\mathrm{def}}{=} \lambda x.x + x$$

is

$$[\![\mathsf{double}]\!]^n = \mathsf{val}(\lambda x.\mathsf{let}\ y \Leftarrow x\ \mathsf{in\ let}\ z \Leftarrow x\ \mathsf{in}\ y+z)$$

Because double is strict, we can translate the body using the alternative rule

$$[\![\lambda x.t]\!]^n = \mathsf{val}(\lambda x.\mathsf{let}\ y \Leftarrow x\ \mathsf{in}\ ([\![t]\!]^n[\mathsf{val}(y)/x]))$$

which simplifies, after applying the let/val rule above, to

$$[\![\mathsf{double}]\!]^n = \mathsf{val}(\lambda x.\mathsf{let}\ y \Leftarrow x\ \mathsf{in}\ y + y)$$

This new translation moves the evaluation of strict arguments up to the front of the function body, and is perfomed in real compilers. Notice how the shared evaluation of the two occurrences of the argument now happens without any updating.

We should also like to be able to pass arguments by value. Consider the source program

$$(\lambda x.x + x)\,(3 + 4)$$

Using the alternative translation above, this translates to

$$(\lambda x.\mathsf{let}\ y \Leftarrow x\ \mathsf{in}\ y + y)\,(\mathsf{let}\ w \Leftarrow \mathsf{val}(\underline{3})\ \mathsf{in}\ \mathsf{let}\ z \Leftarrow \mathsf{val}(\underline{4})\ \mathsf{in}\ w + z)$$

which simplifies to

$$(\lambda x.\mathsf{let}\ y \Leftarrow x\ \mathsf{in}\ y + y)\,(\underline{3} + \underline{4})$$

But we really want to get

$$\mathsf{let}\ z \Leftarrow \underline{3} + \underline{4}\ \mathsf{in}\ (\lambda y.y + y)\,z$$

in which because we have an application of a strict function, we can evaluate the argument before the application and compile the function to expect an evaluated argument. Note that this involves changing the type of the translation of the function.

These are trivial examples, and we have not given any formal proofs. There are several difficulties which must be addressed before we can make the idea work. The first, and possibly the largest, difficulty is deciding exactly which optimisations we wish to perform. Once one has a general framework like this, there are many possible choices. A particular problem is knowing when to generate multiple translations to be used in different contexts (for example, "is it a good idea to generate a special version of this higher order function because we happen to know that in this occurrence it will be passed an argument which is a function which is strict in its second argument?"). We may need to include the notion of a top-level environment in the intermediate language, so that we can generate multiple versions just for top-level definitions. It is also not clear exactly how to combine the strictness analysis with the translation process, in the sense of what properties need to be deduced in what contexts to enable a particular transformation.

Another problem is that justifying the transformed programs will not be as simple as one might hope. A naive idea is that transformed terms should be denotationally equivalent to the default translations. This will, even for the simplest transformations, not be true. The problem is that the default translation and the improved translation of a term will, in general, only be observationally equivalent in all contexts which arise as the translation of call-by-name contexts; not in all contexts which are expressible in Λ_{op}. Thus we shall have to take context into account to prove the correctness of optimisations.

There are also complications arising from trying to use the results of a strictness analysis which is based on the semantics of the source language in reasoning about transformations on the target language, which uses different (pre)domains. It might be better to reformulate the analysis directly in terms of the intermediate language.

As yet, then, the claim that this framework can be used to express and prove the correctness of strictness optimisations is unjustified. The intermediate language does seem, however, to have just enough extra structure to capture what happens in real optimising compilers, and it is hard to see how anything closer to the source language could do (apart from a language which only allowed the **let**-evaluation of terms of ground type). If what happens in compilers is correct, then it should be possible to explain and justify it in this framework. That is our next goal.

Appendix A

Omitted Proofs

Proof of Theorem 3.3.6
We start by proving that we may normalise any derivation containing exactly one occurrence of [sub′]. This is proved by induction on the height of the derivation. Clearly, we only need to consider the case where [sub′] is the last rule applied in the derivation, so that the situation is

$$\dfrac{\begin{array}{c}\Pi\\ \Gamma \vdash s : \psi^\sigma \qquad \psi^\sigma \leq \phi^\sigma\end{array}}{\Gamma \vdash s : \phi^\sigma}\,[\text{sub}']$$

It suffices to consider the cases where ϕ^σ is not a conjunction, since if we can normalise all the non-conjunctive cases, and $\phi^\sigma = \bigwedge_i \phi_i^\sigma$, then $\psi^\sigma \leq \phi_i^\sigma$ for each i, so there is a normal derivation of $\Gamma \vdash s : \phi_i^\sigma$ and hence we are done by [conj′].
We consider cases according to the last rule of Π:

- Case [sub′]. This cannot happen as Π is normal by assumption.
- Case [top′]. In this case there is a normal derivation of $\Gamma \vdash s : \phi^\sigma$ by Lemma 3.3.3.
- Case [var′]. By Lemma 3.3.4, we can produce a normal derivation consisting of one application of [var′].
- Case [absbot′]. We have

$$\dfrac{\dfrac{\begin{array}{c}\Sigma\\ \Gamma, x^\sigma : \mathbf{t}^\sigma \vdash t : \mathbf{f}^\tau\end{array}}{\Gamma \vdash \lambda x^\sigma.t : \mathbf{f}^{\sigma\to\tau}}\,[\text{absbot}'] \qquad \mathbf{f}^{\sigma\to\tau} \leq \phi^{\sigma\to\tau}}{\Gamma \vdash \lambda x^\sigma.t : \phi^{\sigma\to\tau}}\,[\text{sub}']$$

  and we consider the possible forms of $\phi^{\sigma\to\tau}$. If it is $\mathbf{f}^{\sigma\to\tau}$ then we are done immediately. If it is $\mathbf{t}^{\sigma\to\tau}$ then we are done by [top′]. As noted above, we do not need to consider the case of conjunctions so the only remaining possibility

is that $\phi^{\sigma\to\tau} = \varphi^\sigma \to \psi^\tau$. Since $\varphi^\sigma \leq \mathbf{t}^\sigma$, Lemma 3.3.5 gives us a normal derivation Σ' of $\Gamma, x^\sigma : \varphi^\sigma \vdash t : \mathbf{f}^\tau$ which is of the same height as Σ. Thus

$$\dfrac{\begin{array}{c}\Sigma' \\ \Gamma, x^\sigma : \varphi^\sigma \vdash t : \mathbf{f}^\tau\end{array} \qquad \mathbf{f}^\tau \leq \psi^\tau}{\Gamma, x^\sigma : \varphi^\sigma \vdash t : \psi^\tau}\,[\mathrm{sub}']$$

is a shorter derivation than the original and contains exactly one use of [sub′]. It can therefore be normalised by the induction hypothesis, and an application of [abs′] then gives the result.

- Case [app′]. The situation looks like

$$\dfrac{\dfrac{\Gamma \vdash t : \phi^\sigma \to \psi^\tau \qquad \Gamma \vdash s : \phi^\sigma}{\Gamma \vdash ts : \psi^\tau}\,[\mathrm{app}'] \qquad \psi^\tau \leq \chi^\tau}{\Gamma \vdash\ ts : \chi^\tau}\,[\mathrm{sub}']$$

  which we can transform into

$$\dfrac{\dfrac{\Gamma \vdash t : \phi^\sigma \to \psi^\tau \qquad \phi^\sigma \to \psi^\tau \leq \phi^\sigma \to \chi^\tau}{\Gamma \vdash t : \phi^\sigma \to \chi^\tau}\,[\mathrm{sub}'] \qquad \Gamma \vdash s : \phi^\sigma}{\Gamma \vdash\ ts : \chi^\tau}\,[\mathrm{app}']$$

  and it is clear that the derivation of the first premiss of the final inference of the above can be normalised by induction, and hence we are done.

- Case [abs′].

$$\dfrac{\dfrac{\Gamma, x^\sigma : \phi^\sigma \vdash t : \psi^\tau}{\Gamma \vdash \lambda x^\sigma.t : \phi^\sigma \to \psi^\tau}\,[\mathrm{abs}'] \qquad \phi^\sigma \to \psi^\tau \leq \chi^{\sigma\to\tau}}{\Gamma \vdash \lambda x^\sigma.t : \chi^{\sigma\to\tau}}\,[\mathrm{sub}']$$

  If $\chi^{\sigma\to\tau} = \mathbf{t}^{\sigma\to\tau}$ then we just use [top′]. If $\chi^{\sigma\to\tau} = \mathbf{f}^{\sigma\to\tau}$ then by entailment decomposition $\mathbf{t}^\sigma \leq \phi^\sigma$ and $\psi^\tau \leq \mathbf{f}^\tau$. Thus

$$\dfrac{\Gamma, x^\sigma : \phi^\sigma \vdash t : \psi^\tau \qquad \psi^\tau \leq \mathbf{f}^\tau}{\Gamma, x^\sigma : \phi^\sigma \vdash t : \mathbf{f}^\tau}\,[\mathrm{sub}']$$

  which may be normalised by induction hypothesis. Lemma 3.3.5 then gives a normal derivation of $\Gamma, x^\sigma : \mathbf{t}^\sigma \vdash t : \mathbf{f}^\tau$, and an application of [absbot′] gives the result in this subcase.

  The remaining subcase is $\chi^{\sigma\to\tau} = \phi_1^\sigma \to \psi_1^\tau$. By entailment decomposition, we know that either (a) $\mathbf{t}^\tau \leq \psi_1^\tau$ or (b) $\psi^\tau \leq \psi_1^\tau$ and $\phi_1^\sigma \leq \phi^\sigma$. In case (a), by Lemma 3.3.3, we have a normal derivation of $\Gamma, x^\sigma : \psi_1^\sigma \vdash t : \psi_1^\tau$ so we are done by [abs′]. In case (b), we can inductively normalise

$$\dfrac{\Gamma, x^\sigma : \phi^\sigma \vdash t : \psi^\tau \qquad \psi^\tau \leq \psi_1^\tau}{\Gamma, x^\sigma : \phi^\sigma \vdash t : \psi_1^\tau}\,[\mathrm{sub}']$$

  and then apply Lemma 3.3.5 and [abs′] to obtain a normal derivation of the original judgement.

- Case [conj′]. We have

$$\dfrac{\dfrac{\begin{matrix}\Pi_1 & & \Pi_n \\ \Gamma \vdash s : \phi_1^\sigma & \ldots & \Gamma \vdash s : \phi_n^\sigma\end{matrix}}{\Gamma \vdash s : \bigwedge_{i=1}^{n} \phi_i^\sigma}\,[\text{conj}'] \qquad \bigwedge_{i=1}^{n} \phi_i^\sigma \leq \psi^\sigma}{\Gamma \vdash s : \psi^\sigma}\,[\text{sub}']$$

and without loss of generality we can assume that none of the Π_i end with [conj′], as such occurrences can be incorporated into the lower occurrence of [conj′], or with [top′], as we can remove such a Π_i altogether (if all the Π_i end in [top′] then the whole derivation collapses to one use of [top′]). We consider the possible cases for the form of s:

  - If s is a variable or numeral then by Lemma 3.3.4 we can just replace the derivation with an instance of [var′] or [top′].
  - If s is an application $u\,v$ then the only case we have to consider is the one in which every Π_i ends with an occurrence of [app′][1]:

$$\dfrac{\dfrac{\dfrac{\Gamma \vdash u : \phi_1 \to \psi_1 \quad \Gamma \vdash v : \phi_1}{\Gamma \vdash uv : \psi_1} \quad \cdots \quad \dfrac{\Gamma \vdash u : \phi_n \to \psi_n \quad \Gamma \vdash v : \phi_n}{\Gamma \vdash uv : \psi_n}}{\Gamma \vdash uv : \bigwedge_{i=1}^{n} \psi_i} \qquad \bigwedge_{i=1}^{n} \psi_i \leq \chi}{\Gamma \vdash uv : \chi}$$

  We form the following derivation, Σ:

$$\dfrac{\dfrac{\Gamma \vdash u : \phi_1 \to \psi_1 \quad \cdots \quad \Gamma \vdash u : \phi_n \to \psi_n}{\Gamma \vdash u : \bigwedge_{i=1}^{n} (\phi_i \to \psi_i)} \qquad \bigwedge_{i=1}^{n} (\phi_i \to \psi_i) \leq (\bigwedge_{i=1}^{n} \phi_i) \to \chi}{\Gamma \vdash u : (\bigwedge_{i=1}^{n} \phi_i) \to \chi}$$

  which may be normalised inductively to give Σ' so that we can form

$$\dfrac{\begin{matrix}\Sigma' \\ \Gamma \vdash u : (\bigwedge_{i=1}^{n} \phi_i) \to \chi\end{matrix} \qquad \dfrac{\Gamma \vdash v : \phi_1 \cdots \Gamma \vdash v : \phi_n}{\Gamma \vdash v : \bigwedge_{i=1}^{n} \phi_i}}{\Gamma \vdash uv : \chi}$$

  which is normal.

[1] We have omitted type superscripts and rule names for layout reasons.

– If s is $\mathsf{fix}(x^\sigma.u)$ then the only case we have to consider is when all the Π_i end in [fix′]:

$$\dfrac{\dfrac{\Gamma, x:\phi_1 \vdash u:\phi_1 \quad \phi_1 \leq \psi_1}{\Gamma \vdash \mathsf{fix}(x.u):\psi_1} \quad \cdots \quad \dfrac{\Gamma, x:\phi_n \vdash u:\phi_n \quad \phi_n \leq \psi_n}{\Gamma \vdash \mathsf{fix}(x.u):\psi_n}}{\dfrac{\Gamma \vdash \mathsf{fix}(x.u):\bigwedge_{i=1}^{n}\psi_i \qquad \bigwedge_{i=1}^{n}\psi_i \leq \chi}{\Gamma \vdash \mathsf{fix}(x.u):\chi}}$$

Then by Lemma 3.3.5 we have a normal derivation of $\Gamma, x:\bigwedge_i \phi_i \vdash u:\phi_j$ for each j, which is the same height as the original derivation of $\Gamma, x^\sigma : \phi_j \vdash u:\phi_j$. It is also the case that $\bigwedge_i \phi_i \leq \bigwedge_i \psi_i \leq \chi$ so

$$\dfrac{\dfrac{\Gamma, x:\bigwedge_i \phi_i \vdash u:\phi_1 \quad \cdots \quad \Gamma, x:\bigwedge_i \phi_i \vdash u:\phi_n}{\Gamma, x:\bigwedge_i \phi_i \vdash u:\bigwedge_i \phi_i} \qquad \bigwedge_i \phi_i \leq \chi}{\Gamma \vdash \mathsf{fix}(x.u):\chi}$$

is a shorter derivation than the one we started with, and we are done by induction.

– The remaining subcases of [conj′] are similar.

- Case [cond2′]. We have

$$\dfrac{\dfrac{\Gamma \vdash t_1 : \phi^\tau \qquad \Gamma \vdash t_2 : \phi^\tau}{\Gamma \vdash \mathsf{if}\ s\ \mathsf{then}\ t_1\ \mathsf{else}\ t_2 : \phi^\tau}\,[\mathrm{cond}'] \qquad \phi^\tau \leq \psi^\tau}{\Gamma \vdash \mathsf{if}\ s\ \mathsf{then}\ t_1\ \mathsf{else}\ t_2 : \psi^\tau}\,[\mathrm{sub}']$$

and thus for each $i \in \{1,2\}$ we can form Π_i:

$$\dfrac{\Gamma \vdash t_i : \phi^\tau \qquad \phi^\tau \leq \psi^\tau}{\Gamma \vdash t_i : \psi^\tau}\,[\mathrm{sub}']$$

which can be inductively normalised to yield Π_i'. Then

$$\dfrac{\Pi_1' \qquad \Pi_2'}{\Gamma \vdash \mathsf{if}\ s\ \mathsf{then}\ t_1\ \mathsf{else}\ t_2 : \psi^\tau}\,[\mathrm{cond2}']$$

is the normal derivation we wanted.

- The remaining cases are similar.

Having shown that we can remove a single use of [sub′], a second induction on the number of occurrences of [sub′] in the derivation completes the proof. □

Proof of Lemma 4.2.2
Given another cocone $\rho' : \Delta \to D'$, note that the sequence $\langle \rho'_n \circ \rho_n^R \rangle$ is an increasing chain of maps from D to D' since

$$\begin{aligned} \rho'_n \circ \rho_n^R &= \rho'_{n+1} \circ f_n \circ (\rho_{n+1} \circ f_n)^R \\ &= \rho'_{n+1} \circ f_n \circ f_n^R \circ \rho_{n+1}^R \\ &\sqsubseteq \rho'_{n+1} \circ \rho_{n+1}^R \end{aligned}$$

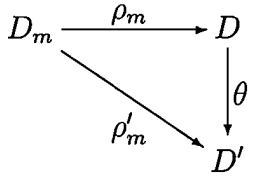
Hence we can define $\theta : D \to D'$ by $\theta = \bigsqcup_n \rho'_n \circ \rho_n^R$.
First we show that θ is an embedding, by constructing its right adjoint. Just as above, $\langle \rho_n \circ \rho_n'^R \rangle$ is an increasing sequence of maps, so we can calculate

$$\begin{aligned} (\bigsqcup \rho_n \circ \rho_n'^R) \circ (\bigsqcup \rho'_n \circ \rho_n^R) &= \bigsqcup \rho_n \circ \rho_n'^R \circ \rho'_n \circ \rho_n^R \\ &= \bigsqcup \rho_n \circ \rho_n^R \\ &= id_D \end{aligned}$$

and

$$\begin{aligned} (\bigsqcup \rho'_n \circ \rho_n^R) \circ (\bigsqcup \rho_n \circ \rho_n'^R) &= \bigsqcup \rho'_n \circ \rho_n^R \circ \rho_n \circ \rho_n'^R \\ &= \bigsqcup \rho'_n \circ \rho_n'^R \\ &\sqsubseteq id_{D'} \end{aligned}$$

Next, we check that θ is a cocone morphism, by checking that each



commutes:

$$\begin{aligned} \theta \circ \rho_m &= (\bigsqcup_n \rho'_n \circ \rho_n^R) \circ \rho_m \\ &= \bigsqcup_{n \geq m} (\rho'_n \circ \rho_n^R \circ \rho_m) \\ &= \bigsqcup_{n \geq m} (\rho'_n \circ \rho_n^R \circ \rho_n \circ f_{mn}) \\ &= \bigsqcup_{n \geq m} (\rho'_n \circ f_{mn}) \\ &= \rho'_m \end{aligned}$$

Finally, we check that θ is unique. If $\theta' : D \lhd D'$ is another mediating morphism, then

$$\theta' = \theta' \circ id_D$$

$$
\begin{aligned}
&= \theta' \circ \bigsqcup_n \rho_n \circ \rho_n^R \\
&= \bigsqcup_n \theta' \circ \rho_n \circ \rho_n^R \\
&= \bigsqcup_n \rho'_n \circ \rho_n^R \\
&= \theta
\end{aligned}
$$

□

Proof of Lemma 4.2.3
Define $|D|$ to be the set of all sequences $\langle d_n \rangle$ such that $d_n \in D_n$ and $d_n = f_n^R(d_{n+1})$ for all n. This is easily seen to form a domain D when given the pointwise order: $\langle d_n \rangle \sqsubseteq \langle d'_n \rangle \Leftrightarrow \forall n. d_n \sqsubseteq d'_n$. Now define $\rho : \Delta \to D$ by, for $d \in D_m$,

$$(\rho_m(d))_n = \begin{cases} f_{mn}(d) & \text{if } n \geq m \\ f_{nm}^R(d) & \text{if } n < m \end{cases}$$

It is easy to verify that each ρ_m is an embedding of D_m into D, with right adjoint given by $\rho_m^R(\langle d_n \rangle) = d_m$. It is also straightforward to check that ρ is indeed a cocone. Finally, it is clear that $\bigsqcup \rho_n \circ \rho_n^R \sqsubseteq id_D$, so we just need the converse. For any m, $((\rho_m \circ \rho_m^R)\langle d_n \rangle)_m = d_m$ and so $(\bigsqcup_i \rho_i \circ \rho_i^R)\langle d_n \rangle)_m \sqsupseteq d_m$ and thus $(\bigsqcup_i \rho_i \circ \rho_i^R) \sqsupseteq id_D$ as required. □

Proof of Lemma 4.2.4
By Lemma 4.2.3, there is a cocone $\rho : \Delta \to D$ such that $\bigsqcup \rho_n \circ \rho_n^R = id_D$; and by Lemma 4.2.2 this is universal. Since initial objects are unique up to isomorphism, there is an isomorphism $\theta : \rho \to \rho'$. Then

$$
\begin{aligned}
\bigsqcup \rho'_n \circ \rho'^R_n &= \bigsqcup (\theta \circ \rho_n) \circ (\theta \circ \rho_n)^R \\
&= \bigsqcup \theta \circ \rho_n \circ \rho_n^R \circ \theta^R \\
&= \theta \circ (\bigsqcup \rho_n \circ \rho_n^R) \circ \theta^{-1} \\
&= \theta \circ \theta^{-1} \\
&= id_{D'}
\end{aligned}
$$

□

Proof of Lemma 4.5.1
If $\rho : \Delta \to D$ is universal in $\mathcal{D}om_E$, then by Lemma 4.2.4 we know that $\bigsqcup_n \rho_n \circ \rho_n^R = id_D$. The proof of Lemma 4.2.2 then gives the result immediately simply by omitting the verification that in that case the mediating morphism is an embedding. □

Proof of Lemma 4.5.2
Let (D, α) be a T-algebra. A T-homomorphism from $(\mathsf{Fix}_{TE}, \eta_{TE})$ to (D, α) is a strict map $h : \mathsf{Fix}_{TE} \to D$ such that the following commutes in $\mathcal{D}om_S$:

$$
\begin{array}{ccc}
T(\mathsf{Fix}_{TE}) & \xrightarrow{T(h)} & T(D) \\
\Big\downarrow{\scriptstyle \eta_{TE}} & & \Big\downarrow{\scriptstyle \alpha} \\
\mathsf{Fix}_{TE} & \xrightarrow[h]{} & D
\end{array}
$$

Firstly, we claim that such an h, if it exists, must be unique. Recall how Fix_{T^E} was constructed as the colimit $\rho : \Delta \to \mathsf{Fix}_{T^E}$ where $\Delta = \langle (T^E)^m(1), (T^E)^m(!) \rangle$. The composite $h \circ \rho_0 : 1 \to D$ must be the unique strict map from 1 to D (this is why we chose earlier to regard the domain constructing operations as functors on $\mathcal{D}om_S$ rather than $\mathcal{D}om$, which does not have an initial object). Furthermore

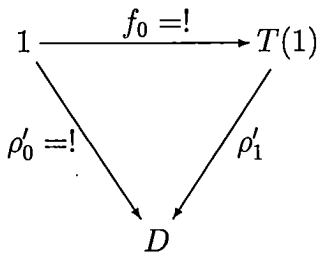
$$\begin{aligned} h \circ \rho_{m+1} &= h \circ \eta_{T^E} \circ T(\rho_m) && \text{as } \eta_{T^E} \text{ is a cocone morphism} \\ &= \alpha \circ T(h) \circ T(\rho_m) && \text{from the diagram above} \\ &= \alpha \circ T(h \circ \rho_m) \end{aligned}$$

so that by induction each $h \circ \rho_m$ is forced by the assumption that h is a T-homomorphism. But then

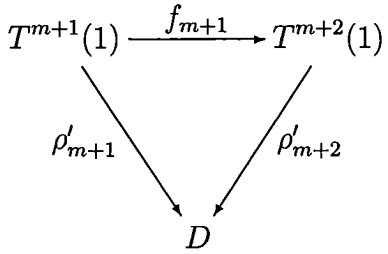
$$\begin{aligned} h &= h \circ id_{\mathsf{Fix}_{T^E}} \\ &= h \circ \bigsqcup \rho_n \circ \rho_n^R \\ &= \bigsqcup (h \circ \rho_n) \circ \rho_n^R \end{aligned}$$

so that h itself is uniquely determined by the composites $h \circ \rho_n$ and is therefore unique. It remains to show the existence of such an h.

Define the sequence $\rho' = \langle \rho'_m \rangle$ of strict maps where $\rho'_m : T^m(1) \to D$ by $\rho'_0 = ! : 1 \to D$ and $\rho'_{m+1} = \alpha \circ T(\rho'_m)$. We claim that $\rho' : \Delta \to D$ is a cocone in $\mathcal{D}om_S$, which we show by induction on m. The base case is the triangle



which certainly commutes. For the induction step we need the commutativity of



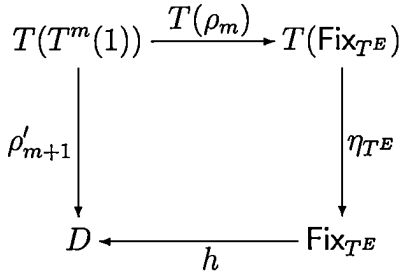
This follows because

$$\begin{aligned} \rho'_{m+2} \circ f_{m+1} &= \alpha \circ T(\rho'_{m+1}) \circ f_{m+1} \\ &= \alpha \circ T(\rho'_{m+1}) \circ T(f_m) \\ &= \alpha \circ T(\rho'_{m+1} \circ f_m) \\ &= \alpha \circ T(\rho'_m) && \text{by induction} \\ &= \rho'_{m+1} \end{aligned}$$

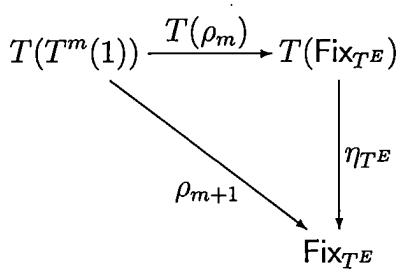
So ρ' is a cocone in $\mathcal{D}om_S$. Now, as our original cocone $\rho : \Delta \to \mathsf{Fix}_{T^E}$ is initial in $\mathcal{D}om_E$, by Lemma 4.5.1 it is initial in $\mathcal{D}om_S$ as well. Therefore, there is a unique mediating strict map $h : \rho \to \rho'$. Finally, we need to check that this h is indeed a T-homomorphism.

In $\mathcal{D}om_E$, $T^E(\rho) : T^E(\Delta) \to T^E(\mathsf{Fix}_{T^E})$ is universal, and hence by Lemma 4.5.1 again, $T(\rho) : T(\Delta) \to T(\mathsf{Fix}_{T^E})$ is universal in $\mathcal{D}om_S$. We also have a cocone $\rho'^{-} : T(\Delta) \to D$ in $\mathcal{D}om_S$, and we claim that both $h \circ \eta_{T^E}$ and $\alpha \circ T(h)$ mediate between these two cocones and are therefore equal by the initiality of $T(\rho)$.

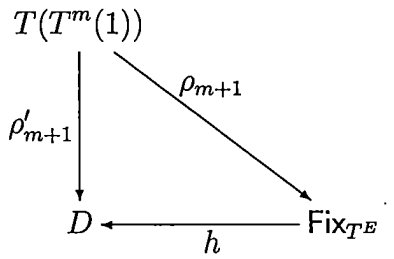
Firstly, we need to show that



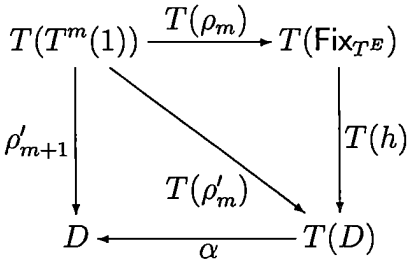
commutes. This follows from the commutativity of



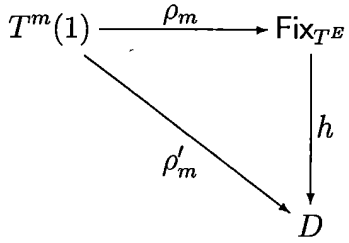
because η_{T^E} is a cocone morphism and



because h is a cocone morphism. So $h \circ \eta_{T^E}$ is a mediating morphism. Similarly, the outer square of



commutes because the two triangles do: the top right hand one commutes because it is T applied to



which commutes because h is a cocone morphism, whilst the bottom left hand one commutes by the definition of ρ'_{m+1}. □

Bibliography

[Abr85] S. Abramsky. Strictness analysis and polymorphic invariance (extended abstract). In H. Ganzinger and N. Jones, editors, *Proceedings of the Workshop on Programs as Data Objects, Copenhagen*, volume 217 of *Lecture Notes in Computer Science*, pages 1–23. Springer-Verlag, October 1985.

[Abr88] S. Abramsky. The lazy lambda calculus. In D. Turner, editor, *Research Directions in Functional Programming*, chapter 4, pages 65–116. Addison-Wesley, 1988.

[Abr90a] S. Abramsky. Abstract interpretation, logical relations and Kan extensions. *Journal of Logic and Computation*, 1(1), 1990.

[Abr90b] S. Abramsky. Computational interpretations of linear logic. Technical Report 90/20, Department of Computing, Imperial College, London, October 1990.

[Abr91] S. Abramsky. Domain theory in logical form. *Annals of Pure and Applied Logic*, 51(1–2):1–78, 1991.

[AJ89] L. Augustsson and T. Johnsson. Parallel graph reduction with the $\langle \nu, G \rangle$-machine. Chalmers University of Technology, March 1989.

[AJ91] S. Abramsky and T. P. Jensen. A relational approach to strictness analysis of higher order polymorphic functions. In *Proceedings of ACM Symposium on Principles of Programming Languages*, 1991.

[AJM93] S. Abramsky, R. Jagadeesan, and P. Malacaria. Games and full abstraction for PCF: Preliminary announcement. Department of Computing, Imperial College, London, July 1993.

[AL91] A. Asperti and G. Longo. *Categories, Types and Structures: An Introduction to Category Theory for the Working Computer Scientist.* The MIT Press, 1991.

[Aug84] L. Augustsson. A compiler for lazy ML. In *Proceedings of the ACM Symposium on Lisp and Functional Programming, Austin*, August 1984.

[Bac78] J. W. Backus. Can programming be liberated from the von-Neuman style? *Communications of the ACM*, 21:613–641, 1978.

[BBdP92] P. N. Benton, G. M. Bierman, and V. C. V. de Paiva. Computational types from a logical perspective. Submitted, November 1992.

[BBHdP92] P. N. Benton, G. M. Bierman, J. M. E. Hyland, and V. C. V. de Paiva. Term assignment for intuitonistic linear logic. Technical Report 262, Computer Laboratory, University of Cambridge, August 1992.

[BCL85] G. Berry, P. L. Curien, and J. J. Lévy. Full abstraction for sequential languages: The state of the art. In M. Nivat and J. C. Reynolds, editors, *Algebraic Methods in Semantics*, pages 89–132. Cambridge University Press, 1985.

[Ben92a] P. N. Benton. Proof theory of the computational lambda calculus. Draft, June 1992.

[Ben92b] P. N. Benton. Strictness logic and polymorphic invariance. In A. Nerode and M. Taitslin, editors, *Proceedings of the Second International Symposium on Logical Foundations of Computer Science, Tver, Russia*, volume 620 of *Lecture Notes in Computer Science*, pages 33–44. Springer-Verlag, July 1992.

[Ben93] P. N. Benton. Strictness properties of lazy algebraic datatypes. In *Proceedings of the Third International Workshop on Static Analysis, Padova, Italy*, volume ?? of *Lecture Notes in Computer Science*. Springer-Verlag, September 1993.

[BFSS90] S. Bainbridge, P. Freyd, A. Scedrov, and P. Scott. Functorial polymorphism. *Theoretical Computer Science*, 70:35–64, 1990.

[BHA86] G. L. Burn, C. L. Hankin, and S. Abramsky. The theory and practice of strictness analysis for higher-order functions. *Science of Computer Programming*, 7:249–278, 1986.

[Bie92] G. M. Bierman. Type systems, linearity and functional languages (ohp slides). In G. Winskel, editor, *Proceedings of the CLICS Workshop, Aarhus University, March 1992*, pages 71–92, May 1992. Available as Aarhus University Technical Report DAIMI PB-397-I.

[BL92] G. L. Burn and D. Le Metayer. Cps-translation and the correctness of optimising compilers. Technical Report DoC92/20, Department of Computing, Imperial College, London, 1992.

[BMM90] K. Bruce, J. C. Mitchell, and A. R. Meyer. The semantics of second-order lambda calculus. In G. Huet, editor, *Logical Foundations of Functional Programming*, pages 213–272. Addison-Wesley, 1990.

[BMS80] R. M. Burstall, D. B. MacQueen, and D. T. Sanella. Hope: an experimental applicative language. Technical Report CSR-62-80, Department of Computer Science, University of Edinburgh, May 1980.

[Bur87] G. L. Burn. *Abstract Interpretation and the Parallel Evaluation of Functional Languages.* PhD thesis, Department of Computing, Imperial College, London, March 1987.

[Bur88] G. L. Burn. A shared memory parallel G-machine based on the evaluation transformer model of computation. In *Proceedings of the Workshop on the Implementation of Lazy Functional Languages, Aspenäs, Göteborg, Sweden*, 5–8 September 1988.

[Bur90a] G. L. Burn. A relationship between abstract interpretation and projection analysis (extended abstract). In *Conference Record of the Seventeenth ACM Symposium on Principles of Programming Languages,San Francisco, California.* ACM, 1990.

[Bur90b] G. L. Burn. Using projection analysis in compiling lazy functional programs. In *Proceedings of the 1990 ACM Conference on Lisp and Functional Programming, Nice.* ACM, June 1990.

[Bur91a] G. L. Burn. The abstract interpretation of higher-order functional languages: From properties to abstract domains. In *Proceedings of the 1991 Glasgow Functional Programming Workshop*, August 1991.

[Bur91b] G. L. Burn. *Lazy Functional Languages: Abstract Interpretation and Compilation.* Research Monographs in Parallel and Distributed Computing. MIT Press, Cambridge, Mass., 1991.

[BW88] R. Bird and P. Wadler. *Introduction to Functional Programming.* Prentice-Hall International, 1988.

[BW90] M. Barr and C. Wells. *Category Theory for Computing Science.* Prentice-Hall International, 1990.

[Car87] L. Cardelli. Basic polymorphic typechecking. *Science of Computer Programming*, 8(2):147–172, 1987.

[CC77] P. Cousot and R. Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Conference Record of the Fourth ACM Symposium on Principles of Programming Languages.* ACM, 1977.

[CC79] P. Cousot and R. Cousot. Systematic design of program analysis frameworks. In *Conference Record of the Sixth ACM Symposium on Principles of Programming Languages.* ACM, 1979.

[CDCV81] M. Coppo, M. Dezani-Ciancaglini, and B. Venneri. Functional characters of solvable terms. *Zeitschrift für Mathematische Logik und Grundlagen der Mathematik*, 27:45–58, 1981.

[Cro92] R. L. Crole. *Programming Metalogics with a Fixpoint Type.* PhD thesis, Computer Laboratory, University of Cambridge, February 1992. Available as technical report 247.

[dB72] N. G. de Bruijn. Lambda calculus notation with nameless dummies, a tool for automatic formula manipulation. *Indag. Math.*, 34:381–392, 1972.

[DM82] L. Damas and R. Milner. Principal types schemes for functional programs. In *Proceedings of the 9th ACM Symposium on Principles of Programming Languages*, 1982.

[DP90] B. A. Davey and H. A. Priestley. *Introduction to Lattices and Order.* Cambridge University Press, 1990.

[EM91] C. Ernoult and A. Mycroft. Uniform ideals and strictness analysis. In *Proceedings of ICALP 91.* Springer-Verlag, 1991.

[Fai82] J. Fairbairn. Ponder and its type system. Technical Report 31, Computer Laboratory, University of Cambridge, November 1982.

[Fai85] J. Fairbairn. *Design and Implementation of a Simple Typed Language Based on the Lambda-Calculus.* PhD thesis, Computer Laboratory, University of Cambridge, May 1985. Also Technical Report 75.

[FM88] Y.-C. Fuh and P. Mishra. Type inference with subtypes. In *Proceedings of the 1988 European Symposium on Programming*, volume 300 of *Lecture Notes in Computer Science.* Springer-Verlag, March 1988.

[FM89] Y.-C. Fuh and P. Mishra. Polymorphic subtype inference: Closing the theory–practice gap. In *Proceedings of TAPSOFT-89*, 1989.

[FW87] J. Fairbairn and S. Wray. TIM: A simple, lazy abstract machine to execute supercombinators. In G. Kahn, editor, *Proceedings of the 1987 Conference on Functional Langauges and Computer Architecture.* LNCS 274, Springer-Verlag, 1987.

[Gir87] J.-Y. Girard. Linear logic. *Theoretical Computer Science*, (50):1–102, 1987.

[GLT89] J.-Y. Girard, Y. Lafont, and P. Taylor. *Proofs and Types.* Cambridge Tracts in Theoretical Computer Science. Cambridge University Press, 1989.

[GMW79] M. J. Gordon, A. J. Milner, and C. P. Wadsworth. *Edinburgh LCF*, volume 78 of *Lecture Notes in Computer Science.* Springer-Verlag, 1979.

[Gom90] C. K. Gomard. Partial type inference for untyped functional programs (extended abstract). In *Proceedings of the 1990 ACM Conference on Lisp and Functional Programming, Nice.* ACM, June 1990.

[HBPJ86] C. L. Hankin, G. L. Burn, and S. L. Peyton Jones. A safe approach to parallel combinator reduction (extended abstract). In *Proceedings of ESOP 86*, number 213 in Lecture Notes in Computer Science. Springer-Verlag, March 1986.

[HF89] R. J. M. Hughes and A. B. Ferguson. An iterative powerdomain construction. In *Proceedings of the 1989 Glasgow Workshop on Functional Programming*, 1989.

[HH90] L. S. Hunt and C. L. Hankin. Fixed points and frontiers: A new perspective. *The Journal of Functional Programming*, 1(1), 1990.

[Hin69] J. R. Hindley. The principal type-scheme of an object in combinatory logic. *Transactions of the American Mathematical Society*, (146):29–60, 1969.

[HO93] M. Hyland and L. Ong. Dialogue games and innocent strategies: An approach to (intensional) full abstraction for PCF. Preliminary announcement. University of Cambridge, July 1993.

[How80] W. A. Howard. The formulae-as-types notion of construction. In J. R. Hindley and J. P. Seldin, editors, *To H. B. Curry: Essays on Combinatory Logic, Lambda Calculus and Formalism.* Academic Press, New York, 1980.

[Hug88] R. J. M. Hughes. Abstract interpretation of first-order polymorphic functions. In *Glasgow Workshop on Functional Programming*, August 1988.

[Hug89] R. J. M. Hughes. Why functional programming matters. *The Computer Journal*, 2(32):98–107, April 1989.

[Hun89] S. Hunt. Frontiers and open sets in abstract interpretation. In *Proceedings of the Conference on Functional Programming Languages and Computer Architecture*, September 1989.

[Hun90] S. Hunt. PERs generalise projections for strictness analysis. Technical Report DOC 14/90, Department of Computing, Imperial College, London, 1990.

[Hun91] S. Hunt. *Abstract Interpretation of Functional Languages: From Theory to Practice.* PhD thesis, Department of Computing, Imperial College, London, October 1991.

[HWA+90] P. Hudak, P. Wadler, Arvind, B. Boutel, J. Fairbairn, J. Fasel, K. Hammond, J. Hughes, T. Johnsson, R. Kieburtz, R. S. Nikhil, S. L. Peyton Jones, M. Reeve, D. Wise, and J. Young. Report on the functional programming language Haskell. Technical report, Department of Computing Science, Glasgow University, April 1990.

[Jen91] T. P. Jensen. Strictness analysis in logical form. In *Proceedings of the 1991 Conference on Functional Programming Languages and Computer Architecture*, 1991.

[Jen92a] T. P. Jensen. *Abstract Interpretation in Logical Form.* PhD thesis, Department of Computing, Imperial College, London, November 1992.

[Jen92b] T. P. Jensen. Disjunctive strictness analysis. In *Proceedings of the 7th Symposium on Logic in Computer Science.* Computer Society Press of the IEEE, 1992.

[JM81] N. D. Jones and S. S. Muchnick. Complexity of flow analysis, inductive assertion synthesis and a language due to Dijkstra. In N. D. Jones and S. S. Muchnick, editors, *Program Flow Analysis: Theory and Applications.* Prentice-Hall International, 1981.

[Joh82] P. T. Johnstone. *Stone Spaces.* Cambridge studies in advanced mathematics. Cambridge University Press, 1982.

[Joh84] T. Johnsson. Efficient compilation of lazy evaluation. In *Proceedings of the ACM Conference on Compiler Construction, Montreal*, June 1984.

[Kah88] G. Kahn. Natural semantics. In K. Fuchi and M. Nivat, editors, *Programming of Future Generation Computers*, pages 237–258. Elsevier Science Publishers B.V. North Holland, 1988.

[Kie87] R. Kieburtz. Abstract semantics. In S. Abramsky and C. L. Hankin, editors, *Abstract Interpretation of Declarative Languages*, chapter 7. Ellis-Horwood, 1987.

[KLB89] H. Kingdon, D. R. Lester, and G. L. Burn. The HDG-machine: A highly distributed graph reducer for a transputer network. Draft, February 1989.

[KM89] T.-M. Kuo and P. Mishra. Strictness analysis: A new perspective based on type inference. In *Proceedings of the 4th International Conference on Functional Programming Languages and Computer Architecture.* ACM, 1989.

[KMM91] P. C. Kanellakis, H. G. Mairson, and J. C. Mitchell. Unification and ML type reconstruction. In *Computational Logic, Essays in Honour of Alan Robinson.* MIT Press, Cambridge, Mass., 1991.

[Lau89] J. Launchbury. *Projection Factorisations in Partial Evaluation.* PhD thesis, Department of Computing, Glasgow University, November 1989.

[LS81] D. Lehmann and M. Smyth. Algebraic specification of data types: A synthetic approach. *Math. Systems Theory*, 14, 1981.

[Mil77] R. Milner. Fully abstract models of typed lambda calculi. *Theoretical Computer Science*, 4:1–22, 1977.

[Mil78] R. Milner. A theory of type polymorphism in programming. *Journal of Computer and Systems Sciences*, 17:348–375, 1978.

[Mit84] J. C. Mitchell. Coercion and type inference (summary). In *Proceedings of the 11th ACM Symposium on Principles of Programming Languages*, pages 175–185, January 1984.

[Mit90] J. C. Mitchell. Type systems for programming languages. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, chapter 8, pages 365–458. Elsevier Science Publishers, 1990.

[MJ85] A. Mycroft and N. D. Jones. A relational framework for abstract interpretation. In H. Ganzinger and N. D. Jones, editors, *Proceedings of the Workshop on Programs as Data Objects, Copenhagen*, volume 217 of *Lecture Notes in Computer Science*. Springer-Verlag, October 1985.

[MN89] U. Martin and T. Nipkow. Boolean ring unification – the story so far. *Journal of Symbolic Computation*, 7:275–293, 1989.

[Mog89] E. Moggi. Computational lambda-calculus and monads. In *Proceedings of the 4th Annual Symposium on Logic in Computer Science, Asiloomar, CA*, pages 14–23, 1989.

[MPS84] D. B. MacQueen, G. D. Plotkin, and R. Sethi. An ideal model for recursive polymorphic types. In *Proceedings of the 11th ACM Symposium on Principles of Programming Languages, Salt Lake City*, 1984.

[MS82] D. B. MacQueen and R. Sethi. A semantic model of types for applicative languages. In *Proceedings of the 1982 ACM Conference on Lisp and Functional Programming, Pittsburgh*, 1982.

[MS92] J. C. Mitchell and A. Scedrov. Notes on sconing and relators. Draft, November 1992.

[MTH90] R. Milner, M. Tofte, and R. Harper. *The Definition of Standard ML.* The MIT Press, 1990.

[Myc80] A. Mycroft. The theory and practice of transforming call-by-need into call-by-value. In *Proceedings of the 4th International Symposium on Programming*, number 83 in Lecture Notes in Computer Science, pages 269–281. Springer-Verlag, April 1980.

[Myc81] A. Mycroft. *Abstract Interpretation and Optimising Transformations for Applicative Programs.* PhD thesis, Department of Computer Science, University of Edinburgh, December 1981.

[Nie85] F. Nielson. Tensor products generalize the relational data flow analysis method. In *Proceedings of the 4th Hungarian Computer Science Conference*, pages 211–225, 1985.

[Nie86] F. Nielson. Strictness analysis and denotational abstract interpretation. Technical Report R 86-9, Institut for Elektroniske Systemer, Aalborg Universitetscenter, Denmark, August 1986.

[NN90] H. R. Nielson and F. Nielson. Context information for lazy code generation. In *LISP and Functional Programming*, June 1990.

[NPS90] B. Nordstrom, K. Petersson, and J. M. Smith. *Programming in Martin-Löf's Type Theory*, volume 7 of *Monographs in Computer Science.* Oxford University Press, 1990.

[Ong88] C.-H. L. Ong. *The Lazy Lambda Calculus: An Investigation into the Foundations of Functional Programming.* PhD thesis, Department of Computing, Imperial College, London, May 1988.

[PF92] W. K.-S. Phoa and M. P. Fourman. A proposed categorical semantics for pure ML. In *Proceedings of ICALP 92*, 1992.

[Pho92] W. K.-S. Phoa. A simple categorical model of ML polymorphism. Draft, 1992.

[Pie91] B. C. Pierce. *Programming with Intersection Types and Bounded Polymorphism.* PhD thesis, School of Computer Science, Carnegie Mellon University, December 1991. Available as CMU-CS-91-205.

[Pit92] A. M. Pitts. A coinduction principle for recursively defined domains. Technical Report 252, Computer Laboratory, University of Cambridge, April 1992.

[Pit93] A. M. Pitts. Computational adequacy via 'mixed' inductive definitions. Talk given at MFPS IX, New Orleans April 1993. Paper available by ftp from theory.doc.ic.ac.uk., July 1993.

[PJ87] S. L. Peyton Jones. *The Implementation of Functional Programming Languages.* Prentice Hall, 1987.

[PJCSH87] S. L. Peyton Jones, C. Clack, J. Salkild, and M. Hardie. GRIP — a high performance architecture for parallel graph reduction. In G. Kahn, editor, *Proceedings of the 1987 Conference on Functional Langauges and Computer Architecture*, volume 274 of *Lecture Notes in Computer Science.* Springer-Verlag, 1987.

[PJL91] S. L. Peyton Jones and J. Launchbury. Unboxed values as first class citizens in a non-strict functional language. In *Proceedings of the 5th International Symposium on Functional Programming Languages and Computer Architecture*, volume 523 of *Lecture Notes in Computer Science*. Springer-Verlag, 1991.

[Plo77] G. D. Plotkin. LCF considered as a programming language. *Theoretical Computer Science*, 5:223–255, 1977.

[Plo79] G. D. Plotkin. Lecture notes for a postgraduate course on domain theory incorporating 'The Pisa Notes' 79,81,82, 1979.

[Plo85] G. D. Plotkin. Types and partial functions. Lecture Notes, Computer Science Department, University of Edinburgh, 1985.

[Rey83] J. C. Reynolds. Types, abstraction and parametric polymorphism. *Information Processing '83*, pages 513–523, 1983.

[Saz76] V. Yu. Sazonov. Expressibility of functions in D. Scott's LCF language. *Algebra i Logika*, 15:308–330, 1976.

[SP82] M. B. Smyth and G. D. Plotkin. The category-theoretic solution of recursive domain equations. *SIAM Journal of Computing*, 11, 1982.

[Sto77] J. E. Stoy. *Denotational Semantics: The Scott-Strachey Approach to Programming Language Theory.* MIT Press, Cambridge, Mass., 1977.

[Sto88] A. Stoughton. *Fully Abstract Models of Programming Languages.* Research Notes in Theoretical Computer Science. Pitman, London, and Wiley, New York, 1988.

[Str67] C. Strachey. Fundamental concepts in programming languages. In *Notes for the International Summer School in Computer Programming, Copenhagen*, 1967.

[Ten91] R. D. Tennent. *Semantics of Programming Languages.* Prentice-Hall International, 1991.

[Tur85] D. A. Turner. Miranda – a non-strict functional language with polymorphic types. In J. P. Jouannaud, editor, *Proceedings of the Conference on Functional Programming Languages and Computer Architecture, Nancy*, volume 201 of *Lecture Notes in Computer Science*, pages 1–16. Springer-Verlag, 1985.

[Vic89] S. Vickers. *Topology Via Logic.* Cambridge Tracts in Theoretical Computer Science. Cambridge University Press, 1989.

[Wad71] C. P. Wadsworth. *Semantics and Pragmatics of the Lambda-Calculus.* PhD thesis, University of Oxford, 1971.

[Wad87] P. Wadler. Strictness analysis on non-flat domains (by abstract interpretation over finite domains). In S. Abramsky and C. L. Hankin, editors, *Abstract Interpretation of Declarative Languages*, chapter 12, pages 266–275. Ellis Horwood Ltd., 1987.

[Wad88] P. Wadler. Strictness analysis aids time analysis. In *Proceedings of the 15th ACM Symposium on Principles of Programming Languages*, January 1988.

[Wad89] P. Wadler. Theorems for free! In *Proceedings of the 4th International Symposium on Functional Programming Languages and Computer Architecture*, September 1989.

[Wad90] P. Wadler. Comprehending monads. In *Proceedings of the 1990 ACM Conference on Lisp and Functional Programming*, pages 61–78. ACM, June 1990.

[Wad91] P. Wadler. Is there a use for linear logic ? In *Proceedings of the ACM SIGPLAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation*, 1991.

[Wel93] J. B. Wells. Typability and type checking in the second-order λ-calculus are equivalent and undecidable (preliminary draft). Boston University, August 1993.

[WH87] P. Wadler and R. J. M. Hughes. Projections for strictness analysis. In G. Kahn, editor, *Functional Programming Languages and Computer Architecture*, volume 274 of *Lecture Notes in Computer Science*, pages 385–407. Springer-Verlag, September 1987.

[Win93] G. Winskel. *The Formal Semantics of Programming Languages. An Introduction.* MIT Press, 1993.

[Wri91a] D. A. Wright. An intensional type discipline. Technical Report R91-3, Department of Computer Science, University of Tasmania, July 1991.

[Wri91b] D. A. Wright. A new technique for strictness analysis. In *Theory and Practice of Software Development.* Springer-Verlag LNCS 494, April 1991.

[WW87] P. Watson and I. Watson. Evaluating functional programs on the FLAGSHIP machine. In G. Kahn, editor, *Proceedings of the 1987 Conference on Functional Languages and Computer Architecture*, volume 274 of *Lecture Notes in Computer Science.* Springer-Verlag, 1987.

[YH86] J. Young and P. Hudak. Finding fixpoints on function spaces. Technical Report YALEEU/DCS/RR-505, Yale University, Department of Computer Science, 1986.

[Wad87] P. Wadler. Strictness analysis on non-flat domains (by abstract interpretation over finite domains). In S. Abramsky and C. L. Hankin, editors, *Abstract Interpretation of Declarative Languages*, chapter 12, pages 266–275. Ellis Horwood Ltd., 1987.

[Wad88] P. Wadler. Strictness analysis aids time analysis. In *Proceedings of the 15th ACM Symposium on Principles of Programming Languages*, January 1988.

[Wad89] P. Wadler. Theorems for free! In *Proceedings of the 4th International Symposium on Functional Programming Languages and Computer Architecture*, September 1989.

[Wad90] P. Wadler. Comprehending monads. In *Proceedings of the 1990 ACM Conference on Lisp and Functional Programming*, pages 61–78. ACM, June 1990.

[Wad91] P. Wadler. Is there a use for linear logic ? In *Proceedings of the ACM SIGPLAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation*, 1991.

[Wel93] J. B. Wells. Typability and type checking in the second-order λ-calculus are equivalent and undecidable (preliminary draft). Boston University, August 1993.

[WH87] P. Wadler and R. J. M. Hughes. Projections for strictness analysis. In G. Kahn, editor, *Functional Programming Languages and Computer Architecture*, volume 274 of *Lecture Notes in Computer Science*, pages 385–407. Springer-Verlag, September 1987.

[Win93] G. Winskel. *The Formal Semantics of Programming Languages. An Introduction.* MIT Press, 1993.

[Wri91a] D. A. Wright. An intensional type discipline. Technical Report R91-3, Department of Computer Science, University of Tasmania, July 1991.

[Wri91b] D. A. Wright. A new technique for strictness analysis. In *Theory and Practice of Software Development.* Springer-Verlag LNCS 494, April 1991.

[WW87] P. Watson and I. Watson. Evaluating functional programs on the FLAGSHIP machine. In G. Kahn, editor, *Proceedings of the 1987 Conference on Functional Languages and Computer Architecture*, volume 274 of *Lecture Notes in Computer Science.* Springer-Verlag, 1987.

[YH86] J. Young and P. Hudak. Finding fixpoints on function spaces. Technical Report YALEEU/DCS/RR-505, Yale University, Department of Computer Science, 1986.

[Zha89] G. Q. Zhang. *Logics of Domains.* PhD thesis, Computer Laboratory, University of Cambridge, December 1989. Technical Report 185.