7 *Typed Operational Reasoning*

By Andrew Pitts

The aim of this chapter is to explain, by example, some methods for reasoning about equivalence of programs based directly upon a type system and an operational semantics for the programming language in question. We will concentrate on methods for reasoning about equivalence of representations of abstract data types. This provides an excellent example: it is easy to appreciate why such methods are useful and at the same time non-trivial problems have to be solved to get a sound reasoning principle in the presence of non-termination and recursion. Rather than just treat abstract data types, we will cover full existential types, using a programming language combining a pure fragment of ML (including records and recursive functions) with System F.

7.1 Introduction

As explained in TAPL Chapter 24, type systems involving existentially quantified type variables provide a useful foundation for explaining and relating various features of programming languages to do with information hiding. To establish the properties of such type-theoretic interpretations of information hiding requires a theory of semantic equivalence for expressions of existential type. Methods involving type-indexed families of *relations* between between expressions have proved very useful in this respect. Study of relational properties of typed calculi goes back to the 'logical relations' for simply typed lambda calculus in Plotkin (1973) and Statman (1985) and discussed in Chapter 6, and the notion of 'relational parametricity' for polymorphic types in Reynolds (1983). More relevant to the kind of example considered in this chapter is Mitchell's principle for establishing the denotational equivalence of programs involving higher-order functions and different implementations of an abstract datatype in terms of the existence of a 'simulation' relation between the implementations (Mitchell, 1991a). This principle was extended by Plotkin and Abadi (1993) to encompass all the (possibly impredicative) existential types of the Girard-Reynolds polymorphic lambda calculus.

One feature of these works is that they develop proof principles for *denotational* models of programming languages. The relevance of such principles to the operational behaviour of programs relies upon 'goodness of fit' results (some published, some not) connecting operational and denotational semantics. Another feature of the above works is that they do not treat the use of general recursive definitions; and so the languages considered are not Turing powerful. It is folklore that a proof principle for denotational equality at existential type, phrased in terms of the existence of certain simulation relations, is still valid in the presence of recursively defined functions of higher type, provided one imposes some 'admissibility' conditions on the notion of relation. In fact using techniques for defining operationally-based logical relations developed in Pitts (2000), we will see in this chapter that suitable admissibility conditions for relations and an associated proof principle for operational equivalence at existential type can be phrased directly, and quite simply, in terms of the syntax and operational semantics of a programming language combining existential types with recursively defined, higher-order functions. The programming language we work with combines a pure fragment of ML (including records and recursive functions) with the polymorphic lambda calculus of Girard (1972) and Reynolds (1974).

7.2 Overview

In order to get the most out of this chapter you should have some familiarity with TAPL Chapters 23 and 24. The material in this chapter is technically quite intricate (especially the definition and properties of the logical relation in §7.6) and it is easy to lose sight of the wood for the trees. So here is an overview of the chapter.

Equivalence of programs One application of formal semantics of programming languages is to give a mathematically precise definition of what it means for one program to be semantically equal to another. In this chapter we use operational semantics (§7.4.2) and discuss a notion of program equivalence called *contextual equivalence* (§7.5).

Extensionality principles In order to reason about program equivalence, it is useful to establish the validity of proof methods for it. The most basic method uses the congruence property—reasoning by 'replacing equals by equals'—which holds of contextual equivalence by construction. In §7.1 we discuss informally some methods for proving contextual equivalence of implementations of abstract datatypes. The discussion culminates with the Extensionality Principle 7.3.6. One goal of this chapter is to give a mathematically precise formulation of this principle and to establish its validity.

Logical relations The Extensionality Principle is phrased in terms of type-respecting *relations* between the terms of our example language. In order to formulate this principle precisely and then prove it we develop an alternative characterisation of contextual equivalence in terms of a certain 'logical relation' (§7.6). The combination of features in our language—higher-order recursive functions and fully impredicative polymorphic types—force us to use a form of logical relation with quite a difficult definition. Chapter 6 presents another use of logical relations with a simpler definition; as such, that chapter provides a useful warm-up for this one.

7.3 Motivating Examples

In this section we motivate the use of logical relations for reasoning about existential types by giving some examples.

To begin, let us recall the syntax for expressions involving existentially quantified type variables from TAPL Chapter 24. If `T` is a type expression and `X` is a type variable, then we write `{∃X,T}` for the corresponding existentially quantified type. Free occurrences of `X` in `T` become bound in this type expression. We write $[\mathtt{X} \mapsto \mathtt{S}]\mathtt{T}$ for the result of substituting a type `S` for all free occurrences of `X` in `T`, renaming bound type variables as necessary to avoid capture.[1] It `t` is a term of type $[\mathtt{X} \mapsto \mathtt{S}]\mathtt{T}$, then we can 'pack' the type `S` and the term `t` together to get a term

$$\texttt{\{*S,t\} as \{}\exists\texttt{X,T\}} \tag{7.1}$$

of the indicated existential type. To eliminate such terms we use the form

$$\texttt{let \{*X,x\}=}\mathtt{t}_1\texttt{ in }\mathtt{t}_2 \tag{7.2}$$

This is a binding construct: free occurrences of the type variable `X` and the value variable `x` in $\mathtt{t}_2$ become bound in the term. The typing of such terms goes as follows:

1. Throughout this chapter we will always identify expressions, be they types or terms, up to renaming of bound variables.

if t_1 has type $\{\exists X, T\}$ and t_2 has type T_2 when we assume the variable x has type T, then *provided* X *does not occur free in* T_2, we can conclude that the term in (7.2) has type T_2.

(Such rules are better presented symbolically, but we postpone doing that until we give a formal definition of the language we will be using, in the next section.) The italicised restriction on free occurrences of X in T_2 in the above rule is what distinguishes an existential type from a type-indexed dependent sum, where there is free access both to the type component as well as the term component of a 'packed' term: see Mitchell and Plotkin (1988), p. 474 *et seq* for a discussion of this point.

Since we wish to consider existential types in the context of an ML-like language, we adopt an eager strategy for evaluating expressions like (7.1) and (7.2). Thus to evaluate the first, one evaluates t to canonical form, v say, and returns the canonical form `{*S,v} as {∃X,T}`; to evaluate the second, one evaluates t_1 to canonical form, `{*S,v} as {∃X,T}` say, and then evaluates $[X \mapsto S][x \mapsto v]t_2$.

7.3.1 EXAMPLE: Consider the existentially quantified record type

```
type Counter = {∃X, {mk:X, inc:X→X, get:X→Int}}
```

where `Int` is a type of integers. Values of type `Counter` consist of some type together with values of the appropriate types implementing `mk`, `inc`, and `get`. For example

```
val counter₁ = {*Int, {mk  = 0,
                       inc = λx:Int.x+1,
                       get = λx:Int.x   } as Counter
```

and

```
val counter₂ = {*Int, {mk  = 0,
                        inc = λx:Int.x-1,
                        get = λx:Int.0-x } as Counter
```

are both values of type `Counter`. The terms

```
let {*X,x} = counter₁  in x.get(x.inc(x.mk))
let {*X,x} = counter₂ in x.get(x.inc(x.mk))
```

(where we use the syntax `r.f` for selecting field `f` of record `r`) are both terms of type `Int` which evaluate to `1`. By contrast, of the terms

```
let {*X,x} = counter₁  in x.get(x.inc(1))
let {*X,x} = counter₂ in x.get(x.inc(1))
```

the first evaluates to `2`, whereas the second evaluates to `0`; but in this case neither term is well-typed. Indeed, it is the case that *any* well-typed closed term involving occurrences of the term $\mathtt{counter}_1$ will exhibit precisely the same evaluation behaviour if we replace those occurrences by $\mathtt{counter}_2$. In other words, $\mathtt{counter}_1$ and $\mathtt{counter}_2$ are equivalent in the following sense. □

7.3.2 DEFINITION [CONTEXTUAL EQUIVALENCE, INFORMALLY]: We write

$$\mathtt{t}_1 =_{\mathrm{ctx}} \mathtt{t}_2 : \mathtt{T}$$

to indicate that two terms $\mathtt{t}_1$ and $\mathtt{t}_2$ of the same type `T` are *contextually equivalent*. By definition, this means that for all well-typed terms $\mathtt{t}[\mathtt{t}_1]$ containing instances of $\mathtt{t}_1$, if $\mathtt{t}[\mathtt{t}_2]$ is the term obtained by replacing those instances by $\mathtt{t}_2$, then $\mathtt{t}[\mathtt{t}_1]$ and $\mathtt{t}[\mathtt{t}_2]$ give exactly the same observable results when evaluated. □

This notion of program equivalence assumes we have already fixed upon a definition of the 'observable results' of evaluating terms. It also presupposes that the meaning of a well-typed term should only depend upon the final result (if any) of evaluating it. This is reasonable for deterministic and non-interactive programming even in the presence of computational effects like side-effecting state or raising exceptions, provided we include those effects as part of the observable results of evaluation. Certainly, contextual equivalence is a widely used notion of program equivalence in the literature and it is the one we adopt here.

For the terms in Example 7.3.1, it is the case that

$$\mathtt{counter}_1 =_{\mathrm{ctx}} \mathtt{counter}_2 : \mathtt{Counter} \tag{7.3}$$

but the quantification over all possible contexts $\mathtt{t}[-]$ in the definition of $=_{\mathrm{ctx}}$ makes a direct proof of this and similar facts rather difficult. Thus one is led to ask whether there are proof principles for contextual equivalence that make proving such equivalences at existential types more tractable. Since values `{*S,v} as {∃X,T}` of a given existential type `{∃X,T}` are specified by pairs of data `S` and `v`, as a first stab at such a proof principle one might try componentwise equivalence. Equivalence in the second component will of course mean contextual equivalence; but in the first component, where the expressions involved are types, what should equivalence mean? If we take it to mean syntactic identity, $=$, (which for us includes renaming of bound variables) we obtain the following proof principle.[2]

2. This and subsequent proof principles for `{∃X,T}` are called *extensionality* principles by analogy with the familiar extensionality principle for functions; it is a convenient terminology, but perhaps the analogy is a little stretched.

7.3.3 PRINCIPLE [EXTENSIONALITY FOR ∃-TYPES, VERSION I]: For each existential type $\mathtt{E} \stackrel{\text{def}}{=} \{\exists \mathtt{X},\mathtt{T}\}$, types $\mathtt{T}_1$, $\mathtt{T}_2$, and values $\mathtt{v}_1$, $\mathtt{v}_2$, if $\mathtt{T}_1 = \mathtt{T}_2$ and $\mathtt{v}_1 =_{\text{ctx}} \mathtt{v}_2 : [\mathtt{X} \mapsto \mathtt{T}_2]\mathtt{T}$, then $(\{{*}\mathtt{T}_1,\mathtt{v}_1\}\ \mathtt{as}\ \mathtt{E}) =_{\text{ctx}} (\{{*}\mathtt{T}_2,\mathtt{v}_2\}\ \mathtt{as}\ \mathtt{E}) : \{\exists \mathtt{X},\mathtt{T}\}$. □

The hypotheses of Principle 7.3.3 are far too strong for it to be very useful. For example, it cannot be used to prove (7.3), since in this case $\mathtt{T}_1 = \mathtt{Int} = \mathtt{T}_2$, but

```
val v₁ = {mk=0, inc=λx:Int.x+1, get=λx:Int.x}
```

and

```
val v₂ = {mk=0, inc=λx:Int.x-1, get=λx:Int.0-x}
```

are clearly not contextually equivalent values of the record type

```
{mk:Int,inc:Int→Int,get:Int→Int}
```

(for example, we get different integers when evaluating $\mathtt{t}[\mathtt{v}_1]$ and $\mathtt{t}[\mathtt{v}_2]$ when $\mathtt{t}[-]$ is $(-.\mathtt{inc})\,0$). However, they do become contextually equivalent if in the second term we use a variant of integers in which the roles of positive and negative are reversed. Such 'integers' are of course in bijection with the usual ones and this leads us to our second version of an extensionality principle for existential types—in which the use of syntactic identity as the notion of type equivalence is replaced by the more flexible one of *bijection*. A bijection $\mathtt{i} : \mathtt{T}_1 \cong \mathtt{T}_2$ means a closed term $\mathtt{i} : \mathtt{T}_1 \to \mathtt{T}_2$ for which there is a closed term $\mathtt{i}^{-1} : \mathtt{T}_2 \to \mathtt{T}_1$ which is a two-sided inverse up to contextual equivalence: $\mathtt{i}^{-1}(\mathtt{i}\ \mathtt{x}_1) =_{\text{ctx}} \mathtt{x}_1 : \mathtt{T}_1$ and $\mathtt{i}(\mathtt{i}^{-1}\ \mathtt{x}_2) =_{\text{ctx}} \mathtt{x}_2 : \mathtt{T}_2$.

7.3.4 PRINCIPLE [EXTENSIONALITY FOR ∃-TYPES, VERSION II]: For each existential type $\mathtt{E} \stackrel{\text{def}}{=} \{\exists \mathtt{X},\mathtt{T}\}$, types $\mathtt{T}_1$, $\mathtt{T}_2$, and values $\mathtt{v}_1$, $\mathtt{v}_2$, if there is a *bijection* $\mathtt{i} : \mathtt{T}_1 \cong \mathtt{T}_2$ such that $\mathtt{T}(\mathtt{i})\ \mathtt{v}_1 =_{\text{ctx}} \mathtt{v}_2 : [\mathtt{X} \mapsto \mathtt{T}_2]\mathtt{T}$, then

$$(\{{*}\mathtt{T}_1,\mathtt{v}_1\}\ \mathtt{as}\ \mathtt{E}) =_{\text{ctx}} (\{{*}\mathtt{T}_2,\mathtt{v}_2\}\ \mathtt{as}\ \mathtt{E}) : \{\exists \mathtt{X},\mathtt{T}\}.$$

In stating this principle we have used the notation $\mathtt{T}(\mathtt{i})$ for the 'action' of types $\mathtt{T}$ on bijections $\mathtt{i}$: given a type $\mathtt{T}$, possibly containing free occurrences of a type variable $\mathtt{X}$, one can define an induced bijection $\mathtt{T}(\mathtt{i}) : [\mathtt{X} \mapsto \mathtt{T}_1]\mathtt{T} \cong [\mathtt{X} \mapsto \mathtt{T}_2]\mathtt{T}$ (with inverse $\mathtt{T}(\mathtt{i}^{-1})$). For example, if $\mathtt{T}$ is the type

```
{mk:X, inc:X→X, get:X→Int}
```

then $\mathtt{T}(\mathtt{i})$ is

```
λx:{ mk:T₁, inc:T₁→T₁, get:T₁→Int}.
    { mk = i(x.mk),
      inc = λx₂:T₂.i(x.inc(i⁻¹ x₂)),
      get = λx₂:T₂.x.get(i⁻¹ x₂)) }
```

and $\mathtt{T(i^{-1})}$ is

$$\begin{aligned}&\lambda\mathtt{x}\!:\!\{\,\mathtt{mk}\!:\!\mathtt{T_2},\ \mathtt{inc}\!:\!\mathtt{T_2}\!\to\!\mathtt{T_2},\ \mathtt{get}\!:\!\mathtt{T_2}\!\to\!\mathtt{Int}\}.\\&\quad\{\,\mathtt{mk} = \mathtt{i^{-1}(x.mk)},\\&\quad\ \ \mathtt{inc} = \lambda\mathtt{x_1}\!:\!\mathtt{T_1}.\mathtt{i^{-1}(x.inc(i\ x_1))},\\&\quad\ \ \mathtt{get} = \lambda\mathtt{x_1}\!:\!\mathtt{T_1}.\mathtt{x.get(i\ x_1))}\ \}.\end{aligned}$$

(In general, if `T` is a simple type then the definition of `T(i)` and $\mathtt{T(i^{-1})}$ can be done by induction on the structure of `T`; for recursively defined types, the definition of the induced bijection is not so straightforward.) □

We can use this second version of the extensionality principle for existential types to prove the contextual equivalence in (7.3), using the bijection

$$\mathtt{i} \stackrel{\text{def}}{=} (\lambda\mathtt{x}\!:\!\mathtt{Int}.0\mathtt{-x}) : \mathtt{Int} \cong \mathtt{Int}.$$

This does indeed satisfy $\mathtt{T(i)}\ \mathtt{v_1} =_{\text{ctx}} \mathtt{v_2} : \mathtt{Int}$ when $\mathtt{v_1}$, $\mathtt{v_2}$, and `T` are defined as above. (Of course these contextual equivalences, and indeed the fact that this particular term `i` is a bijection, all require proof; but the methods developed in this chapter render this straightforward.) However, the use of bijections between types is still too restrictive for proving many common examples of contextual equivalence of abstract datatype implementations, such as the following.

7.3.5 EXAMPLE: Consider the following existentially quantified record type, where `Bool` is a type of booleans.

```
type Semaphore = {∃X, {bit:X, flip:X→X, read:X→Bool}}
```

The following terms have type `Semaphore`:

```
val semaphore₁ =
    {*Bool, {bit  = true
             flip = λx:Bool.not x,
             read = λx:Bool.x      } as Semaphore;
val semaphore₂ =
    {*Int,  {bit  = 1,
             flip = λx:Int.0-2*x,
             read = λx:Int.x >= 0} as Semaphore
```

There is no bijection $\mathtt{Bool} \cong \mathtt{Int}$, so one cannot use Principle 7.3.4 to prove

$$\mathtt{semaphore_1} =_{\text{ctx}} \mathtt{semaphore_2} : \mathtt{Semaphore}. \tag{7.4}$$

Nevertheless, this contextual equivalence does hold. An informal argument for this makes use of the following relation $\mathtt{r} : \mathtt{Bool} \leftrightarrow \mathtt{Int}$ between values of type $\mathtt{Bool}$ and of type $\mathtt{Int}$.

$$\begin{aligned} r \stackrel{\text{def}}{=} & \quad \{(\mathtt{true}, m) \mid m = (-2)^n \text{ for some even } n \geq 0\} \\ & \cup \ \{(\mathtt{false}, m) \mid m = (-2)^n \text{ for some odd } n \geq 0\}. \end{aligned}$$

Write $\mathtt{s}_i$ for the second component of $\mathtt{semaphore}_i$ ($i = 1, 2$). Then

- $\mathtt{s_1.bit}$ evaluates to $\mathtt{true}$; $\mathtt{s_2.bit}$ evaluates to $\mathtt{1}$; and $(\mathtt{true}, \mathtt{1}) \in r$;
- if $(\mathtt{t_1}, \mathtt{t_2}) \in r$, then $\mathtt{(s_1.flip)t_1}$ and $\mathtt{(s_2.flip)t_2}$ evaluate to a pair of values which are again r-related;
- if $(\mathtt{t_1}, \mathtt{t_2}) \in r$, then $\mathtt{(s_1.read)t_1}$ and $\mathtt{(s_2.read)t_2}$ evaluate to the same boolean value.

The informal argument for the contextual equivalence (7.4) goes as follows: "any context $\mathtt{t}[-]$ which is well-typed whenever its hole '$-$' is filled with a term of type $\mathtt{Semaphore}$ can only make use of a term placed in its hole by opening it as an abstract pair $\mathtt{\{X,x\}}$ and applying the methods $\mathtt{bit}$, $\mathtt{flip}$, and $\mathtt{read}$ in some combination; therefore the above observations about r are enough to show that $\mathtt{t}[\mathtt{semaphore}_1]$ and $\mathtt{t}[\mathtt{semaphore}_2]$ always have the same evaluation behaviour." □

The validity of this informal argument and in particular the assumptions it makes about the way a context can 'use' its hole are far from immediate and need formal justification. Leaving that for later, at least we can state the relational principle a bit more precisely.

7.3.6 PRINCIPLE [EXTENSIONALITY FOR $\exists$-TYPES, FINAL VERSION]: For each existential type $\mathtt{E} \stackrel{\text{def}}{=} \{\exists\mathtt{X},\mathtt{T}\}$, types $\mathtt{T_1}$, $\mathtt{T_2}$, and values $\mathtt{v_1}$, $\mathtt{v_2}$, if there is a *relation* $r : \mathtt{T_1} \leftrightarrow \mathtt{T_2}$ between terms of type $\mathtt{T_1}$ and of type $\mathtt{T_2}$, such that $(\mathtt{v_1}, \mathtt{v_2}) \in \mathtt{T}[r]$, then $(\{{*}\mathtt{T_1},\mathtt{v_1}\}\ \mathtt{as}\ \mathtt{E}) =_{\text{ctx}} (\{{*}\mathtt{T_2},\mathtt{v_2}\}\ \mathtt{as}\ \mathtt{E}) : \{\exists\mathtt{X},\mathtt{T}\}$. □

Evidently this principle presupposes the existence of an 'action' of types on term-relations that sends relations $r : \mathtt{T_1} \leftrightarrow \mathtt{T_2}$ to relations $\mathtt{T}[r] : [\mathtt{X} \mapsto \mathtt{T_1}]\mathtt{T} \leftrightarrow [\mathtt{X} \mapsto \mathtt{T_2}]\mathtt{T}$ and with certain other properties. It is the definition of this action that is at the heart of the matter. It has to be phrased with some care in order for the above extensionality principle to be valid for languages involving non-termination of evaluation (through the presence of fixpoint recursion for example). We will give a precise definition in §7.6 (Definition 7.6.9) for a language combining impredicative polymorphism with fixpoint recursion at the level of terms. How best to define such relational actions in the presence of recursion at the level of types is still a matter for research (see Exercise 7.8.1).

7.3.7 NOTE: Principle 7.3.4 generalises Principle 7.3.3, because if $\mathtt{T}_1 = \mathtt{T}_2$, then the identity function $\mathtt{i} \stackrel{\text{def}}{=} \lambda \mathtt{x}\!:\!\mathtt{T}_1\,.\,\mathtt{x}$ is a bijection $\mathtt{T}_1 \cong \mathtt{T}_2$ satisfying

$$(\mathtt{T}(\mathtt{i})\ \mathtt{v}) =_{\text{ctx}} \mathtt{v} \quad \text{(for any } \mathtt{v}\text{)}$$

so that $\mathtt{v}_1 =_{\text{ctx}} \mathtt{v}_2$ implies $(\mathtt{T}(\mathtt{i})\ \mathtt{v}_1) =_{\text{ctx}} \mathtt{v}_2$. Principle 7.3.6 generalises Principle 7.3.4, because each bijection $\mathtt{i} : \mathtt{T}_1 \cong \mathtt{T}_2$ can be replaced by its *graph*

$$\mathtt{r}_{\mathtt{i}} \stackrel{\text{def}}{=} \{(\mathtt{u}_1, \mathtt{u}_2) \mid \mathtt{i}\ \mathtt{u}_1 =_{\text{ctx}} \mathtt{u}_2\}$$

which in fact has the property that $(\mathtt{v}_1, \mathtt{v}_2) \in \mathtt{T}[\mathtt{r}_{\mathtt{i}}]$ if and only if $(\mathtt{T}(\mathtt{i})\ \mathtt{v}_1)$ is contextually equivalent to $\mathtt{v}_2$. □

As mentioned in the Introduction, Principle 7.3.6 is an operational generalisation of similar principles for the denotational semantics of abstract datatypes over simply typed lambda calculus (Mitchell, 1991a) and relationally parametric models of the polymorphic lambda calculus (Plotkin and Abadi, 1993). It permits many examples of contextual equivalence at existential types to be proved rather easily. Nevertheless, we will see in §7.7 that it is incomplete for the particular ML-like language we consider here, in the sense that $(\{{*}\mathtt{T}_1, \mathtt{v}_1\}\ \mathtt{as}\ \mathtt{E}) =_{\text{ctx}} (\{{*}\mathtt{T}_2, \mathtt{v}_2\}\ \mathtt{as}\ \mathtt{E}) : \{\exists \mathtt{X}, \mathtt{T}\}$ can hold even though there is no relation $\mathtt{r}$ for which $(\mathtt{v}_1, \mathtt{v}_2) \in \mathtt{T}[\mathtt{r}]$ holds (see Example 7.7.4).

7.4 The Language

In this section we define a small, ML-like programming language that we will use in the rest of the chapter. It combines Girard's *System F* (1972) (in other words, the *polymorphic lambda calculus* of Reynolds, 1974) with recursively defined functions, record types and ground types; in common with ML (Milner, Tofte, Harper, and MacQueen, 1997), evaluation order is strict (i.e. left-to-right, call-by-value). We will call the language F_{ML}. Its syntax and type system are specified in Figure 7-1 and its operational semantics in Figure 7-2.

7.4.1 Syntax

In Figure 7-1, `X` and `x` respectively range over disjoint countably infinite sets of *type variables* and *value variables*; `l` ranges over a countably infinite set of *field labels*; `c` ranges over the constants `true`, `false` and $\mathtt{n}$ (for $\mathtt{n} \in \mathbb{Z}$); `Gnd` is either the type of booleans `Bool` or the type of integers `Int`; and `op` ranges

Syntax

$\mathtt{t}$ ::=		*terms:*
	$\mathtt{v}$	*value*
	$\mathtt{if\ v\ then\ t\ else\ t}$	*conditional*
	$\mathtt{op(v_i}^{\ i\in 1..n}\mathtt{)}$	*operation*
	$\mathtt{v\ v}$	*application*
	$\mathtt{v.l}$	*projection*
	$\mathtt{v\ T}$	*type application*
	$\mathtt{let\ \{*X,x\}=v\ in\ t}$	*unpacking*
	$\mathtt{let\ x=t\ in\ t}$	*sequencing*
$\mathtt{v}$ ::=		*values:*
	$\mathtt{x}$	*value variable*
	$\mathtt{c}$	*constant*
	$\mathtt{fun\ x(x:T)=t:T}$	*recursive function*
	$\mathtt{\{l_i=v_i}^{\ i\in 1..n}\mathtt{\}}$	*record value*
	$\lambda\mathtt{X.v}$	*type abstraction*
	$\mathtt{\{*T,v\}\ as\ \{\exists X,T\}}$	*package value*
$\mathtt{T}$ ::=		*types:*
	$\mathtt{X}$	*type variable*
	$\mathtt{Gnd}$	*ground type*
	$\mathtt{T\to T}$	*function type*
	$\mathtt{\{l_i:T_i}^{\ i\in 1..n}\mathtt{\}}$	*record type*
	$\forall\mathtt{X.T}$	*universally quantified type*
	$\mathtt{\{\exists X,T\}}$	*existentially quantified type*
Γ ::=		*typing contexts:*
	$\emptyset$	*empty context*
	$\Gamma, \mathtt{x:T}$	*non-empty context*
	$\Gamma, \mathtt{X}$	*non-empty context*

Typing terms $\boxed{\Gamma \vdash \mathtt{t : T}}$

$$\frac{\mathtt{x:T} \in \Gamma}{\Gamma \vdash \mathtt{x : T}} \qquad \text{(T-VAR)}$$

$$\Gamma \vdash \mathtt{c} : \mathit{Typeof}(\mathtt{c}) \qquad \text{(T-CONST)}$$

$$\frac{\Gamma, \mathtt{f:T}, \mathtt{x:T_1} \vdash \mathtt{t : T_2} \quad \mathtt{T} = \mathtt{T_1\to T_2}}{\Gamma \vdash \mathtt{fun\ f(x:T_1)=t:T_2 : T}} \qquad \text{(T-FUN)}$$

$$\frac{(\Gamma \vdash \mathtt{v_i : T_i})^{\ i\in 1..n}}{\Gamma \vdash \mathtt{\{l_i=v_i}^{\ i\in 1..n}\mathtt{\} : \{l_i:T_i}^{\ i\in 1..n}\mathtt{\}}} \qquad \text{(T-RCD)}$$

$$\frac{\Gamma, \mathtt{X} \vdash \mathtt{v : T} \qquad \mathtt{X} \notin \mathit{ftv}(\Gamma)}{\Gamma \vdash \lambda\mathtt{X.v} : \forall\mathtt{X.T}} \qquad \text{(T-TABS)}$$

$$\frac{\Gamma \vdash \mathtt{v_1} : [\mathtt{X} \mapsto \mathtt{T_1}]\mathtt{T} \quad \mathtt{T'} = \mathtt{\{\exists X,T\}}}{\Gamma \vdash \mathtt{\{*T_1,v_1\}\ as\ T' : T'}} \qquad \text{(T-PACK)}$$

$$\frac{\Gamma \vdash \mathtt{v : Bool} \qquad \Gamma \vdash \mathtt{t_1 : T} \qquad \Gamma \vdash \mathtt{t_2 : T}}{\Gamma \vdash \mathtt{if\ v\ then\ t_1\ else\ t_2 : T}} \qquad \text{(T-IF)}$$

$$\frac{\mathtt{op:Gnd_1,\ldots,Gnd_n\to Gnd} \qquad (\Gamma \vdash \mathtt{v_i : Gnd_i})^{\ i\in 1..n}}{\Gamma \vdash \mathtt{op(v_i}^{\ i\in 1..n}\mathtt{) : Gnd}} \qquad \text{(T-OP)}$$

$$\frac{\Gamma \vdash \mathtt{v_1 : T_1\to T_2} \qquad \Gamma \vdash \mathtt{v_2 : T_1}}{\Gamma \vdash \mathtt{v_1\ v_2 : T_2}} \qquad \text{(T-APP)}$$

$$\frac{\Gamma \vdash \mathtt{v : \{l_i:T_i}^{\ i\in 1..n}\mathtt{\}}}{\Gamma \vdash \mathtt{v.l_j : T_j}} \qquad \text{(T-PROJ)}$$

$$\frac{\Gamma \vdash \mathtt{v} : \forall\mathtt{X.T}}{\Gamma \vdash \mathtt{v\ T_1} : [\mathtt{X} \mapsto \mathtt{T_1}]\mathtt{T}} \qquad \text{(T-TAPP)}$$

$$\frac{\Gamma, \mathtt{X}, \mathtt{x:T} \vdash \mathtt{t : T_1} \qquad \mathtt{X} \notin \mathit{ftv}(\Gamma, \mathtt{T_1}) \qquad \Gamma \vdash \mathtt{v : \{\exists X,T\}}}{\Gamma \vdash \mathtt{let\ \{*X,x\}=v\ in\ t : T_1}} \qquad \text{(T-UNPACK)}$$

$$\frac{\Gamma \vdash \mathtt{t_1 : T_1} \qquad \Gamma, \mathtt{x:T_1} \vdash \mathtt{t_2 : T_2}}{\Gamma \vdash \mathtt{let\ x=t_1\ in\ t_2 : T_2}} \qquad \text{(T-SEQ)}$$

Figure 7-1: F_{ML} **syntax and typing**

over a fixed collection of arithmetic and boolean operations (such as `+`, `=`, `not`, etc).

To simplify the definition of the language's operational semantics we employ the now quite common device of using a syntax for terms that is in a 'reduced' (or 'A-normal') form, with all sequential evaluation expressed via `let`-expressions. For example, the general form of (left-to-right, call-by-value) function application is coded by

$$\mathtt{t}_1\ \mathtt{t}_2 \stackrel{\text{def}}{=} \mathtt{let}\ \mathtt{x}_1\mathtt{=}\mathtt{t}_1\ \mathtt{in}\ (\mathtt{let}\ \mathtt{x}_2\mathtt{=}\mathtt{t}_2\ \mathtt{in}\ \mathtt{x}_1\ \mathtt{x}_2). \tag{7.5}$$

As a further simplification, function abstraction and recursive function declaration have been rolled into the one form $\mathtt{fun}\ \mathtt{f}\,(\mathtt{x}:\mathtt{T}_1) = \mathtt{t}:\mathtt{T}_2$, which corresponds to the expressions

	$\mathtt{let\ fun\ f\ (x:T_1) = t:T_2\ in\ f\ end}$	in Standard ML
or	$\mathtt{let\ rec\ f\ (x:T_1) = t:T_2\ in\ f}$	in Ocaml.

Ordinary function abstraction can be coded as

$$\lambda\mathtt{x}:\mathtt{T}_1.\mathtt{t} \stackrel{\text{def}}{=} \mathtt{fun}\ \mathtt{f}\,(\mathtt{x}:\mathtt{T}_1) = \mathtt{t}:\mathtt{T}_2 \tag{7.6}$$

where `f` does not occur freely in `t` (and $\mathtt{T}_2$ is the type of `t`, given `f` has type $\mathtt{T}_1 \to \mathtt{T}_2$ and `x` has type $\mathtt{T}_1$). In what follows we shall use the abbreviations (7.5) and (7.6) without further comment. We shall also use infix notation for application of constants that stand for arithmetic or boolean operators such as `+`, `=`, etc.

7.4.1 REMARK [VALUE-RESTRICTION]: Note that the operation $\lambda\mathtt{X}.(-)$ of polymorphic generalisation is restricted to apply only to values. This is a real restriction since for a non-value term `t`, one cannot define $\lambda\mathtt{X}.\mathtt{t}$ to be the term $\mathtt{let\ x{=}t\ in}\ \lambda\mathtt{X}.\mathtt{x}$, since the latter will in general be an ill-typed term. In an ML-like language $\lambda\mathtt{X}.\mathtt{t}$ is not yet fully evaluated if `t` is a non-value; and thus evaluation must go under type abstraction $\lambda\mathtt{X}.(-)$ and work on terms at types with free type variables. By imposing the restriction that $\lambda\mathtt{X}.\mathtt{t}$ is only well-formed when `t` is a value we can restrict attention to the evaluation of closed terms of *closed* type, simplifying the technical development. The restriction does not seem to affect the expressiveness of F_{ML} in practice and is comparable to the 'value-restriction' on `let`-bound polymorphism that occurs in the (1997) revision of Standard ML (Milner, Tofte, Harper, and MacQueen, 1997). However, this restriction does have an effect on the properties of F_{ML}. For example, with the restriction the type $\forall\mathtt{X}.\mathtt{X}$ contains no closed values (see Exercise 7.7.6); whereas without the restriction there are closed values of that type, such as $\lambda\mathtt{X}.\ (\mathtt{fun\ f\,(x{:}Bool) = f\ x : X})\ \mathtt{true}$.

Frame stack syntax

$\mathtt{S}$ ::=		*frame stacks:*
	Id	*nil stack*
	$\mathtt{S} \circ (\mathtt{x.t})$	*stack cons*

Typing frame stacks $\boxed{\Gamma \vdash \mathtt{S} : \mathtt{T}_1 \multimap \mathtt{T}_2}$

$$\Gamma \vdash \mathit{Id} : \mathtt{T} \multimap \mathtt{T} \qquad \text{(S-Nil)}$$

$$\frac{\Gamma, \mathtt{x}{:}\mathtt{T}_1 \vdash \mathtt{t} : \mathtt{T}_2 \qquad \Gamma \vdash \mathtt{S} : \mathtt{T}_2 \multimap \mathtt{T}_3}{\Gamma \vdash \mathtt{S} \circ (\mathtt{x.t}) : \mathtt{T}_1 \multimap \mathtt{T}_3} \qquad \text{(S-Cons)}$$

Primitive reductions $\boxed{\mathtt{t}_1 \rightsquigarrow \mathtt{t}_2}$

$$\mathtt{if\ true\ then\ t_1\ else\ t_2} \rightsquigarrow \mathtt{t}_1 \qquad \text{(R-IfTrue)}$$

$$\mathtt{if\ false\ then\ t_1\ else\ t_2} \rightsquigarrow \mathtt{t}_2 \qquad \text{(R-IfFalse)}$$

$$\frac{\text{the value of } \mathit{op}(\mathtt{c}_i\ {}^{i \in 1..n}) \text{ is } \mathtt{c}}{\mathtt{op}(\mathtt{c}_i\ {}^{i \in 1..n}) \rightsquigarrow \mathtt{c}} \qquad \text{(R-Op)}$$

$$\frac{\mathtt{v}_1 \text{ is } \mathtt{fun\ f(x{:}T_1)=t{:}T_2}}{\mathtt{v}_1\ \mathtt{v}_2 \rightsquigarrow [\mathtt{f} \mapsto \mathtt{v}_1][\mathtt{x} \mapsto \mathtt{v}_2]\mathtt{t}} \qquad \text{(R-AppAbs)}$$

$$\{\mathtt{l}_i{=}\mathtt{v}_i\ {}^{i \in 1..n}\}.\mathtt{j} \rightsquigarrow \mathtt{v}_j \qquad \text{(R-ProjRcd)}$$

$$(\lambda \mathtt{X.v})\,\mathtt{T} \rightsquigarrow [\mathtt{X} \mapsto \mathtt{T}]\mathtt{v} \qquad \text{(R-TappTabs)}$$

$$\frac{\mathtt{v} \text{ is } \{{*}\mathtt{T}_1, \mathtt{v}_1\}\ \mathtt{as}\ \{\exists \mathtt{X}, \mathtt{T}\}}{\mathtt{let}\ \{{*}\mathtt{X}, \mathtt{x}\}{=}\mathtt{v}\ \mathtt{in}\ \mathtt{t} \rightsquigarrow [\mathtt{X} \mapsto \mathtt{T}_1][\mathtt{x} \mapsto \mathtt{v}_1]\mathtt{t}} \qquad \text{(R-UnpackPack)}$$

Termination $\boxed{\langle \mathtt{S}, \mathtt{t}\rangle \downarrow \quad \mathit{and} \quad \mathtt{t}\downarrow}$

$$\langle \mathit{Id}, \mathtt{v}\rangle \downarrow \qquad \text{(S-NilVal)}$$

$$\frac{\langle \mathtt{S}, [\mathtt{x} \mapsto \mathtt{v}]\mathtt{t}\rangle \downarrow}{\langle \mathtt{S} \circ (\mathtt{x.t}), \mathtt{v}\rangle \downarrow} \qquad \text{(S-ConsVal)}$$

$$\frac{\langle \mathtt{S} \circ (\mathtt{x.t}_2), \mathtt{t}_1\rangle \downarrow}{\langle \mathtt{S}, \mathtt{let\ x{=}t_1\ in\ t_2}\rangle \downarrow} \qquad \text{(S-Seq)}$$

$$\frac{\mathtt{t}_1 \rightsquigarrow \mathtt{t}_2 \qquad \langle \mathtt{S}, \mathtt{t}_2\rangle \downarrow}{\langle \mathtt{S}, \mathtt{t}_1\rangle \downarrow} \qquad \text{(S-Red)}$$

$$\frac{\langle \mathit{Id}, \mathtt{t}\rangle \downarrow}{\mathtt{t}\downarrow} \qquad \text{(Term)}$$

Figure 7-2: F_{ML} **operational semantics**

The ‘emptiness’ of $\forall \mathtt{X.X}$ plays a role in the properties explored in Example 7.7.4 and Remark 7.7.7. □

7.4.2 Operational semantics

Although we do not do so, the operational semantics of F_{ML} could be specified in the style of the Definition of Standard ML (Milner, Tofte, Harper, and MacQueen, 1997) as a syntax-directed, inductively defined relation between terms and values.[3] Here we are interested primarily in the notion of contextual equivalence (Definition 7.3.2) that this evaluation relation determines by observing the results of evaluating terms in context. Because evaluation in

3. That Definition uses environments assigning values to value variables. For reasons of technical convenience we eliminate the use of environments by substituting them into the term and only considering the evaluation relation between *closed* terms and values.

Binding constructs

$\texttt{let}\ \{\texttt{*X,x}\}\texttt{=v in}\ (-)$
$\texttt{let x=t in}\ (-)$
$\texttt{fun f(x:T}_1\texttt{)=}(-\texttt{:T}_2)$
$\lambda\texttt{X.}(-)$
$\forall\texttt{X.}(-)$
$\{\exists\texttt{X,}(-)\}$
$\mathrm{S}\circ(\texttt{x.}(-))$

We identify expressions up to renaming of bound value and type variables.

Notation for free variable sets

$\boxed{ftv(\mathtt{E})}$ is the finite set of free type variables of the expression E (a type, a term, or a frame stack);

$\boxed{fv(\mathtt{E})}$ is the finite set of free value variables of an expression E (a term, or a frame stack, but not a type, since types do not contain occurrences of value variables).

Closed types, terms and frame stacks

A type T is *closed* if $ftv(\mathtt{T}) = \emptyset$.

A term or frame stack E is *closed* if $fv(\mathtt{E}) = \emptyset$ (even if $ftv(\mathtt{E}) \neq \emptyset$).

Notation for substitution

$\boxed{[\mathtt{X} \mapsto \mathtt{T}]\mathtt{E}}$ denotes the result of capture-avoiding substitution of a type T for all free occurrences of a type variable X in E (a type, a term, or a frame stack);

$\boxed{[\mathtt{x} \mapsto \mathtt{v}]\mathtt{E}}$ denotes the result of capture-avoiding substitution of a value v for all free occurrences of the value variable x in a term or frame stack E.

(Note that as their name suggests, value variables stand for unknown *values*—the substitution of a non-value term for a variable makes no sense syntactically, in that it may result in an ill-formed expression.)

Figure 7-3: Binding, free variables and substitution

F_{ML} is strict and the language has a sufficiently expressive collection of constructs for deconstructing values, it turns out that the notion of contextual equivalence is not affected much by the choice of what to observe of evaluation. Most reasonable choices give rise to the same equivalence as the one we adopt (see Exercise 7.5.10 below), which is based upon observing *termination*: whether or not a term evaluates to some value, we care not which. So instead of defining the relation of evaluation between terms and values, we proceed directly to a definition of the termination relation, $\mathtt{t}\downarrow$, for F_{ML}. This is given in Figure 7-2, using an auxiliary notion of *frame stack*. (The conventions and notations used in Figure 7-2 in connection with binding, free variables and substitution are summarised in Figure 7-3.)

Frame stacks are finite lists of individual ‘evaluation frames’. They provide a convenient syntax for the notion of *evaluation context* $\mathtt{E}[-]$ (Felleisen and Hieb, 1992; Wright and Felleisen, 1994). Every closed term can be decomposed uniquely as $\mathtt{E}[\mathtt{t}]$ where the evaluation context $\mathtt{E}[-]$ is a context with a unique hole $(-)$ occurring in the place where the next step of evaluation (called a *primitive reduction* in Figure 7-2), if any, will take place. With F_{ML}'s reduced syntax, such evaluation contexts turn out to be just nested sequences of the `let`-construct

$$\mathtt{E}[-] = \mathtt{let}\ \mathtt{x}_1\mathtt{=(...(let}\ \mathtt{x}_n\mathtt{=}(-)\ \mathtt{in}\ \mathtt{t}_n\mathtt{)...)}\ \mathtt{in}\ \mathtt{t}_1.$$

The corresponding frame stack

$$S = Id \circ (\mathtt{x}_1\mathtt{.t}_1) \circ \cdots \circ (\mathtt{x}_n\mathtt{.t}_n)$$

records this sequence as a list of *evaluation frames*, $\mathtt{x}_i\mathtt{.t}_i$ (with free occurrences of $\mathtt{x}_i$ in $\mathtt{t}_i$ being bound in $\mathtt{x}_i\mathtt{.t}_i$). Under this correspondence it can be shown that $\mathtt{E}[\mathtt{t}]$ evaluates to some value in the standard evaluation-style (or 'big-step') structural operational semantics if and only if $\langle S, \mathtt{t}\rangle\!\downarrow$ holds, for the relation $\langle -, -\rangle\!\downarrow$ defined in Figure 7-2. Not only does the use of frame stacks enable a conveniently syntax-directed inductive definition of termination, but also frame stacks play a big role in §7.6 when defining the logical relation that we use to establish properties of F_{ML} contextual equivalence.

7.4.2 EXERCISE [RECOMMENDED, ★★]: Suppose we define a relation $\langle S_1, \mathtt{t}_1\rangle \longrightarrow \langle S_2, \mathtt{t}_2\rangle$ by cases according to the structure of the term $\mathtt{t}_1$ and the frame stack S_1, as follows:

- $\langle S \circ (\mathtt{x.t}), \mathtt{v}\rangle \longrightarrow \langle S, [\mathtt{x} \mapsto \mathtt{v}]\mathtt{t}\rangle$
- $\langle S, \mathtt{let}\ \mathtt{x=t}_1\ \mathtt{in}\ \mathtt{t}_2\rangle \longrightarrow \langle S \circ (\mathtt{x.t}_2), \mathtt{t}_1\rangle$
- $\langle S, \mathtt{t}_1\rangle \longrightarrow \langle S, \mathtt{t}_2\rangle$, if $\mathtt{t}_1 \leadsto \mathtt{t}_2$.

Show that

$$\langle S'@S, \mathtt{t}\rangle\!\downarrow \quad \Leftrightarrow \quad (\exists \mathtt{v})\ \langle S, \mathtt{t}\rangle \longrightarrow^* \langle Id, \mathtt{v}\rangle\ \&\ \langle S', \mathtt{v}\rangle\!\downarrow \tag{7.7}$$

where $\longrightarrow^*$ denotes the reflexive-transitive closure of the $\longrightarrow$ relation, and $S'@S$ is the frame stack obtained by appending the two lists of evaluation frames S' and S. Deduce that $\mathtt{t}\!\downarrow$ holds if and only if there is some value $\mathtt{v}$ with $\langle Id, \mathtt{t}\rangle \longrightarrow^* \langle Id, \mathtt{v}\rangle$. □

7.4.3 Typing

We will consider the termination relation only for frame stacks and terms that are *well-typed*. A term $\mathtt{t}$ is well-typed with respect to a particular typing context Γ if a typing judgement

$$\Gamma \vdash \mathtt{t} : \mathtt{T} \tag{7.8}$$

can be derived for some type $\mathtt{T}$ using the rules in Figure 7-1. We identify typing contexts Γ up to rearranging their constituent hypotheses ('$\mathtt{X}$' or '$\mathtt{x} : \mathtt{X}$') and eliminating duplicates. Thus a typical typing context looks like

$$\Gamma = \mathtt{X}_1, \ldots, \mathtt{X}_m, \mathtt{x}_1 : \mathtt{T}_1, \ldots, \mathtt{x}_n : \mathtt{T}_n$$

where the type variables $\mathtt{X}_i$ and the value variables $\mathtt{x}_j$ are all distinct (and $m = 0$ or $n = 0$ is allowed). The typing judgements that are derivable from the rules all have the property that the free type variables of $\mathtt{T}$ and each $\mathtt{T}_j$ occur in the set $\{\mathtt{X}_1, \ldots, \mathtt{X}_m\}$, and the free value variables of $\mathtt{t}$ occur in the set $\{\mathtt{x}_1, \ldots, \mathtt{x}_n\}$. This is ensured by including some explicit side-conditions about free variable occurrences in the typing rules (T-ABS) and (T-UNPACK). In TAPL Chapters 23 and 24 such side-conditions are implicit, being subsumed by extra well-formedness conditions for typing judgements. Also, we have chosen to include sufficient explicit type information in terms to ensure that for any given Γ and $\mathtt{t}$, there is at most one $\mathtt{T}$ for which (7.8) holds. Apart from such minor differences, the rules in Figure 7-1 for inductively generating the valid F_{ML} typing judgements are all quite standard.

The judgement for typing frame stacks takes the form

$$\Gamma \vdash S : \mathtt{T}_1 \multimap \mathtt{T}_2 \tag{7.9}$$

where, in terms of the evaluation context corresponding to S, $\mathtt{T}_2$ is the overall type of the context, given that $\mathtt{T}_1$ is the type of the hole. The rules for generating this judgement are given in Figure 7-2. Unlike for terms, we have not included explicit type information in the syntax of frame stacks; for example, *Id* is not tagged with a type. However, it is not hard to see that, given Γ, S, and $\mathtt{T}_1$, there is at most one $\mathtt{T}_2$ for which (7.9) holds. This property is enough for our purposes, since the argument type of a frame stack will always be supplied in any particular situation in which we use it.

7.4.3 EXERCISE [★, ↛]: Write $\Gamma \vdash \langle S, \mathtt{t} \rangle : \mathtt{T}$ to mean that $\Gamma \vdash S : \mathtt{T}' \multimap \mathtt{T}$ and $\Gamma \vdash \mathtt{t} : \mathtt{T}'$ hold for some type $\mathtt{T}'$. Using the relation $\longrightarrow$ from Exercise 7.4.2, show that if $\emptyset \vdash \langle S_1, \mathtt{t}_1 \rangle : \mathtt{T}$ and $\langle S_1, \mathtt{t}_1 \rangle \longrightarrow \langle S_2, \mathtt{t}_2 \rangle$, then $\emptyset \vdash \langle S_2, \mathtt{t}_2 \rangle : \mathtt{T}$. □

7.4.4 Unwinding recursive functions

In what follows we will need a finiteness property of recursively defined functions with respect to the termination relation. This *unwinding property*, as it is called, is a syntactic analogue of the fact that the denotation of a recursively defined function is constructed as the least upper bound (*lub*) of finite approximations obtained by successively unfolding its definition starting with the *bottom* denotation, i.e. the totally undefined partial function. This gives rise to the useful principle of *Scott induction* in denotational semantics: given an *admissible* property of denotations, i.e. one closed under the formation of lubs of increasing chains, to show that it holds of the denotation of recursively defined data it suffices to show that it holds of bottom and is closed under application of the function that defines the data as a fixed point. Here we use a syntactic analogue of Scott induction for recursively defined functions, $\mathtt{fun\ f(x:T_1) = u:T_2}$, in order to prove the 'fundamental property' (Lemma 7.6.17) of the logical relation constructed in §7.6.

The proof of the unwinding property that we give here is made easier by our syntax-directed definition of termination using frame stacks. For statements and proofs of similar properties see for example: Mason et al. (1996), Section 4.3, Pitts and Stark (1998), Theorem 3.2, Birkedal and Harper (1999), Section 3.1, and Lassen (1998), Section 4.5.

7.4.4 THEOREM [UNWINDING]: Given any closed recursive function value $\mathtt{F} \stackrel{\text{def}}{=} \mathtt{fun\ f(x:T_1)=u:T_2}$, define the following abbreviations[4] :

$$\begin{aligned} \mathtt{F}_0 &\stackrel{\text{def}}{=} \mathtt{fun\ f(x:T_1) = (f\ x) : T_2} \\ \mathtt{F}_{n+1} &\stackrel{\text{def}}{=} \mathtt{fun\ f(x:T_1) = [f \mapsto F_n]u : T_2} \end{aligned}$$

Thus $\mathtt{F}_0$ is a closed function value describing a function of type $\mathtt{T_1 \to T_2}$ that diverges when applied to any argument, and the $\mathtt{F}_n$ are obtained from this by repeatedly substituting for the the value variable $\mathtt{f}$ in the body $\mathtt{u}$ of the original function value $\mathtt{F}$. Then for all terms $\mathtt{t}$ containing at most $\mathtt{f}$ free we have $[\mathtt{f} \mapsto \mathtt{F}]\mathtt{t}\downarrow$ if and only if $(\exists n)\ [\mathtt{f} \mapsto \mathtt{F}_n]\mathtt{t}\downarrow$. □

Proof: By definition of the relation $\mathtt{t}\downarrow$ in terms of the relation $\langle S, \mathtt{t}\rangle\downarrow$ (via rule (TERM) in Figure 7-2), it suffices to prove the more general property that for all terms $\mathtt{t}$ and frame stacks S (containing at most $\mathtt{f}$ free) we have

$$\langle [\mathtt{f} \mapsto \mathtt{F}]S, [\mathtt{f} \mapsto \mathtt{F}]\mathtt{t}\rangle\downarrow \;\Leftrightarrow\; (\exists n)\ \langle [\mathtt{f} \mapsto \mathtt{F}_n]S, [\mathtt{f} \mapsto \mathtt{F}_n]\mathtt{t}\rangle\downarrow \tag{7.10}$$

The proof of (7.10) is via a series of straightforward, if somewhat tedious, inductions that we leave as an exercise. □

7.4.5 EXERCISE [★★★, ↛]: This exercise leads you through a proof of (7.10). First prove that

$$\langle [\mathtt{f} \mapsto \mathtt{F}_n]S, [\mathtt{f} \mapsto \mathtt{F}_n]\mathtt{t}\rangle\downarrow \;\Rightarrow\; \langle [\mathtt{f} \mapsto \mathtt{F}]S, [\mathtt{f} \mapsto \mathtt{F}]\mathtt{t}\rangle\downarrow \tag{7.11}$$

4. Note that in the definition of $\mathtt{F}_{n+1}$, the outer binding instance of $\mathtt{f}$ is a dummy, since $\mathtt{f}$ does not occur free in $[\mathtt{f} \mapsto \mathtt{F}_n]\mathtt{u}$.

holds for all $\mathtt{n}$, S and $\mathtt{t}$ by induction on the derivation of $\langle[\mathtt{f} \mapsto \mathtt{F}_\mathtt{n}]S, [\mathtt{f} \mapsto \mathtt{F}_\mathtt{n}]\mathtt{t}\rangle\downarrow$ from the rules in Figure 7-2. Conversely show that

$$\langle[\mathtt{f} \mapsto \mathtt{F}]S, [\mathtt{f} \mapsto \mathtt{F}]\mathtt{t}\rangle\downarrow \;\Rightarrow\; (\exists \mathtt{n})\; \langle[\mathtt{f} \mapsto \mathtt{F}_\mathtt{n}]S, [\mathtt{f} \mapsto \mathtt{F}_\mathtt{n}]\mathtt{t}\rangle\downarrow \tag{7.12}$$

holds for all S and $\mathtt{t}$, by induction on the derivation of $\langle[\mathtt{f} \mapsto \mathtt{F}]S, [\mathtt{f} \mapsto \mathtt{F}]\mathtt{t}\rangle\downarrow$ from the rules. To do this, you will first need to prove by induction on $\mathtt{n}$ that

$$\langle[\mathtt{f} \mapsto \mathtt{F}_\mathtt{n}]S, [\mathtt{f} \mapsto \mathtt{F}_\mathtt{n}]\mathtt{t}\rangle\downarrow \;\Rightarrow\; \langle[\mathtt{f} \mapsto \mathtt{F}_{\mathtt{n}+1}]S, [\mathtt{f} \mapsto \mathtt{F}_{\mathtt{n}+1}]\mathtt{t}\rangle\downarrow \tag{7.13}$$

holds for all $\mathtt{n}$, S and $\mathtt{t}$; the base case $\mathtt{n} = 0$ involves yet another induction, this time over the derivation of $\langle[\mathtt{f} \mapsto \mathtt{F}_0]S, [\mathtt{f} \mapsto \mathtt{F}_0]\mathtt{t}\rangle\downarrow$ from the rules. □

7.5 Contextual Equivalence

Definition 7.3.2 gave an informal definition of the notion of *contextual equivalence* that applies to any (typed) programming language. In giving a precise definition of this notion for the F_{ML} language we will take the more abstract, relational approach of Gordon (1998) and Lassen (1998) that avoids the explicit use of program contexts $\mathtt{t}[-]$ in favour of congruence relations. For one thing, program contexts are an inconveniently concrete notion, because substitution of terms $\mathtt{t}'$ for the hole '$-$' in a context $\mathtt{t}[-]$ to produce a term $\mathtt{t}[\mathtt{t}']$ may involve the capture of free variables in $\mathtt{t}'$ by binders in $\mathtt{t}[-]$. For example, when we replace the hole '$-$' in the context `fun f(x:T) = f` $[-]$ by the term `f x`, its free value variables are captured by the `fun`-binder. Consequently, contexts have to be treated more concretely than terms since renaming their bound variables may not preserve their meaning. For example, if we identified `fun f(x:T) = f` $[-]$ with `fun g(x:T) = g` $[-]$ (where `f` and `g` are distinct value variables), then we should have to identify the results of filling the hole with `f x`, that is, we should have to identify the syntactically unequal terms `fun f(x:T) = f(f x)` and `fun g(x:T) = g(f x)`. But more than this, the abstract treatment of contextual equivalence that we use focuses attention upon the key features of this kind of program equality, namely that it is a congruence and is 'adequate' for observing termination. In a nutshell, we will define contextual equivalence to be the largest type-respecting congruence relation between F_{ML} terms that is adequate for observing termination.

7.5.1 DEFINITION: A *type-respecting binary relation* between F_{ML} terms is a set R of quadruples $(\Gamma, \mathtt{t}, \mathtt{t}', \mathtt{T})$, each consisting of a typing context, two terms and a type satisfying $\Gamma \vdash \mathtt{t} : \mathtt{T}$ and $\Gamma \vdash \mathtt{t}' : \mathtt{T}$. Figure 7-4 defines the properties of *reflexivity*, *symmetry*, *transitivity*, *substitutivity*, and *compatibility* for such relations; R has one of these properties if it is closed under the axioms and rules under the corresponding heading in the figure. In these figures, and elsewhere, we write $\Gamma \vdash \mathtt{t}\ R\ \mathtt{t}' : \mathtt{T}$ instead of $(\Gamma, \mathtt{t}, \mathtt{t}', \mathtt{T}) \in R$. We say that R is

- an *equivalence relation* if it has the reflexivity, symmetry and transitivity properties;
- a *congruence relation* if it is an equivalence relation with the *substitutivity* and *compatibility* properties;
- *adequate* (for the termination relation $\downarrow$ defined in Figure 7-2) if whenever $\emptyset \vdash \mathtt{t}\ R\ \mathtt{t}' : \mathtt{T}$ holds, then $\mathtt{t}\downarrow$ holds if and only if $\mathtt{t}'\downarrow$ does. □

7.5.2 DEFINITION: We will need to use the following constructions on type-respecting binary relations.

(i) The *identity* relation is $Id \stackrel{\text{def}}{=} \{(\Gamma, \mathtt{t}, \mathtt{t}, \mathtt{T}) \mid \Gamma \vdash \mathtt{t} : \mathtt{T}\}$.

(ii) The *reciprocal* of the relation R is $R^{op} \stackrel{\text{def}}{=} \{(\Gamma, \mathtt{t}', \mathtt{t}, \mathtt{T}) \mid \Gamma \vdash \mathtt{t}\ R\ \mathtt{t}' : \mathtt{T}\}$.

(iii) The *composition* of relations R_1 and R_2 is
$R_1 \circ R_2 \stackrel{\text{def}}{=} \{(\Gamma, \mathtt{t}, \mathtt{t}'', \mathtt{T}) \mid \exists \mathtt{t}'.\ \Gamma \vdash \mathtt{t}\ R_1\ \mathtt{t}' : \mathtt{T}\ \&\ \Gamma \vdash \mathtt{t}'\ R_2\ \mathtt{t}'' : \mathtt{T}\}$.

(iv) The *transitive closure* of the relation R is the countable union $R^+ \stackrel{\text{def}}{=} \bigcup_{i \in \mathbb{N}} R_i$, where $R_0 = R$ and $R_{i+1} = R \circ R_i$.

(v) The *open extension* of the relation R is denoted R° and consists of all quadruples $(\Gamma, \mathtt{t}, \mathtt{t}', \mathtt{T})$ such that $\emptyset \vdash \sigma(\mathtt{t})\ R\ \sigma(\mathtt{t}') : \sigma(\mathtt{T})$ holds for all Γ-closing substitutions σ. If $\Gamma = \mathtt{X}_1, \ldots, \mathtt{X}_m, \mathtt{x}_1 : \mathtt{T}_1, \ldots, \mathtt{x}_n : \mathtt{T}_n$, then a Γ-*closing substitution* is given by a function $[\mathtt{X}_i \mapsto \mathtt{T}_i \mid i = 1..m]$ mapping the type variables $\mathtt{X}_i$ to closed types $\mathtt{T}_i$ and by a function $[\mathtt{x}_j \mapsto \mathtt{v}_j \mid j = 1..n]$ mapping the value variables $\mathtt{x}_j$ to closed values $\mathtt{v}_j$ of appropriate type, namely satisfying $\emptyset \vdash \mathtt{v}_j : [\mathtt{X}_i \mapsto \mathtt{T}_i \mid i = 1..m]\mathtt{T}_j$.

(Note that R° only depends on the quadruples of the form $(\emptyset, \mathtt{t}, \mathtt{t}', \mathtt{T})$ in R.) □

We wish to define contextual equivalence to be the largest adequate congruence relation, but it is not immediately clear why a largest such relation exists. Therefore we give a theorem rather than a definition.

7.5.3 THEOREM [F_{ML} CONTEXTUAL EQUIVALENCE, $=_{\mathrm{ctx}}$]: There is a largest type-respecting binary relation between F_{ML} terms that is a congruence and adequate. We call it *contextual equivalence* and write it $=_{\mathrm{ctx}}$. □

Reflexivity

$$\frac{\Gamma \vdash \mathtt{t} : \mathtt{T}}{\Gamma \vdash \mathtt{t} \; R \; \mathtt{t} : \mathtt{T}}$$

Symmetry

$$\frac{\Gamma \vdash \mathtt{t} \; R \; \mathtt{t}' : \mathtt{T}}{\Gamma \vdash \mathtt{t}' \; R \; \mathtt{t} : \mathtt{T}}$$

Transitivity

$$\frac{\Gamma \vdash \mathtt{t} \; R \; \mathtt{t}' : \mathtt{T} \qquad \Gamma \vdash \mathtt{t}' \; R \; \mathtt{t}'' : \mathtt{T}}{\Gamma \vdash \mathtt{t} \; R \; \mathtt{t}'' : \mathtt{T}}$$

Substitutivity

$$\frac{\Gamma \vdash \mathtt{v} \; R \; \mathtt{v}' : \mathtt{T_1} \qquad \Gamma, \mathtt{x} : \mathtt{T_1} \vdash \mathtt{t} \; R \; \mathtt{t}' : \mathtt{T_2}}{\Gamma \vdash [\mathtt{x} \mapsto \mathtt{v}]\mathtt{t} \; R \; [\mathtt{x} \mapsto \mathtt{v}']\mathtt{t}' : \mathtt{T_2}}$$

$$\frac{\Gamma, \mathtt{X} \vdash \mathtt{t} \; R \; \mathtt{t}' : \mathtt{T}}{\Gamma \vdash [\mathtt{X} \mapsto \mathtt{T_1}]\mathtt{t} \; R \; [\mathtt{X} \mapsto \mathtt{T_1}]\mathtt{t}' : [\mathtt{X} \mapsto \mathtt{T_1}]\mathtt{T}}$$

Compatibility

$$\frac{(\mathtt{x}\mathtt{:}\mathtt{T}) \in \Gamma}{\Gamma \vdash \mathtt{x} \; R \; \mathtt{x} : \mathtt{T}}$$

$$\Gamma \vdash \mathtt{c} \; R \; \mathtt{c} : \mathit{Typeof}(\mathtt{c})$$

$$\frac{\Gamma, \mathtt{f}\mathtt{:}\mathtt{T_1} \to \mathtt{T_2}, \mathtt{x}\mathtt{:}\mathtt{T_1} \vdash \mathtt{t} \; R \; \mathtt{t}' : \mathtt{T_2}}{\begin{array}{c}\Gamma \vdash \mathtt{fun\ f(x:T_1)=t:T_2} \; R \\ \mathtt{fun\ f(x:T_1)=t':T_2} : \mathtt{T_1} \to \mathtt{T_2}\end{array}}$$

$$\frac{(\Gamma \vdash \mathtt{v_i} \; R \; \mathtt{v}'_{\mathtt{i}} : \mathtt{T_i})^{\;\mathtt{i} \in 1..n}}{\begin{array}{c}\Gamma \vdash \{\mathtt{l_i}\mathtt{=}\mathtt{v_i}^{\;\mathtt{i} \in 1..n}\} \; R \; \{\mathtt{l_i}\mathtt{=}\mathtt{v}'_{\mathtt{i}}{}^{\;\mathtt{i} \in 1..n}\} \\ : \{\mathtt{l_i}\mathtt{:}\mathtt{T_i}^{\;\mathtt{i} \in 1..n}\}\end{array}}$$

$$\frac{\Gamma, \mathtt{X} \vdash \mathtt{v} \; R \; \mathtt{v}' : \mathtt{T} \qquad \mathtt{X} \notin \mathit{ftv}(\Gamma)}{\Gamma \vdash \lambda \mathtt{X.v} \; R \; \lambda \mathtt{X.v}' : \forall \mathtt{X.T}}$$

$$\frac{\Gamma \vdash \mathtt{v_1} \; R \; \mathtt{v}'_1 : [\mathtt{X} \mapsto \mathtt{T_1}]\mathtt{T}}{\begin{array}{c}\Gamma \vdash \{\mathtt{*T_1,v_1}\} \; \mathtt{as} \; \{\exists\mathtt{X,T}\} \; R \\ \{\mathtt{*T_1,v}'_1\} \; \mathtt{as} \; \{\exists\mathtt{X,T}\} : \{\exists\mathtt{X,T}\}\end{array}}$$

$$\frac{\begin{array}{c}\Gamma \vdash \mathtt{v} \; R \; \mathtt{v}' : \mathtt{Bool} \\ \Gamma \vdash \mathtt{t_1} \; R \; \mathtt{t}'_1 : \mathtt{T} \qquad \Gamma \vdash \mathtt{t_2} \; R \; \mathtt{t}'_2 : \mathtt{T}\end{array}}{\begin{array}{c}\Gamma \vdash \mathtt{if\ v\ then\ t_1\ else\ t_2} \; R \\ \mathtt{if\ v'\ then\ t'_1\ else\ t'_2} : \mathtt{T}\end{array}}$$

$$\frac{\begin{array}{c}\mathtt{op}\mathtt{:}\mathtt{Gnd_1}, \ldots, \mathtt{Gnd_n} \to \mathtt{Gnd} \\ (\Gamma \vdash \mathtt{v_i} \; R \; \mathtt{v}'_{\mathtt{i}} : \mathtt{Gnd_i})^{\;\mathtt{i} \in 1..n}\end{array}}{\Gamma \vdash \mathtt{op}(\mathtt{v_i}^{\;\mathtt{i} \in 1..n}) \; R \; \mathtt{op}(\mathtt{v}'_{\mathtt{i}}{}^{\;\mathtt{i} \in 1..n}) : \mathtt{Gnd}}$$

$$\frac{\Gamma \vdash \mathtt{v_1} \; R \; \mathtt{v}'_1 : \mathtt{T_1} \to \mathtt{T_2} \qquad \Gamma \vdash \mathtt{v_2} \; R \; \mathtt{v}'_2 : \mathtt{T_1}}{\Gamma \vdash \mathtt{v_1\ v_2} \; R \; \mathtt{v}'_1 \; \mathtt{v}'_2 : \mathtt{T_2}}$$

$$\frac{\Gamma \vdash \mathtt{v} \; R \; \mathtt{v}' : \{\mathtt{l_i}\mathtt{:}\mathtt{T_i}^{\;\mathtt{i} \in 1..n}\}}{\Gamma \vdash \mathtt{v.l_j} \; R \; \mathtt{v'.l_j} : \mathtt{T_j}}$$

$$\frac{\Gamma \vdash \mathtt{v} \; R \; \mathtt{v}' : \forall \mathtt{X.T}}{\Gamma \vdash \mathtt{v\ T_1} \; R \; \mathtt{v'\ T_1} : [\mathtt{X} \mapsto \mathtt{T_1}]\mathtt{T}}$$

$$\frac{\begin{array}{c}\Gamma, \mathtt{X}, \mathtt{x}\mathtt{:}\mathtt{T} \vdash \mathtt{t} \; R \; \mathtt{t}' : \mathtt{T_1} \\ \mathtt{X} \notin \mathit{ftv}(\Gamma, \mathtt{T_1}) \qquad \Gamma \vdash \mathtt{v} \; R \; \mathtt{v}' : \{\exists\mathtt{X,T}\}\end{array}}{\begin{array}{c}\Gamma \vdash \mathtt{let\ \{*X,x\}=v\ in\ t} \; R \\ \mathtt{let\ \{*X,x\}=v'\ in\ t'} : \mathtt{T_1}\end{array}}$$

$$\frac{\Gamma \vdash \mathtt{t_1} \; R \; \mathtt{t}'_1 : \mathtt{T_1} \qquad \Gamma, \mathtt{x}\mathtt{:}\mathtt{T_1} \vdash \mathtt{t_2} \; R \; \mathtt{t}'_2 : \mathtt{T_2}}{\Gamma \vdash \mathtt{let\ x=t_1\ in\ t_2} \; R \; \mathtt{let\ x=t'_1\ in\ t'_2} : \mathtt{T_2}}$$

Figure 7-4: Properties of a type-respecting relation R between F_{ML} terms

Proof: The proof makes use of the following series of facts, only the last of which is not entirely straightforward to prove (see Exercise 7.5.4).

(i) The identity relation *Id* is an adequate congruence relation.

(ii) The collection of adequate relations is closed under taking unions.

(iii) Every compatible relation is reflexive, i.e. contains *Id*.

(iv) The set of all of compatible relations is closed under the operations of composition and reciprocation; similarly for the set of all substitutive relations and the set of all adequate relations.

(v) If the union of a non-empty family of compatible relations is transitive, it is also compatible; similarly, if the union of a non-empty family of reflexive and substitutive relations is transitive, it is also (reflexive and) substitutive.

Let $=_{\mathrm{ctx}}$ be the union of the family of relations that are adequate, compatible and substitutive. Note that this family is non-empty by (i). By (ii), $=_{\mathrm{ctx}}$ is adequate. So it suffices to show that it is a congruence relation. It is certainly reflexive by (i); and (iv) implies that it is also symmetric and transitive. So it just remains to show that it is compatible and substitutive, and this follows from (v), whose proof needs (iii). □

7.5.4 EXERCISE [★★]: Prove properties (iii) and (v) stated in the above proof. □

It is not easy to use either the formulation in terms of contexts in Definition 7.3.2 or the more abstract characterisation of Theorem 7.5.3 to prove that a particular pair of terms are contextually equivalent. For example, it is not easy to see from these characterisations that terms in the primitive reduction relation of Figure 7-2 are contextually equivalent (Corollary 7.5.8). That this is so follows from the coincidence of $=_{\mathrm{ctx}}$ with a notion of equivalence popularised by Mason and Talcott (1991).

7.5.5 DEFINITION [CIU-EQUIVALENCE, $=_{\mathrm{ciu}}$]: Two closed F_{ML} terms of the same (closed) type are *ciu-equivalent* if they have the same termination behaviour when paired with any frame stack (a 'use' of the terms); the relation is extended to open terms via closing substitutions (or 'closed instantiations'—thus we arrive at an explanation of the rather cryptic name for this equivalence).

More formally, we define $=_{\mathrm{ciu}}$ to be the type-respecting relation R° (using the operation from Definition 7.5.2(v)), where R consists of quadruples $(\emptyset, \mathtt{t}, \mathtt{t}', \mathtt{T})$ satisfying $\emptyset \vdash \mathtt{t} : \mathtt{T}$, $\emptyset \vdash \mathtt{t}' : \mathtt{T}$, and $\forall S.\ \langle S, \mathtt{t}\rangle\!\downarrow \;\Leftrightarrow\; \langle S, \mathtt{t}'\rangle\!\downarrow$. □

7.5.6 LEMMA: For any frame stack S and term $\mathtt{t}$, define a term $S[\mathtt{t}]$ by induction of the length of the stack S as follows:

$$\left.\begin{aligned} Id[\mathtt{t}] &\stackrel{\mathrm{def}}{=} \mathtt{t} \\ S \circ (\mathtt{x}.\mathtt{t}')[\mathtt{t}] &\stackrel{\mathrm{def}}{=} S[\mathtt{let}\ \mathtt{x}{=}\mathtt{t}\ \mathtt{in}\ \mathtt{t}'] \end{aligned}\right\} \tag{7.14}$$

Then $\langle S, \mathtt{t}\rangle\!\downarrow$ if and only if $S[\mathtt{t}]\!\downarrow$ (i.e. $\langle Id, S[\mathtt{t}]\rangle\!\downarrow$). □

Proof: This is proved by induction on the length of S. The base case $S = Id$ is trivial. The induction step follows from the fact that $\langle S, \texttt{let x=t in t}'\rangle\downarrow$ holds if and only if it was derived using rule (S-SEQ) in Figure 7-4, if and only if $\langle S \circ (\texttt{x.t}'), \texttt{t}\rangle\downarrow$ holds. □

7.5.7 THEOREM [CIU THEOREM FOR F_{ML}]: The relations of contextual equivalence and ciu-equivalence coincide. □

Proof: We first show that $=_{ctx}$ is contained in $=_{ciu}$. Suppose

$$\Gamma \vdash \texttt{t} =_{ctx} \texttt{t}' : \texttt{T}. \tag{7.15}$$

Since $=_{ctx}$ satisfies the substitutivity and reflexivity properties from Figure 7-4, it follows that

$$\emptyset \vdash \sigma(\texttt{t}) =_{ctx} \sigma(\texttt{t}') : \sigma(\texttt{T}) \tag{7.16}$$

for any Γ-closing substitution σ. For any frame stack S, since $=_{ctx}$ satisfies the compatibility (and reflexivity) properties from Figure 7-4, from (7.16) we deduce that $\emptyset \vdash S[\sigma(\texttt{t})] =_{ctx} S[\sigma(\texttt{t}')] : \sigma(\texttt{T})$ (using the notation of (7.14)). Since $=_{ctx}$ is adequate, this means that $S[\sigma(\texttt{t})]\downarrow$ if and only if $S[\sigma(\texttt{t}')]\downarrow$; hence by Lemma 7.5.6, $\langle S, \sigma(\texttt{t})\rangle\downarrow$ if and only if $\langle S, \sigma(\texttt{t}')\rangle\downarrow$. As this holds for all σ and S, we have $\Gamma \vdash \texttt{t} =_{ciu} \texttt{t}' : \texttt{T}$, as required.

To complete the proof of the theorem we have to show conversely that $=_{ciu}$ is contained in $=_{ctx}$. We can deduce this as a corollary of a stronger characterisation of $=_{ctx}$ in terms of logical relations (Theorem 7.6.25) that we establish later; so we postpone the rest of this proof until then. □

7.5.8 COROLLARY [CONVERSIONS]: The following are valid contextual equivalences:

(i) $\Gamma \vdash \texttt{if true then } \texttt{t}_1 \texttt{ else } \texttt{t}_2 =_{ctx} \texttt{t}_1 : \texttt{T}$ and
$\Gamma \vdash \texttt{if false then } \texttt{t}_1 \texttt{ else } \texttt{t}_2 =_{ctx} \texttt{t}_2 : \texttt{T}$, where $\Gamma \vdash \texttt{t}_i : \texttt{T}$ for $i = 1, 2$.

(ii) $\Gamma \vdash \texttt{op}(\texttt{c}_i\ ^{i\in 1..n}) =_{ctx} \texttt{c} : \texttt{Gnd}$, where c is the value of $op(\texttt{c}_i\ ^{i\in 1..n})$ and $Typeof(\texttt{c}) = \texttt{Gnd}$.

(iii) $\Gamma \vdash \texttt{v}_1\ \texttt{v}_2 =_{ctx} [\texttt{f} \mapsto \texttt{v}_1][\texttt{x} \mapsto \texttt{v}_2]\texttt{t} : \texttt{T}_2$,
where $\texttt{v}_1 = \texttt{fun f(x:T}_1\texttt{)=t:T}_2$.

(iv) $\Gamma \vdash \{\texttt{l}_i\texttt{=v}_i\ ^{i\in 1..n}\}\texttt{.j} =_{ctx} \texttt{v}_j : \texttt{T}_j$,
where $\Gamma \vdash \{\texttt{l}_i\texttt{=v}_i\ ^{i\in 1..n}\} : \{\texttt{l}_i\texttt{:T}_i\ ^{i\in 1..n}\}$.

(v) $\Gamma \vdash (\lambda\texttt{X.v})\ \texttt{T}_1 =_{ctx} [\texttt{X} \mapsto \texttt{T}_1]\texttt{v} : [\texttt{X} \mapsto \texttt{T}_1]\texttt{T}$, where $\Gamma \vdash \texttt{v} : \forall\texttt{X.T}$.

(vi) $\Gamma \vdash \mathtt{let}\ \{*\mathtt{X},\mathtt{x}\}\mathtt{=}(\{*\mathtt{T}_1,\mathtt{v}_1\}\ \mathtt{as}\ \{\exists\mathtt{X},\mathtt{T}\})\ \mathtt{in}\ \mathtt{t} =_{\mathrm{ctx}} [\mathtt{X} \mapsto \mathtt{T}_1][\mathtt{x} \mapsto \mathtt{v}_1]\mathtt{t} : \mathtt{T}_2$, where $\Gamma, \mathtt{X}, \mathtt{x}{:}\mathtt{T} \vdash \mathtt{t} : \mathtt{T}_2$ with $\mathtt{X} \notin \mathit{ftv}(\Gamma, \mathtt{T}_2)$.

(vii) $\Gamma \vdash \mathtt{let}\ \mathtt{x}\mathtt{=}\mathtt{v}\ \mathtt{in}\ \mathtt{t} =_{\mathrm{ctx}} [\mathtt{x} \mapsto \mathtt{v}]\mathtt{t} : \mathtt{T}_2$, where $\Gamma \vdash \mathtt{v} : \mathtt{T}_1$ and $\Gamma, \mathtt{x}{:}\mathtt{T}_1 \vdash \mathtt{t} : \mathtt{T}_2$.

(viii) $\Gamma \vdash \mathtt{let}\ \mathtt{x}_1\mathtt{=}\mathtt{t}_1\ \mathtt{in}\ (\mathtt{let}\ \mathtt{x}_2\mathtt{=}\mathtt{t}_2\ \mathtt{in}\ \mathtt{t}) =_{\mathrm{ctx}}$ $\mathtt{let}\ \mathtt{x}_2\mathtt{=}(\mathtt{let}\ \mathtt{x}_1\mathtt{=}\mathtt{t}_1\ \mathtt{in}\ \mathtt{t}_2)\ \mathtt{in}\ \mathtt{t} : \mathtt{T}$, where $\Gamma \vdash \mathtt{t}_1 : \mathtt{T}_1$, $\Gamma, \mathtt{x}_1{:}\mathtt{T}_1 \vdash \mathtt{t}_2 : \mathtt{T}_2$ and $\Gamma, \mathtt{x}_2{:}\mathtt{T}_2 \vdash \mathtt{t} : \mathtt{T}$. □

Proof: These are all ciu-equivalences, so we can just apply Theorem 7.5.7 (using the difficult half of the theorem whose proof we have postponed to §7.6!). The ciu-equivalences all follow easily from the definition of the termination relation (Figure 7-2) except for the last one, where one can apply property (7.7) from Exercise 7.4.2 to reduce proving (viii) for $=_{\mathrm{ciu}}$ to the special case when $\mathtt{t}_1$ is a value: see the following exercise. □

7.5.9 Exercise [★, ↛]: Given

$$\begin{aligned} \emptyset &\vdash \mathtt{t}_1 : \mathtt{T}_1 \\ \mathtt{x}_1{:}\mathtt{T}_1 &\vdash \mathtt{t}_2 : \mathtt{T}_2 \\ \mathtt{x}_2{:}\mathtt{T}_2 &\vdash \mathtt{t} : \mathtt{T} \end{aligned}$$

use property (7.7) to show for all frame stacks S that

$$\langle S \circ (\mathtt{x}_1.\mathtt{let}\ \mathtt{x}_2\mathtt{=}\mathtt{t}_2\ \mathtt{in}\ \mathtt{t}), \mathtt{t}_1\rangle\downarrow \quad \text{iff} \quad \langle S \circ (\mathtt{x}_2.\mathtt{t}) \circ (\mathtt{x}_1.\mathtt{t}_2), \mathtt{t}_1\rangle\downarrow.$$

Deduce part (viii) of Corollary 7.5.8. □

7.5.10 Exercise [★★]: Recall from Definition 7.5.1 the notion of adequate type-respecting binary relation. Call a type-respecting binary relation $\mathcal{R}$ $\mathtt{true}$-*adequate* if whenever $\emptyset \vdash \mathtt{t}\ \mathcal{R}\ \mathtt{t}' : \mathtt{Bool}$ holds, then $\langle \mathit{Id}, \mathtt{t}\rangle \longrightarrow^* \langle \mathit{Id}, \mathtt{true}\rangle$ holds if and only if $\langle \mathit{Id}, \mathtt{t}'\rangle \longrightarrow^* \langle \mathit{Id}, \mathtt{true}\rangle$ does. Here $\longrightarrow^*$ is the relation defined in Exercise 7.4.2. One can adapt the proof of Theorem 7.5.3 to show that there is a largest type-respecting binary relation $=_{\mathrm{ctx}}^{\mathtt{true}}$ between F_{ML} terms that is a congruence and $\mathtt{true}$-adequate. Show that $=_{\mathrm{ctx}}^{\mathtt{true}}$ coincides with contextual equivalence, $=_{\mathrm{ctx}}$. □

7.6 An Operationally-Based Logical Relation

We now have a precise definition of contextual equivalence for F_{ML} terms. Before showing that the Extensionality Principle 7.3.6 holds for existential types in F_{ML}, we first have to give a precise definition of the action of types on term-relations, $r \mapsto T[r]$, mentioned in the principle. That is the topic of this section. We will end up with a characterisation of $=_{ctx}$ in terms of a logical relation which yields several useful extensionality properties of contextual equivalence.

7.6.1 NOTATION: Let *Typ* denote the set of closed F_{ML} types. Given $T \in \mathit{Typ}$, let

- $\mathit{Term}(T)$ denote the set of closed terms of type T, i.e. those terms t for which $\emptyset \vdash t : T$ holds;
- $\mathit{Val}(T)$ denote the subset of $\mathit{Term}(T)$ whose elements are values;
- $\mathit{Stack}(T)$ denote the set of closed frame stacks whose argument type is T, i.e. those frame stacks S for which $\emptyset \vdash S : T \multimap T'$ holds for some $T' \in \mathit{Typ}$.

Given $T, T' \in \mathit{Typ}$, let

- $\mathit{TRel}(T, T')$ denote the set of all subsets of $\mathit{Term}(T) \times \mathit{Term}(T')$; we call its elements *term-relations*;
- $\mathit{VRel}(T, T')$ denote the set of all subsets of $\mathit{Val}(T) \times \mathit{Val}(T')$; we call its elements *value-relations*;
- $\mathit{SRel}(T, T')$ denote the the set of all subsets of $\mathit{Stack}(T) \times \mathit{Stack}(T')$; we call its elements *stack-relations*. □

Note that every value-relation is also a term-relation (since values are particular sorts of term): $\mathit{VRel}(T, T') \subseteq \mathit{TRel}(T, T')$. On the other hand we can obtain a value-relation from a term-relation just by restricting attention to values: given $r \in \mathit{TRel}(T, T')$, define $r^v \in \mathit{VRel}(T, T')$ by

$$r^v \stackrel{\text{def}}{=} \{(v, v') \in \mathit{Val}(T) \times \mathit{Val}(T') \mid (v, v') \in r\}. \tag{7.17}$$

We will be particularly interested in term-relations r that are indistinguishable, as far as termination properties are concerned, from their value restrictions, r^v. Definition 7.6.3 makes this precise, using a Galois connection between term-relations and stack-relations. The definition may appear to be rather mysterious; its nature will emerge as we develop the action of types on term-relations and its properties. First we recall for the reader what is meant in general by a 'Galois connection'.

7.6.2 DEFINITION: A *Galois connection* between partially ordered sets $(P, \leq_P)$ and $(Q, \leq_Q)$ is specified by a pair of functions $f : P \to Q$ and $g : Q \to P$ satisfying $q \leq_Q f(p)$ if and only if $p \leq_P g(q)$, for all $p \in P$ and $q \in Q$. □

7.6.3 DEFINITION [CLOSED AND VALUABLE TERM-RELATIONS]: Let $\mathtt{T}, \mathtt{T}' \in \mathit{Typ}$ be closed types. Given a term-relation $r \in \mathit{TRel}(\mathtt{T}, \mathtt{T}')$, define a stack-relation $r^s \in \mathit{SRel}(\mathtt{T}, \mathtt{T}')$ by

$(S, S') \in r^s$ if and only if for all $(\mathtt{t}, \mathtt{t}') \in r$, $\langle S, \mathtt{t} \rangle \downarrow$ holds if and only if $\langle S', \mathtt{t}' \rangle \downarrow$ does.

Conversely, given a stack-relation $s \in \mathit{SRel}(\mathtt{T}, \mathtt{T}')$, define a term-relation $s^t \in \mathit{TRel}(\mathtt{T}, \mathtt{T}')$ by

$(\mathtt{t}, \mathtt{t}') \in s^t$ if and only if for all $(S, S') \in s$, $\langle S, \mathtt{t} \rangle \downarrow$ holds if and only if $\langle S', \mathtt{t}' \rangle \downarrow$ does.

Call a term-relation $r \in \mathit{TRel}(\mathtt{T}, \mathtt{T}')$ *closed* if it satisfies $r = r^{st}$ and *valuable* if it satisfies $r = r^{vst}$. □

7.6.4 NOTE: The operator $(-)^{st}$ is denoted $(-)^{\top\top}$ in Pitts; Pitts (1998; 2000). □

7.6.5 LEMMA: The operations $(-)^s$ and $(-)^t$ for turning term-relations into stack-relations and *vice versa*, form a Galois connection:

$$s \subseteq r^s \quad \text{if and only if} \quad r \subseteq s^t. \tag{7.18}$$

Hence the operator $(-)^{st}$ on term-relations is monotone ($r_1 \subseteq r_2$ implies $(r_1)^{st} \subseteq (r_2)^{st}$), inflationary ($r \subseteq r^{st}$), and idempotent ($(r^{st})^{st} = r^{st}$). □

Proof: If $s \subseteq r^s$, then for any $(\mathtt{t}, \mathtt{t}') \in r$ we have for all $(S, S') \in s$ that $(S, S') \in r^s$, so $\langle S, \mathtt{t} \rangle \downarrow$ iff $\langle S', \mathtt{t}' \rangle \downarrow$; hence $(\mathtt{t}, \mathtt{t}') \in s^t$. Thus $s \subseteq r^s$ implies $r \subseteq s^t$. The converse implication holds by a similar argument. Once we have (7.18), the other properties follow by standard arguments true of any Galois connection, which we give in case the reader has not seen them before.

Thus for any term-relation r, since $r^s \subseteq r^s$, from (7.18) we conclude that $r \subseteq r^{st}$; so $(-)^{st}$ is inflationary (and symmetrically, so is the operator $(-)^{ts}$ on stack-relations).

Now we can deduce that $(-)^s$ and $(-)^t$ are order-reversing. For if $r_1 \subseteq r_2$, then $r_1 \subseteq r_2 \subseteq r_2^{st}$, so by (7.18), $r_2^s \subseteq r_1^s$. Similarly, $s_1 \subseteq s_2$ implies $s_2^t \subseteq s_1^t$. Hence $(-)^{st}$ is monotone (and so is $(-)^{ts}$).

Finally, for idempotence, in view of the inflationary property we just have to show $(r^{st})^{st} \subseteq r^{st}$. But applying (7.18) to $r^{st} \subseteq r^{st}$ we get $r^s \subseteq (r^{st})^s$; applying the order-reversing operator $(-)^t$ to this yields $(r^{st})^{st} \subseteq r^{st}$, as required. □

7.6.6 COROLLARY: Every valuable term-relation is in particular a closed term-relation. □

Proof: Note that because $(-)^{st}$ is idempotent (by the above lemma), any term-relation of the form r^{st} is closed. Thus valuable term-relations (ones satisfying $r = r^{vst}$) are in particular closed. □

The following exercise establishes a supply of valuable term-relations that we will need later.

7.6.7 EXERCISE [RECOMMENDED, ★★]: Given any value-relation $r \in \mathit{VRel}(\mathtt{T}, \mathtt{T}')$, show that r^{st} is valuable, i.e. satisfies $r^{st} = (r^{st})^{vst}$. □

Closed term-relations (and hence also valuable term-relations) have excellent 'admissibility' properties that we record in the following lemma.

7.6.8 LEMMA: If $r \in \mathit{TRel}(\mathtt{T}, \mathtt{T}')$ satisfies $r = r^{st}$ (and in particular if it is valuable), then it has the following properties.

Equivalence-respecting If $(\mathtt{t}, \mathtt{t}') \in r$, $\emptyset \vdash \mathtt{t} =_{\mathrm{ciu}} \mathtt{t}_1 : \mathtt{T}$, and $\emptyset \vdash \mathtt{t}' =_{\mathrm{ciu}} \mathtt{t}'_1 : \mathtt{T}$, then $(\mathtt{t}_1, \mathtt{t}'_1) \in r$.

Admissibility Given recursive function values $\mathtt{F} \stackrel{\text{def}}{=} \mathtt{fun\,f(x:T_1)=u:T_2}$ and $\mathtt{F}' \stackrel{\text{def}}{=} \mathtt{fun\,f(x:T_1)=u':T_2}$, let $\mathtt{F}_n$ and $\mathtt{F}'_n$ $(n = 0, 1, \ldots)$ be their 'unwindings', as in Theorem 7.4.4. If $([\mathtt{x} \mapsto \mathtt{F}_n]\mathtt{t}, [\mathtt{x} \mapsto \mathtt{F}'_n]\mathtt{t}') \in r$ for all $n = 0, 1, \ldots$, then $([\mathtt{x} \mapsto \mathtt{F}]\mathtt{t}, [\mathtt{x} \mapsto \mathtt{F}']\mathtt{t}') \in r$. □

Proof: Suppose $(\mathtt{t}, \mathtt{t}') \in r$, $\emptyset \vdash \mathtt{t} =_{\mathrm{ciu}} \mathtt{t}_1 : \mathtt{T}$ and $\emptyset \vdash \mathtt{t}' =_{\mathrm{ciu}} \mathtt{t}'_1 : \mathtt{T}$. To see that $(\mathtt{t}_1, \mathtt{t}'_1) \in r$, since $r = (r^s)^t$, it suffices to show for all $(S, S') \in r^s$ that $\langle S, \mathtt{t}_1 \rangle \downarrow$ iff $\langle S', \mathtt{t}'_1 \rangle \downarrow$. But

$$\begin{aligned}
\langle S, \mathtt{t}_1 \rangle \downarrow &\text{ iff } \langle S, \mathtt{t} \rangle \downarrow && (\text{since } \emptyset \vdash \mathtt{t} =_{\mathrm{ciu}} \mathtt{t}_1 : \mathtt{T}) \\
&\text{ iff } \langle S', \mathtt{t}' \rangle \downarrow && (\text{since } (S, S') \in r^s \text{ and } (\mathtt{t}, \mathtt{t}') \in r) \\
&\text{ iff } \langle S', \mathtt{t}'_1 \rangle \downarrow && (\text{since } \emptyset \vdash \mathtt{t}' =_{\mathrm{ciu}} \mathtt{t}'_1 : \mathtt{T}).
\end{aligned}$$

For the **Admissibility** property we apply the Unwinding Theorem. Suppose $([\mathtt{x} \mapsto \mathtt{F}_n]\mathtt{t}, [\mathtt{x} \mapsto \mathtt{F}'_n]\mathtt{t}') \in r$ holds for all $n = 0, 1, \ldots$. Then for any $(S, S') \in r^s$ we have

$$\begin{aligned}
&\langle S, [\mathtt{x} \mapsto \mathtt{F}]\mathtt{t} \rangle \downarrow \\
&\text{iff for some } n, \langle S, [\mathtt{x} \mapsto \mathtt{F}_n]\mathtt{t} \rangle \downarrow && (\text{by Theorem 7.4.4}) \\
&\text{iff for some } n, \langle S', [\mathtt{x} \mapsto \mathtt{F}'_n]\mathtt{t}' \rangle \downarrow && (\text{since } (S, S') \in r^s \text{ and} \\
& && \ ([\mathtt{x} \mapsto \mathtt{F}_n]\mathtt{t}, [\mathtt{x} \mapsto \mathtt{F}'_n]\mathtt{t}') \in r) \\
&\text{iff } \langle S, [\mathtt{x} \mapsto \mathtt{F}']\mathtt{t}' \rangle \downarrow && (\text{by Theorem 7.4.4 again})
\end{aligned}$$

and therefore $([\mathtt{x} \mapsto \mathtt{F}]\mathtt{t}, [\mathtt{x} \mapsto \mathtt{F}']\mathtt{t}') \in (r^s)^t$; but $r^{st} = r$. □

$\boxed{Id_{\mathtt{Gnd}}} \in VRel(\mathtt{Gnd}, \mathtt{Gnd})$
is $\{(\mathtt{c}, \mathtt{c}) \mid Typeof(\mathtt{c}) = \mathtt{Gnd}\}$.

$\boxed{\mathtt{fun}(r_1, r_2)} \in VRel(\mathtt{T}_1 \to \mathtt{T}_2, \mathtt{T}'_1 \to \mathtt{T}'_2)$,
given $r_1 \in TRel(\mathtt{T}_1, \mathtt{T}'_1)$ and $r_2 \in TRel(\mathtt{T}_2, \mathtt{T}'_2)$,
is defined by:

$(\mathtt{v}, \mathtt{v}') \in \mathtt{fun}(r_1, r_2)$ if and only if for all $(\mathtt{v}_1, \mathtt{v}'_1) \in (r_1)^v$, it is the case that $(\mathtt{v}\ \mathtt{v}_1, \mathtt{v}'\ \mathtt{v}'_1) \in r_2$.

$\boxed{\{\mathtt{l}_i{=}r_i\ ^{i \in 1..n}\}} \in VRel(\{\mathtt{l}_i{:}\mathtt{T}_i\ ^{i \in 1..n}\}, \{\mathtt{l}_i{:}\mathtt{T}'_i\ ^{i \in 1..n}\})$
given $(r_i \in TRel(\mathtt{T}_i, \mathtt{T}'_i)\ ^{i \in 1..n})$,
is defined by:

$(\mathtt{v}, \mathtt{v}') \in \{\mathtt{l}_i{=}r_i\ ^{i \in 1..n}\}$ if and only if for all $i \in 1..n$, it is the case that $(\mathtt{v}.\mathtt{l}_i, \mathtt{v}'.\mathtt{l}_i) \in r_i$.

$\boxed{\lambda r.\,R(r)} \in VRel(\forall \mathtt{X}.\mathtt{T}, \forall \mathtt{X}.\mathtt{T}')$,
given $R(r) \in TRel([\mathtt{X} \mapsto \mathtt{T}_1]\mathtt{T}, [\mathtt{X} \mapsto \mathtt{T}'_1]\mathtt{T}')$ for $r \in TRel(\mathtt{T}_1, \mathtt{T}'_1)$ and $\mathtt{T}_1, \mathtt{T}'_1 \in Typ$,
is defined by:

$(\mathtt{v}, \mathtt{v}') \in \lambda r.\,R(r)$ if and only if for all $\mathtt{T}_1, \mathtt{T}'_1 \in Typ$ and all $r \in TRel(\mathtt{T}_1, \mathtt{T}'_1)$, it is the case that $(\mathtt{v}\ \mathtt{T}_1, \mathtt{v}'\ \mathtt{T}'_1) \in R(r)$.

$\boxed{\{\exists r, R(r)\}} \in VRel(\{\exists \mathtt{X}, \mathtt{T}\}, \{\exists \mathtt{X}, \mathtt{T}'\})$,
given $R(r) \in TRel([\mathtt{X} \mapsto \mathtt{T}_1]\mathtt{T}, [\mathtt{X} \mapsto \mathtt{T}'_1]\mathtt{T}')$ for $r \in TRel(\mathtt{T}_1, \mathtt{T}'_1)$ and $\mathtt{T}_1, \mathtt{T}'_1 \in Typ$,
is defined by:

$(\mathtt{v}, \mathtt{v}') \in \{\exists r, R(r)\}$ if and only if there exist $\mathtt{T}_1, \mathtt{T}'_1 \in Typ$, $r \in TRel(\mathtt{T}_1, \mathtt{T}'_1)$ and $(\mathtt{v}_1, \mathtt{v}'_1) \in R(r)$ with $\mathtt{v} = \{{*}\mathtt{T}_1, \mathtt{v}_1\}\ \mathtt{as}\ \{\exists \mathtt{X}, \mathtt{T}\}$ and $\mathtt{v}' = \{{*}\mathtt{T}'_1, \mathtt{v}'_1\}\ \mathtt{as}\ \{\exists \mathtt{X}, \mathtt{T}'\}$.

Figure 7-5: Type-directed constructions on term-relations

7.6.9 DEFINITION [ACTION OF TYPES ON TERM-RELATIONS]: The action of types on term-relations takes the following form: if $\mathtt{T}(\overline{\mathtt{X}})$ is a type whose free type variables lie among the list $\overline{\mathtt{X}} = \mathtt{X}_1, \ldots, \mathtt{X}_n$, then given a corresponding list of term relations $r_1 \in TRel(\mathtt{T}_1, \mathtt{T}'_1), \ldots, r_n \in TRel(\mathtt{T}_n, \mathtt{T}'_n)$, we define a term relation $\mathtt{T}[\overline{r}] \in TRel([\overline{\mathtt{X}} \mapsto \overline{\mathtt{T}}]\mathtt{T}, [\overline{\mathtt{X}} \mapsto \overline{\mathtt{T}'}]\mathtt{T})$. The definition is by induction on the structure of $\mathtt{T}$ as follows.

$$\begin{aligned}
\mathtt{X}_i[\overline{r}] &\stackrel{\text{def}}{=} (r_i)^{v\,s\,t} \\
\mathtt{Gnd}[\overline{r}] &\stackrel{\text{def}}{=} (Id_{\mathtt{Gnd}})^{s\,t} \\
(\mathtt{T}_1 \to \mathtt{T}_2)[\overline{r}] &\stackrel{\text{def}}{=} \mathtt{fun}(\mathtt{T}_1[\overline{r}], \mathtt{T}_2[\overline{r}])^{s\,t} \\
\{\mathtt{l}_i{:}\mathtt{T}_i\ ^{i \in 1..n}\}[\overline{r}] &\stackrel{\text{def}}{=} \{\mathtt{l}_i{=}\mathtt{T}_i[\overline{r}]\ ^{i \in 1..n}\}^{s\,t} \\
(\forall \mathtt{X}.\mathtt{T})[\overline{r}] &\stackrel{\text{def}}{=} (\lambda r.\,\mathtt{T}[r, \overline{r}])^{s\,t} \\
\{\exists \mathtt{X}, \mathtt{T}\}[\overline{r}] &\stackrel{\text{def}}{=} \{\exists r, \mathtt{T}[r, \overline{r}]\}^{s\,t}
\end{aligned}$$

In addition to the operations on term-, value- and stack-relations given in Definition 7.6.3, these definitions make use of the operations for constructing value-relations from term-relations given in Figure 7-5. □

We can use the action of types on term-relations to define a type-respecting binary relation between *open* terms (in the sense of Definition 7.5.1) by insisting that if we substitute related terms for the free value variables, the resulting terms are still related. This 'mapping related things to related things' property is the common characteristic of the wide variety of constructs called *logical relations* that have arisen since the seminal work of Plotkin (1973) and Statman (1985) concerning simply typed λ-calculus; see also Chapter 6.

7.6.10 DEFINITION [LOGICAL RELATION, Δ]: Given $\Gamma \vdash \mathtt{t} : \mathtt{T}$ and $\Gamma \vdash \mathtt{t}' : \mathtt{T}$, with $\Gamma = \mathtt{X}_1, \ldots, \mathtt{X}_m, \mathtt{x}_1 : \mathtt{T}_1, \ldots, \mathtt{x}_n : \mathtt{T}_n$ say, we write $\Gamma \vdash \mathtt{t}\ \Delta\ \mathtt{t}' : \mathtt{T}$ to mean that for all Γ-closing substitutions σ, σ' (cf. Definition 7.5.2(v)) and all families of term-relations $\overline{r} = (r_i \in \mathit{TRel}(\sigma(\mathtt{X}_i), \sigma'(\mathtt{X}_i))\ ^{i \in 1..m})$, if $(\sigma(\mathtt{x}_j), \sigma'(\mathtt{x}_j)) \in \mathtt{T}_j[\overline{r}]^v$ holds for each $j = 1, \ldots, n$, then $(\sigma(\mathtt{t}), \sigma'(\mathtt{t}')) \in \mathtt{T}[\overline{r}]$. □

7.6.11 REMARK: Since it is far from straightforward, the form of Definitions 7.6.9 and 7.6.10 deserves some explanation. These definitions embody certain extensionality and parametricity properties (see §7.7 and Theorem 7.7.8) that we wish to show hold for F_{ML} contextual equivalence: eventually we show that the above logical relation Δ coincides with contextual equivalence (Theorem 7.6.25). To get that coincidence we have to formulate the definition of Δ so that it satisfies the crucial property of Lemma 7.6.17 below (the so-called fundamental property of the logical relation) and is adequate (Lemma 7.6.24). The definition of the action of types on term-relations in Definition 7.6.9 is carefully formulated to ensure these properties hold.

First of all, note the use of closing substitutions to reduce the logical relation for open terms to that for closed ones. This builds in the 'instantiation' aspect of ciu-equivalence that we wish to prove of contextual equivalence. (It also means that the logical relation has the 'monotonicity' property considered in Chapter 6.)

Secondly, we want $\mathtt{T}[\overline{r}]$ to always be a closed term-relation, because then it has the equivalence-respecting and admissibility properties noted in Lemma 7.6.8. This accounts for the use of $(-)^{s\,t}$ in the definition. The $(-)^s$ and $(-)^t$ operators build into the logical relation a delicate interplay between terms and frame stacks. Of course this relies on the formulation of the operational semantics of F_{ML} in §7.4.2: although more traditional 'big-step' or 'small-step' operational semantics lead to the same termination relation (cf. Exercise 7.4.2), the pairing between frame stacks and terms defined in Figure 7-2 is ideal for our purposes.

Lastly, the call-by-value nature of F_{ML} dictates that relational parametricity properties of polymorphic types should be with respect to term-relations that are valuable; but instead of letting r range over such relations in the definition of $(\forall\mathtt{X.T})[\overline{r}]$ and $\{\exists\mathtt{X,T}\}[\overline{r}]$ we have used an equivalent formulation in which r ranges over all term-relations (of appropriate type), but type variables $\mathtt{X}$ are interpreted using the closure of the value-restriction operator $(-)^v$: for in fact as r ranges over all term-relations, r^{vst} ranges over all valuable term-relations. □

The rest of this section is devoted to showing that contextual equivalence and ciu-eqivalence coincide with the logical relation.

7.6.12 LEMMA: Each of the term relations $\mathtt{T}[\overline{r}]$ defined in Definition 7.6.9 is valuable, i.e. satisfies $\mathtt{T}[\overline{r}] = \mathtt{T}[\overline{r}]^{vst}$, and hence in particular by Corollary 7.6.6 is closed. □

Proof: It is immediate from the definition that each $\mathtt{T}[\overline{r}]$ is of the form r^{st} for some *value*-relation r; so just apply Exercise 7.6.7. □

The following lemma helps with calculations involving the action on term-relations of function types. We give its proof in detail since it typifies the kind of reasoning needed when working with the Galois connection given by the $(-)^s$ and $(-)^t$ operators. (For related properties for record and $\forall$-types, see Exercise 7.6.14.)

7.6.13 LEMMA: The operation $\mathtt{fun}(-,-)$ from Definition 7.6.9(ii) satisfies

$$\mathtt{fun}(r_1,(r_2)^{st})^{stv} = \mathtt{fun}(r_1,(r_2)^{st}) \tag{7.19}$$

$$\mathtt{fun}((r_1)^{vst},(r_2)^{st}) = \mathtt{fun}(r_1,(r_2)^{st}). \tag{7.20}$$

□

Proof: To prove (7.19), first note that since $(-)^{st}$ is inflationary (Lemma 7.6.5) we have $\mathtt{fun}(r_1,(r_2)^{st}) \subseteq \mathtt{fun}(r_1,(r_2)^{st})^{st}$; and since $\mathtt{fun}(r_1,(r_2)^{st})$ is a value-relation, it follows that $\mathtt{fun}(r_1,(r_2)^{st}) \subseteq \mathtt{fun}(r_1,(r_2)^{st})^{stv}$. For the reverse inclusion it suffices to prove

$$\mathtt{fun}(r_1,(r_2)^{st})^{st} \subseteq \mathtt{fun}(r_1,(r_2)^{st}) \tag{7.21}$$

and then apply $(-)^v$ to both sides (noting that $\mathtt{fun}(r_1,(r_2)^{st})$, being a value-relation, is equal to $\mathtt{fun}(r_1,(r_2)^{st})^v$). For (7.21) we use the following simple property of the termination relation (Figure 7-2) with respect to application:

$$\langle S \circ (\mathtt{f.f}\ v_1), v\rangle\!\downarrow\ \Leftrightarrow\ \langle S, v\ v_1\rangle\!\downarrow$$

and hence

$$(\langle S, \mathtt{v}\ \mathtt{v}_1\rangle\downarrow \Leftrightarrow \langle S', \mathtt{v}'\ \mathtt{v}_1'\rangle\downarrow) \;\Leftrightarrow \\ (\langle S \circ (\mathtt{f}\,.\,\mathtt{f}\ \mathtt{v}_1), \mathtt{v}\rangle\downarrow \Leftrightarrow \langle S' \circ (\mathtt{f}\,.\,\mathtt{f}\ \mathtt{v}_1'), \mathtt{v}'\rangle\downarrow) \tag{7.22}$$

If $(\mathtt{v}, \mathtt{v}') \in \mathtt{fun}(r_1, (r_2)^{st})$ and $(\mathtt{v}_1, \mathtt{v}_1') \in (r_1)^v$, then we have $(\mathtt{v}\ \mathtt{v}_1, \mathtt{v}'\ \mathtt{v}_1') \in (r_2^s)^t$ by definition of the $\mathtt{fun}(-,-)$ operation on term-relations (Figure 7-5). So if $(S, S') \in (r_2)^s$, then

$$\langle S, \mathtt{v}\ \mathtt{v}_1\rangle\downarrow \;\Leftrightarrow\; \langle S', \mathtt{v}'\ \mathtt{v}_1'\rangle\downarrow$$

and hence by (7.22)

$$\langle S \circ (\mathtt{f}\,.\,\mathtt{f}\ \mathtt{v}_1), \mathtt{v}\rangle\downarrow \;\Leftrightarrow\; \langle S' \circ (\mathtt{f}\,.\,\mathtt{f}\ \mathtt{v}_1'), \mathtt{v}'\rangle\downarrow.$$

Since this holds for all $(\mathtt{v}, \mathtt{v}') \in \mathtt{fun}(r_1, (r_2)^{st})$, we deduce that

$$(S, S') \in (r_2)^s \;\;\&\;\; (\mathtt{v}_1, \mathtt{v}_1') \in (r_1)^v \;\Rightarrow \\ (S \circ (\mathtt{f}\,.\,\mathtt{f}\ \mathtt{v}_1), S' \circ (\mathtt{f}\,.\,\mathtt{f}\ \mathtt{v}_1')) \in \mathtt{fun}(r_1, (r_2)^{st})^s.$$

So for any $(S, S') \in (r_2)^s$ and $(\mathtt{v}_1, \mathtt{v}_1') \in (r_1)^v$, since $(S \circ (\mathtt{f}\,.\,\mathtt{f}\ \mathtt{v}_1), S' \circ (\mathtt{f}\,.\,\mathtt{f}\ \mathtt{v}_1')) \in \mathtt{fun}(r_1, (r_2)^{st})^s$ it follows that if

$$(\mathtt{v}, \mathtt{v}') \in \mathtt{fun}(r_1, (r_2)^{st})^{st} \tag{7.23}$$

then $\langle S \circ (\mathtt{f}\,.\,\mathtt{f}\ \mathtt{v}_1), \mathtt{v}\rangle\downarrow \;\Leftrightarrow\; \langle S' \circ (\mathtt{f}\,.\,\mathtt{f}\ \mathtt{v}_1'), \mathtt{v}'\rangle\downarrow$, and hence by (7.22) that $\langle S, \mathtt{v}\ \mathtt{v}_1\rangle\downarrow \Leftrightarrow \langle S', \mathtt{v}'\ \mathtt{v}_1'\rangle\downarrow$. Since this holds for all $(S, S') \in (r_2)^s$, it follows that $(\mathtt{v}\ \mathtt{v}_1, \mathtt{v}'\ \mathtt{v}_1') \in (r_2)^{st}$ whenever $(\mathtt{v}_1, \mathtt{v}_1') \in (r_1)^v$. So $(\mathtt{v}, \mathtt{v}') \in \mathtt{fun}(r_1, (r_2)^{st})$ whenever (7.23) holds; thus we have proved the inclusion in (7.21), as required.

Turning to the proof of (7.20), first note that since since $(-)^{st}$ is inflationary, we have $(r_1)^v \subseteq (r_1)^{vst}$. So since $\mathtt{fun}(-,-)$ is clearly order-reversing in its first argument, we have $\mathtt{fun}((r_1)^{vst}, (r_2)^{st}) \subseteq \mathtt{fun}((r_1)^v, (r_2)^{st})$; and $\mathtt{fun}((r_1)^v, (r_2)^{st}) = \mathtt{fun}(r_1, (r_2)^{st})$, because $\mathtt{fun}(-,-)$ only depends upon the values related by its first argument. Thus to prove (7.20), we just have to show

$$\mathtt{fun}(r_1, (r_2)^{st}) \subseteq \mathtt{fun}((r_1)^{vst}, (r_2)^{st}). \tag{7.24}$$

For this we use the following fact about termination

$$\langle S \circ (\mathtt{x}\,.\,\mathtt{v}\ \mathtt{x}), \mathtt{v}_1\rangle\downarrow \Leftrightarrow \langle S, \mathtt{v}\ \mathtt{v}_1\rangle\downarrow$$

which is immediate from the definition in Figure 7-2. From this it follows that

$$(\langle S, \mathtt{v}\,\mathtt{v}_1\rangle\!\downarrow \Leftrightarrow \langle S', \mathtt{v}'\,\mathtt{v}_1'\rangle\!\downarrow) \;\Leftrightarrow\; (\langle S \circ (\mathtt{x}\,.\,\mathtt{v}\,\mathtt{x}), \mathtt{v}_1\rangle\!\downarrow \Leftrightarrow \langle S' \circ (\mathtt{x}\,.\,\mathtt{v}'\,\mathtt{x}), \mathtt{v}_1'\rangle\!\downarrow) \tag{7.25}$$

If $(\mathtt{v}, \mathtt{v}') \in \mathtt{fun}(r_1, (r_2)^{st})$ and $(\mathtt{v}_1, \mathtt{v}_1') \in (r_1)^v$, then by definition of $\mathtt{fun}(-,-)$ we have $(\mathtt{v}\,\mathtt{v}_1, \mathtt{v}'\,\mathtt{v}_1') \in (r_2)^{st}$. So if $(S, S') \in (r_2)^s$, then

$$\langle S, \mathtt{v}\,\mathtt{v}_1\rangle\!\downarrow \;\Leftrightarrow\; \langle S', \mathtt{v}'\,\mathtt{v}_1'\rangle\!\downarrow$$

and hence by (7.25) we have

$$\langle S \circ (\mathtt{x}\,.\,\mathtt{v}\,\mathtt{x}), \mathtt{v}_1\rangle\!\downarrow \;\Leftrightarrow\; \langle S' \circ (\mathtt{x}\,.\,\mathtt{v}'\,\mathtt{x}), \mathtt{v}_1'\rangle\!\downarrow.$$

Since this holds for all $(\mathtt{v}_1, \mathtt{v}_1') \in (r_1)^v$, we deduce that

$$(S, S') \in (r_2)^s \;\;\&\;\; (\mathtt{v}, \mathtt{v}') \in \mathtt{fun}(r_1, (r_2)^{st}) \;\Rightarrow\; (S \circ (\mathtt{x}\,.\,\mathtt{v}\,\mathtt{x}), S' \circ (\mathtt{x}\,.\,\mathtt{v}'\,\mathtt{x})) \in (r_1)^{vs}.$$

So for any $(S, S') \in (r_2)^s$ and $(\mathtt{v}, \mathtt{v}') \in \mathtt{fun}(r_1, (r_2)^{st})$, since $(S \circ (\mathtt{x}\,.\,\mathtt{v}\,\mathtt{x}), S' \circ (\mathtt{x}\,.\,\mathtt{v}'\,\mathtt{x})) \in (r_1)^{vs}$, it follows for any $(\mathtt{v}_1, \mathtt{v}_1') \in ((r_1)^{vst})^v \subseteq ((r_1)^{vs})^t$ that we have $\langle S \circ (\mathtt{x}\,.\,\mathtt{v}\,\mathtt{x}), \mathtt{v}_1\rangle\!\downarrow \Leftrightarrow \langle S' \circ (\mathtt{x}\,.\,\mathtt{v}'\,\mathtt{x}), \mathtt{v}_1'\rangle\!\downarrow$, and hence by (7.25) that $\langle S, \mathtt{v}\,\mathtt{v}_1\rangle\!\downarrow \Leftrightarrow \langle S', \mathtt{v}'\,\mathtt{v}_1'\rangle\!\downarrow$. Since this holds for all $(S, S') \in (r_2)^s$, it follows that $(\mathtt{v}\,\mathtt{v}_1, \mathtt{v}'\,\mathtt{v}_1') \in (r_2)^{st}$. Hence $(\mathtt{v}, \mathtt{v}') \in \mathtt{fun}((r_1)^{vst}, (r_2)^{st})$ whenever $(\mathtt{v}, \mathtt{v}') \in \mathtt{fun}(r_1, (r_2)^{st})$, as required for (7.24). □

7.6.14 EXERCISE [RECOMMENDED, ★]: Show that constructions (iii) and (iv) in Definition 7.6.9 satisfy

$$\{\mathtt{l}_i{=}(r_i)^{st\ i\in 1..n}\}^{stv} = \{\mathtt{l}_i{=}(r_i)^{st\ i\in 1..n}\} \tag{7.26}$$

$$(\lambda r\,.\,R(r)^{st})^{stv} = \lambda r\,.\,R(r)^{st}. \tag{7.27}$$

(Cf. the proof of Lemma (7.6.13).) □

7.6.15 LEMMA: For all ground types $\mathtt{Gnd}$, $(Id_{\mathtt{Gnd}})^{stv} = Id_{\mathtt{Gnd}}$. □

Proof: Since $(-)^{st}$ is idempotent (Lemma 7.6.5), we have $Id_{\mathtt{Gnd}} \subseteq (Id_{\mathtt{Gnd}})^{st}$; and since $Id_{\mathtt{Gnd}}$ is a value-relation it follows that $Id_{\mathtt{Gnd}} \subseteq (Id_{\mathtt{Gnd}})^{stv}$. To prove the reverse inclusion, for each constant $\mathtt{c}$ of type $\mathtt{Gnd}$ consider

$$\mathtt{diverge} \stackrel{\text{def}}{=} (\mathtt{fun\ f\,(b{:}Bool) = f\ b : Bool})\,\mathtt{true}$$

$$S_{\mathtt{c}} \stackrel{\text{def}}{=} Id \circ (\mathtt{x.\ if\ x{=}c\ then\ true\ else\ diverge}).$$

Note that for all constants $\mathtt{c}'$ of type $\mathtt{Gnd}$

$$\langle S_{\mathtt{c}}, \mathtt{c}' \rangle \downarrow \;\Leftrightarrow\; \mathtt{c} = \mathtt{c}'. \tag{7.28}$$

Furthermore, since $(\mathtt{c}', \mathtt{c}'') \in \mathit{Id}_{\mathtt{Gnd}}$ iff $\mathtt{c}' = \mathtt{c}''$, we have that $(S_{\mathtt{c}}, S_{\mathtt{c}}) \in (\mathit{Id}_{\mathtt{Gnd}})^{s}$; so if the constants $\mathtt{c}$ and $\mathtt{c}'$ satisfy $(\mathtt{c}, \mathtt{c}') \in (\mathit{Id}_{\mathtt{Gnd}})^{s\,t}$, then $\langle S_{\mathtt{c}}, \mathtt{c} \rangle \downarrow \Leftrightarrow \langle S_{\mathtt{c}}, \mathtt{c}' \rangle \downarrow$. So by (7.28), $(\mathtt{c}, \mathtt{c}') \in (\mathit{Id}_{\mathtt{Gnd}})^{s\,t}$ implies $\mathtt{c} = \mathtt{c}'$; thus $(\mathit{Id}_{\mathtt{Gnd}})^{s\,t\,v} \subseteq \mathit{Id}_{\mathtt{Gnd}}$. □

7.6.16 LEMMA: The action of types on term-relations of Definition 7.6.9 has the following substitution property. For any types $\mathtt{T}$ and $\mathtt{T}'$ with $\mathit{ftv}(\mathtt{T}) \subseteq \mathtt{X}, \overline{\mathtt{X}}$ and $\mathit{ftv}(\mathtt{T}') \subseteq \overline{\mathtt{X}}$, it is the case that $([\mathtt{X} \mapsto \mathtt{T}']\mathtt{T})[\overline{\mathtt{r}}] = \mathtt{T}[\mathtt{T}'[\overline{\mathtt{r}}], \overline{\mathtt{r}}]$. □

Proof: This follows by induction on the structure of the type $\mathtt{T}$; for the base case when $\mathtt{T} = \mathtt{X}$, use Lemma 7.6.12. □

7.6.17 LEMMA [FUNDAMENTAL PROPERTY OF THE LOGICAL RELATION]: The logical relation Δ of Definition 7.6.10 has the substitutivity and compatibility properties defined in Figure 7-4. □

Proof: The first substitutivity property in Figure 7-4 (closure under substituting values for value variables) holds for Δ because of the way it is defined in terms of closing substitutions. The second substitutivity property (closure under substituting types for types variables) holds for Δ because of Lemma 7.6.16.

Now consider the compatibility properties given in Figure 7-4. There is one for each clause in the grammar of F_{ML} terms and values (Figure 7-1). We consider each in turn, giving the details in some cases and setting the others as exercises (with solutions).

Value variables: This case is immediate from the definition of Δ in Definition 7.6.10.

Constants: We have to show for each constant $\mathtt{c}$, with $\mathit{Typeof}(\mathtt{c}) = \mathtt{Gnd}$ say, that $(\mathtt{c}, \mathtt{c}) \in \mathtt{Gnd}[\overline{\mathtt{r}}] = (\mathit{Id}_{\mathtt{Gnd}})^{s\,t}$. But by definition of $\mathit{Id}_{\mathtt{Gnd}}$ (Figure 7-5), $(\mathtt{c}, \mathtt{c}) \in \mathit{Id}_{\mathtt{Gnd}}$; and $\mathit{Id}_{\mathtt{Gnd}} \subseteq (\mathit{Id}_{\mathtt{Gnd}})^{s\,t}$ by Lemma 7.6.5.

Recursive functions: Using property (7.19) and the fact that each $\mathtt{T}[\overline{\mathtt{r}}]$ is valuable and hence closed (Lemma 7.6.12), the compatibility property for recursive functions reduces to proving the property in Exercise 7.6.18.

Record values: This case follows from the property in Exercise 7.6.19.

Type abstractions: This case follows from the property in Exercise 7.6.20.

Package values: This case follows easily from the definition of $\{\exists \mathtt{r}, R(\mathtt{r})\}$ in Figure 7-5, using Lemma 7.6.16.

Conditionals: This case follows from the property in Exercise 7.6.21.

Operations: In view of Lemma 7.6.15, this compatibility property follows once we prove $(\mathtt{op}(\mathtt{c_i}\ ^{i\in 1..n}), \mathtt{op}(\mathtt{c_i}\ ^{i\in 1..n})) \in (\mathit{Id}_{\mathtt{Gnd}})^{st}$ for any (suitably typed) constants $\mathtt{c_i}$ and operator $\mathtt{op}$. But if the value of $\mathtt{op}(\mathtt{c_i}\ ^{i\in 1..n})$ is the constant $\mathtt{c}$ say, then for any S

$$\langle S, \mathtt{op}(\mathtt{c_i}\ ^{i\in 1..n})\rangle\downarrow\ \Leftrightarrow\ \langle S, \mathtt{c}\rangle\downarrow.$$

Hence for any $(S, S') \in (\mathit{Id}_{\mathtt{Gnd'}})^{s}$ (where $\mathtt{Gnd}' = \mathit{Typeof}(\mathtt{c})$), we have

$$\begin{aligned}\langle S, \mathtt{op}(\mathtt{c_i}\ ^{i\in 1..n})\rangle\downarrow &\Leftrightarrow \langle S, \mathtt{c}\rangle\downarrow\\ &\Leftrightarrow \langle S', \mathtt{c}\rangle\downarrow && \text{(since } (\mathtt{c},\mathtt{c}) \in \mathit{Id}_{\mathtt{Gnd'}})\\ &\Leftrightarrow \langle S', \mathtt{op}(\mathtt{c_i}\ ^{i\in 1..n})\rangle\downarrow.\end{aligned}$$

So we do indeed have $(\mathtt{op}(\mathtt{c_i}\ ^{i\in 1..n}), \mathtt{op}(\mathtt{c_i}\ ^{i\in 1..n})) \in (\mathit{Id}_{\mathtt{Gnd}})^{st}$.

Applications: This case amounts to proving that if recursive function values $\mathtt{v}$ and $\mathtt{v}'$ satisfy $(\mathtt{v}, \mathtt{v}') \in \mathtt{fun}(r_1, r_2)^{st}$ for some closed term-relations r_1 and r_2, then for any $(\mathtt{v_1}, \mathtt{v_1'}) \in r_1$ it is the case that $(\mathtt{v}\ \mathtt{v_1}, \mathtt{v}'\ \mathtt{v_1'}) \in r_2$. But this property follows immediately from the definition of $\mathtt{fun}(-,-)$ using the first part of Lemma 7.6.13: for

$$\begin{aligned}(\mathtt{v}, \mathtt{v}') \in\ & \mathtt{fun}(r_1, r_2)^{stv}\\ &= \mathtt{fun}(r_1, (r_2)^{st})^{stv} && \text{(since } r_2 \text{ is closed)}\\ &= \mathtt{fun}(r_1, (r_2)^{st}) && \text{(by (7.19))}\\ &= \mathtt{fun}(r_1, r_2) && \text{(since } r_2 \text{ is closed).}\end{aligned}$$

Projections: This case is similar to the previous one, but using property (7.26) from Exercise 7.6.14 rather than (7.19).

Type applications: This case is similar to the previous one, using property (7.27) from Exercise 7.6.14.

Unpacking: This case follows from the property in Exercise 7.6.22.

Sequencing: This case follows from the property in Exercise 7.6.23. □

7.6.18 EXERCISE [RECOMMENDED, ★★★]: Suppose

$$\begin{aligned}\mathtt{F} &\overset{\text{def}}{=} \mathtt{fun\ f(x{:}T_1)=t{:}T_2} \in \mathit{Val}(\mathtt{T_1}\rightarrow\mathtt{T_2})\\ \mathtt{F}' &\overset{\text{def}}{=} \mathtt{fun\ f(x{:}T_1')=t'{:}T_2'} \in \mathit{Val}(\mathtt{T_1'}\rightarrow\mathtt{T_2'})\\ r_1 &\in \mathit{TRel}(\mathtt{T_1}, \mathtt{T_1'})\\ r_2 &\in \mathit{TRel}(\mathtt{T_2}, \mathtt{T_2'})\end{aligned}$$

satisfy $r_2 = (r_2)^{st}$ and

$$\begin{aligned}&([\mathtt{f} \mapsto \mathtt{v}][\mathtt{x} \mapsto \mathtt{v_1}]\mathtt{t}, [\mathtt{f} \mapsto \mathtt{v}'][\mathtt{x} \mapsto \mathtt{v_1'}]\mathtt{t}') \in r_2,\\ &\text{for all } (\mathtt{v}, \mathtt{v}') \in \mathtt{fun}(r_1, r_2) \text{ and } (\mathtt{v_1}, \mathtt{v_1'}) \in (r_1)^{v}.\end{aligned} \tag{7.29}$$

Use the admissibility property of valuable term-relations established in Lemma 7.6.8 to show that $(\mathtt{F}, \mathtt{F}') \in \mathtt{fun}(r_1, r_2)$. □

7.6.19 EXERCISE [★★]: Suppose for $i \in 1..n$ that $v_i \in Val(T_i)$, $v_i' \in Val(T_i')$ and $r_i \in TRel(T_i, T_i')$ with $r_i = (r_i)^{st}$. Putting

$$v \overset{\text{def}}{=} \{l_i{=}v_i\ ^{i\in 1..n}\} \in Val(\{l_i{:}T_i\ ^{i\in 1..n}\})$$

$$v' \overset{\text{def}}{=} \{l_i{=}v_i'\ ^{i\in 1..n}\} \in Val(\{l_i{:}T_i'\ ^{i\in 1..n}\})$$

show that if $(v_i, v_i') \in r_i$ for $i \in 1..n$, then (v, v') is in the value-relation $\{l_i{=}r_i\ ^{i\in 1..n}\}$ defined in Figure 7-5. □

7.6.20 EXERCISE [★★]: Let T and T' be types with at most X free. For each $T_1, T_1' \in Typ$ and $r \in TRel(T_1, T_1')$ suppose we are given a closed term-relation $R(r)$ in $TRel([X \mapsto T_1]T, [X \mapsto T_1']T')$ (i.e. $R(r) = R(r)^{st}$). Show that if the values v and v' satisfy

$$X \vdash v : T$$

$$X \vdash v' : T'$$

$$\forall T_1, T_1' \in Typ, r \in TRel(T_1, T_1').\ ([X \mapsto T_1]v, [X \mapsto T_1']v') \in R(r)$$

then $(\lambda X.v, \lambda X.v')$ is in the value-relation $\lambda r.\,R(r)$ defined in Figure 7-5. □

7.6.21 EXERCISE [★★]: Suppose $(v, v') \in (Id_{\mathtt{Bool}})^{st}$ and $(t_1, t_1'), (t_2, t_2') \in r$, where $r \in TRel(T, T')$ is closed (i.e. $r = (r)^{st}$). Show that

$$(\mathtt{if}\ v\ \mathtt{then}\ t_1\ \mathtt{else}\ t_2, \mathtt{if}\ v'\ \mathtt{then}\ t_1'\ \mathtt{else}\ t_2')$$

is in r. □

7.6.22 EXERCISE [★★]: Let T and T' be types with at most X free. For each $T_1, T_1' \in Typ$ and $r_1 \in TRel(T_1, T_1')$ suppose we are given a closed term-relation $R(r_1) = R(r_1)^{st}$ in $TRel([X \mapsto T_1]T, [X \mapsto T_1']T')$. Suppose we are also given a closed term-relation $r_2 = (r_2)^{st} \in TRel(T_2, T_2')$ for some closed types $T_2, T_2' \in Typ$. Show that if the terms t, t' satisfy

$$X, x : T \vdash t : T_2$$

$$X, x : T' \vdash t' : T_2'$$

$$\forall T_1, T_1' \in Typ, r_1 \in TRel(T_1, T_1'), (v_1, v_1') \in (r_1)^{v}.$$

$$([X \mapsto T_1][x \mapsto v_1]t, [X \mapsto T_1][x \mapsto v_1]t) \in r_2$$

then whenever $(v, v') \in \{\exists r_1, R(r_1)\}^{stv}$, it is also the case that

$$(\mathtt{let}\ \{{*}X, x\}{=}v\ \mathtt{in}\ t, \mathtt{let}\ \{{*}X, x\}{=}v'\ \mathtt{in}\ t')$$

is in r_2. □

7.6.23 EXERCISE [★★]: Suppose we are given $r_1 \in TRel(\mathtt{T}_1, \mathtt{T}_1')$, $r_2 \in TRel(\mathtt{T}_2, \mathtt{T}_2')$ with r_1 valuable (i.e. $r_1 = (r_1)^{vst}$) and r_2 closed (i.e. $r_2 = (r_2)^{st}$). Show that if the terms $\mathtt{t}_2, \mathtt{t}_2'$ satisfy

$$\mathtt{x} : \mathtt{T}_1 \vdash \mathtt{t}_2 : \mathtt{T}_2$$
$$\mathtt{x} : \mathtt{T}_1' \vdash \mathtt{t}_2' : \mathtt{T}_2'$$
$$\forall (\mathtt{v}_1, \mathtt{v}_1') \in (r_1)^v.\ ([\mathtt{x} \mapsto \mathtt{v}_1]\mathtt{t}_2, [\mathtt{x} \mapsto \mathtt{v}_1']\mathtt{t}_2') \in r_2$$

then whenever $(\mathtt{t}_1, \mathtt{t}_1') \in r_1$, it is also the case that

$$(\mathtt{let}\ \mathtt{x}{=}\mathtt{t}_1\ \mathtt{in}\ \mathtt{t}_2, \mathtt{let}\ \mathtt{x}{=}\mathtt{t}_1'\ \mathtt{in}\ \mathtt{t}_2')$$

is in r_2. □

7.6.24 LEMMA [ADEQUACY]: The logical relation Δ is adequate (Definition 7.5.1). □

Proof: Suppose $\emptyset \vdash \mathtt{t}\ \Delta\ \mathtt{t}' : \mathtt{T}$; we have to show that $\mathtt{t}\downarrow$ holds iff $\mathtt{t}'\downarrow$ does, or equivalently that

$$\langle Id, \mathtt{t}\rangle\downarrow \quad \text{iff} \quad \langle Id, \mathtt{t}'\rangle\downarrow. \tag{7.30}$$

Unravelling Definition 7.6.10, the assumption that the closed terms $\mathtt{t}$ and $\mathtt{t}'$ of closed type $\mathtt{T}$ are Δ-related means that $(\mathtt{t}, \mathtt{t}') \in \mathtt{T}[]$, the latter being the action of the type $\mathtt{T}$ on the empty list of term-relations. By Lemma 7.6.12, $\mathtt{T}[]$ is valuable; so $(\mathtt{t}, \mathtt{t}') \in \mathtt{T}[]^{vst}$. Hence to prove (7.30), it suffices to show that $(Id, Id) \in (\mathtt{T}[]^v)^s$; but for any $(\mathtt{v}, \mathtt{v}') \in \mathtt{T}[]^v$,

$$\langle Id, \mathtt{v}\rangle\downarrow \quad \text{iff} \quad \langle Id, \mathtt{v}'\rangle\downarrow$$

holds trivially by axiom (S-NILVAL) in Figure 7-2. □

We are finally able to put all the pieces together and prove the main result of this section. At the same time we complete the proof of Theorem 7.5.7.

7.6.25 THEOREM [$=_{\mathrm{ctx}}$ EQUALS Δ EQUALS $=_{\mathrm{ciu}}$]: F_{ML} contextual equivalence, $=_{\mathrm{ctx}}$, (defined in Theorem 7.5.3) coincides with the logical relation Δ of Definition 7.6.10 and with ciu-equivalence, $=_{\mathrm{ciu}}$ (Definition 7.5.5): $\Gamma \vdash \mathtt{t} =_{\mathrm{ctx}} \mathtt{t}' : \mathtt{T}$ holds if and only if $\Gamma \vdash \mathtt{t}\ \Delta\ \mathtt{t}' : \mathtt{T}$ does, if and only if $\Gamma \vdash \mathtt{t} =_{\mathrm{ciu}} \mathtt{t}' : \mathtt{T}$ does. □

Proof: It suffices to show that the following chain of inclusions holds:

$$=_{\mathrm{ctx}} \overset{(1)}{\subseteq} =_{\mathrm{ciu}} \overset{(3)}{\subseteq} \Delta \overset{(2)}{\subseteq} =_{\mathrm{ctx}}.$$

(1) This is the half of Theorem 7.5.7 that we have already proved in §7.5.

(2) We have not yet shown that Δ is an equivalence relation; and in fact we will only deduce this once we have shown that it coincides with $=_{\mathrm{ctx}}$ and $=_{\mathrm{ciu}}$ (which are easily seen to be equivalence relations). However, we have shown that Δ is compatible, substitutive and adequate (Lemmas 7.6.17 and 7.6.24). In the proof of Theorem 7.5.3 we constructed $=_{\mathrm{ctx}}$ as the union of all such type-respecting relations, without regard to whether they were also equivalence relations; therefore Δ is contained in $=_{\mathrm{ctx}}$.

(3) Noting how $=_{\mathrm{ciu}}$ and Δ are defined on open terms via substitutions, we can combine the first part of Lemma 7.6.8 with Lemma 7.6.12 to conclude that

$$\Gamma \vdash \mathtt{t} =_{\mathrm{ciu}} \mathtt{t}' : \mathtt{T} \ \& \ \Gamma \vdash \mathtt{t}' \,\Delta\, \mathtt{t}'' : \mathtt{T} \Rightarrow \Gamma \vdash \mathtt{t} \,\Delta\, \mathtt{t}'' : \mathtt{T}. \tag{7.31}$$

We noted in the proof of Theorem 7.5.3 that every compatible term-relation is reflexive. (This is easily proved by induction on the structure of terms.) So since Δ is compatible (Lemma 7.6.17) it is in particular reflexive. So we can take $\mathtt{t}' = \mathtt{t}''$ in (7.31) to deduce that $\Gamma \vdash \mathtt{t} =_{\mathrm{ctx}} \mathtt{t}' : \mathtt{T}$ implies $\Gamma \vdash \mathtt{t} \,\Delta\, \mathtt{t}' : \mathtt{T}$. □

7.7 Operational Extensionality

In this section we develop some of the consequences of Theorem 7.6.25. Now that we know that contextual equivalence coincides with ciu-equivalence (Theorem 7.5.7), when giving general properties of $=_{\mathrm{ctx}}$ we restrict attention to closed terms of closed type where possible, since the corresponding property for open terms can be obtained via closing substitutions.

7.7.1 THEOREM [EXTENSIONALITY FOR VALUES]: We give extensionality principles for the various types of value; for package values, the principle is a formalisation of the final one discussed in the Introduction (Principle 7.3.6).

1. CONSTANTS: Given constants $\mathtt{c}$, $\mathtt{c}'$ of the same ground type, $\mathtt{Gnd}$ say, $\emptyset \vdash \mathtt{c} =_{\mathrm{ctx}} \mathtt{c}' : \mathtt{Gnd}$ holds if and only if $\mathtt{c} = \mathtt{c}'$.

2. FUNCTIONS: Given $\mathtt{f}{:}\mathtt{T}_1{\to}\mathtt{T}_2, \mathtt{x}{:}\mathtt{T}_1 \vdash \mathtt{t} : \mathtt{T}_2$ and $\mathtt{f}{:}\mathtt{T}_1{\to}\mathtt{T}_2, \mathtt{x}{:}\mathtt{T}_1 \vdash \mathtt{t}' : \mathtt{T}_2$, writing $\mathtt{v}$ and $\mathtt{v}'$ for the recursive function values $\mathtt{fun\ f(x{:}T_1){=}t{:}T_2}$ and $\mathtt{fun\ f(x{:}T_1){=}t'{:}T_2}$ respectively, then $\emptyset \vdash \mathtt{v} =_{\mathrm{ctx}} \mathtt{v}' : \mathtt{T}_1{\to}\mathtt{T}_2$ if and only if for all $\emptyset \vdash \mathtt{v}_1 : \mathtt{T}_1$, it is the case that $\emptyset \vdash [\mathtt{f} \mapsto \mathtt{v}][\mathtt{x} \mapsto \mathtt{v}_1]\mathtt{t} =_{\mathrm{ctx}} [\mathtt{f} \mapsto \mathtt{v}'][\mathtt{x} \mapsto \mathtt{v}_1]\mathtt{t}' : \mathtt{T}_2$.

3. RECORDS: Given values $\emptyset \vdash \mathtt{v}_i : \mathtt{T}_i$ and $\emptyset \vdash \mathtt{v}'_i : \mathtt{T}_i$ for $i \in 1..n$, then $\emptyset \vdash \{\mathtt{l}_i\mathtt{=}\mathtt{v}_i\ ^{i\in 1..n}\} =_{\mathrm{ctx}} \{\mathtt{l}_i\mathtt{=}\mathtt{v}'_i\ ^{i\in 1..n}\} : \{\mathtt{l}_i\mathtt{:}\mathtt{T}_i\ ^{i\in 1..n}\}$ if and only if for each $i \in 1..n$, $\emptyset \vdash \mathtt{v}_i =_{\mathrm{ctx}} \mathtt{v}'_i : \mathtt{T}_i$.

4. TYPE ABSTRACTIONS: Given $\mathtt{X} \vdash \mathtt{v} : \mathtt{T}$ and $\mathtt{X} \vdash \mathtt{v}' : \mathtt{T}$, then $\emptyset \vdash \lambda\mathtt{X}.\mathtt{v} =_{\mathrm{ctx}} \lambda\mathtt{X}.\mathtt{v}' : \forall\mathtt{X}.\mathtt{T}$ if and only if for all closed types $\mathtt{T}'$, $\emptyset \vdash [\mathtt{X} \mapsto \mathtt{T}']\mathtt{v} =_{\mathrm{ctx}} [\mathtt{X} \mapsto \mathtt{T}']\mathtt{v}' : [\mathtt{X} \mapsto \mathtt{T}']\mathtt{T}$.

5. PACKAGES: For any closed existential type $\{\exists\mathtt{X},\mathtt{T}\}$, closed types $\mathtt{T}_1$, $\mathtt{T}_2$, and values $\emptyset \vdash \mathtt{v}_i : [\mathtt{X} \mapsto \mathtt{T}_i]\mathtt{T}$ $(i = 1, 2)$,

$$\emptyset \vdash \{\mathtt{*T}_1,\mathtt{v}_1\}\ \mathtt{as}\ \{\exists\mathtt{X},\mathtt{T}\} =_{\mathrm{ctx}} \{\mathtt{*T}_2,\mathtt{v}_2\}\ \mathtt{as}\ \{\exists\mathtt{X},\mathtt{T}\} : \{\exists\mathtt{X},\mathtt{T}\}$$

holds if there is some term-relation $r \in \mathit{TRel}(\mathtt{T}_1, \mathtt{T}_2)$ with $(\mathtt{v}_1, \mathtt{v}_2) \in \mathtt{T}[r]$. □

Proof:

1. The property for constants follows from Lemma 7.6.15 combined with Theorem 7.6.25.

2. Suppose for all $\emptyset \vdash \mathtt{v}_1 : \mathtt{T}_1$ that

$$\emptyset \vdash [\mathtt{f} \mapsto \mathtt{v}][\mathtt{x} \mapsto \mathtt{v}_1]\mathtt{t} =_{\mathrm{ctx}} [\mathtt{f} \mapsto \mathtt{v}'][\mathtt{x} \mapsto \mathtt{v}_1]\mathtt{t}' : \mathtt{T}_2 \tag{7.32}$$

where $\mathtt{v}$ and $\mathtt{v}'$ are as in part 2 of the theorem. To show $\emptyset \vdash \mathtt{v} =_{\mathrm{ctx}} \mathtt{v}' : \mathtt{T}_1{\to}\mathtt{T}_2$, by Theorem 7.6.25 it suffices to show $\emptyset \vdash \mathtt{v}\ \Delta\ \mathtt{v}' : \mathtt{T}_1{\to}\mathtt{T}_2$, i.e. that $(\mathtt{v},\mathtt{v}') \in (\mathtt{T}_1{\to}\mathtt{T}_2)[] = \mathtt{fun}(\mathtt{T}_1[], \mathtt{T}_2[])^{st}$. In fact we show that $(\mathtt{v},\mathtt{v}') \in \mathtt{fun}(\mathtt{T}_1[], \mathtt{T}_2[])$. For this we have to prove that if $(\mathtt{v}_1, \mathtt{v}'_1) \in \mathtt{T}_1[]^{v}$, then $(\mathtt{v}\ \mathtt{v}_1, \mathtt{v}'\ \mathtt{v}'_1) \in \mathtt{T}_2[]$. By Theorem 7.6.25 again, this is the same as showing: if $\emptyset \vdash \mathtt{v}_1 =_{\mathrm{ctx}} \mathtt{v}'_1 : \mathtt{T}_1$, then $\emptyset \vdash \mathtt{v}\ \mathtt{v}_1 =_{\mathrm{ctx}} \mathtt{v}'\ \mathtt{v}'_1 : \mathtt{T}_2$. As noted in Corollary 7.5.8, we can turn the primitive reduction for function application into a ciu-equivalence and hence by Theorem 7.6.25 into a contextual equivalence:

$$\emptyset \vdash \mathtt{v}\ \mathtt{v}_1 =_{\mathrm{ctx}} [\mathtt{f} \mapsto \mathtt{v}][\mathtt{x} \mapsto \mathtt{v}_1]\mathtt{t} : \mathtt{T}_2 \tag{7.33}$$

and similarly for $\mathtt{v}'\ \mathtt{v}'_1$. Therefore we just need to show: if $\emptyset \vdash \mathtt{v}_1 =_{\mathrm{ctx}} \mathtt{v}'_1 : \mathtt{T}_1$, then $\emptyset \vdash [\mathtt{f} \mapsto \mathtt{v}][\mathtt{x} \mapsto \mathtt{v}_1]\mathtt{t} =_{\mathrm{ctx}} [\mathtt{f} \mapsto \mathtt{v}'][\mathtt{x} \mapsto \mathtt{v}'_1]\mathtt{t}' : \mathtt{T}_2$. But this follows from the assumption (7.32) using the reflexivity and substitutivity properties of $=_{\mathrm{ctx}}$. So we have established one half (the difficult half) of the property in 2. For the converse, if $\emptyset \vdash \mathtt{v} =_{\mathrm{ctx}} \mathtt{v}' : \mathtt{T}_1{\to}\mathtt{T}_2$, then for any $\emptyset \vdash \mathtt{v}_1 : \mathtt{T}_1$, the compatibility properties of $=_{\mathrm{ctx}}$ give $\emptyset \vdash \mathtt{v}\ \mathtt{v}_1 =_{\mathrm{ctx}} \mathtt{v}'\ \mathtt{v}_1 : \mathtt{T}_2$; and then as before, we can compose with (7.33) to get (7.32).

3. We leave the extensionality property for records as an exercise (Exercise 7.7.2).

4. For the property for type abstractions, suppose

$$\forall \mathtt{T}' \in \mathit{Typ}. \quad \emptyset \vdash [\mathtt{X} \mapsto \mathtt{T}']\mathtt{v} =_{\mathrm{ctx}} [\mathtt{X} \mapsto \mathtt{T}']\mathtt{v}' \mathbin{\mathtt{:}} [\mathtt{X} \mapsto \mathtt{T}']\mathtt{T}. \tag{7.34}$$

   Note that since Δ coincides with $=_{\mathrm{ctx}}$ (Theorem 7.6.25) it is reflexive and hence $\mathtt{X} \vdash \mathtt{v}\ \Delta\ \mathtt{v} \mathbin{\mathtt{:}} \mathtt{T}$ holds. According to Definition 7.6.10 this means that for all $\mathtt{T}_1, \mathtt{T}_1' \in \mathit{Typ}$ and $r \in \mathit{TRel}(\mathtt{T}_1, \mathtt{T}_1')$, $([\mathtt{X} \mapsto \mathtt{T}_1]\mathtt{v}, [\mathtt{X} \mapsto \mathtt{T}_1']\mathtt{v}) \in \mathtt{T}[r]$. Since $\mathtt{T}[r]$ is closed (Lemma 7.6.12), we can combine (7.34) with the first part of Lemma 7.6.8 (using $=_{\mathrm{ctx}}$ in place of $=_{\mathrm{ciu}}$ by virtue of Theorem 7.6.25) to conclude that $([\mathtt{X} \mapsto \mathtt{T}_1]\mathtt{v}, [\mathtt{X} \mapsto \mathtt{T}_1']\mathtt{v}') \in \mathtt{T}[r]$ for all r. Then using the equivalence in Corollary 7.5.8(v), we have

$$\forall \mathtt{T}_1, \mathtt{T}_1' \in \mathit{Typ}, r \in \mathit{TRel}(\mathtt{T}_1, \mathtt{T}_1'). \quad (\,(\lambda\mathtt{X.v})\,\mathtt{T}_1,\ (\lambda\mathtt{X.v}')\,\mathtt{T}_1') \in \mathtt{T}[r]$$

   and hence $(\lambda\mathtt{X.v}, \lambda\mathtt{X.v}')$ is in $\lambda r\,.\,\mathtt{T}[r]$. Since $\lambda r\,.\,\mathtt{T}[r] \subseteq (\lambda r\,.\,\mathtt{T}[r])^{st}$ and the latter is equal to $(\forall\mathtt{X.T})[]$ by definition, we have $\emptyset \vdash \lambda\mathtt{X.v}\ \Delta\ \lambda\mathtt{X.v}' \mathbin{\mathtt{:}} \forall\mathtt{X.T}$, and hence by Theorem 7.6.25, $\emptyset \vdash \lambda\mathtt{X.v} =_{\mathrm{ctx}} \lambda\mathtt{X.v}' \mathbin{\mathtt{:}} \forall\mathtt{X.T}$. So we have established one half (the difficult half) of the property in 4. The argument for the other half is similar to that for property 2, using Corollary 7.5.8(v) and the congruence properties of $=_{\mathrm{ctx}}$.

5. Finally, let us consider the extensionality property for package values. (Note that unlike the other four, this only gives a sufficient condition for contextual equivalence; Example 7.7.4 below shows that the condition is not necessary.) If $(\mathtt{v}_1, \mathtt{v}_2) \in \mathtt{T}[r]$, then from Definition 7.6.9 we have

$$\begin{aligned}(\{{*}\mathtt{T}_1,\mathtt{v}_1\}\ \mathtt{as}\ \{\exists\mathtt{X},\mathtt{T}\}, \{{*}\mathtt{T}_2,\mathtt{v}_2\}\ \mathtt{as}\ \{\exists\mathtt{X},\mathtt{T}\}) &\in \{\exists r, \mathtt{T}[r]\}\\ &\subseteq \{\exists r, \mathtt{T}[r]\}^{st}\\ &= \{\exists\mathtt{X},\mathtt{T}\}[].\end{aligned}$$

   Thus $\emptyset \vdash \{{*}\mathtt{T}_1,\mathtt{v}_1\}\ \mathtt{as}\ \{\exists\mathtt{X},\mathtt{T}\}\ \Delta\ \{{*}\mathtt{T}_2,\mathtt{v}_2\}\ \mathtt{as}\ \{\exists\mathtt{X},\mathtt{T}\} \mathbin{\mathtt{:}} \{\exists\mathtt{X},\mathtt{T}\}$ and we can apply Theorem 7.6.25 to get the desired contextual equivalence. □

7.7.2 EXERCISE [★★, ↛]: Use Theorem 7.6.25, Corollary 7.5.8 and the definition of the term-relation $\{\mathtt{l}_i\mathtt{=}r_i\ ^{i\in 1..n}\}$ in Definition 7.6.9 to deduce extensionality property 3 of Theorem 7.7.1. □

To see how Theorem 7.7.1(5) can be used in practice, we will apply it to establish the contextual equivalence of Example 7.3.5 from the Introduction.

7.7.3 EXAMPLE: Recall from Example 7.3.5 the type $\mathtt{Semaphore}$ and its values $\mathtt{semaphore}_1$, $\mathtt{semaphore}_2$. To prove $\emptyset \vdash \mathtt{semaphore}_1 =_{\mathrm{ctx}} \mathtt{semaphore}_2 : \mathtt{Semaphore}$ using Theorem 7.7.1(5), it suffices to show that $(\mathtt{v}_1, \mathtt{v}_2) \in \mathtt{T}[r]$ where

$$\begin{aligned}
\mathtt{T} &\stackrel{\text{def}}{=} \{\mathtt{bit{:}X,\ flip{:}X{\to}X,\ read{:}X{\to}Bool}\} \\
\mathtt{v}_1 &\stackrel{\text{def}}{=} \{\mathtt{bit{=}true,\ flip{=}\lambda x{:}Bool.not\ x,\ read{=}\lambda x{:}Int.x}\} \\
\mathtt{v}_2 &\stackrel{\text{def}}{=} \{\mathtt{bit{=}1,\ flip{=}\lambda x{:}Int.0{-}2{*}x,\ read{=}\lambda x{:}Int.x >= 0}\}
\end{aligned}$$

and $r \in \mathit{VRel}(\mathtt{Bool}, \mathtt{Int})$ is

$$\begin{aligned}
r \stackrel{\text{def}}{=}\ &\{(\mathtt{true}, m) \mid m = (-2)^n \text{ for some even } n \geq 0\} \cup \\
&\{(\mathtt{false}, m) \mid m = (-2)^n \text{ for some odd } n \geq 0\}.
\end{aligned}$$

Since r is a value-relation, we can use Lemma 7.6.13 to slightly simplify $\mathtt{T}[r]$:

$$\begin{aligned}
\mathtt{T}[r] &\stackrel{\text{def}}{=} \{\mathtt{bit}{=}r^{st}, \mathtt{flip}{=}\mathtt{fun}(r^{st}, r^{st})^{st}, \mathtt{read}{=}\mathtt{fun}(r^{st}, \mathit{Id}^{st}_{\mathtt{Bool}})^{st}\}^{st} \\
&= \{\mathtt{bit}{=}r^{st}, \mathtt{flip}{=}\mathtt{fun}(r, r^{st})^{st}, \mathtt{read}{=}\mathtt{fun}(r, \mathit{Id}^{st}_{\mathtt{Bool}})^{st}\}^{st}.
\end{aligned}$$

So since $(-)^{st}$ is inflationary, to prove $(\mathtt{v}_1, \mathtt{v}_2) \in \mathtt{T}[r]$, it suffices to show

$$\begin{aligned}
(\mathtt{true}, 1) &\in r \\
(\mathtt{\lambda x{:}Bool.not\ x}, \mathtt{\lambda x{:}Int.0{-}2{*}x}) &\in \mathtt{fun}(r, r^{st}) \\
(\mathtt{\lambda x{:}Int.x}, \mathtt{\lambda x{:}Int.x >= 0}) &\in \mathtt{fun}(r, \mathit{Id}^{st}_{\mathtt{Bool}}).
\end{aligned}$$

These follow from the definition of r—the first trivially and the second two once we combine the definition of $\mathtt{fun}(-,-)$ with the fact (Lemma 7.6.8) that closed relations such as r^{st} and $\mathit{Id}^{st}_{\mathtt{Bool}}$ respect ciu-equivalence. For example, if $(\mathtt{v}_1, \mathtt{v}_1') \in r$, then $(\mathtt{\lambda x{:}Bool.not\ x})\,\mathtt{v}_1$ and $(\mathtt{\lambda x{:}Int.0{-}2{*}x})\,\mathtt{v}_1'$ are ciu-equivalent to r-related values $\mathtt{v}_2$ and $\mathtt{v}_2'$; then since $(\mathtt{v}_2, \mathtt{v}_2') \in r \subseteq r^{st}$ and the latter is closed, we have $((\mathtt{\lambda x{:}Bool.not\ x})\,\mathtt{v}_1,\ (\mathtt{\lambda x{:}Int.0{-}2{*}x})\,\mathtt{v}_1') \in r^{st}$. As this holds for all $(\mathtt{v}_1, \mathtt{v}_1') \in r$, we have $(\mathtt{\lambda x{:}Bool.not\ x}, \mathtt{\lambda x{:}Int.0{-}2{*}x})$ in $\mathtt{fun}(r, r^{st})$. □

Theorem 7.7.1(5) gives a sufficient condition for contextual equivalence of package values, but the condition is not necessary: it can be the case that $\{\mathtt{*\,T}_1\mathtt{,\ v}_1\}\ \mathtt{as}\ \{\exists\mathtt{X,\ T}\}$ is contextually equivalent to $\{\mathtt{*\,T}_2\mathtt{,\ v}_2\}\ \mathtt{as}\ \{\exists\mathtt{X,\ T}\}$ even though there is no $r \in \mathit{TRel}(\mathtt{T}_1, \mathtt{T}_2)$ with $(\mathtt{v}_1, \mathtt{v}_2) \in \mathtt{T}[r]$. The rest of this section is devoted to giving an example of this unpleasant phenomenon (based on a suggestion of Ian Stark arising out of our joint work on logical relations for functions and dynamically allocated names in Pitts and Stark, 1993).

7.7.4 EXAMPLE: Consider the following types and terms.

$$\begin{aligned}
\mathtt{P} &\stackrel{\text{def}}{=} \mathtt{(X{\to}Bool){\to}Bool}\\
\mathtt{Q} &\stackrel{\text{def}}{=} \mathtt{\{\exists X,P\}}\\
\mathtt{N} &\stackrel{\text{def}}{=} \mathtt{\forall X.X}\\
\mathtt{diverge} &\stackrel{\text{def}}{=} \mathtt{(fun\,f\,(b{:}Bool) = f\,b : Bool)\,true}\\
\mathtt{G} &\stackrel{\text{def}}{=} \mathtt{fun\,g\,(f{:}N{\to}Bool) = diverge : Bool}\\
\mathtt{G}' &\stackrel{\text{def}}{=} \mathtt{fun\,g\,(f{:}Bool{\to}Bool) =}\\
&\qquad\mathtt{(if\,f\,true\,then}\\
&\qquad\qquad\mathtt{if\,f\,false\,then\,diverge\,else\,true}\\
&\qquad\mathtt{else\,diverge) : Bool}.
\end{aligned}$$

Thus $\mathtt{N}$ is a type with no values (Exercise 7.7.6); $\mathtt{G}$ is a function that diverges when applied to any value of type $\mathtt{N{\to}Bool}$; and $\mathtt{G}'$ is a function that diverges when applied to any value of type $\mathtt{Bool{\to}Bool}$ except ones (such as the identity function) that map $\mathtt{true}$ to $\mathtt{true}$ and $\mathtt{false}$ to $\mathtt{false}$, in which case it returns $\mathtt{true}$. We claim that

(i) there is no $r \in \mathit{TRel}(\mathtt{N},\mathtt{Bool})$ for which $(\mathtt{G},\mathtt{G}') \in \mathtt{P}[r]$ holds,

(ii) but nevertheless $\emptyset \vdash \mathtt{\{*N,G\}\ as\ Q} =_{\mathrm{ctx}} \mathtt{\{*Bool,G'\}\ as\ Q : Q}$. □

Proof: For (i) note that the definition of $\mathtt{N}$ implies that $\mathit{Val}(\mathtt{N}) = \emptyset$, i.e. there are no closed values of type $\mathtt{N}$ (Exercise 7.7.6). So any $r \in \mathit{TRel}(\mathtt{N},\mathtt{Bool})$ satisfies $r^v = \emptyset$. Now

$$\begin{aligned}
\mathtt{P}[r]^v &\stackrel{\text{def}}{=} \mathtt{((X{\to}Bool){\to}Bool)}[r]^v\\
&\stackrel{\text{def}}{=} \mathtt{fun}(\mathtt{(X{\to}Bool)}[r], \mathit{Id}^{st}_{\mathtt{Bool}})^{stv}\\
&= \mathtt{fun}(\mathtt{(X{\to}Bool)}[r], \mathit{Id}^{st}_{\mathtt{Bool}}) && \text{using (7.19)}\\
&\stackrel{\text{def}}{=} \mathtt{fun}(\mathtt{fun}(r^{vst}, \mathit{Id}^{st}_{\mathtt{Bool}})^{st}, \mathit{Id}^{st}_{\mathtt{Bool}})\\
&= \mathtt{fun}(\mathtt{fun}(r^{vst}, \mathit{Id}^{st}_{\mathtt{Bool}})^{stv}, \mathit{Id}^{st}_{\mathtt{Bool}}) && \text{by definition of } \mathtt{fun}(-,-)\\
&= \mathtt{fun}(\mathtt{fun}(r^{vst}, \mathit{Id}^{st}_{\mathtt{Bool}}), \mathit{Id}^{st}_{\mathtt{Bool}}) && \text{using (7.19)}\\
&= \mathtt{fun}(\mathtt{fun}(r, \mathit{Id}^{st}_{\mathtt{Bool}}), \mathit{Id}^{st}_{\mathtt{Bool}}) && \text{using (7.20)}\\
&= \mathtt{fun}(\mathtt{fun}(r^{v}, \mathit{Id}^{st}_{\mathtt{Bool}}), \mathit{Id}^{st}_{\mathtt{Bool}}) && \text{by definition of } \mathtt{fun}(-,-).
\end{aligned}$$

Since $r^v = \emptyset$, we have $\mathtt{fun}(r^v, \mathit{Id}^{st}_{\mathtt{Bool}}) = \mathit{Val}(\mathtt{N{\to}Bool}) \times \mathit{Val}(\mathtt{Bool{\to}Bool})$; and we know by Theorem 7.6.25 that $\mathit{Id}^{st}_{\mathtt{Bool}}$ is the relation $\{(\mathtt{t},\mathtt{t}') \mid \emptyset \vdash \mathtt{t} =_{\mathrm{ctx}}$

$\mathtt{t'}$: $\mathtt{Bool}\}$. Therefore

$$\begin{aligned}\mathtt{P}[\mathtt{r}]^v = \{(\mathtt{v}, \mathtt{v'}) \mid\ & \emptyset \vdash \mathtt{v}\,\mathtt{v}_1 =_{\mathrm{ctx}} \mathtt{v'}\,\mathtt{v}'_1 : \mathtt{Bool} \\ & \text{for all } \mathtt{v}_1 \in \mathit{Val}(\mathtt{N}{\to}\mathtt{Bool}) \text{ and } \mathtt{v}'_1 \in \mathit{Val}(\mathtt{Bool}{\to}\mathtt{Bool})\,\}.\end{aligned}$$

However, $\emptyset \vdash \mathtt{G}\,\mathtt{v}_1 =_{\mathrm{ctx}} \mathtt{G'}\,\mathtt{v}'_1 : \mathtt{Bool}$ does not hold if we take $\mathtt{v}_1$ and $\mathtt{v}'_1$ to be the values

$$\mathtt{v}_1 \stackrel{\text{def}}{=} \mathtt{fun\ f\,(x\!:\!N) = diverge : Bool}$$

$$\mathtt{v}'_1 \stackrel{\text{def}}{=} \mathtt{fun\ f\,(x\!:\!Bool) = x : Bool}$$

since evaluation of $\mathtt{G}\,\mathtt{v}_1$ does not terminate, whereas evaluation of $\mathtt{G'}\,\mathtt{v}'_1$ does. Therefore $(\mathtt{G}, \mathtt{G'}) \notin \mathtt{P}[\mathtt{r}]^v$, for any $\mathtt{r} \in \mathit{TRel}(\mathtt{N}, \mathtt{Bool})$.

Turning to the proof of (ii), now we know that it cannot be deduced from the extensionality principle for package values in Theorem 7.7.1, we have to prove this contextual equivalence by brute force. The termination relation defined in Fig. 7-2 provides a possible strategy (if rather a tedious one) for proving ciu-equivalences and hence contextual equivalences—by what one might call *termination induction*. Thus to prove (ii) it suffices to prove that the two terms are ciu-equivalent:

$$\forall S.\quad \langle S, \mathtt{\{*N,G\}\ as\ Q}\rangle\downarrow\ \Leftrightarrow\ \langle S, \mathtt{\{*Bool,G'\}\ as\ Q}\rangle\downarrow.$$

Attempting to do this by induction on the derivation of terminations $\langle -, -\rangle\downarrow$ (for all S simultaneously), one rapidly realises that a stronger induction hypothesis is needed: prove for all frame stacks S and terms $\mathtt{t}$ that

$$\begin{aligned}&\langle[\mathtt{x} \mapsto \mathtt{\{*N,G\}\ as\ Q}]S, [\mathtt{x} \mapsto \mathtt{\{*N,G\}\ as\ Q}]\mathtt{t}\rangle\downarrow\\ &\text{if and only if } \langle[\mathtt{x} \mapsto \mathtt{\{*Bool,G'\}\ as\ Q}]S, [\mathtt{x} \mapsto \mathtt{\{*Bool,G'\}\ as\ Q}]\mathtt{t}\rangle\downarrow.\end{aligned}$$

It is possible to prove this by induction on the definition of the termination relation in Fig. 7-2 (for all S and $\mathtt{t}$ simultaneously). We omit the details except to note that the only difficult induction step is for the primitive reduction (R-UNPACKPACK) in Fig. 7.4.2 in the case that $\mathtt{t}$ is the form $\mathtt{let\,\{*X,g\}=x\ in\ t'}$. For that step, one can first show for all frame stacks S and terms $\mathtt{t}$ that

$$\begin{aligned}&\langle[\mathtt{X} \mapsto \mathtt{N}][\mathtt{g} \mapsto \mathtt{G}]S, [\mathtt{X} \mapsto \mathtt{N}][\mathtt{g} \mapsto \mathtt{G}]\mathtt{t}\rangle\downarrow\\ &\text{if and only if } \langle[\mathtt{X} \mapsto \mathtt{Bool}][\mathtt{g} \mapsto \mathtt{G'}]S, [\mathtt{X} \mapsto \mathtt{Bool}][\mathtt{g} \mapsto \mathtt{G'}]\mathtt{t}\rangle\downarrow.\end{aligned}$$

This also is proved by induction on the definition of the termination relation. Once again we omit the details except to note that now the only difficult induction step is for the primitive reduction (R-APPABS) in the case that $\mathtt{t}$ is of the form $\mathtt{g}\ \mathtt{v}$ for some value $\mathtt{v}$. To prove that step one can use Lemma 7.7.5

below. This lemma lies at the heart of the reason why the contextual equivalence in (ii) is valid: if an argument supplied to $\mathtt{G}'$ is sufficiently polymorphic (which is guaranteed by the existential abstraction), then when specialised to $\mathtt{Bool}$ it cannot have the functionality ($\mathtt{true} \mapsto \mathtt{true}$, $\mathtt{false} \mapsto \mathtt{false}$) needed to distinguish $\mathtt{G}'$ from the divergent behaviour of $\mathtt{G}$. □

7.7.5 LEMMA: For any value $\mathtt{v}$ satisfying $\mathtt{X},\ \mathtt{g{:}P} \vdash \mathtt{v} : \mathtt{X}{\to}\mathtt{Bool}$, evaluation of $\mathtt{G}'([\mathtt{X} \mapsto \mathtt{Bool}][\mathtt{g} \mapsto \mathtt{G}']\mathtt{v})$ does not terminate. □

Proof: To prove this we can use the logical relation from the previous section. Consider the following value-relation in $\mathit{VRel}(\mathtt{Bool},\mathtt{Bool})$:

$$r \stackrel{\text{def}}{=} \{(\mathtt{true},\mathtt{true}),(\mathtt{false},\mathtt{false}),(\mathtt{true},\mathtt{false})\}.$$

Note that

$$(\mathtt{X}{\to}\mathtt{Bool})[r]^{v} \stackrel{\text{def}}{=} \mathtt{fun}(r^{v\,s\,t}, \mathit{Id}^{s\,t}_{\mathtt{Bool}})^{s\,t\,v} \stackrel{(7.20)}{=} \mathtt{fun}(r, \mathit{Id}^{s\,t}_{\mathtt{Bool}})^{s\,t\,v} \stackrel{(7.19)}{=} \mathtt{fun}(r, \mathit{Id}^{s\,t}_{\mathtt{Bool}}) \quad (7.35)$$

and hence

$$\begin{aligned}\mathtt{P}[r]^{v} &\stackrel{\text{def}}{=} \mathtt{fun}((\mathtt{X}{\to}\mathtt{Bool})[r], \mathit{Id}^{s\,t}_{\mathtt{Bool}})^{s\,t\,v} = \mathtt{fun}((\mathtt{X}{\to}\mathtt{Bool})[r]^{v}, \mathit{Id}^{s\,t}_{\mathtt{Bool}})^{s\,t\,v}\\ &\stackrel{(7.35)}{=} \mathtt{fun}(\mathtt{fun}(r, \mathit{Id}^{s\,t}_{\mathtt{Bool}}), \mathit{Id}^{s\,t}_{\mathtt{Bool}})^{s\,t\,v} \stackrel{(7.19)}{=} \mathtt{fun}(\mathtt{fun}(r, \mathit{Id}^{s\,t}_{\mathtt{Bool}}), \mathit{Id}^{s\,t}_{\mathtt{Bool}}). \quad (7.36)\end{aligned}$$

If $(\mathtt{v}_1, \mathtt{v}_1') \in \mathtt{fun}(r, \mathit{Id}^{s\,t}_{\mathtt{Bool}})$, since $(\mathtt{true},\mathtt{true}),(\mathtt{false},\mathtt{false}) \in r$ and $\mathit{Id}^{s\,t}_{\mathtt{Bool}}$ is contextual equivalence (Theorem 7.6.25) we get

$$\emptyset \vdash \mathtt{v}_1\ \mathtt{true} =_{\mathrm{ctx}} \mathtt{v}_1'\ \mathtt{true} : \mathtt{Bool}$$
$$\emptyset \vdash \mathtt{v}_1\ \mathtt{false} =_{\mathrm{ctx}} \mathtt{v}_1'\ \mathtt{false} : \mathtt{Bool}.$$

So using Corollary 7.5.8(iii) and the congruence properties of $=_{\mathrm{ctx}}$, we have

$$\begin{aligned}\mathtt{G}'\,\mathtt{v}_1 =_{\mathrm{ctx}}\ & (\mathtt{if}\ \mathtt{v}_1\ \mathtt{true}\ \mathtt{then}\\ &\quad \mathtt{if}\ \mathtt{v}_1\ \mathtt{false}\ \mathtt{then}\ \mathtt{diverge}\ \mathtt{else}\ \mathtt{true}\\ &\ \ \mathtt{else}\ \mathtt{diverge})\\ =_{\mathrm{ctx}}\ & (\mathtt{if}\ \mathtt{v}_1'\ \mathtt{true}\ \mathtt{then}\\ &\quad \mathtt{if}\ \mathtt{v}_1'\ \mathtt{false}\ \mathtt{then}\ \mathtt{diverge}\ \mathtt{else}\ \mathtt{true}\\ &\ \ \mathtt{else}\ \mathtt{diverge})\\ =_{\mathrm{ctx}}\ & \mathtt{G}'\,\mathtt{v}_1'\end{aligned}$$

Therefore $(\mathtt{G'}\ \mathtt{v}_1, \mathtt{G'}\ \mathtt{v}'_1) \in Id^{s\,t}_{\mathtt{Bool}}$ whenever $(\mathtt{v}_1, \mathtt{v}'_1) \in \mathtt{fun}(r, Id^{s\,t}_{\mathtt{Bool}})$; and so $(\mathtt{G'}, \mathtt{G'}) \in \mathtt{P}[r]^v$, by (7.36). Hence using Lemma 7.6.17 we have

$$([\mathtt{X} \mapsto \mathtt{Bool}][\mathtt{g} \mapsto \mathtt{G'}]\mathtt{v}, [\mathtt{X} \mapsto \mathtt{Bool}][\mathtt{g} \mapsto \mathtt{G'}]\mathtt{v}) \in (\mathtt{X}{\to}\mathtt{Bool})[r]^v$$
$$= \mathtt{fun}(r, Id^{s\,t}_{\mathtt{Bool}}) \quad \text{by (7.35).}$$

So since $(\mathtt{true}, \mathtt{false}) \in r$, we get

$$([\mathtt{X} \mapsto \mathtt{Bool}][\mathtt{g} \mapsto \mathtt{G'}]\mathtt{v}\ \mathtt{true}, [\mathtt{X} \mapsto \mathtt{Bool}][\mathtt{g} \mapsto \mathtt{G'}]\mathtt{v}\ \mathtt{false}) \in Id^{s\,t}_{\mathtt{Bool}}.$$

Thus $([\mathtt{X} \mapsto \mathtt{Bool}][\mathtt{g} \mapsto \mathtt{G'}]\mathtt{v})\,\mathtt{true}$ and $([\mathtt{X} \mapsto \mathtt{Bool}][\mathtt{g} \mapsto \mathtt{G'}]\mathtt{v})\,\mathtt{false}$ are contextually equivalent closed terms of type `Bool`. Therefore it cannot be the case that the first evaluates to `true` and the second to `false` (cf. Exercise 7.5.10); but in that case, by definition of $\mathtt{G'}$, it must be that evaluation of $\mathtt{G'}([\mathtt{X} \mapsto \mathtt{Bool}][\mathtt{g} \mapsto \mathtt{G'}]\mathtt{v})$ does not terminate. □

7.7.6 EXERCISE [★, ↛]: By considering the possible typing derivations from the rules in Figure 7-1, show that there is no value `v` satisfying $\emptyset \vdash \mathtt{v} : \forall\mathtt{X}.\mathtt{X}$. (Note that the syntactic restriction on values of universally quantified type mentioned in Remark 7.4.1 plays a crucial role here.) □

7.7.7 REMARK [THE ROLE OF NON-TERMINATION]: Example 7.7.4 shows that the logical relation presented here is incomplete for proving contextual equivalence of F_{ML} values of existential type. The example makes use of the fact that, because of the presence of recursive function values, evaluation of F_{ML} terms need not terminate. However, it seems that the source of the incompleteness has more to do with the existence of types with no values (such as $\forall\mathtt{X}.\mathtt{X}$) than with non-termination. Eijiro Sumii (private communication) has suggested the following, 'terminating' version of Example 7.7.4:

$$\begin{aligned}
\mathtt{P} &\stackrel{\text{def}}{=} (\mathtt{X}{\to}\mathtt{Bool}){\to}\mathtt{Bool}\\
\mathtt{Q} &\stackrel{\text{def}}{=} \{\exists\mathtt{X},\mathtt{P}\}\\
\mathtt{N} &\stackrel{\text{def}}{=} \forall\mathtt{X}.\mathtt{X}\\
\mathtt{H} &\stackrel{\text{def}}{=} \lambda\mathtt{f}{:}\mathtt{N}{\to}\mathtt{Bool}.\ \mathtt{false}\\
\mathtt{H'} &\stackrel{\text{def}}{=} \lambda\mathtt{f}{:}\mathtt{Bool}{\to}\mathtt{Bool}.\\
&\qquad (\mathtt{if}\ \mathtt{f}\ \mathtt{true}\ \mathtt{then}\\
&\qquad\quad \mathtt{if}\ \mathtt{f}\ \mathtt{false}\ \mathtt{then}\ \mathtt{false}\ \mathtt{else}\ \mathtt{true}\\
&\qquad \mathtt{else}\ \mathtt{false}) : \mathtt{Bool}.
\end{aligned}$$

Consider a version of F_{ML} with only non-recursive function abstractions (i.e. with $\lambda\mathtt{x}{:}\mathtt{T}.\mathtt{t}$ rather than $\mathtt{fun}\ \mathtt{f}(\mathtt{x}{:}\mathtt{T}) = \mathtt{t}{:}\mathtt{T'}$). Evaluation is terminating in this version. So to be non-trivial, contextual equivalence should be formulated in terms of observing convergence to the same ground value in all contexts of ground type. Making corresponding changes to the definition of the operations $(-)^s$ and $(-)^t$ on term- and stack-relations, one could develop a logical relation for this terminating version of F_{ML}. It seems that properties (i) and (ii) in Example 7.7.4 are also true of $\mathtt{H}$ and $\mathtt{H}'$ in this version (the first by the same argument we gave, but the second by a different argument that nevertheless hinges on the observation at the end of the proof of Example 7.7.4). We leave investigating this as an extended exercise for the reader. □

The proof of Lemma 7.7.5 exploits 'relational parametricity' properties of polymorphic types in F_{ML}. In fact Theorem 7.6.25 tells us far more about the properties of type abstraction values than just the extensionality property of Theorem 7.7.1(4).

7.7.8 THEOREM [RELATIONAL PARAMETRICITY FOR $\forall$-TYPES]: Given $\mathtt{X} \vdash \mathtt{v} : \mathtt{T}$ and $\mathtt{X} \vdash \mathtt{v}' : \mathtt{T}$, then $\emptyset \vdash \lambda\mathtt{X.v} =_{\mathrm{ctx}} \lambda\mathtt{X.v}' : \forall\mathtt{X.T}$ if and only if for all closed types $\mathtt{T}_1, \mathtt{T}_1' \in \mathit{Typ}$ and all term-relations $r \in \mathit{TRel}(\mathtt{T}_1, \mathtt{T}_1')$ it is the case that $([\mathtt{X} \mapsto \mathtt{T}_1]\mathtt{v}, [\mathtt{X} \mapsto \mathtt{T}_1']\mathtt{v}') \in \mathtt{T}[r]$. □

Proof: By Theorem 7.6.25, we have that $\emptyset \vdash \lambda\mathtt{X.v} =_{\mathrm{ctx}} \lambda\mathtt{X.v}' : \forall\mathtt{X.T}$ iff $\emptyset \vdash \lambda\mathtt{X.v} \;\Delta\; \lambda\mathtt{X.v}' : \forall\mathtt{X.T}$, i.e. iff $(\lambda\mathtt{X.v}, \lambda\mathtt{X.v}') \in (\forall\mathtt{X.T})[] = (\lambda r.\mathtt{T}[r])^{st}$. Since $\lambda\mathtt{X.v}$ and $\lambda\mathtt{X.v}'$ are values, the latter is the case iff $(\lambda\mathtt{X.v}, \lambda\mathtt{X.v}') \in (\lambda r.\mathtt{T}[r])^{stv}$, and by Lemma 7.6.12 and Exercise 7.6.14 $(\lambda r.\mathtt{T}[r])^{stv} = \lambda r.\mathtt{T}[r]$. Hence $\emptyset \vdash \lambda\mathtt{X.v} =_{\mathrm{ctx}} \lambda\mathtt{X.v}' : \forall\mathtt{X.T}$ iff $(\lambda\mathtt{X.v}, \lambda\mathtt{X.v}') \in \lambda r.\mathtt{T}[r]$. By definition (Figure 7-5), this is the case iff for all for all closed types $\mathtt{T}_1, \mathtt{T}_1' \in \mathit{Typ}$ and all term-relations $r \in \mathit{TRel}(\mathtt{T}_1, \mathtt{T}_1')$, $(\,(\lambda\mathtt{X.v})\,\mathtt{T}_1,\ (\lambda\mathtt{X.v}')\,\mathtt{T}_1') \in \mathtt{T}[r]$; and the latter holds iff $([\mathtt{X} \mapsto \mathtt{T}_1]\mathtt{v}, [\mathtt{X} \mapsto \mathtt{T}_1']\mathtt{v}') \in \mathtt{T}[r]$, because $(\lambda\mathtt{X.v})\,\mathtt{T}_1 =_{\mathrm{ciu}} [\mathtt{X} \mapsto \mathtt{T}_1]\mathtt{v}$ and $(\lambda\mathtt{X.v}')\,\mathtt{T}_1' =_{\mathrm{ciu}} [\mathtt{X} \mapsto \mathtt{T}_1']\mathtt{v}'$ (so that we can use Lemmas 7.6.8 and 7.6.12). □

The force of Theorem 7.7.1(4) is to give a method for establishing that two type abstraction values are contextually equivalent. By contrast, the force of Theorem 7.7.8 is to give us useful properties of families of values parameterised by type variables. Given such a value, $\mathtt{X} \vdash \mathtt{v} : \mathtt{T}$, since $=_{\mathrm{ctx}}$ is reflexive, we have $\emptyset \vdash \lambda\mathtt{X.v} =_{\mathrm{ctx}} \lambda\mathtt{X.v} : \forall\mathtt{X.T}$; hence the theorem has the following corollary.

7.7.9 COROLLARY: Given a value $\mathtt{X} \vdash \mathtt{v} : \mathtt{T}$, for all $\mathtt{T}_1, \mathtt{T}_1' \in \mathit{Typ}$ and all $r \in \mathit{TRel}(\mathtt{T}_1, \mathtt{T}_1')$, it is the case that $([\mathtt{X} \mapsto \mathtt{T}_1]\mathtt{v}, [\mathtt{X} \mapsto \mathtt{T}_1']\mathtt{v}) \in \mathtt{T}[r]$. □

Such 'relational parametricity' properties can often be exploited for proving contextual equivalences: we already saw an example in the proof of Lemma 7.7.5 and other examples can be found in Pitts (2000),Bierman et al. (2000), and Johann (2002). However, the strict nature of function application and type abstraction in F_{ML} means that it does not satisfy all the parametricity properties one might expect. For example, in Pitts (2000), § 7 it is shown that

$$\{\exists X, T\} \cong \forall Y.(\forall X.T \to Y) \to Y$$

holds in the polymorphic version of PCF (Plotkin, 1977) studied in that paper (where $\cong$ is 'bijection up to contextual equivalence'—see Principle 7.3.4). However this bijection does not hold in general for F_{ML} (Exercise 7.7.10).

7.7.10 EXERCISE [★★★]: Consider the type $N \stackrel{\text{def}}{=} \forall X.X$ from Example 7.7.4 that you showed has no closed values in Exercise 7.7.6. Show that there cannot exist values

$$i \in \mathit{Val}(\{\exists X, N\} \to \forall Y.(\forall X.N \to Y) \to Y)$$
$$j \in \mathit{Val}((\forall Y.(\forall X.N \to Y) \to Y) \to \{\exists X, N\})$$

that are mutually inverse, in the sense that

$$p : \{\exists X, N\} \vdash j\,(i\,p) =_{\mathrm{ctx}} p : \{\exists X, N\}$$
$$y : \forall Y.(\forall X.N \to Y) \to Y \vdash i\,(j\,y) =_{\mathrm{ctx}} y : \forall Y.(\forall X.N \to Y) \to Y.$$

□

7.7.11 EXERCISE [★★★, ↛]: Verify the claim made in Note 7.3.7 that Principle 7.3.4 is a special case of Principle 7.3.6. To do so, you will first have to give a definition of the action of F_{ML} types on bijections mentioned in Principle 7.3.4. □

7.8 Notes

This chapter is a revised and expanded version of Pitts (1998) and also draws on material from Pitts (2000).

In discussing typed operational reasoning we have focused on reasoning about *contextual equivalence* of program phrases. Being by construction a congruence, contextual equivalence permits us to use the usual forms of equational reasoning (replacing equals by equals) when deriving equivalences between phrases. However, its definition does not lend itself to establishing the basic laws that are needed to get such reasoning going. We studied two characterisations of contextual equivalence in order to get round this problem: *ciu-equivalence* (Definition 7.5.5) and a certain kind of operationally-based *logical relation* (Definition 7.6.10).

The informal notion of contextual equivalence (Definition 7.3.2) has been studied for a wide variety of programming languages. If the language's operational semantics involves non-determinism—usually because the language supports some form of concurrent or interactive computation—then contextual equivalence tends to identify too many programs and various co-inductive notions of *bisimilarity* are used instead (see the text book by Sangiorgi and Walker, 2001, for example). But even if we remain within the realm of languages with deterministic operational semantics, one may ask to what extent the results of this chapter are stable with respect to adding further features such as recursive datatypes, mutable state, and object-oriented features à la Objective Caml.

Ciu-equivalence has the advantage of being quite robust in this respect—it can provide a characterisation of contextual equivalence in the presence of such features (Honsell, Mason, Smith, and Talcott, 1995; Talcott, 1998). However, its usefulness is mainly limited to establishing basic laws such as the conversions in Corollary 7.5.8; it cannot be used directly to establish extensionality properties such as those in Theorem 7.7.1 without resorting to tedious 'termination inductions' of the kind we sketched in the proof of Example 7.7.4. Ciu-equivalence is quite closely related to some notions of 'applicative bisimilarity' that have been applied to functional and object-based languages (Gordon, 1995, 1998), in that their congruence properties can both be established using a clever technique due to Howe (1996). The advantage of applicative bisimilarity is that it has extensionality built into its definition; so when it does coincide with contextual equivalence, this provides a method of establishing some extensionality properties for $=_{\mathrm{ctx}}$ (such as (1)–(4) in Theorem 7.7.1, but not, as far as I know, property (5) for package values).

The kind of operationally-based logical relation we developed in this chapter provides a very powerful analysis of contextual equivalence. We used it to prove not only conversions and simple extensionality principles for F_{ML}, but also quite subtle properties of $=_{\mathrm{ctx}}$ such as Theorems 7.7.1(5) and 7.7.8. Similar logical relations can be used to prove some properties of ML-style references and of linear types: see Pitts and Stark (1998), (2000), and (2002). Unfortunately, the characteristic feature of logical relations—that functions are related iff they map related arguments to related results—makes it difficult to define them in the presence of 'recursive features'. I mean by the latter programming language features which in a denotational semantics lead one to have to solve domain equations in which the defined domain occurs both positively (to the left of an even number of function space constructions) and negatively (to the left of an odd number of function space constructions). Recursive datatypes involving function types can lead to such domain equations; as does the use of references to functions in ML. Suitable logical relations can be defined in the denotational semantics of languages with such features using techniques such as those in Pitts (1996), but they tell us properties of denotational equality, which is often a poor (if safe) approximation to contextual equivalence. For this reason people have tried to develop syntactical analogues of these denotational logical relations: see Birkedal and Harper (1999). The unwinding theorem (Theorem 7.4.4) provides the basis for such an approach. However, it seems like a fresh idea is needed to make further progress. Therefore I set a last exercise, whose solution is not included.

7.8.1 EXERCISE [★★★★..., ↛]: Extend F_{ML} with *iso-recursive types*, `μX.T`, as in Figure 20-1 of TAPL Chapter 20. By finding an operationally-based logical relation as in §7.6 or otherwise, try to prove the kind of properties of contextual equivalence for this extended language that we developed for F_{ML} in this chapter. (For the special case of iso-recursive types `μX.T` for which `T` contains no negative occurrences of `X`, albeit for a non-strict functional language, see Johann (2002). The generalised ideal model of recursive polymorphic in Vouillon and Melliès (2004) uses the same kind of Galois connection as we used in §7.6 and may well shed light on this exercise.) □

Unfortunately, this is an invalid definition, because logical equivalence is defined by induction on types, and there is no guarantee that the type $[\mathtt{X} \mapsto \mathtt{T}']\mathtt{T}$ is smaller than $\forall\mathtt{X}.\mathtt{T}$.

The problem is tied to the issue of *impredicativity* (TAPL §23.10). In both $\forall\mathtt{X}.\mathtt{T}$, the domain of the quantified type variable $\mathtt{X}$ includes the very type being defined. This prevents the obvious definition from being well-founded.

For the simple definition attempt given above, the impredicativity problem is fatal. Sometimes we can save the simple definition by changing the language to be predicative. However, for an impredicative language, a more sophisticated definition is required.

The solution to the problem is Girard's method of candidates (Girard, Lafont, and Taylor, 1989). An explanation of Girard's method is beyond the scope of this discussion, but informally the method works as follows: Instead of quantifying over types, the definition of logical equivalence quantifies over possible interpretations of types called *candidates.* Importantly, candidates come equipped with their own notion of logical equivalence that can be defined independently (*i.e.,* without reference to the general definition of logical equivalence). Thus, the definition of logical equivalence may refer to arbitrary candidates and remain well-founded.

7.4.2 HINT: First prove

$$\langle S_1, t_1\rangle \longrightarrow \langle S_2, t_2\rangle \Rightarrow (\forall S)(\langle S@S_2, t_2\rangle\downarrow \Rightarrow \langle S@S_1, t_1\rangle\downarrow)$$

by considering the different cases for $\longrightarrow$. Deduce the 'if' part of (7.7) from this. For the 'only if' part, show that

$$\{(S,t) \mid (\exists S_1, S_2, v)\ S = S_1@S_2\ \&\ \langle S_2, t\rangle \longrightarrow^* \langle Id, v\rangle\ \&\ \langle S_1, v\rangle\downarrow\}$$

is closed under the axiom and rules in Figure 7-2 inductively defining the termination relation.

7.5.4 SOLUTION: For property (iii), assuming R is compatible, argue by induction on the derivation of $\Gamma \vdash \mathtt{t} : \mathtt{T}$ that this typing judgement implies that $\Gamma \vdash \mathtt{t}\ R\ \mathtt{t} : \mathtt{T}$ holds. For property (v), if $R = \bigcup_{i\in I} R_i$ with $I \neq \emptyset$ and each R_i compatible, first note that by (iii), R is reflexive since it contains at least one relation R_i. For each of the compatibility properties in Figure 7-4 with a *single* hypothesis, it is clear that R has this property because each of the R_i does. For compatibility properties with multiple hypotheses, we can break them down into a chain of single-hypothesis compatibilities and appeal to the transitivity of R (which we are assuming). For example consider the compatibility property for function application. It suffices to show that R satisfies

$$\frac{\Gamma \vdash \mathtt{v}_1 \; R \; \mathtt{v}_1' : \mathtt{T}_1 \to \mathtt{T}_2 \qquad \Gamma \vdash \mathtt{v}_2 : \mathtt{T}_1}{\Gamma \vdash \mathtt{v}_1 \; \mathtt{v}_2 \; R \; \mathtt{v}_1' \; \mathtt{v}_2 : \mathtt{T}_2} \tag{A.1}$$

and

$$\frac{\Gamma \vdash \mathtt{v}_1 : \mathtt{T}_1 \to \mathtt{T}_2 \qquad \Gamma \vdash \mathtt{v}_2 \; R \; \mathtt{v}_2' : \mathtt{T}_1}{\Gamma \vdash \mathtt{v}_1 \; \mathtt{v}_2 \; R \; \mathtt{v}_1 \; \mathtt{v}_2' : \mathtt{T}_2}. \tag{A.2}$$

For then if $\Gamma \vdash \mathtt{v}_1 \; R \; \mathtt{v}_1' : \mathtt{T}_1 \to \mathtt{T}_2$ and $\Gamma \vdash \mathtt{v}_2 \; R \; \mathtt{v}_2' : \mathtt{T}_1$, we get

$$\begin{array}{ll} \Gamma \vdash \mathtt{v}_1 \; \mathtt{v}_2 \; R \; \mathtt{v}_1' \; \mathtt{v}_2 : \mathtt{T}_2 & \text{by (A.1), since } \Gamma \vdash \mathtt{v}_2 : \mathtt{T}_1 \\ \Gamma \vdash \mathtt{v}_1' \; \mathtt{v}_2 \; R \; \mathtt{v}_1' \; \mathtt{v}_2' : \mathtt{T}_2 & \text{by (A.2), since } \Gamma \vdash \mathtt{v}_1' : \mathtt{T}_1 \to \mathtt{T}_2. \end{array}$$

and hence $\Gamma \vdash \mathtt{v}_1 \; \mathtt{v}_2 \; R \; \mathtt{v}_1' \; \mathtt{v}_2' : \mathtt{T}_2$ by transitivity. Each of the single-hypothesis properties (A.1) and (A.2) holds of R because they hold for each R_i: each is a special case of the compatibility property for function application because each R_i, being compatible, is also reflexive by (iii).

7.5.10 SOLUTION: Consider the frame stacks

$$S \stackrel{\text{def}}{=} \mathit{Id} \circ (\mathtt{x.\,(fun\,f\,(x'\!:\!Bool)\;=\;if\;x'\;then\;true\;else\;f\;x')\;x})$$

$$S_{\mathtt{T}} \stackrel{\text{def}}{=} \mathit{Id} \circ (\mathtt{x.\,(fun\,f\,(x'\!:\!T)\;=\;true)\;x})$$

Note that $\emptyset \vdash S : \mathtt{Bool} \multimap \mathtt{Bool}$ and $\emptyset \vdash S_{\mathtt{T}} : \mathtt{T} \multimap \mathtt{Bool}$. It is not hard to see for all $\emptyset \vdash \mathtt{b} : \mathtt{Bool}$ that

$$S[\mathtt{b}]\downarrow \text{ iff } \langle \mathit{Id}, \mathtt{b} \rangle \longrightarrow^{*} \langle \mathit{Id}, \mathtt{true} \rangle \tag{A.3}$$

and for all $\emptyset \vdash \mathtt{t} : \mathtt{T}$ that

$$\mathtt{t}\downarrow \text{ iff } \langle \mathit{Id}, S_{\mathtt{T}}[\mathtt{t}] \rangle \longrightarrow^{*} \langle \mathit{Id}, \mathtt{true} \rangle \tag{A.4}$$

From (A.3) and the fact that $=_{\text{ctx}}$ is a congruence (so that $\emptyset \vdash \mathtt{b} =_{\text{ctx}} \mathtt{b}' : \mathtt{Bool}$ implies $\emptyset \vdash S[\mathtt{b}] =_{\text{ctx}} S[\mathtt{b}'] : \mathtt{Bool}$) it follows that $=_{\text{ctx}}$ is $\mathtt{true}$-adequate; hence it is contained in $=_{\text{ctx}}^{\mathtt{true}}$. Similarly, (A.4) and the fact that $=_{\text{ctx}}^{\mathtt{true}}$ is a congruence implies that it is adequate and hence contained in $=_{\text{ctx}}$.

7.6.7 SOLUTION: Since $(-)^{st}$ is inflationary we have $r \subseteq r^{st}$; and since r only relates values, this implies $r \subseteq r^{stv}$. Then since $(-)^{st}$ is monotone, we have $r^{st} \subseteq r^{stvst}$. Conversely, since $(r')^{v} \subseteq r'$ for any r', we have $r^{stv} \subseteq r^{st}$; and then since $(-)^{st}$ is monotone and idempotent, $r^{stvst} \subseteq r^{stst} = r^{st}$.

7.6.14 HINT: The proof of (7.26) is just like the proof of (7.21), using the following property of the termination relation:

$$(\langle S, \mathtt{v.l}\rangle\downarrow \Leftrightarrow \langle S', \mathtt{v'.l}\rangle\downarrow) \text{ iff } (\langle S\circ(\mathtt{x.x.l}), \mathtt{v}\rangle\downarrow \Leftrightarrow \langle S'\circ(\mathtt{x.x.l}), \mathtt{v'}\rangle\downarrow).$$

Similarly, the proof of (7.27) follows from:

$$(\langle S, \mathtt{v\ T}\rangle\downarrow \Leftrightarrow \langle S', \mathtt{v'\ T'}\rangle\downarrow) \text{ iff } (\langle S\circ(\mathtt{x.x\ T}), \mathtt{v}\rangle\downarrow \Leftrightarrow \langle S'\circ(\mathtt{x.x\ T'}), \mathtt{v'}\rangle\downarrow).$$

7.6.18 SOLUTION: It suffices to show

$$(\forall n = 0, 1, \ldots)\ (\mathtt{F}_n, \mathtt{F}'_n) \in \mathtt{fun}(r_1, r_2) \tag{A.5}$$

where $\mathtt{F}_n$ and $\mathtt{F}'_n$ are the unwindings associated with $\mathtt{F}$ and $\mathtt{F}'$ respectively, as in Theorem 7.4.4. For if (A.5) holds, then using the fact that $(-)^{st}$ is inflationary

$$(\mathtt{F}_n, \mathtt{F}'_n) \in \mathtt{fun}(r_1, r_2) \subseteq \mathtt{fun}(r_1, r_2)^{st}$$

for each n; so by the **Admissibility** property in Lemma 7.6.8 we have $(\mathtt{F}, \mathtt{F}') \in \mathtt{fun}(r_1, r_2)^{st}$. Thus $(\mathtt{F}, \mathtt{F}') \in \mathtt{fun}(r_1, r_2)^{stv} = \mathtt{fun}(r_1, r_2)$ by Lemma 7.6.13, since $(r_2)^{st} = r_2$. (A.5) is proved by induction on n:

Base case $n = 0$: By definition of $\mathtt{F}_0$, $\langle S, \mathtt{F}_0\ \mathtt{v}_1\rangle\downarrow$ does not hold for any $S \in \mathit{Stack}(\mathtt{T}_2)$ and $\mathtt{v}_1 \in \mathit{Val}(\mathtt{T}_1)$; similarly for $\mathtt{F}'_0$. Hence for all $(\mathtt{v}_1, \mathtt{v}'_1) \in (r_1)^v$, $(\mathtt{F}_0\ \mathtt{v}_1, \mathtt{F}'_0\ \mathtt{v}'_1) \in s^t$ for any $s \in \mathit{SRel}(\mathtt{T}_2, \mathtt{T}'_2)$ and hence in particular for $s = (r_2)^s$. So $(\mathtt{F}_0\ \mathtt{v}_1, \mathtt{F}'_0\ \mathtt{v}'_1) \in (r_2)^{st} = r_2$ for all $(\mathtt{v}_1, \mathtt{v}'_1) \in (r_1)^v$. Therefore $(\mathtt{F}_0, \mathtt{F}'_0) \in \mathtt{fun}(r_1, r_2)$.

Induction step: Suppose $(\mathtt{F}_n, \mathtt{F}'_n) \in \mathtt{fun}(r_1, r_2)$. Then for any $(\mathtt{v}_1, \mathtt{v}'_1) \in (r_1)^v$, from (7.29) we have

$$([\mathtt{f} \mapsto \mathtt{F}_n][\mathtt{x} \mapsto \mathtt{v}_1]\mathtt{t}, [\mathtt{f} \mapsto \mathtt{F}'_n][\mathtt{x} \mapsto \mathtt{v}'_1]\mathtt{t}') \in r_2.$$

By definition of $\mathtt{F}_{n+1}$ and Corollary 7.5.8 we have $\emptyset \vdash \mathtt{F}_{n+1}\mathtt{v}_1 =_{\text{ctx}} [\mathtt{f} \mapsto \mathtt{F}_n][\mathtt{x} \mapsto \mathtt{v}_1]\mathtt{t}$; and similarly, $\emptyset \vdash \mathtt{F}'_{n+1}\mathtt{v}'_1 =_{\text{ctx}} [\mathtt{f} \mapsto \mathtt{F}'_n][\mathtt{x} \mapsto \mathtt{v}'_1]\mathtt{t}'$. So since r_2 is closed, we can apply the **Equivalence-respecting** property in Lemma 7.6.8 to conclude that $(\mathtt{F}_{n+1}\mathtt{v}_1, \mathtt{F}'_{n+1}\mathtt{v}'_1) \in r_2$. Since this holds for any $(\mathtt{v}_1, \mathtt{v}'_1) \in (r_1)^v$, we have $(\mathtt{F}_{n+1}, \mathtt{F}'_{n+1}) \in \mathtt{fun}(r_1, r_2)$.

7.6.19 SOLUTION: To show $(\mathtt{v}, \mathtt{v}') \in \{\mathtt{l}_i{=}r_i\ ^{i\in 1..n}\}$ we have to show $(\mathtt{v.l}_i, \mathtt{v'.l}_i) \in r_i$ for each $i \in 1..n$. Since each r_i is closed, this is equivalent to showing $(\mathtt{v.l}_i, \mathtt{v'.l}_i) \in (r_i)^{st}$, i.e. that $\langle S, \mathtt{v.l}_i\rangle\downarrow \Leftrightarrow \langle S', \mathtt{v'.l}_i\rangle\downarrow$ holds for all (S, S') in $(r_i)^s$. But by definition of $\mathtt{v}$, $\langle S, \mathtt{v.l}_i\rangle\downarrow \Leftrightarrow \langle S, \mathtt{v}_i\rangle\downarrow$; and similarly for $\mathtt{v}'$. So it suffices to show $\langle S, \mathtt{v}_i\rangle\downarrow \Leftrightarrow \langle S', \mathtt{v}'_i\rangle$; and this holds because by assumption $(\mathtt{v}_i, \mathtt{v}'_i) \in r_i$ and $(S, S') \in (r_i)^s$.

7.6.20 SOLUTION: To show $(\lambda\mathtt{X.v}, \lambda\mathtt{X.v}') \in \lambda\mathtt{r}.R(\mathtt{r})$ we have to show for each $\mathtt{T}_1, \mathtt{T}_1' \in \mathit{Typ}$ and $\mathtt{r} \in \mathit{TRel}(\mathtt{T}_1, \mathtt{T}_1')$ that $(\,(\lambda\mathtt{X.v})\,\mathtt{T},\ (\lambda\mathtt{X.v}')\,\mathtt{T}') \in R(\mathtt{r})$. Since each $R(\mathtt{r})$ is closed, this is equivalent to showing $(\,(\lambda\mathtt{X.v})\,\mathtt{T},\ (\lambda\mathtt{X.v}')\,\mathtt{T}') \in R(\mathtt{r})^{st}$, i.e. that $\langle S,\ (\lambda\mathtt{X.v})\,\mathtt{T}\rangle\downarrow\ \Leftrightarrow\ \langle S',\ (\lambda\mathtt{X.v}')\,\mathtt{T}'\rangle\downarrow$ holds for all $(S, S') \in R(\mathtt{r})^s$. But $\langle S,\ (\lambda\mathtt{X.v})\,\mathtt{T}\rangle\downarrow \Leftrightarrow \langle S, [\mathtt{X} \mapsto \mathtt{T}_1]\mathtt{v}\rangle\downarrow$; and similarly for $\mathtt{v}'$. So it suffices to show $\langle S, [\mathtt{X} \mapsto \mathtt{T}_1]\mathtt{v}\rangle\downarrow \Leftrightarrow \langle S, [\mathtt{X} \mapsto \mathtt{T}_1']\mathtt{v}'\rangle\downarrow$; and this holds because by assumption $([\mathtt{X} \mapsto \mathtt{T}_1]\mathtt{v}, [\mathtt{X} \mapsto \mathtt{T}_1]\mathtt{v}) \in R(\mathtt{r})$ and $(S, S') \in R(\mathtt{r})^s$.

7.6.21 HINT: To show $(\mathtt{if\ v\ then\ t_1\ else\ t_2}, \mathtt{if\ v'\ then\ t_1'\ else\ t_2'}) \in \mathtt{r} = (\mathtt{r})^{st}$ it suffices to show for all $(S, S') \in (\mathtt{r})^s$ that

$$\langle S, \mathtt{if\ v\ then\ t_1\ else\ t_2}\rangle\downarrow \Leftrightarrow \langle S', \mathtt{if\ v'\ then\ t_1'\ else\ t_2'}\rangle\downarrow$$

or equivalently that

$$\langle S \circ (\mathtt{x.if\ x\ then\ t_1\ else\ t_2}), \mathtt{v}\rangle\downarrow \Leftrightarrow \\ \langle S' \circ (\mathtt{x.if\ x\ then\ t_1'\ else\ t_2'}), \mathtt{v}'\rangle\downarrow.$$

Do this by proving that

$$(S\circ(\mathtt{x.if\ x\ then\ t_1\ else\ t_2}), S'\circ(\mathtt{x.if\ x\ then\ t_1'\ else\ t_2'}) \in (\mathit{Id}_{\mathtt{Bool}})^s.$$

7.6.22 SOLUTION: For any $(S, S') \in (\mathtt{r}_2)^s$ it follows from the assumptions on $\mathtt{t}, \mathtt{t}'$ and the definition of $\{\exists\mathtt{r}_1, R(\mathtt{r}_1)\}$ (Figure 7-5) that

$$(S \circ (\mathtt{y.let\ \{*X,x\}=y\ in\ t}), S' \circ (\mathtt{y.let\ \{*X,x\}=y\ in\ t'}))$$

is in $\{\exists\mathtt{r}_1, R(\mathtt{r}_1)\}^s$. Hence if $(\mathtt{v}, \mathtt{v}') \in \{\exists\mathtt{r}_1, R(\mathtt{r}_1)\}^{stv} \subseteq (\{\exists\mathtt{r}_1, R(\mathtt{r}_1)\}^s)^t$, then

$$\langle S\circ(\mathtt{y.let\ \{*X,x\}=y\ in\ t}), \mathtt{v}\rangle\downarrow\ \Leftrightarrow\ \langle S'\circ(\mathtt{y.let\ \{*X,x\}=y\ in\ t'}), \mathtt{v}'\rangle\downarrow$$

and so $\langle S, \mathtt{let\ \{*X,x\}=v\ in\ t}\rangle\downarrow \Leftrightarrow \langle S, \mathtt{let\ \{*X,x\}=v'\ in\ t'}\rangle\downarrow$. Since this holds for all $(S, S') \in (\mathtt{r}_2)^s$, we deduce that

$$(\mathtt{let\ \{*X,x\}=v\ in\ t}, \mathtt{let\ \{*X,x\}=v\ in\ t}) \in (\mathtt{r}_2)^{st} = \mathtt{r}_2.$$

7.6.23 SOLUTION: For any $(S, S') \in (\mathtt{r}_2)^s$ it follows from the assumptions on $\mathtt{t}, \mathtt{t}'$ that $(S \circ (\mathtt{x.t_2}), S' \circ (\mathtt{x.t_2'})) \in (\mathtt{r}_1)^{vs}$. Since $((\mathtt{r}_1)^{vs})^t = \mathtt{r}_1$, if $(\mathtt{t}_1, \mathtt{t}_1') \in \mathtt{r}_1$ then we get $\langle S \circ (\mathtt{x.t_2}), \mathtt{t}_1\rangle\downarrow \Leftrightarrow \langle S' \circ (\mathtt{x.t_2'}), \mathtt{t}_1'\rangle\downarrow$, and hence that

$$\langle S, \mathtt{let\ x=t_1\ in\ t_2}\rangle\downarrow\ \Leftrightarrow\ \langle S', \mathtt{let\ x=t_1'\ in\ t_2'}\rangle\downarrow.$$

Since this holds for all $(S, S') \in (\mathtt{r}_2)^s$, we deduce that

$$(\mathtt{let\ x=t_1\ in\ t_2}, \mathtt{let\ x=t_1'\ in\ t_2'}) \in (\mathtt{r}_2)^{st} = \mathtt{r}_2.$$

7.7.10 SOLUTION: Since `N` has no closed values, neither does `{∃X,N}`. On the other hand

```
val v = λY.fun f(x:∀X.N→Y) = (f x):Y
```

is a closed value of type `∀Y.(∀X.N→Y)→Y`. If `i` and `j` were to exist with the stated properties we could use them to construct from `v` a closed value of type `{∃X,N}`, which is impossible. (For `i(j v)` and `v` are ciu-equivalent (Theorem 7.5.7); so since $\mathtt{v}\downarrow$, we also have $\mathtt{i(j\ v)}\downarrow$. Hence by Exercise 7.4.2, $\langle Id, \mathtt{j\ v}\rangle \longrightarrow^{*} \langle Id, \mathtt{v}'\rangle$ for some $\mathtt{v}'$, which is a closed value of type `{∃X,N}`, by Exercise 7.4.3.)

I.1 SOLUTION: We can introduce a type family for rectangular matrices thus:

```
Matrix      :: Nat → Nat → *
idmatrix    : Πn:Nat. Matrix n n
multmatrix  : Πl:Nat. Πm:Nat. Πn:Nat.
                      Matrix l m → Matrix m n → Matrix l n
```

Suppose we have a dependent type for ranges of integers: `{n...m}` denotes the type of integers between `n` and `m` inclusive, either `n` or `m` may be omitted. A possible typing for dates is given by:

```
Year  =  {2003..} :: *
Month =  {1..12}  :: *
Day   :: Month → *
```

where

$$
\begin{array}{lcll}
\mathtt{Day(n)} & = & \mathtt{\{1..31\}} & \text{if } n \in \{1,3,5,7,8,10,12\} \\
\mathtt{Day(n)} & = & \mathtt{\{1..30\}} & \text{if } n \in \{4,6,9,11\} \\
\mathtt{Day(2)} & = & \mathtt{\{1..29\}} &
\end{array}
$$

A date is then given by an element of the Σ-type (see page 56):

```
Date :: Σy:Year. Σm:Month. Day(m)
```

Of course, we could gain more accuracy by making the type of days also depend on the year.

I.2 SOLUTION: A type representing the constructive axiom of choice for a predicate `P` is:

```
(Πa:A.Σb:B. P(a,b)) → (Σf:A→B. Πx:A. P(x, f x))
```

It can be shown in Martin-Löf's type theory that this type is inhabited (Martin-Löf, 1984).

I.3 SOLUTION: `Σa:A.Σb:B.Id(f a, g b)`

I.4 SOLUTION: Here are some terms representing β-reduction and its closure on lambda terms:

```
Eval     :: ΠA:Ty. Tm A → Tm A → *
evalAppAbs : ΠA:Ty. ΠB:Ty.
              Πt1:(Tm A → Tm B).
              Πt2:(Tm A) → Eval (app (lam t1) t2) (t1 t2)
evalLam  :  ΠA:Ty. ΠB:Ty.
             Πft1,ft1':(Tm A → Tm B).
              (Πx: Tm A. Eval (ft1 x) (ft1' x))
                 → Eval (lam ft1) (lam ft2)
evalApp1 :  ΠA:Ty. ΠB:Ty.
             Πt1,t1':(Tm (arrow A B)).
              Πt2: Tm B. Eval t1 t1' → Eval (app t1 t2) (app t1' t2)
evalApp2 :  ΠA:Ty. ΠB:Ty.
             Πt1: (Tm (arrow A B)).
              Πt2,t2': Tm B. Eval t2 t2' → Eval (app t1 t2) (app t1 t2')
```

2.5.4 SOLUTION: We give the solution in the syntax of the implementation.

```
eqSucc = λx:Prf(nat).λy:Prf(nat).λh:Prf(eq nat x y).
       h(λz:Prf(nat).eq nat (succ x) (succ z)) (eqRefl nat (succ x))
      : Πx:Prf(nat).Πy:Prf(nat).Prf(eq nat x y) →
                                      Prf(eq nat (succ x) (succ y));

addAssoc = λx:Prf(nat).λy:Prf(nat).λz:Prf(nat).
          eq nat (add x (add y z)) (add (add x y) z);

proofOfAddAssoc = λx:Prf(nat).λy:Prf(nat).λz:Prf(nat).
      natInd (λx1:Prf(nat).addAssoc x1 y z)
          (eqRefl nat (add y z))
          (λx1:Prf(nat).λp:Prf(addAssoc x1 y z).
               eqSucc (add x1 (add y z)) (add (add x1 y) z) p)
          x
  : Πx:Prf(nat).Πy:Prf(nat).Πz:Prf(nat).Prf(addAssoc x y z);
```

2.6.1 SOLUTION: Let $i : \mathit{Syntax}(\lambda\mathrm{LF}) \to \mathit{Syntax}(\lambda\mathrm{P})$ be the obvious mapping between the syntaxes, which collapses each λLF λ-construct to the single λP λ-operator, etc. (Except that type and term variables have disoint images). Then we would hope to show:

1. $\Gamma \vdash_{\lambda\mathrm{LF}} \mathtt{t} : \mathtt{T} \iff i(\Gamma) \vdash_{\lambda\mathrm{P}} i(\mathtt{t}) : i(\mathtt{T})$
2. $\Gamma \vdash_{\lambda\mathrm{LF}} \mathtt{T} :: \mathtt{K} \iff i(\Gamma) \vdash_{\lambda\mathrm{P}} i(\mathtt{T}) : i(\mathtt{K})$

References

Martín Abadi, Luca Cardelli, Pierre-Louis Curien, and Jean-Jacques Lévy. Explicit substitutions. *Journal of Functional Programming*, 1(4):375–416, 1991. Summary in *ACM Symposium on Principles of Programming Languages (POPL), San Francisco, California*, 1990.

Rolf Adams, Walter Tichy, and Annette Weinert. The cost of selective recompilation and environment processing. *ACM Transactions on Software Engineering and Methodology*, 3(1):3–28, January 1994.

Amal Ahmed, Limin Jia, and David Walker. Reasoning about hierarchical storage. In *IEEE Symposium on Logic in Computer Science (LICS), Ottawa, Canada*, pages 33–44, Ottawa, Canada, June 2003.

Amal Ahmed and David Walker. The logical approach to stack typing. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI), San Diego, California*, New Orleans, January 2003.

Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley, Reading, MA, USA, 1986.

Alex Aiken, Jeffrey S. Foster, John Kodumal, and Tachio Terauchi. Checking and inferring local non-aliasing. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI), San Diego, California*, San Diego, California, June 2003.

Alexander Aiken, Manuel Fähndrich, and Raph Levien. Better static memory management: Improving region-based analysis of higher-order languages. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI), La Jolla, California*, pages 174–185, 1995. URL `http://theory.stanford.edu/~aiken/publications/papers/pldi95.ps`.

Alexander Aiken and Edward L. Wimmers. Solving systems of set constraints (extended abstract). In *IEEE Symposium on Logic in Computer Science (LICS)*, pages 329–340, Santa Cruz, California, 22–25 June 1992. IEEE Computer Society Press.

Alexander Aiken and Edward L. Wimmers. Type inclusion constraints and type inference. In *ACM Symposium on Functional Programming Languages and Computer Architecture (FPCA)*, pages 31–41, 1993.

Thorsten Altenkirch. *Constructions, Inductive Types and Strong Normalization*. PhD thesis, LFCS, University of Edinburgh, 1993.

Roberto M. Amadio and Luca Cardelli. Subtyping recursive types. *ACM Transactions on Programming Languages and Systems*, 15(4):575–631, 1993. Summary in *ACM Symposium on Principles of Programming Languages (POPL), Orlando, Florida*, pp. 104–118; also DEC/Compaq Systems Research Center Research Report number 62, August 1990.

Torben Amtoft, Flemming Nielson, and Hanne Riis Nielson. *Type and Effect Systems: Behaviours for Concurrency*. Imperial College Press, 1999.

Davide Ancona and Elena Zucca. A theory of mixin modules: Basic and derived operators. *Mathematical Structures in Computer Science*, 8(4):401–446, August 1998.

Davide Ancona and Elena Zucca. A calculus of module systems. *Journal of Functional Programming*, 12(2):91–132, March 2002.

Andrew W. Appel. Foundational proof-carrying code. In *IEEE Symposium on Logic in Computer Science (LICS)*, pages 247–258, June 2001.

Andrew W. Appel and Amy P. Felty. A semantic model of types and machine instructions for proof-carrying code. In *ACM Symposium on Principles of Programming Languages (POPL), Boston, Massachusetts*, pages 243–253. ACM Press, January 2000.

David Aspinall. Subtyping with singleton types. In *Computer Science Logic (CSL), Kazimierz, Poland*, pages 1–15. Springer-Verlag, 1994.

David Aspinall and Martin Hofmann. Another type system for in-place update. In *European Symp. on Programming (ESOP)*, number 2305 in Lecture Notes in Computer Science, pages 36–52, Grenoble, France, April 2002. Springer.

Lennart Augustsson. Cayenne — a language with dependent types. In *International Conference on Functional Programming (ICFP), Baltimore, Maryland, USA*, pages 239–250, 1998.

Franz Baader and Jörg Siekmann. Unification theory. In D. M. Gabbay, C. J. Hogger, and J. A. Robinson, editors, *Handbook of Logic in Artificial Intelligence and Logic Programming*, volume 2, Deduction Methodologies, pages 41–125. Oxford University Press, Oxford, UK, 1994.

Henry G. Baker. Lively linear lisp – look ma, no garbage! *ACM SIGPLAN Notices*, 27 (8):89–98, 1992.

Henk P. Barendregt. *The Lambda Calculus*. North Holland, revised edition, 1984.

Henk P. Barendregt. Introduction to generalized type systems. *Journal of Functional Programming*, 1(2):125–154, 1991.

Henk P. Barendregt. Lambda calculi with types. In Abramsky, Gabbay, and Maibaum, editors, *Handbook of Logic in Computer Science*, volume II. Oxford University Press, 1992.

Erik Barendsen and Sjaak Smetsers. Conventional and uniqueness typing in graph rewrite systems. In Shyamasundar, editor, *Foundations of Software Technology and Theoretical Computer Science*, volume 761 of *Lecture Notes in Computer Science*, pages 41–51, Bombay, India, 1993. Springer-Verlag.

Bruno Barras, Samuel Boutin, Cristina Cornes, Judicael Courant, Jean-Christophe Filliatre, Eduardo Gimenez, Hugo Herbelin, Gerard Huet, Cesar Munoz, Chetan Murthy, Catherine Parent, Christine Paulin-Mohring, Amokrane Saibi, and Benjamin Werner. The Coq proof assistant reference manual : Version 6.1. Technical Report RT-0203, Inria (Institut National de Recherche en Informatique et en Automatique), France, 1997.

Lujo Bauer, Andrew W. Appel, and Edward W. Felten. Mechanisms for secure modular programming in java. Technical Report TR-603-99, Princeton University, 1999. URL `citeseer.nj.nec.com/bauer99mechanisms.html`.

Stephan Bellantoni and Stephan Cook. A new recursion-theoretic characterization of polytime functions. *Computational Complexity*, 2(2):97–110, 1992.

Stephan Bellantoni, K.-H. Niggl, and H. Schwichtenberg. Higher type recursion, ramification and polynomial time. *Annals of Pure and Applied Logic*, 104:17–30, 2000.

Stefano Berardi. Towards a mathematical analysis of the Coquand-Huet calculus of constructions and the other systems in Barendregt's cube. Technical report, Department of Computer Science, CMU, and Dipartimento Matematica, Universita di Torino, 1988.

Bernard Berthomieu. Tagged types. a theory of order sorted types for tagged expressions. Research Report 93083, LAAS, 7, avenue du Colonel Roche, 31077 Toulouse, France, March 1993.

Bernard Berthomieu and Camille le Moniès de Sagazan. A calculus of tagged types, with applications to process languages. In *Workshop on Types for Program Analysis*, pages 1–15, May 1995.

Edoardo Biagioni, Nicholas Haines, Robert Harper, Peter Lee, Brian G. Milnes, and Eliot B. Moss. Signatures for a protocol stack: A systems application of Standard ML. In *LISP and Functional Programming*, Orlando, FL, June 1994.

G. M. Bierman, A. M. Pitts, and C. V. Russo. Operational properties of Lily, a polymorphic linear lambda calculus with recursion. In *Fourth International Workshop on Higher Order Operational Techniques in Semantics, Montréal*, volume 41 of *Electronic Notes in Theoretical Computer Science*. Elsevier, September 2000. URL `http://www.elsevier.nl/locate/entcs/volume41.html`.

L. Birkedal and R. W. Harper. Constructing interpretations of recursive types in an operational setting. *Information and Computation*, 155:3–63, 1999.

Lars Birkedal and Mads Tofte. A constraint-based region inference algorithm. *Theoretical Computer Science*, 258:299–392, 2001. URL `http://www.it-c.dk/people/birkedal/papers/conria.ps.gz`.

Lars Birkedal, Mads Tofte, and Magnus Vejlstrup. From region inference to von Neumann machines via region representation inference. In *ACM Symposium on Principles of Programming Languages (POPL), St. Petersburg Beach, Florida*, pages 171–183, 1996.

Matthias Blume. *The SML/NJ Compilation and Library Manager*, May 2002. URL `http://www.smlnj.org/doc/CM/index.html`.

Matthias Blume and Andrew W. Appel. Hierarchical modularity. *ACM Transactions on Programming Languages and Systems*, 21(4):813–847, 1999. URL `citeseer.nj.nec.com/blume98hierarchical.html`.

Daniel Bonniot. Type-checking multi-methods in ML (a modular approach). In *Workshop on Foundations of Object-Oriented Languages (FOOL), informal proceedings*, January 2002.

François Bourdoncle and Stephan Merz. Type-checking higher-order polymorphic multi-methods. In *ACM Symposium on Principles of Programming Languages (POPL), Paris, France*, pages 302–315, 15–17 January 1997.

Gilad Bracha and William R. Cook. Mixin-based inheritance. In *European Conference on Object-Oriented Programming (ECOOP)*, pages 303–311, Ottawa, Canada, October 1990.

Michael Brandt and Fritz Henglein. Coinductive axiomatization of recursive type equality and subtyping. In Roger Hindley, editor, *Proc. 3d Int'l Conf. on Typed Lambda Calculi and Applications (TLCA), Nancy, France, April 2–4, 1997*, volume 1210 of *Lecture Notes in Computer Science (LNCS)*, pages 63–81. Springer-Verlag, April 1997. Full version in Fundamenta Informaticae, Vol. 33, pp. 309–338, 1998.

Val Breazu-Tannen, Thierry Coquand, Carl Gunter, and Andre Scedrov. Inheritance as implicit coercion. *Information and Computation*, 93:172–221, 1991. Also in Carl A. Gunter and John C. Mitchell, editors, *Theoretical Aspects of Object-Oriented Programming: Types, Semantics, and Language Design*, MIT Press, 1994.

Kim B. Bruce. Typing in object-oriented languages: Achieving expressibility and safety, 1995. Available through `http://www.cs.williams.edu/˜kim`.

Kim B. Bruce, Luca Cardelli, Giuseppe Castagna, the Hopkins Objects Group (Jonathan Eifrig, Scott Smith, Valery Trifonov), Gary T. Leavens, and Benjamin Pierce. On binary methods. *Theory and Practice of Object Systems*, 1(3):221–242, 1996.

Kim B. Bruce, Luca Cardelli, and Benjamin C. Pierce. Comparing object encodings. In *Theoretical Aspects of Computer Software (TACS), Sendai, Japan*, September 1997. An earlier version was presented as an invited lecture at the Third International Workshop on Foundations of Object Oriented Languages (FOOL 3), July 1996.

T. H. Brus, M. C. J. D. van Eekelen, M. O. van Leer, and M. J. Plasmeijer. Clean: A language for functional graph rewriting. In G. Kahn, editor, *Functional Programming Languages and Computer Architecture*, pages 364–384. Springer-Verlag, Berlin, DE, 1987. ISBN 3-540-18317-5. Lecture Notes in Computer Science 274; Proceedings of Conference held at Portland, OR.

Rod Burstall and Butler Lampson. A kernel language for abstract data types and modules. In G. Kahn, D. MacQueen, and G. Plotkin, editors, *Semantics of Data Types*, volume 173 of *Lecture Notes in Computer Science*, pages 1–50. Springer-Verlag, 1984.

Rod Burstall, David MacQueen, and Donald Sannella. HOPE: an experimental applicative language. In *Proceedings of the 1980 LISP Conference*, pages 136–143, Stanford, California, 1980. Stanford University.

Cristiano Calcagno. Stratified operational semantics for safety and correctness of region calculus. In *ACM Symposium on Principles of Programming Languages (POPL), London, England*, pages 155–165, 2001. URL `ftp://ftp.disi.unige.it/person/CalcagnoC/regions.ps`.

Cristiano Calcagno, Simon Helsen, and Peter Thiemann. Syntactic type soundness results for the region calculus. *Information and Computation*, 173(2):199–221, 2002. URL `http://www.swen.uwaterloo.ca/~shelsen/calcagno-helsen-thiemann-iandc-20%01.pdf`.

Luca Cardelli. A polymorphic λ-calculus with Type:Type. Research report 10, DEC/Compaq Systems Research Center, May 1986.

Luca Cardelli. Phase distinctions in type theory. unpublished manuscript, 1988a.

Luca Cardelli. Typechecking dependent types and subtypes. In M. Boscarol, L. Carlucci Aiello, and G. Levi, editors, *Foundations of Logic and Functional Programming, Workshop Proceedings, Trento, Italy, (Dec. 1986)*, volume 306 of *Lecture Notes in Computer Science*, pages 45–57. Springer-Verlag, 1988b.

Luca Cardelli. Program fragments, linking, and modularization. In *ACM Symposium on Principles of Programming Languages (POPL), Paris, France*, pages 266–277, Paris, France, January 1997. ACM Press.

Luca Cardelli, James Donahue, Mick Jordan, Bill Kalsow, and Greg Nelson. The Modula-3 type system. In *Proceedings of the Sixteenth Annual ACM Symposium on Principles of Programming Languages*, pages 202–212, January 1989.

Luca Cardelli and Xavier Leroy. Abstract types and the dot notation. In *Proceedings of the IFIP TC2 Working Conference on Programming Concepts and Methods*. North Holland, 1990. Also appeared as DEC/Compaq SRC technical report 56.

Luca Cardelli and Giuseppe Longo. A semantic basis for Quest. *Journal of Functional Programming*, 1(4):417–458, October 1991. Summary in ACM Conference on Lisp and Functional Programming, June 1990. Also available as DEC/Compaq SRC Research Report 55, Feb. 1990.

Luca Cardelli and John Mitchell. Operations on records. *Mathematical Structures in Computer Science*, 1:3–48, 1991. Also in Carl A. Gunter and John C. Mitchell, editors, *Theoretical Aspects of Object-Oriented Programming: Types, Semantics, and Language Design*, MIT Press, 1994; available as DEC/Compaq Systems Research Center Research Report #48, August, 1989, and in the proceedings of MFPS '89, Springer LNCS volume 442.

J. Cartmell. Generalised algebraic theories and contextual categories. *Annals of Pure and Applied Logic*, 32:209–243, 1986.

R. Cartwright and M. Fagan. Soft typing. In *Proceedings of the SIGPLAN '91 Conference on Programming Language Design and Implementation*, pages 278–292, June 1991. Also available as SIGPLAN Notices 26(6) June 1991.

Iliano Cervesato, Joshua S. Hodas, and Frank Pfenning. Efficient resource management for linear logic proof search. *Theoretical Computer Science*, 232(1–2):133–163, February 2000.

Iliano Cervesato and Frank Pfenning. A linear logical framework. *Information and Computation*, 179(1):19–75, November 2002.

Sagar Chaki, Sriram K. Rajamani, and Jakob Rehof. Types as models: Model checking message-passing programs. In *ACM Symposium on Principles of Programming Languages (POPL), Portland, Oregon*, pages 45–57, 2002.

Jawahar Chirimar, Carl A. Gunter, and Jon G. Riecke. Reference counting as a computational interpretation of linear logic. *Journal of Functional Programming*, 6(2): 195–244, March 1996.

Morten Voetmann Christiansen and Per Velschow. Region-based memory management in Java. Master's thesis, Department of Computer Science, University of Copenhagen (DIKU), 1998. URL `ftp://ftp.diku.dk/diku/semantics/papers/D-395.ps.gz`.

Alonzo Church. *The Calculi of Lambda Conversion*. Princeton University Press, 1941.

Alonzo Church. The weak theory of implication. *Kontroliertes Denken: Untersuchungen zum Logikkalk ul und zur Logik der Einzelwissenschaften*, pages 22–37, 1951.

Dominique Clement, Joelle Despeyroux, Thierry Despeyroux, and Gilles Kahn. A simple applicative language: Mini-ML. In *ACM Conference on LISP and Functional Programming*, pages 13–27, 1986.

Christopher Colby, Peter Lee, George C. Necula, Fred Blau, Mark Plesko, and Kenneth Cline. A certifying compiler for Java. *ACM SIGPLAN Notices*, 35(5):95–107, May 2000. ISSN 0362-1340.

Hubert Comon. Constraints in term algebras (short survey). In *Conference on Algebraic Methodology and Software Technology (AMAST)*, Workshops in Computing. Springer-Verlag, 1993.

Robert L. Constable et al. *Implementing Mathematics with the NuPRL Proof Development System*. Prentice-Hall, Englewood Cliffs, NJ, 1986.

Catarina Coquand. The AGDA proof system homepage, 1998. At `http://www.cs.chalmers.se/~catarina/agda/`.

Thierry Coquand. An analysis of Girard's paradox. In *IEEE Symposium on Logic in Computer Science (LICS)*, pages 227–236, 1986.

Thierry Coquand. An algorithm for testing conversion in type theory. In Gérard Huet and Gordon Plotkin, editors, *Logical Frameworks*, pages 255–279. Cambridge University Press, 1991.

Thierry Coquand. Pattern matching with dependent types. In *Proceedings of the Workshop on Types for Proofs and Programs, Baastad*. Informal proceedings available by ftp from `ftp://ftp.cs.chalmers.se/pub/cs-reports/baastad.92/proc.ps.Z`, 1992.

Thierry Coquand and Gérard Huet. The Calculus of Constructions. *Information and Computation*, 76(2/3):95–120, February/March 1988.

Thierry Coquand, Randy Pollack, and Makoto Takeyama. A logical framework with dependently typed records. In *Proc. Typed Lambda Calculi and Applications (TLCA 2003)*, pages 105–119, 2003. Available as LNCS 2701.

Judicaël Courant. Strong normalization with singleton types. In *Proceeedings of the Second Workshop on Intersection Types and Related Systems (ITRS '02)*, volume 70 of *ENTCS*, 2002.

Erik Crank and Matthias Felleisen. Parameter-passing and the lambda calculus. In *ACM Symposium on Principles of Programming Languages (POPL), Orlando, Florida*, pages 233–244, January 1991.

Karl Crary. A simple proof technique for certain parametricity results. In *International Conference on Functional Programming (ICFP), Paris, France*, pages 82 – 89, 1999.

Karl Crary. Toward a foundational typed assembly language. In *ACM Symposium on Principles of Programming Languages (POPL), New Orleans, Louisiana*, pages 198–212, January 2003.

Karl Crary, Robert Harper, and Sidd Puri. What is a recursive module? In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 50–63, May 1999.

Karl Crary, Stephanie Weirich, and J. Gregory Morrisett. Intensional polymorphism in type-erasure semantics. In *International Conference on Functional Programming (ICFP), Baltimore, Maryland, USA*, pages 301–312, 1998.

Pavel Curtis. *Constrained Quantification in Polymorphic Type Analysis*. PhD thesis, Cornell University, February 1990.

Luis Damas and Robin Milner. Principal type schemes for functional programs. In *ACM Symposium on Principles of Programming Languages (POPL), Albuquerque, New Mexico*, pages 207–212, 1982.

Olivier Danvy. Functional unparsing. *Journal of Functional Programming*, 8(6):621–625, 1998.

Nicolas G. de Bruijn. A survey of the project AUTOMATH. In J. P. Seldin and J. R. Hindley, editors, *To H. B. Curry: Essays in Combinatory Logic, Lambda Calculus, and Formalism*, pages 589–606. Academic Press, 1980.

Rob DeLine and Manuel Fähndrich. Enforcing high-level protocols in low-level software. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 59–69, Snowbird, Utah, June 2001.

James Donahue and Alan Demers. Data types are values. *ACM Transactions on Programming Languages and Systems*, 7(3):426–445, July 1985.

Kosta Došen and Peter Schroeder-Heister, editors. *Substructural Logics*, chapter A historical introduction to substructural logics, pages 1–30. Oxford University Press, 1993.

Derek Dreyer, Karl Crary, and Robert Harper. A type system for higher-order modules. In *ACM Symposium on Principles of Programming Languages (POPL), New Orleans, Louisiana*, pages 236–249, New Orleans, January 2003.

Dirk Dussart, Fritz Henglein, and Christian Mossin. Polymorphic recursion and subtype qualifications: Polymorphic binding-time analysis in polynomial time. In *International Symposium on Static Analysis (SAS)*, volume 983 of *Lecture Notes in Computer Science*, pages 118–135, 1995.

Martin Emms and Hans Leiß. Extending the type checker for SML by polymorphic recursion — A correctness proof. Technical Report 96-101, Centrum für Informations- und Sprachverarbeitung, Universität München, 1996.

Thomas Erhard. A categorical semantics of constructions. In *IEEE Symposium on Logic in Computer Science (LICS)*, pages 264–273, July 1988.

Manuel Fähndrich and Rob DeLine. Adoption and focus: Practical linear types for imperative programming. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 13–24, Berlin, June 2002.

Manuel Fähndrich, Jakob Rehof, and Manuvir Das. Scalable context-sensitive flow analysis using instantiation constraints. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI), Vancouver, British Columbia, Canada*, pages 253–263, 2000.

Matthias Felleisen and Robert Hieb. A revised report on the syntactic theories of sequential control and state. *Theoretical Computer Science*, 103(2):235–271, 1992.

Manuel Fähndrich. BANE*: A Library for Scalable Constraint-Based Program Analysis*. PhD thesis, University of California at Berkeley, 1999.

Manuel Fähndrich, Jakob Rehof, and Manuvir Das. Scalable context-sensitive flow analysis using instantiation constraints. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI), Vancouver, British Columbia, Canada*, June 2000.

Kathleen Fisher and John H. Reppy. The design of a class mechanism for Moby. In *SIGPLAN Conference on Programming Language Design and Implementation (PDLI)*, pages 37–49, 1999.

Cormac Flanagan and Shaz Qadeer. A type and effect system for atomicity. In *Proceedings of the ACM SIGPLAN 2003 conference on Programming language design and implementation*, pages 338–349. ACM Press, 2003. ISBN 1-58113-662-5.

Matthew Flatt and Matthias Felleisen. Units: Cool modules for HOT languages. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI), Montreal, Canada*, pages 236–248, 1998.

Matthew Fluet. Monadic regions. In *Proc. 2nd Workshop on Semantics, Program Analysis and Computing Environments for Memory Management (SPACE), Venice, Italy*, January 2004. URL `http://www.diku.dk/topps/space2004`.

Matthew Fluet and Riccardo Pucella. Phantom types and subtyping. In *IFIP International Conference on Theoretical Computer Science (TCS)*, pages 448–460, August 2002.

Jeffrey S. Foster, Tachio Terauchi, and Alex Aiken. Flow-sensitive type qualifiers. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI), Berlin, Germany*, pages 1–12, June 2002.

Alexandre Frey. Satisfying subtype inequalities in polynomial space. In Pascal Van Hentenryck, editor, *International Symposium on Static Analysis (SAS)*, volume 1302 of *Lecture Notes in Computer Science*, pages 265–277. Springer-Verlag, September 1997.

You-Chin Fuh and Prateek Mishra. Type inference with subtypes. In H. Ganzinger, editor, *European Symp. on Programming (ESOP)*, volume 300 of *Lecture Notes in Computer Science*, pages 94–114. Springer-Verlag, 1988.

Jun P. Furuse and Jacques Garrigue. A label-selective lambda-calculus with optional arguments and its compilation method. RIMS Preprint 1041, Kyoto University, October 1995.

Ronald Garcia, Jaakko Jarvi, Andrew Lumsdaine, Jeremy Siek, and Jeremia h Willcock. A comparative study of language support for generic programming. In *Proc. Object-Oriented Programing: Systems, Languages, and Applications (OOPSLA)*, pages 115–134. ACM Press, 2003. ISBN 1-58113-712-5.

Jacques Garrigue. Programming with polymorphic variants. In *Workshop on ML*, September 1998.

Jacques Garrigue. Code reuse through polymorphic variants. In *Workshop on Foundations of Software Engineering*, November 2000.

Jacques Garrigue. Simple type inference for structural polymorphism. In *Workshop on Foundations of Object-Oriented Languages (FOOL), informal proceedings*, January 2002.

Jacques Garrigue. Relaxing the value restriction. Draft., August 2003.

Jacques Garrigue and Didier Rémy. Extending ML with semi-explicit higher-order polymorphism. *Information and Computation*, 155(1):134–169, 1999.

Jaques Garrigue and Hassan Aït-Kaci. The typed polymorphic label-selective lambda-calculus. In *ACM Symposium on Principles of Programming Languages (POPL), Portland, Oregon*, pages 35–47, 1994.

Benedict R. Gaster. *Records, variants and qualified types*. PhD thesis, University of Nottingham, July 1998.

Benedict R. Gaster and Mark P. Jones. A polymorphic type system for extensible records and variants. Technical Report NOTTCS-TR-96-3, Department of Computer Science, University of Nottingham, November 1996.

David Gay and Alexander Aiken. Language support for regions. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI), Snowbird, Utah*, pages 70–80, 2001. URL `http://www.cs.berkeley.edu/~dgay/papers/pldi01.ps`.

Giorgio Ghelli and Benjamin Pierce. Bounded existentials and minimal typing. *Theoretical Computer Science*, 193:75–96, 1998. Circulated in manuscript form in 1992.

David K. Gifford and John M. Lucassen. Integrating functional and imperative programming. In *LISP and Functional Programming*, pages 28–38, 1986.

Jean-Yves Girard. *Interprétation fonctionnelle et élimination des coupures de l'arithmétique d'ordre supérieur*. Thèse d'état, University of Paris VII, 1972. Summary in *Proceedings of the Second Scandinavian Logic Symposium* (J.E. Fenstad, editor), North-Holland, 1971 (pp. 63–92).

Jean-Yves Girard. Linear logic. *Theoretical Computer Science*, 50:1–102, 1987.

Jean-Yves Girard. Light linear logic. *Information and Computation*, 143:175–204, 1998.

Jean-Yves Girard, Yves Lafont, and Paul Taylor. *Proofs and Types*, volume 7 of *Cambridge Tracts in Theoretical Computer Science*. Cambridge University Press, 1989.

Neal Glew. Type dispatch for named hierarchical types. In *International Conference on Functional Programming (ICFP), Paris, France*, pages 172–182, 1999.

GNU. GNU C library, version 2.2.5, 2001. URL `http://www.gnu.org/manual/glibc-2.2.5/html_mono/libc.html`.

Healfdene Goguen. *A Typed Operational Semantics for Type Theory*. PhD thesis, LFCS, University of Edinburgh, 1994. Report ESC-LFCS-94-304.

A. D. Gordon. Bisimilarity as a theory of functional programming. In *Eleventh Conference on the Mathematical Foundations of Programming Semantics, New Orleans, 1995*, volume 1 of *Electronic Notes in Theoretical Computer Science*. Elsevier, 1995.

A. D. Gordon. Operational equivalences for untyped and polymorphic object calculi. In A. D. Gordon and A. M. Pitts, editors, *Higher Order Operational Techniques in Semantics*, Publications of the Newton Institute, pages 9–54. Cambridge University Press, 1998.

Andrew D. Gordon and Alan Jeffrey. Authenticity by typing for security protocols. In *Proc. 14th IEEE Computer Security Foundations Workshop (CSFW 2001), Cape Breton*, pages 145–159, 2001a.

Andrew D. Gordon and Alan Jeffrey. Typing correspondence assertions for communiation protocols. *Electronic Notes in Theoretical Computer Science*, 45:22 pages, 2001b. `http://www.elsevier.nl/locate/entcs/volume45.html`.

Andrew D. Gordon and Alan Jeffrey. Types and effects for asymmetric cryptographic protocols. In *Proc. 15th IEEE Computer Security Foundations Workshop (CSFW 2002), Cape Breton*, pages 77–91, 2002.

Andrew D. Gordon and Don Syme. Typing a multi-language intermediate code. In *ACM Symposium on Principles of Programming Languages (POPL), London, England*, pages 248–260, London, United Kingdom, January 2001.

Michael J. Gordon, Robin Milner, and Christopher P. Wadsworth. *Edinburgh LCF*. Springer-Verlag LNCS 78, 1979.

John Gough. *Compiling for the .NET Common Language Runtime*. .NET series. Prentice Hall, Upper Saddle River, New Jersey, 2002. ISBN 0130622966.

Dan Grossman, Greg Morrisett, Trevor Jim, Michael Hicks, Yanling Wang, and James Cheney. Region-based memory management in Cyclone. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI), Berlin, Germany*, pages 282–293. ACM Press, 2002.

Jörgen Gustavsson and Josef Svenningsson. Constraint abstractions. In *Symposium on Programs as Data Objects*, volume 2053 of *Lecture Notes in Computer Science*. Springer-Verlag, May 2001.

Niels Hallenberg, Martin Elsman, and Mads Tofte. Combining region inference and garbage collection. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI), Berlin, Germany*. ACM Press, June 2002. Berlin, Germany.

Thomas Hallgren and Aarne Ranta. An extensible proof text editor (abstract). In *Logic for Programming and Automated Reasoning (LPAR'2000)*, volume 1955 of *Lecture Notes in Computer Science*, pages 70–84. Springer-Verlag, 2000.

Nadeem Hamid, Zhong Shao, Valery Trifonov, Stefan Monnier, and Zhaozhong Ni. A syntactic approach to foundational proof-carrying code. In *IEEE Symposium on Logic in Computer Science (LICS)*, pages 89–100, July 2002.

David R. Hanson. Fast allocation and deallocation of memory based on object lifetimes. *Software—Practice and Experience*, 20(1):5–12, 1990.

Thérèse Hardin, Luc Maranget, and Bruno Pagano. Functional runtimes within the lambda-sigma calculus. *Journal of Functional Programming*, 8(2), March 1998.

Robert Harper. On equivalence and canonical forms in the LF type theory. (Submitted for publication.), August 2002.

Robert Harper, Furio Honsell, and Gordon Plotkin. A framework for defining logics. *Journal of the ACM*, 40(1):143–184, 1993. Summary in LICS'87.

Robert Harper and Mark Lillibridge. A type-theoretic approach to higher-order modules with sharing. In *ACM Symposium on Principles of Programming Languages (POPL), Portland, Oregon*, pages 123–137, January 1994.

Robert Harper and John C. Mitchell. On the type structure of Standard ML. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 15(2):211–252, April 1993. An earlier version appeared in the Proceedings of the Fifteenth ACM Symposium on Principles of Programming Languages, San Diego, CA, January 1988, under the title "The Essence of ML" (Mitchell and Harper).

Robert Harper, John C. Mitchell, and Eugenio Moggi. Higher-order modules and the phase distinction. In *ACM Symposium on Principles of Programming Languages (POPL), San Francisco, California*, pages 341–354, January 1990.

Robert Harper and Robert Pollack. Type checking with universes. *Theoretical Computer Science*, 89:107–136, 1991.

Robert Harper and Christopher Stone. A type-theoretic interpretation of Standard ML. In Gordon Plotkin, Colin Stirling, and Mads Tofte, editors, *Proof, Language and Interaction: Essays in Honour of Robin Milner*. MIT Press, 2000.

Nevin Heintze. Set based analysis of ML programs. In *Proc. ACM Conf. on LISP and Functional Programming (LFP)*, pages 306–317. ACM, ACM Press, June 1994.

Nevin Heintze. Control-flow analysis and type systems. In Alan Mycroft, editor, *International Symposium on Static Analysis (SAS), Glasgow, Scotland*, volume 983 of *Lecture Notes in Computer Science*, pages 189–206. Springer-Verlag, 1995.

Simon Helsen and Peter Thiemann. Syntactic type soundness for the region calculus. In Alan Jeffrey, editor, *ACM Workshop on Higher Order Operational Techniques in Semantics*, volume 41(3) of *Electronic Notes in Theoretical Computer Science*, pages 1–20. Elsevier, September 2000. URL `http://www.elsevier.nl/locate/entcs/volume41.html`.

Simon Helsen and Peter Thiemann. Polymorphic specialization for ML. Available from www.helsen.org, 2003.

Fritz Henglein. *Polymorphic Type Inference and Semi-Unification*. PhD thesis, Rutgers University, April 1989. Available as NYU Technical Report 443, May 1989, from New York University, Courant Institute of Mathematical Sciences, Department of Computer Science, 251 Mercer St., New York, N.Y. 10012, USA.

Fritz Henglein. Type inference with polymorphic recursion. *ACM Transactions on Programming Languages and Systems*, 15(2):253–289, 1993.

Fritz Henglein, Henning Makholm, and Henning Niss. A direct approach to control-flow sensitive region-based memory management. In *ACM SIGPLAN Conference on Principles and Practice of Declarative Programming (PPDP)*, pages 175–186, 2001. URL `http://www.diku.dk/~hniss/publications/ppdp2001-abstract.html`.

Fritz Henglein and Christian Mossin. Polymorphic binding-time analysis. In *European Symp. on Programming (ESOP)*, pages 287–301, 1994.

Tom Hirschowitz and Xavier Leroy. Mixin modules in a call-by-value setting. In *European Symp. on Programming (ESOP)*, pages 6–20, 2002. URL `citeseer.nj.nec.com/article/hirschowitz02mixin.html`.

C. A. R. Hoare. Proof of correctness of data representation. *Acta Informatica*, 1:271–281, 1972.

Martin Hofmann. A mixed modal/linear lambda calculus with applications to bellantoni-cook safe recursion. In *Conference for Computer Science Logic*, pages 275–294, Aarhus, Denmark, August 1997a.

Martin Hofmann. Syntax and semantics of dependent types. In *Semantics and Logic of Computation*. Cambridge University Press, Cambridge, England, 1997b.

Martin Hofmann. Linear types and non-size-increasing polynomial time computation. In *IEEE Symposium on Logic in Computer Science (LICS)*, pages 464–473, Los Alamitos, CA, June 1999.

Martin Hofmann. Safe recursion with higher types and BCK-algebra. *Annals of Pure and Applied Logic*, pages 113–166, 2000.

F. Honsell, I. A. Mason, S. F. Smith, and C. L. Talcott. A variable typed logic of effects. *Information and Computation*, 119(1):55–90, 1995.

William A. Howard. Hereditarily majorizable functionals of finite type. In Anne Sjerp Troelstra, editor, *Metamathematical Investigation of Intuitionistic Arithmetic and Analysis*, volume 344 of *Lecture Notes in Mathematics*, pages 454–461. Springer-Verlag, Berlin, 1973. Appendix.

William A. Howard. The formulas-as-types notion of construction. In J. P. Seldin and J. R. Hindley, editors, *To H. B. Curry: Essays on Combinatory Logic, Lambda Calculus, and Formalism*, pages 479–490. Academic Press, New York, 1980. Reprint of 1969 article.

D. J. Howe. Proving congruence of bisimulation in functional programming languages. *Information and Computation*, 124(2):103–112, 1996.

Gérard Huet. *Résolution d'equations dans les langages d'ordre 1,2, ...,ω*. Thèse de Doctorat d'Etat, Université de Paris 7 (France), 1976.

Atsushi Igarashi and Naoki Kobayashi. A generic type system for the pi-calculus. In *ACM Symposium on Principles of Programming Languages (POPL), London, England*, pages 128–141, 2001.

Atsushi Igarashi and Naoki Kobayashi. Resource usage analysis. In *ACM Symposium on Principles of Programming Languages (POPL), Portland, Oregon*, pages 331–342, Portland, Oregon, January 2002.

Atsushi Igarashi and Benjamin C. Pierce. Foundations for virtual types. In *European Conference on Object-Oriented Programming (ECOOP)*, 1999. Also in informal proceedings of the Sixth International Workshop on Foundations of Object-Oriented Languages (FOOL). Full version to appear in *Information and Computation*.

Samin Ishtiaq and Peter O'Hearn. Bi as an assertion language for mutable data structures. In *ACM Symposium on Principles of Programming Languages (POPL), London, England*, 2001.

Bart Jacobs. *Categorical Logic and Type Theory*. Studies in Logic and the Foundations of Mathematics 141. Elsevier, 1999.

Lalita A. Jategaonkar. ML with extended pattern matching and subtypes. Master's thesis, MIT, August 1989.

Lalita A. Jategaonkar and John C. Mitchell. ML with extended pattern matching and subtypes (preliminary version). In *Proceedings of the ACM Conference on Lisp and Functional Programming*, pages 198–211, Snowbird, Utah, July 1988.

Kathleen Jensen and Niklaus Wirth. *Pascal User Manual and Report*. Springer-Verlag, second edition, 1975.

Trevor Jim. What are principal typings and what are they good for? In ACM, editor, *ACM Symposium on Principles of Programming Languages (POPL), St. Petersburg Beach, Florida*, pages 42–53, 1996.

Trevor Jim, J. Greg Morrisett, Dan Grossman, Michael W. Hicks, James Cheney, and Yanling Wang. Cyclone: A safe dialect of C. In *Proceedings of the General Track: 2002 USENIX Annual Technical Conference*, pages 275–288. USENIX Association, 2002.

Trevor Jim and Jens Palsberg. Type inference in systems of recursive types with subtyping. Manuscript, 1999.

P. Johann. A generalization of short-cut fusion and its correctness proof. *Higher-Order and Symbolic Computation*, 15(4):273–300, 2002.

Mark P. Jones. *Qualified Types: Theory and Practice*. Cambridge University Press, Cambridge, England, 1994.

Mark P. Jones. Using parameterized signatures to express modular structure. In *ACM Symposium on Principles of Programming Languages (POPL), St. Petersburg Beach, Florida*, January 21–24, 1996.

Mark P. Jones. Typing Haskell in Haskell. In *Haskell Workshop*, October 1999.

Mark P. Jones and John C. Peterson. The Hugs 98 user manual. `http://www.haskell.org/hugs/`, September 1999.

Mark P. Jones and Simon Peyton Jones. Lightweight extensible records for Haskell. In *Haskell Workshop*, October 1999.

Jean-Pierre Jouannaud and Claude Kirchner. Solving equations in abstract algebras: a rule-based survey of unification. In Jean-Louis Lassez and Gordon Plotkin, editors, *Computational Logic. Essays in honor of Alan Robinson*, chapter 8, pages 257–321. MIT Press, 1991.

Pierre Jouvelot and David Gifford. Algebraic reconstruction of types and effects. In *ACM Symposium on Principles of Programming Languages (POPL), Orlando, Florida*, pages 303–310, January 1991.

Pierre Jouvelot and David K. Gifford. Reasoning about continuations with control effects. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI), Portland, Oregon*, pages 218–226, 1989.

Achim Jung and Allen Stoughton. Studying the fully abstract model of PCF within its continuous function model. In M. Bezem and J.M. Groote, editors, *Proc. Typed Lambda Calculi and Applications (TLCA)*, volume 664 of *Lecture Notes in Computer Science*, pages 230–244. Springer-Verlag, 1993.

L.S. van Benthem Jutting, James McKinna, and Robert Pollack. Checking algorithms for Pure Type Systems. In Henk Barendregt and Tobias Nipkow, editors, *Proceedings of the International Workshop on Types for Proofs and Programs*, pages 19–61, Nijmegen, The Netherlands, May 1994. Springer-Verlag LNCS 806.

Assaf J. Kfoury, Jerzy Tiuryn, and Pawel Urzyczyn. ML typability is DEXPTIME-complete. In *Proc. 15th Colloq. on Trees in Algebra and Programming*, pages 206–220. Springer LNCS 431, 1990.

Assaf J. Kfoury, Jerzy Tiuryn, and Pawel Urzyczyn. The undecidability of the semi-unification problem. *Information and Computation*, 102(1):83–101, January 1993. Summary in *STOC 1990*.

Assaf J. Kfoury, Jerzy Tiuryn, and Pawel Urzyczyn. An analysis of ML typability. *Journal of the ACM*, 41(2):368–398, March 1994.

Claude Kirchner and F. Klay. Syntactic theories and unification. In *IEEE Symposium on Logic in Computer Science (LICS)*, pages 270–277, June 1990.

Kevin Knight. Unification: a multidisciplinary survey. *ACM Computing Surveys*, 21 (1):93–124, March 1989.

Naoki Kobayashi. Quasi-linear types. In *ACM Symposium on Principles of Programming Languages (POPL), San Antonio, Texas*, pages 29–42, San Antonio, January 1999.

Dexter Kozen, Jens Palsberg, and Michael I. Schwartzbach. Efficient recursive subtyping. *Mathematical Structures in Computer Science*, 5(1):113–125, 1995.

Viktor Kuncak and Martin Rinard. Structural subtyping of non-recursive types is decidable. In *IEEE Symposium on Logic in Computer Science (LICS)*, June 2003.

Yves Lafont. The linear abstract machine. *Theoretical Computer Science*, 59:157–180, 1988.

Joachim Lambek. The mathematics of sentence structure. *American Mathematical Monthly*, 65:154–170, 1958.

B. Lampson and R. Burstall. Pebble, a kernel language for modules and abstract data types. *Information and Computation*, 76:278–346, February/March 1988.

S. B. Lassen. *Relational Reasoning about Functions and Nondeterminism*. PhD thesis, Department of Computer Science, University of Aarhus, 1998.

Jean-Louis Lassez, Michael J. Maher, and Kim G. Marriott. Unification revisited. In Jack Minker, editor, *Foundations of Deductive Databases and Logic Programming*, chapter 15, pages 587–625. Morgan Kaufmann, 1988.

Oukseh Lee and Kwangkeun Yi. Proofs about a folklore let-polymorphic type inference algorithm. *ACM Transactions on Programming Languages and Systems*, 20(4): 707–723, July 1998.

Daniel Leivant. Stratified functional programs and computational complexity. In *ACM Symposium on Principles of Programming Languages (POPL), Charleston, South Carolina*, pages 325–333, jan 1993.

Xavier Leroy. Polymorphic typing of an algorithmic language. Research Report 1778, INRIA, October 1992.

Xavier Leroy. Manifest types, modules and separate compilation. In *ACM Symposium on Principles of Programming Languages (POPL), Portland, Oregon*, pages 109–122, January 1994.

Xavier Leroy. Applicative functors and fully transparent higher-order modules. In *Proceedings of the Twenty-Second ACM Symposium on Principles of Programming Languages (POPL), Portland, Oregon*, pages 142–153, San Francisco, California, January 1995.

Xavier Leroy. A syntactic theory of type generativity and sharing. *Journal of Functional Programming*, 6(5):667–698, 1996a.

Xavier Leroy. A syntactic theory of type generativity and sharing. *Journal of Functional Programming*, 6(5):667–698, September 1996b.

Xavier Leroy. The Objective Caml system: Documentation and user's manual, 2000. With Damien Doligez, Jacques Garrigue, Didier Rémy, and Jérôme Vouillon. Available from `http://caml.inria.fr`.

Xavier Leroy and François Pessaux. Type-based analysis of uncaught exceptions. *ACM Transactions on Programming Languages and Systems*, 22(2):340–377, March 2000. Summary in *ACM Symposium on Principles of Programming Languages (POPL), San Antonio, Texas*, 1999.

Mark Lillibridge. *Translucent Sums: A Foundation for Higher-Order Module Systems*. PhD thesis, School of Computer Science, Carnegie Mellon University, Pittsburgh, PA, May 1997.

Tim Lindholm and Frank Yellin. *The Java Virtual Machine Specification*. The Java Series. Addison-Wesley, Reading, MA, USA, January 1997. ISBN 0-201-63452-X. URL `http://www.aw.com/cp/javaseries.html`.

Barbara Liskov. A history of CLU. *ACM SIGPLAN Notices*, 28(3):133–147, 1993.

Ralph Loader. Finitary PCF is not decidable. *Theoretical Computer Science*, 266(1-2): 341–364, September 2001.

J.M. Lucassen. *Types and Effects towards the Integration of Functional and Imperative Programming*. PhD thesis, Massachusetts Institute of Technology (MIT), August 1987. Technical Report MIT-LCS-TR-408.

John M. Lucassen and David K. Gifford. Polymorphic effect systems. In *ACM Symposium on Principles of Programming Languages (POPL), San Diego, California*, pages 47–57, 1988.

Z. Luo. *Computation and Reasoning: A Type Theory for Computer Science*. Number 11 in International Series of Monographs on Computer Science. Oxford University Press, 1994.

Zhaohui Luo and Robert Pollack. The LEGO proof development system: A user's manual. Technical Report ECS-LFCS-92-211, University of Edinburgh, May 1992.

David MacQueen. Modules for Standard ML. In *ACM Conference on LISP and Functional Programming*, pages 198–207, 1984.

David MacQueen. Using dependent types to express modular structure. In *ACM Symposium on Principles of Programming Languages (POPL), St. Petersburg Beach, Florida*, pages 277–286, January 1986.

David B. MacQueen and Mads Tofte. A semantics for higher-order functors. In *European Symposium on Programming*, pages 409–423. Springer-Verlag, LNCS 788, April 1994.

Lena Magnusson and Bengt Nordström. The ALF proof editor and its proof engine. In *Types for Proofs and Programs*, volume 806 of *Lecture Notes in Computer Science*, pages 213–237. Springer-Verlag, 1994.

Harry G. Mairson, Paris C. Kanellakis, and John C. Mitchell. Unification and ML type reconstruction. In Jean-Louis Lassez and Gordon Plotkin, editors, *Computational Logic: Essays in Honor of Alan Robinson*, pages 444–478. MIT Press, 1991.

Henning Makholm. Region-based memory management in Prolog. Master's thesis, Department of Computer Science, University of Copenhagen (DIKU), March 2000. URL `ftp://ftp.diku.dk/diku/semantics/papers/D-421.ps.gz`. DIKU-TR-00/09.

Henning Makholm. *A language-independend framework for region inference*. PhD thesis, Department of Computer Science, University of Copenhagen (DIKU), 2003.

Henning Makholm and Kostis Sagonas. On enabling the WAM with region support. In *International Conference on Logic Programming*, pages 163–178, 2002.

Alberto Martelli and Ugo Montanari. Unification in linear time and space: A structured presentation. Internal Report B76-16, Istituto di Elaborazione delle Informazione, Consiglio Nazionale delle Ricerche, Pisa, July 1976.

Alberto Martelli and Ugo Montanari. An efficient unification algorithm. *ACM Transactions on Programming Languages and Systems*, 4(2):258–282, 1982.

Per Martin-Löf. *Intuitionistic Type Theory*. Bibliopolis, 1984.

I. A. Mason, S. F. Smith, and C. L. Talcott. From operational semantics to domain theory. *Information and Computation*, 128(1):26–47, 1996.

I. A. Mason and C. L. Talcott. Equivalence in functional languages with effects. *Journal of Functional Programming*, 1:287–327, 1991.

David McAllester. On the complexity analysis of static analyses. *Journal of the ACM*, 49(4):512–537, July 2002.

David McAllester. A logical algorithm for ML type inference. In *Rewriting Techniques and Applications (RTA)*, volume 2706 of *Lecture Notes in Computer Science*, pages 436–451. Springer-Verlag, June 2003.

Conor McBride. *Dependently Typed Functional Programs and their Proofs*. PhD thesis, LFCS, University of Edinburgh, 2000.

Conor McBride and James McKinna. The view from the left. *Journal of Functional Programming*, pages 69–111, 2004.

James McKinna and Robert Pollack. Pure Type Sytems formalized. In M. Bezem and J. F. Groote, editors, *Proceedings of the International Conference on Typed Lambda Calculi and Applications*, pages 289–305. Springer-Verlag LNCS 664, March 1993.

David Melski and Thomas Reps. Interconvertibility of a class of set constraints and context-free language reachability. *Theoretical Computer Science*, 248(1–2), November 2000.

Robin Milner. A theory of type polymorphism in programming. *Journal of Computer and System Sciences*, 17:348–375, August 1978.

Robin Milner, Mads Tofte, and Robert Harper. *The Definition of Standard ML*. MIT Press, 1990.

Robin Milner, Mads Tofte, Robert Harper, and David MacQueen. *The Definition of Standard ML (Revised)*. MIT Press, 1997.

Yasuhiko Minamide. A functional representation of data structures with a hole. In *ACM Symposium on Principles of Programming Languages (POPL), San Diego, California*, pages 75–84, January 1998.

Yasuhiko Minamide, Greg Morrisett, and Robert Harper. Typed closure conversion. In *ACM Symposium on Principles of Programming Languages (POPL), St. Petersburg Beach, Florida*, pages 271–283, January 1996.

A. Miquel. *Le Calcul des Constructions implicite: syntaxe et sémantique*. PhD thesis, University Paris 7, 2001.

J. C. Mitchell. On the equivalence of data representations. In V. Lifschitz, editor, *Artificial Intelligence and Mathematical Theory of Computation: Papers in Honor of John McCarthy*, pages 305–330. Academic Press, 1991a.

John C. Mitchell. Coercion and type inference (summary). In *ACM Symposium on Principles of Programming Languages (POPL), Salt Lake City, Utah*, pages 175–185, January 1984.

John C. Mitchell. Representation independence and data abstraction. In *ACM Symposium on Principles of Programming Languages (POPL), St. Petersburg Beach, Florida*, pages 263–276, 1986.

John C. Mitchell. Type inference with simple subtypes. *Journal of Functional Programming*, 1(3):245–286, July 1991b.

John C. Mitchell. *Foundations for Programming Languages*. MIT Press, 1996.

John C. Mitchell and Gordon D. Plotkin. Abstract types have existential types. *ACM Trans. on Programming Languages and Systems*, 10(3):470–502, 1988. Summary in *ACM Symposium on Principles of Programming Languages (POPL), New Orleans, Louisiana*, 1985.

Eugenio Moggi. Computational lambda-calculus and monads. In *Fourth Annual Symposium on Logic in Computer Science (Asilomar, CA)*, pages 14–23. IEEE Computer Society Press, June 1989. Full version, titled *Notions of Computation and Monads*, in Information and Computation, 93(1), pp. 55–92, 1991.

Shaw-Kwei Moh. The deduction theorems and two new logical systems. *Methodos*, 2: 56–75, 1950.

Christine Mohring. Algorithm development in the calculus of constructions. In *IEEE Symposium on Logic in Computer Science (LICS)*, pages 84–91, 1986.

Stefan Monnier, Bratin Saha, and Zhong Shao. Principled scavenging. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI), Snowbird, Utah*, pages 81–91, June 2001.

Greg Morrisett, Karl Crary, Neal Glew, and David Walker. Stack-based typed assembly language. *Journal of Functional Programming*, 12(1):43–88, January 2002.

Greg Morrisett, David Walker, Karl Crary, and Neal Glew. From System-F to typed assembly language. *ACM Transactions on Programming Languages and Systems*, 21 (3):527–568, May 1999.

Christian Mossin. *Flow Analysis of Typed Higher-Order Programs*. PhD thesis, DIKU, University of Copenhagen, Copenhagen, Denmark, 1997. URL `http://www.diku.dk/research/published/97-1.ps.gz`. Technical Report DIKU-TR-97/1.

Martin Müller. A constraint-based recast of ML-polymorphism. In *International Workshop on Unification*, June 1994. Technical Report 94-R-243, CRIN, Nancy, France. `http://www.ps.uni-sb.de/Papers/abstracts/UNIF94.ps`.

Martin Müller. Notes on HM(X). `http://www.ps.uni-sb.de/~mmueller/papers/HMX.ps.gz`, August 1998.

Martin Müller, Joachim Niehren, and Ralf Treinen. The first-order theory of ordering constraints over feature trees. *Discrete Mathematics and Theoretical Computer Science*, 4(2):193–234, 2001.

Martin Müller and Susumu Nishimura. Type inference for first-class messages with feature constraints. In Jieh Hsiang and Atsushi Ohori, editors, *Asian Computer Science Conference (ASIAN 98)*, volume 1538 of *Lecture Notes in Computer Science*, pages 169–187, Manila, The Philippines, December 1998. Springer-Verlag.

Alan Mycroft. Polymorphic type schemes and recursive definitions. In M. Paul and B. Robinet, editors, *Proceedings of the International Symposium on Programming*, volume 167 of *LNCS*, pages 217–228, Toulouse, France, April 1984. Springer.

George C. Necula. Proof-carrying code. In *ACM Symposium on Principles of Programming Languages (POPL), Paris, France*, pages 106–119, 15–17 January 1997.

George C. Necula. *Compiling with Proofs*. PhD thesis, Carnegie Mellon University, September 1998. Also available as CMU-CS-98-154.

George C. Necula. Translation validation for an optimizing compiler. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI), Vancouver, British Columbia, Canada*, pages 83–94. ACM SIGPLAN, 18–21 June 2000.

George C. Necula and Peter Lee. Safe kernel extensions without run-time checking. In *2nd Symposium on Operating Systems Design and Implementation (OSDI '96), October 28–31, 1996, Seattle, WA*, pages 229–243, Berkeley, CA, USA, October 1996. USENIX press.

George C. Necula and Peter Lee. The design and implementation of a certifying compiler. In *Proceedings of the ACM SIGPLAN'98 Conference on Programming Language Design and Implementation (PLDI)*, pages 333–344, Montreal, Canada, 17–19 June 1998a.

George C. Necula and Peter Lee. Efficient representation and validation of logical proofs. In *IEEE Symposium on Logic in Computer Science (LICS)*, pages 93–104. IEEE Computer Society Press, 1998b.

Joachim Niehren, Martin Müller, and Andreas Podelski. Inclusion constraints over non-empty sets of trees. In Max Dauchet, editor, *International Joint Conference on Theory and Practice of Software Development (TAPSOFT)*, volume 1214 of *Lecture Notes in Computer Science*, pages 217–231. Springer-Verlag, April 1997.

Joachim Niehren and Tim Priesnitz. Non-structural subtype entailment in automata theory. *Information and Computation*, 2003. To appear.

Flemming Nielson and Hanne Riis Nielson. From CML to its process algebra. *Theoretical Computer Science*, 155:179–219, 1996.

Flemming Nielson, Hanne Riis Nielson, and C. L. Hankin. *Principles of Program Analysis*. Springer-Verlag, 1999.

Flemming Nielson, Hanne Riis Nielson, and Helmut Seidl. A succinct solver for ALFP. *Nordic Journal of Computing*, 9(4):335–372, 2002.

Hanne Riis Nielson and Flemming Nielson. Higher-order concurrent programs with finite communication topology. In *ACM Symposium on Principles of Programming Languages (POPL), Portland, Oregon*, pages 84–97, January 1994.

Susumu Nishimura. Static typing for dynamic messages. In *ACM Symposium on Principles of Programming Languages (POPL), San Diego, California*, pages 266–278, New York, 1998. ACM Press.

Henning Niss. *Regions are Imperative: Unscoped Regions and Control-Flow Sensitive Memory Management*. PhD thesis, Department of Computer Science, University of Copenhagen (DIKU), 2002.

E.G.J.M.H. Nöcker and J.E.W. Smetsers. Partially strict non-recursive data types. *Journal of Functional Programming*, 3(2):191–215, 1993.

Erick G. M. H. Nöcker, Sjaak E. W. Smetsers, Marko C. J. D. van Eekelen, and Marinus J. Plasmeijer. Concurrent clean. In E. H. L. Aarts, J. van Leeuwen, and M. Rem, editors, *PARLE '91, Parallel Architectures and Languages Europe, Volume I: Parallel Architectures and Algorithms*, volume 505 of *Lecture Notes in Computer Science*, pages 202–219, Eindhoven, The Netherlands, June 10–13, 1991. Springer, Berlin.

Martin Odersky. Observers for linear types. In B. Krieg-Brückner, editor, *European Symp. on Programming (ESOP)*, pages 390–407. Springer-Verlag, February 1992. Lecture Notes in Computer Science 582.

Martin Odersky, Vincent Cremet, Christine Rockl, and Matthias Zenger. A nominal theory of objects with dependent types. In *Workshop on Foundations of Object-Oriented Languages (FOOL), informal proceedings*, 2003.

Martin Odersky, Martin Sulzmann, and Martin Wehr. Type inference with constrained types. *Theory and Practice of Object Systems*, 5(1):35–55, 1999. Summary in *Workshop on Foundations of Object-Oriented Languages (FOOL), informal proceedings*, 1997.

Peter O'Hearn. On bunched typing. *Journal of Functional Programming*, 13(4):747–796, 2003.

Peter O'Hearn and David Pym. The logic of bunched implications. *Bulletin of Symbolic Logic*, 5(2):215–244, 1999.

Atsushi Ohori. A polymorphic record calculus and its compilation. *ACM Transactions on Programming Languages and Systems*, 17(6):844–895, November 1995.

Atsushi Ohori and Peter Buneman. Type inference in a database programming language. In *1988 ACM Conference on Lisp and Functional Programming*, pages 174–183, Snowbird, Utah, July 1988. Revised manuscript, September, 1988.

Atsushi Ohori and Peter Buneman. Static type inference for parametric classes. In *ACM Symposium on Object Oriented Programming: Systems, Languages, and Applications (OOPSLA)*, pages 445–456, October 1989. Also in Carl A. Gunter and John C. Mitchell, editors, *Theoretical Aspects of Object-Oriented Programming: Types, Semantics, and Language Design*, MIT Press, 1994.

I. E. Orlov. The calculus of compatibility of propositions (in russian). *Matematicheskii Sbornik*, 35:263–286, 1928.

S. Owre, S. Rajan, J.M. Rushby, N. Shankar, and M.K. Srivas. PVS: Combining specification, proof checking, and model checking. In Rajeev Alur and Thomas A. Henzinger, editors, *Computer-Aided Verification, CAV '96*, volume 1102 of *Lecture Notes in Computer Science*, pages 411–414, New Brunswick, NJ, July/August 1996. Springer-Verlag.

Jens Palsberg. Efficient inference of object types. *Information and Computation*, 123(2): 198–209, 1995.

Jens Palsberg. Type-based analysis and applications. In *Proceedings of the 2001 ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering*, pages 20–27. ACM Press, 2001. ISBN 1-58113-413-4.

Jens Palsberg and Patrick O'Keefe. A type system equivalent to flow analysis. In *ACM Symposium on Principles of Programming Languages (POPL), San Francisco, California*, pages 367–378, 1995.

Jens Palsberg and Michael Schwartzbach. Type substitution for object-oriented programming. In N. Meyrowitz, editor, *Proc. Object-Oriented Programming: Systems, Languages (OOPSLA), and Applications and European Conf. on Object-Oriented Programming (ECOOP)*, pages 151–160, Ottawa, Canada, October 1990. ACM Press.

Jens Palsberg and Michael I. Schwartzbach. *Object-Oriented Type Systems*. Wiley, 1994.

Jens Palsberg, Mitchell Wand, and Patrick M. O'Keefe. Type inference with non-structural subtyping. *Formal Aspects of Computing*, 9:49–67, 1997.

David Parnas. The criteria to be used in decomposing systems into modules. *Communications of the ACM*, 14(1):221–227, 1972.

Michael S. Paterson and Mark N. Wegman. Linear unification. *Journal of Computer and System Sciences*, 16:158–167, 1978.

Christine Paulin-Mohring. Extracting F_ω's programs from proofs in the calculus of constructions. In *ACM Symposium on Principles of Programming Languages (POPL), Austin, Texas*, pages 89–104, January 1989.

Leaf Petersen, Perry Cheng, Robert Harper, and Chris Stone. Implementing the TILT internal language. Technical Report CMU-CS-00-180, Department of Computer Science, Carnegie Mellon University, 2000.

Leaf Petersen, Robert Harper, Karl Crary, and Frank Pfenning. A type theory for memory allocation and data layout. In *ACM Symposium on Principles of Programming Languages (POPL), New Orleans, Louisiana*, pages 172–184, New Orleans, LA, January 2003.

Simon Peyton Jones. Special issue: Haskell 98 language and libraries. *Journal of Functional Programming*, 13, January 2003.

Frank Pfenning and Rowan Davies. A judgmental reconstruction of modal logic. *Mathematical Structures in Computer Science*, 11(4):511–540, 2001.

Frank Pfenning and Carsten Schürmann. Algorithms for equality and unification in the presence of notational definitions. In T. Altenkirch, W. Naraschewski, and B. Reus, editors, *Types for Proofs and Programs*, volume 1657 of *Lecture Notes in Computer Science*. Springer-Verlag, 1998.

Benjamin C. Pierce. *Types and Programming Languages*. MIT Press, 2002.

Benjamin C. Pierce and David N. Turner. Object-oriented programming without recursive types. In *ACM Symposium on Principles of Programming Languages (POPL), Charleston, South Carolina*, January 1993.

A. M. Pitts. Relational properties of domains. *Information and Computation*, 127:66–90, 1996.

A. M. Pitts. Existential types: Logical relations and operational equivalence. In K. G. Larsen, S. Skyum, and G. Winskel, editors, *International Colloquium on Automata, Languages and Programming*, volume 1443 of *Lecture Notes in Computer Science*, pages 309–326. Springer-Verlag, Berlin, 1998.

A. M. Pitts. Operational semantics and program equivalence. In G. Barthe, P. Dybjer, and J. Saraiva, editors, *Applied Semantics, Advanced Lectures*, volume 2395 of *Lecture Notes in Computer Science, Tutorial*, pages 378–412. Springer-Verlag, 2002. International Summer School, APPSEM 2000, Caminha, Portugal, September 9–15, 2000.

A. M. Pitts and I. D. B. Stark. Operational reasoning for functions with local state. In A. D. Gordon and A. M. Pitts, editors, *Higher Order Operational Techniques in Semantics*, Publications of the Newton Institute, pages 227–273. Cambridge University Press, 1998.

Andrew M. Pitts. Parametric polymorphism and operational equivalence. *Mathematical Structures in Computer Science*, 10:321–359, 2000.

Andrew M. Pitts and Ian D. B. Stark. Observable properties of higher order functions that dynamically create local names, or: What's new? In *Mathematical Foundations of Computer Science, Proc. 18th Int. Symp., Gdańsk, 1993*, volume 711 of *Lecture Notes in Computer Science*, pages 122–141. Springer-Verlag, Berlin, 1993.

Gordon Plotkin and Martín Abadi. A logic for parametric polymorphism. In M. Bezem and J. F. Groote, editors, *Typed Lambda Calculi and Applications (TLCA), Utrecht, The Netherlands*, number 664 in Lecture Notes in Computer Science, pages 361–375. Springer-Verlag, March 1993.

Gordon D. Plotkin. Lambda-definability and logical relations. Memorandum SAI–RM–4, University of Edinburgh, Edinburgh, Scotland, October 1973.

Gordon D. Plotkin. LCF considered as a programming language. *Theoretical Computer Science*, 5:223–255, 1977.

Gordon D. Plotkin. Lambda-definability in the full type hierarchy. In Jonathan P. Seldin and J. Roger Hindley, editors, *To H. B. Curry: Essays on Combinatory Logic, Lambda Calculus and Formalism*, pages 363–373. Academic Press, London, 1980.

Jeff Polakow and Frank Pfenning. Natural deduction for intuitionistic non-commutative linear logic. In *International Conference on Typed Lambda Calculi*, number 1581 in Lecture Notes in Computer Science, pages 295–309. Springer-Verlag, April 1999.

Erik Poll. Expansion Postponement for Normalising Pure Type Systems. *Journal of Functional Programming*, 8(1):89–96, 1998.

Robert Pollack. *The Theory of LEGO: A Proof Checker for the Extended Calculus of Constructions*. PhD thesis, University of Edinburgh, 1994.

Sally Popkorn. *First Steps in Modal Logic*. Cambridge University Press, 1994.

François Pottier. A versatile constraint-based type inference system. *Nordic Journal of Computing*, 7(4):312–347, November 2000.

François Pottier. A semi-syntactic soundness proof for HM(X). Research Report 4150, INRIA, March 2001a.

François Pottier. Simplifying subtyping constraints: a theory. *Information and Computation*, 170(2):153–183, November 2001b.

François Pottier. A constraint-based presentation and generalization of rows. In *IEEE Symposium on Logic in Computer Science (LICS)*, pages 331–340, June 2003.

François Pottier and Vincent Simonet. Information flow inference for ML. *ACM Transactions on Programming Languages and Systems*, 25(1):117–158, January 2003.

François Pottier, Christian Skalka, and Scott Smith. A systematic approach to static access control. In *European Symp. on Programming (ESOP)*, volume 2028 of *Lecture Notes in Computer Science*, pages 30–45. Springer-Verlag, April 2001.

Vaughan Pratt and Jerzy Tiuryn. Satisfiability of inequalities in a poset. *Fundamenta Informaticae*, 28(1–2):165–182, 1996.

William Pugh and Grant Weddell. Two-directional record layout for multiple inheritance. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 85–91, 1990.

Sriram K. Rajamani and Jakob Rehof. A behavioral module system for the pi-calculus. In *International Symposium on Static Analysis (SAS)*, volume 2126 of *Lecture Notes in Computer Science*, pages 375–394. Springer-Verlag, 2001.

Sriram K. Rajamani and Jakob Rehof. Conformance checking for models of asynchronous message passing software. In *International Conference on Computer Aided Verification*, pages 166–179, 2002.

Jakob Rehof. Minimal typings in atomic subtyping. In *ACM Symposium on Principles of Programming Languages (POPL), Paris, France*, pages 278–291, January 1997.

Jakob Rehof and Manuel Fähndrich. Type-based flow analysis: From polymorphic subtyping to CFL reachability. In *ACM Symposium on Principles of Programming Languages (POPL), London, England*, pages 54–66, 2001.

Alastair Reid, Matthew Flatt, Leigh Stoller, Jay Lepreau, and Eric Eide. Knit: Component composition for systems software. In *Proc. of the 4th Operating Systems Design and Implementation (OSDI)*, pages 347–360, October 2000. URL `citeseer.nj.nec.com/reid00knit.html`.

Didier Rémy. *Algèbres Touffues. Application au Typage Polymorphe des Objets Enregistrements dans les Langages Fonctionnels*. PhD thesis, Université Paris VII, 1990.

Didier Rémy. Extending ML type system with a sorted equational theory. Research Report 1766, Institut National de Recherche en Informatique et Automatisme, Rocquencourt, BP 105, 78 153 Le Chesnay Cedex, France, 1992a.

Didier Rémy. Projective ML. In *ACM Symposium on Lisp and Functional Programming (LFP)*, pages 66–75, 1992b.

Didier Rémy. Syntactic theories and the algebra of record terms. Research Report 1869, Institut National de Recherche en Informatique et Automatisme, Rocquencourt, BP 105, 78 153 Le Chesnay Cedex, France, 1993a.

Didier Rémy. Type inference for records in a natural extension of ML. In Carl A. Gunter and John C. Mitchell, editors, *Theoretical Aspects Of Object-Oriented Programming. Types, Semantics and Language Design*. MIT Press, 1993b.

Didier Rémy. Programming objects with ML-ART: An extension to ML with abstract and record types. In Masami Hagiya and John C. Mitchell, editors, *International Symposium on Theoretical Aspects of Computer Software (TACS)*, pages 321–346, Sendai, Japan, April 1994. Springer-Verlag.

Didier Rémy and Jérôme Vouillon. Objective ML: An effective object-oriented extension to ML. *Theory And Practice of Object Systems*, 4(1):27–50, 1998. Summary in *ACM Symposium on Principles of Programming Languages (POPL), Paris, France*, 1997.

Greg Restall. *An introduction to substructural logics*. Routledge, January 2000.

Greg Restall. Handbook of the history and philosophy of logic. To appear., 2001.

John C. Reynolds. Automatic computation of data set definitions. In A. J. H. Morrell, editor, *Information Processing 68*, volume 1, pages 456–461, Edinburgh, Scotland, 1969. North Holland.

John C. Reynolds. Towards a theory of type structure. In *Proc. Colloque sur la Programmation*, pages 408–425, New York, 1974. Springer-Verlag LNCS 19.

John C. Reynolds. Syntactic control of interference. In *ACM Symposium on Principles of Programming Languages (POPL), Tucson, Arizona*, pages 39–46, 1978. Reprinted in O'Hearn and Tennent, *ALGOL-like Languages*, vol. 1, pages 273–286, Birkhäuser, 1997.

John C. Reynolds. Types, abstraction, and parametric polymorphism. In R. E. A. Mason, editor, *Information Processing 83*, pages 513–523, Amsterdam, 1983. Elsevier Science Publishers B. V. (North-Holland).

John C. Reynolds. Syntactic control of interference, part 2. Report CMU-CS-89-130, Carnegie Mellon University, April 1989.

John C. Reynolds. Intuitionistic reasoning about shared mutable data structure. In *Millennial perspectives in computer science*, Palgrove, 2000.

Didier Rémy. Type checking records and variants in a natural extension of ML. In *ACM Symposium on Principles of Programming Languages (POPL), Austin, Texas*, pages 77–88, 1989.

J. Alan Robinson. Computational logic: The unification computation. *Machine Intelligence*, 6:63–72, 1971.

Douglas T. Ross. The AED free storage package. *Communications of the ACM*, 10(8): 481–492, 1967.

Claudio V. Russo. *Types for Modules*. PhD thesis, Edinburgh University, Edinburgh, Scotland, 1998. LFCS Thesis ECS–LFCS–98–389.

Claudio V. Russo. Non-dependent types for standard ML modules. In *Principles and Practice of Declarative Programming*, pages 80–97, 1999. URL `citeseer.nj.nec.com/russo99nondependent.html`.

Claudio V. Russo. Recursive structures for Standard ML. In *International Conference on Functional Programming (ICFP), Firenze, Italy*, pages 50–61, September 2001.

Amr Sabry. What is a purely functional language? *Journal of Functional Programming*, 8(1):1–22, January 1998.

Bratin Saha, Nevin Heintze, and Dino Oliva. Subtransitive CFA using types. Technical Report YALEU/DCS/TR-1166, Yale University, Department of Computer Science, October 1998.

D. Sangiorgi and D. Walker. *The π-calculus: a Theory of Mobile Processes*. Cambridge University Press, 2001.

Donald Sannella, Stefan Sokolowski, and Andrzej Tarlecki. Toward formal development of programs from algebraic specifications: Parameterisation revisited. *Acta Informatica*, 29(8):689–736, 1992.

Fred B. Schneider. Enforceable security policies. *ACM Transactions on Information and System Security*, 3(1):30–50, February 2000.

Jacob T. Schwartz. Optimization of very high level languages (parts I and II). *Computer Languages*, 1(2 & 3):161–194, 197–218, 1975.

Jonathan Seldin. Curry's anticipation of the types used in programming languages. Presented at Ann. Meet. Can. Soc. Hist. and Phil. Math, Toronto. Available from Seldin's homepage, 2002.

Miley Semmelroth and Amr Sabry. Monadic encapsulation in ML. In *International Conference on Functional Programming (ICFP), Paris, France*, pages 8–17, 1999. URL `http://www.cs.indiana.edu/~sabry/papers/ml-encap.ps`.

Peter Sestoft. Replacing function parameters by global variables. Technical Report 88-7-2, DIKU, University of Copenhagen, October 1988. SE88.

Peter Sestoft. Replacing function parameters by global variables. In *Proc. Functional Programming Languages and Computer Architecture (FPCA), London, England*, pages 39–53. ACM Press, September 1989.

Peter Sestoft. Moscow ML, 2003. URL `http://www.dina.dk/~sestoft/mosml.html`.

Paula Severi and Erik Poll. Pure type systems with definitions. In *Proceedings of Logical Foundations of Computer Science (LFCS)*, pages 316–328. Springer-Verlag, 1994. LNCS volume 813.

Zhong Shao. An overview of the FLINT/ML compiler. In *Proc. 1997 ACM SIGPLAN Workshop on Types in Compilation (TIC'97)*, Amsterdam, The Netherlands, June 1997.

Zhong Shao. Typed cross-module compilation. In *International Conference on Functional Programming (ICFP), Baltimore, Maryland, USA*, pages 141–152, September 1998.

Zhong Shao. Transparent modules with fully syntactic signatures. In *Proc. 1999 ACM SIGPLAN International Conference on Functional Programming (ICFP'99)*. ACM Press, September 1999.

Zhong Shao, Christopher League, and Stefan Monnier. Implementing typed intermediate languages. In *Proceedings of the 1998 ACM SIGPLAN International Conference on Functional Programming*, pages 313–323, Baltimore, MD, September 1998. ACM SIGPLAN.

Olin Shivers. Control flow analysis in Scheme. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI), Atlanta, Georgia*, pages 164–174, 1988.

Olin Shivers. *Control-Flow Analysis of Higher-Order Languages or Taming Lambda*. PhD thesis, Carnegie Mellon University, May 1991.

Vincent Simonet. Type inference with structural subtyping: a faithful formalization of an efficient constraint solver. In *Asian Symposium on Programming Languages and Systems*, November 2003.

Christian Skalka and François Pottier. Syntactic type soundness for $\mathsf{HM}(\mathsf{X})$. In *Workshop on Types in Programming (TIP'02)*, volume 75 of *Electronic Notes in Theoretical Computer Science*, July 2002.

Frederick Smith, David Walker, and Greg Morrisett. Alias types. In Gert Smolka, editor, *Ninth European Symposium on Programming*, volume 1782 of *Lecture Notes in Computer Science*, pages 366–381. Springer-Verlag, April 2000.

Geoffrey S. Smith. Principal type schemes for functional programs with overloading and subtyping. *Science of Computer Programming*, 23(2–3):197–226, December 1994.

Jan Smith, Bengt Nordström, and Kent Petersson. *Programming in Martin-Löf's Type Theory. An Introduction*. Oxford University Press, 1990.

Richard Statman. Logical relations and the typed λ-calculus. *Information and Control*, 65(2–3):85–97, May–June 1985.

Guy L. Steele, Jr. *Common Lisp: The Language*. Digital Press, 1990.

Christopher A. Stone. *Singleton Kinds and Singleton Types*. PhD thesis, Carnegie Mellon University, Pittsburgh, PA, August 2000.

Christopher A. Stone and Robert Harper. Deciding type equivalence in a language with singleton kinds. In *ACM Symposium on Principles of Programming Languages (POPL), Boston, Massachusetts*, pages 214–227, January 19–21, 2000.

Thomas Streicher. *Semantics of Type Theory*. Springer-Verlag, 1991.

Zhendong Su, Alexander Aiken, Joachim Niehren, Tim Priesnitz, and Ralf Treinen. The first-order theory of subtyping constraints. In *ACM Symposium on Principles of Programming Languages (POPL), Portland, Oregon*, pages 203–216, January 2002.

Martin Sulzmann. *A general framework for Hindley/Milner type systems with constraints*. PhD thesis, Yale University, Department of Computer Science, May 2000.

Martin Sulzmann, Martin Müller, and Christoph Zenger. Hindley/Milner style type systems in constraint form. Research Report ACRC–99–009, University of South Australia, School of Computer and Information Science, July 1999.

Sun. *Java™ 2 Platform Micro Edition (J2ME™) Technology for Creating Mobile Devices – White Paper*. Sun Microsystems, May 2000. http://java.sun.com/products/kvm/wp/KVMwp.pdf.

William W. Tait. Intensional interpretations of functionals of finite type I. *Journal of Symbolic Logic*, 32(2):198–212, June 1967.

C. Talcott. Reasoning about functions with effects. In A. D. Gordon and A. M. Pitts, editors, *Higher Order Operational Techniques in Semantics*, Publications of the Newton Institute, pages 347–390. Cambridge University Press, 1998.

Jean-Pierre Talpin and Pierre Jouvelot. Polymorphic type, region and effect inference. *Journal of Functional Programming*, 2(2):245–271, 1992.

Jean-Pierre Talpin and Pierre Jouvelot. The type and effect discipline. *Information and Computation*, 111:245–296, 1994.

David Tarditi, Greg Morrisett, Perry Cheng, Christopher Stone, Robert Harper, and Peter Lee. TIL : A type-directed optimizing compiler for ML. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI), Philadephia, Pennsylvania*, pages 181–192, May 21–24 1996.

Robert Endre Tarjan. Efficiency of a good but not linear set union algorithm. *Journal of the ACM*, 22(2):215–225, April 1975.

Robert Endre Tarjan. Applications of path compression on balanced trees. *Journal of the ACM*, 26(4):690–715, October 1979.

J. Terlouw. Een nadere bewijstheoretische analyse van GSTTs. Manuscript, University of Nijmegen, Netherlands, 1989.

Kresten Krab Thorup. Genericity in Java with virtual types. In *European Conference on Object-Oriented Programming (ECOOP)*, volume 1241 of *Lecture Notes in Computer Science*, pages 444–471, Jyväskylä, Finland, June 1997. Springer-Verlag.

Jerzy Tiuryn. Subtype inequalities. In *IEEE Symposium on Logic in Computer Science (LICS)*, pages 308–317, June 1992.

Jerzy Tiuryn and Mitchell Wand. Type reconstruction with recursive types and atomic subtyping. In *Proceedings of TAPSOFT '93*, volume 668 of *Lecture Notes in Computer Science*, pages 686–701. Springer-Verlag, April 1993.

Tofte and Birkedal. A region inference algorithm. *ACMTOPLAS: ACM Transactions on Programming Languages and Systems*, 20, 1998.

Mads Tofte. *Operational Semantics and Polymorphic Type Inference*. PhD thesis, Computer Science Department, Edinburgh University, 1988. CST-52-88.

Mads Tofte, Lars Birkedal, Martin Elsman, and Niels Hallenberg. Region-based memory management in perspective. In *International Conference on Principles and Practice of Declarative Programming*, pages 175–186, 2001a.

Mads Tofte, Lars Birkedal, Martin Elsman, Niels Hallenberg, Tommy Højfeld Olesen, and Peter Sestoft. Programming with regions in the ML Kit (for version 4). Technical report, IT University of Copenhagen, October 2001b.

Mads Tofte and Jean-Pierre Talpin. Implementing the call-by-value lambda-calculus using a stack of regions. In *ACM Symposium on Principles of Programming Languages (POPL), Portland, Oregon*, January 1994.

Mads Tofte and Jean-Pierre Talpin. Region-based memory management. *Information and Computation*, 132(2):109–176, 1 February 1997.

Mads Torgersen. Virtual types are statically safe. In *Proceedings of the 5th Workshop on Foundations of Object-Oriented Languages (FOOL)*, San Diego, CA, January 1998.

Valery Trifonov and Scott Smith. Subtyping constrained types. In *Proceedings of the Third International Static Analysis Symposium*, volume 1145 of *LNCS*, pages 349–365. Springer-Verlag, September 1996.

David N. Turner and Philip Wadler. Operational interpretations of linear logic. *Theoretical Computer Science*, 227:231–248, 1999. Special issue on linear logic.

David N. Turner, Philip Wadler, and Christian Mossin. Once upon a type. In *ACM Symposium on Functional Programming Languages and Computer Architecture (FPCA)*, San Diego, California, 1995.

Diederik T. van Daalen. *The Language Theory of Automath*. PhD thesis, Technische Hogeschool Eindhoven, 1980.

Robbert van Renesse, Kenneth P. Birman, Mark Hayden, Alexey Vaysburd, and David Karr. Building adaptive systems using Ensemble. *Software: Practice and Experience*, 28(9):963–979, August 1998.

J. Vouillon and A. Melliès. Semantic types: A fresh look at the ideal model for types. In *ACM Symposium on Principles of Programming Languages (POPL), Venice, Italy*, pages 52–63. ACM Press, 2004.

Philip Wadler. Theorems for free! In *Functional Programming Languages and Computer Architecture*, pages 347–359. ACM Press, September 1989. Imperial College, London.

Philip Wadler. Linear types can change the world. In *TC 2 Working Conference on Programming Concepts and Methods (Preprint)*, pages 546–566, 1990.

Philip Wadler. The marriage of effects and monads. *ACM Transactions on Computational Logic*, pages 1–31, 2003.

R. Wahbe, S. Lucco, T. E. Anderson, and S. L. Graham. Efficient software-based fault isolation. In *14th ACM Symposium on Operating Systems Principles*, pages 203–216. ACM, December 1993.

David Walker, Karl Crary, and Greg Morrisett. Typed memory management via static capabilities. *ACM Transactions on Programming Languages and Systems*, 22(4):701–771, July 2000. URL `http://www.cs.princeton.edu/~dpw/capabilities-toplas.pdf`.

David Walker and Greg Morrisett. Alias types for recursive data structures. *Lecture Notes in Computer Science*, 2071:177–206, 2001.

David Walker and Kevin Watkins. On regions and linear types. In *International Conference on Functional Programming (ICFP), Firenze, Italy*, pages 181–192, 2001. URL `http://www.cs.princeton.edu/~dpw/papers/lr.pdf`.

M. Wand. A simple algorithm and proof for type inference. *Fundamenta Informaticae*, X:115–122, 1987a.

Mitchell Wand. Complete type inference for simple objects. In *Proceedings of the IEEE Symposium on Logic in Computer Science*, Ithaca, NY, June 1987b.

Mitchell Wand. Corrigendum: Complete type inference for simple objects. In *Proceedings of the IEEE Symposium on Logic in Computer Science*, 1988.

Mitchell Wand. Type inference for objects with instance variables and inheritance. In Carl A. Gunter and John C. Mitchell, editors, *Theoretical Aspects of Object-Oriented Programming: Types, Semantics, and Language Design*, pages 97–120. MIT Press, 1994.

Daniel C. Wang and Andrew W. Appel. Type-preserving garbage collectors. In *ACM Symposium on Principles of Programming Languages (POPL), London, England*, pages 166–178, January 2001.

Keith Wansbrough and Simon Peyton Jones. Once upon a polymorphic type. In *ACM Symposium on Principles of Programming Languages (POPL), San Antonio, Texas*, pages 15–28, San Antonio, January 1999.

J. B. Wells. Typability and type checking in system F are equivalent and undecidable. *Annals of Pure and Applied Logic*, 98(1–3):111–156, 1999.

J. B. Wells. The essence of principal typings. In *International Colloquium on Automata, Languages and Programming*, volume 2380 of *Lecture Notes in Computer Science*, pages 913–925. Springer-Verlag, 2002.

Benjamin Werner. *Une Théorie des Constructions Inductives*. PhD thesis, Université Paris 7, May 1994.

Niklaus Wirth. *Systematic Programming: An Introduction*. Prentice Hall, 1973.

Niklaus Wirth. *Programming in Modula-2*. Texts and Monographs in Computer Science. Springer-Verlag, 1983.

A. K. Wright and R. Cartwright. A practical soft type system for scheme. In *Proceedings of the 1994 ACM Conference on Lisp and Functional Programming*. ACM, June 1994. Also available as LISP Pointers VII(3) July-September 1994.

Andrew K. Wright. Simple imperative polymorphism. *Lisp and Symbolic Computation*, 8(4):343–355, 1995.

Andrew K. Wright and Matthias Felleisen. A syntactic approach to type soundness. *Information and Computation*, 115(1):38–94, 15 November 1994.

Hongwei Xi. *Dependent Types in Practical Programming*. PhD thesis, Carnegie Mellon University, 1998.

Hongwei Xi and Robert Harper. A dependently typed assembly language. In *International Conference on Functional Programming (ICFP), Firenze, Italy*, 2001.

Hongwei Xi and Frank Pfenning. Dependent types in practical programming. In *ACM Symposium on Principles of Programming Languages (POPL), San Antonio, Texas*, ACM SIGPLAN Notices, pages 214–227, 1999.

Christoph Zenger. Indexed types. *Theoretical Computer Science*, 187:147–165, 1997.

Christoph Zenger. *Indizierte Typen*. PhD thesis, Universität Karlsruhe, July 1998.

Jan Zwanenburg. Pure type systems with subtyping. In J.-Y. Girard, editor, *Typed Lambda Calculus and Applications (TLCA)*, pages 381–396. Springer-Verlag, 1999. Lecture Notes in Computer Science, volume 1581.