Type Systems for Programming Languages[1] (DRAFT)

Robert Harper
School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213-3891

E-mail: `rwh@cs.cmu.edu`
WWW: `http://www.cs.cmu.edu/People/rwh`

Fall, 1997

[1] These are course notes Computer Science 15–814 at Carnegie Mellon University. This is an incomplete, working draft, not intended for publication. Citations to the literature are spotty at best; no results presented here should be considered original unless explicitly stated otherwise. *Please do not distribute these notes without the permission of the author.*

Contents

Foreword

These notes were prepared for use in the graduate course Computer Science 15–814: *Type Systems for Programming Languages* at Carnegie Mellon University. Their purpose is to provide a unified account of the role of type theory in programming language design and implementation. The stress is on the use of types as a tool for analyzing programming language features and studying their implementation.

A number of excellent books and articles are available as background reading for this course. Of particular relevance are *Proofs and Types* by Jean-Yves Girard [?], *Intuitionistic Type Theory* by Per Martin-Löf [?], *Semantics of Programming Languages* by Carl Gunter [?], and *The Formal Semantics of Programming Languages* by Glynn Winskel [?]. Other sources are mentioned at the end of each chapter, but no attempt is made to provide a comprehensive list of sources.

This is the first draft of a planned text on type theory for programming languages. The author welcomes suggestions and corrections. Please direct correspondence to `rwh@cs.cmu.edu`.

The author is grateful to those who have contributed to these notes by making comments, suggestions, or corrections: Lars Birkedal, Perry Cheng, Herb Derby, Jürgen Dingel, Matthias Felleisen, Andrzej Filinski, Rajit Manohar, Greg Morrisett, Chris Stone.

Part I

Type Structure

Chapter 1

Basic Types

1.1 Introduction

We begin with an extremely simple language of arithmetic and logical expressions to illustrate the main features of the type-theoretic approach to programming language definition. In subsequent chapters we will extend this language with additional types corresponding to more interesting language features.

1.2 Statics

The abstract syntax of $\mathcal{L}^{\mathtt{Int},\mathtt{Bool}}$ is defined inductively as follows:

$$
\begin{array}{lcll}
\textit{Types} & \tau & ::= & \mathtt{Int} \mid \mathtt{Bool} \\
\textit{Expressions} & e & ::= & \overline{n} \mid \mathtt{true} \mid \mathtt{false} \mid e_1 + e_2 \mid e_1 - e_2 \mid e_1 = e_2 \mid \\
 & & & \mathtt{if}_\tau\, e_1 \,\mathtt{then}\, e_2 \,\mathtt{else}\, e_3 \,\mathtt{fi}
\end{array}
$$

A *typing judgement* for $\mathcal{L}^{\mathtt{Int},\mathtt{Bool}}$ has the form $e : \tau$. The rules for deriving typing judgements are as follows:

$$\overline{n} : \mathtt{Int} \quad (n \in \omega) \tag{T-NUM}$$

$$\mathtt{true} : \mathtt{Bool} \tag{T-TRUE}$$

$$\mathtt{false} : \mathtt{Bool} \tag{T-FALSE}$$

$$\frac{e_1 : \mathtt{Int} \qquad e_2 : \mathtt{Int}}{e_1 + e_2 : \mathtt{Int}} \tag{T-PLUS}$$

$$\frac{e_1 : \mathtt{Int} \qquad e_2 : \mathtt{Int}}{e_1 - e_2 : \mathtt{Int}} \tag{T-MINUS}$$

$$\frac{e_1 : \mathtt{Int} \qquad e_2 : \mathtt{Int}}{e_1 = e_2 : \mathtt{Bool}} \tag{T-EQL}$$

$$\frac{e_1 : \mathtt{Bool} \qquad e_2 : \tau \qquad e_3 : \tau}{\mathtt{if}_\tau\, e_1\, \mathtt{then}\, e_2\, \mathtt{else}\, e_3\, \mathtt{fi} : \tau} \tag{T-IF}$$

These rules are said to be *syntax-directed* because at most one rule applies to a given expression, and that rule is determined by the outermost form of that expression.

1.3 Dynamics

1.3.1 Contextual Semantics

A *contextual semantics* is given by a set of *values*, a set of *evaluation contexts*, and a *primitive reduction relation*. Values are "fully evaluated" expressions. Evaluation contexts determine the order of evaluation of sub-expressions of a non-value expression. They are defined as follows:

$$\begin{array}{llcl}
\textit{Values} & v & ::= & \overline{n} \mid \mathtt{true} \mid \mathtt{false} \\
\textit{EvaluationContexts} & E & ::= & \bullet \mid E+e \mid v+E \mid E-e \mid v-E \mid E=e \mid v=E \mid \\
 & & & \mathtt{if}_\tau\, E_1\, \mathtt{then}\, e_2\, \mathtt{else}\, e_3\, \mathtt{fi}
\end{array}$$

An evaluation context is an expression fragment with a designated "hole", written "$\bullet$". If E is an evaluation context and e is an expression, then $E[e]$ is the expression obtained by "filling the hole" in E with the expression e, *i.e.*the expression obtained by replacing the occurrence of $\bullet$ in E by e.

The primitive reduction relation defines the behavior of the primitive operations ("primop's") on values:

$$\begin{array}{rcl}
\overline{n}_1 + \overline{n}_2 & \leadsto & \overline{n_1 + n_2} \\
\overline{n}_1 - \overline{n}_2 & \leadsto & \overline{n_1 - n_2} \\
\overline{n}_1 = \overline{n}_2 & \leadsto & \begin{cases} \mathtt{true} & \textit{if } n_1 = n_2 \\ \mathtt{false} & \textit{otherwise} \end{cases} \\
\mathtt{if}_\tau\, \mathtt{true}\, \mathtt{then}\, e_1\, \mathtt{else}\, e_2\, \mathtt{fi} & \leadsto & e_1 \\
\mathtt{if}_\tau\, \mathtt{false}\, \mathtt{then}\, e_1\, \mathtt{else}\, e_2\, \mathtt{fi} & \leadsto & e_2
\end{array}$$

An expression e such that $e \leadsto e'$ for some e' is sometimes called a *redex*, in which case e' is called its *contractum*. We shall occasionally use the metavariable r to range over redices and c over their contracta.

The *one-step evaluation* relation $e \mapsto e'$ is defined to hold iff $e = E[r]$, $e' = E[c]$, and $r \leadsto c$. The *multi-step evaluation* relation $e \mapsto^* e'$ is the reflexive and transitive closure of the one-step evaluation relation. We say that the expression e *evaluates to* the value v iff $e \mapsto^* v$.

Lemma 1.1

1. *If v is a value, then $v = E[e]$ iff $E = \bullet$ and $e = v$.*

2. *If e is not a value, then there is at most one E such that $e = E[r]$ and r is a redex.*

Theorem 1.2 (Determinacy)
For any closed expression e there is at most one value v such that $e \mapsto^* v$.

Exercise 1.3
Structured operational semantics (SOS) *is an alternative method of defining the one-step evaluation in which the "search" for a redex and the actual reduction steps are defined simultaneously by a direct axiomatization of the one-step evaluation relation. The general pattern is illustrated by the following rules for addition:*

$$\frac{e_1 \mapsto e_1'}{e_1 + e_2 \mapsto e_1' + e_2} \qquad (\textsc{sos-add-l})$$

$$\frac{e_2 \mapsto e_2'}{v_1 + e_2 \mapsto v_1 + e_2'} \qquad (\textsc{sos-add-r})$$

$$\overline{n}_1 + \overline{n}_2 \mapsto \overline{n_1 + n_2} \qquad (\textsc{sos-add})$$

Give a complete definition of the one-step evaluation relation for $\mathcal{L}^{\mathtt{Int,Bool}}$ using structured operational semantics, and prove that this relation coincides with the contextual semantics.

1.3.2 Evaluation Semantics

An evaluation semantics is given by an inductive definition of the *evaluation* relation $e \Downarrow v$. The rules for $\mathcal{L}^{\mathtt{Int,Bool}}$ are as follows:

$$\overline{n} \Downarrow \overline{n} \qquad (\textsc{ns-num})$$

$$\mathtt{true} \Downarrow \mathtt{true} \qquad (\textsc{ns-true})$$

$$\mathtt{false} \Downarrow \mathtt{false} \qquad (\textsc{ns-false})$$

$$\frac{e_1 \Downarrow \overline{n}_1 \qquad e_2 \Downarrow \overline{n}_2}{e_1 + e_2 \Downarrow \overline{n_1 + n_2}} \qquad (\textsc{ns-plus})$$

$$\frac{e_1 \Downarrow \overline{n}_1 \qquad e_2 \Downarrow \overline{n}_2}{e_1 - e_2 \Downarrow \overline{n_1 - n_2}} \qquad \text{(NS-MINUS)}$$

$$\frac{e_1 \Downarrow \overline{n}_1 \qquad e_2 \Downarrow \overline{n}_2}{e_1 = e_2 \Downarrow \texttt{true}} \quad (n_1 = n_2) \qquad \text{(NS-EQL-1)}$$

$$\frac{e_1 \Downarrow \overline{n}_1 \qquad e_2 \Downarrow \overline{n}_2}{e_1 = e_2 \Downarrow \texttt{false}} \quad (n_1 \neq n_2) \qquad \text{(NS-EQL-2)}$$

$$\frac{e \Downarrow \texttt{true} \qquad e_1 \Downarrow v}{\texttt{if}_\tau\, e\, \texttt{then}\, e_1\, \texttt{else}\, e_2\, \texttt{fi} \Downarrow v} \qquad \text{(NS-IF-1)}$$

$$\frac{e \Downarrow \texttt{false} \qquad e_2 \Downarrow v}{\texttt{if}_\tau\, e\, \texttt{then}\, e_1\, \texttt{else}\, e_2\, \texttt{fi} \Downarrow v} \qquad \text{(NS-IF-2)}$$

Theorem 1.4 (Determinacy)
For any closed expression e there is at most one v such that $e \Downarrow v$.

The evaluation semantics is equivalent to the contextual semantics in the sense that $e \Downarrow v$ iff $e \mapsto^* v$.

Exercise 1.5
Prove the equivalence of the evaluation and contextual semantics for $\mathcal{L}^{\texttt{Int,Bool}}$ by establishing the following two claims:

1. *The relation $e \mapsto^* v$ is closed under the defining conditions of the $\Downarrow$ relation, the smallest relation closed under these conditions. Therefore, if $e \Downarrow v$, then $e \mapsto^* v$.*

2. *The $\Downarrow$ relation is closed under "head expansion". That is, if $e \Downarrow v$ and $e' \mapsto e$, then $e' \Downarrow v$.*

3. *If $e \mapsto^* v$, then $e \Downarrow v$.*

The proof of equivalence of the contextual and evaluation semantics simplifies the proofs of some useful properties of the dynamic semantics of $\mathcal{L}^{\texttt{Int,Bool}}$.

Exercise 1.6
Prove the following facts about the contextual semantics:

1. *If $e_1 + e_2 \mapsto^* v$, then $e_1 \mapsto^* v_1$ and $e_2 \mapsto^* v_2$ for some values v_1 and v_2. (A similar property holds of the other primitives of $\mathcal{L}^{\texttt{Int,Bool}}$.)*

2. *If $\texttt{if}_\tau\, e\, \texttt{then}\, e_1\, \texttt{else}\, e_2\, \texttt{fi} \mapsto^* v$, then $e \mapsto^* v'$ for some value v'.*

1.4 Type Soundness

Lemma 1.7 (Replacement)
If $E[e] : \tau$ then there exists τ_e such that $e : \tau_e$ and $E[e'] : \tau$ for every e' such that $e' : \tau_e$.

Proof: By induction on the derivation of $E[e] : \tau$, making use of the syntax-directed nature of the rules. ■

Lemma 1.8 (Subject Reduction)
If $e \leadsto e'$ and $e : \tau$, then $e' : \tau$.

Proof: By inspection of the primitive reduction steps. ■

Theorem 1.9 (Preservation)
If $e : \tau$ and $e \mapsto e'$, then $e' : \tau$.

Proof: By Lemmas 1.7 and 1.8. ■

Lemma 1.10 (Canonical Forms)
If v : `Int`, then $v = \overline{n}$ for some n. If v : `Bool`, then either $v =$ `true` or $v =$ `false`.

Proof: Inspection of the typing rules and the definition of values. ■

Theorem 1.11 (Progress)
If $e : \tau$, then either e is a value, or there exists e' such that $e \mapsto e'$.

Proof: By induction on typing derivations. Let us consider the case $e = e_1 + e_2$; the other cases follow by similar reasoning. Since $e : \tau$, it follows that $\tau = \mathtt{Int}$ and $e_1 : \mathtt{Int}$ and $e_2 : \mathtt{Int}$. We proceed by cases on the form of e_1 and e_2. If both are values, then by the canonical forms lemma $e_1 = \overline{n}_1$ and $e_2 = \overline{n}_2$ for some n_1 and n_2. Taking $E = \bullet$, and $e' = \overline{n_1 + n_2}$, it follows that $e = E[e] \mapsto E[e'] = e'$. If e_1 is a value and e_2 is not, then by the inductive hypothesis $e_2 \mapsto e_2'$ for some e_2', and hence $e_2 = E_2[r]$ and $e_2' = E_2[c]$. Taking $E = e_1 + E_2$, we observe that $e = E[r] \mapsto E[c] = e'$. If e_1 is not a value, then by inductive hypothesis there exists E_1, r, and c such that $e_1 = E_1[r] \mapsto E_1[c] = e_1'$. Taking $E = E_1 + e_2$, we observe that $e = E[r] \mapsto E[c] = e'$. ■

The progress theorem (and not the preservation theorem!) is the analogue of Milner's soundness theorem for ML, which states that "well-typed programs cannot go wrong". In the present setting soundness is expressed more accurately by "well-typed programs cannot get stuck" — evaluation cannot terminate in an expression other than a value.

It is difficult to state soundness for evaluation semantics. While it is straightforward to prove preservation in the form that if $e : \tau$ and $e \Downarrow v$, then $v : \tau$, it is harder to see what is the appropriate analogue of progress. The typical approach is to "instrument" the rules to account explicitly for type errors, then prove that these rules cannot apply in a well-typed program. This approach is somewhat unsatisfactory in that the instrumentation is *ad hoc*, and included only for the sake of a proof. We outline the method in the following exercise.

Exercise 1.12
In this exercise we explore the standard formalization of type soundness in the setting of evaluation semantics. We begin by introducing the notion of an answer*, the ultimate result of evaluation. For the present purposes answers may be defined as follows:*

$$\textit{Answers} \quad a \quad ::= \quad \mathsf{Ok}(v) \mid \mathsf{Wrong}$$

We define $a : \tau$ iff $a = \mathsf{Ok}(v)$ and $v : \tau$; thus Wrong *is ill-typed.*

1. *Re-write the evaluation semantics of $\mathcal{L}^{\texttt{Int,Bool}}$ as a relation $e \Downarrow a$, where a is an answer of the form $\mathsf{Ok}(v)$ for some value v.*

2. *Instrument the evaluation semantics of $\mathcal{L}^{\texttt{Int,Bool}}$ with rules $e \Downarrow \mathsf{Wrong}$ corresponding to "run-time type errors". For example,*

$$\frac{e_1 \Downarrow \mathsf{Ok}(\texttt{true})}{e_1 + e_2 \Downarrow \mathsf{Wrong}} \qquad (\textsc{ns-plus-wrong-left})$$

3. *Prove that if $e : \tau$ and $e \Downarrow a$, then $a : \tau$. How are rules such as* NS-PLUS-WRONG-LEFT *handled?*

Exercise 1.13
Consider adding integer division to $\mathcal{L}^{\texttt{Int,Bool}}$. Since division by zero is undefined, how will you define an operational semantics to handle this case? What happens to the progress and preservation theorems?

1.5 References

Mitchell's book [**?**] is a comprehensive introduction to type systems for programming languages, covering operational, denotational, and axiomatic aspects. Other basic references are Gunter's book [**?**] and Winskel's book [**?**].

Plotkin's Aarhus lecture notes on operational semantics [**?**] contain a thorough account of a wide range of programming language features using structured operational semantics. The use of evaluation semantics (under the name "natural semantics") was popularized by Gilles Kahn [**?**, **?**]. However, Martin-Löf made earlier use of this approach in his highly influential constructive type theory [**?**, **?**].

The definition of Standard ML constitutes an extensive experiment in programming language specification based on type theory and operational semantics [**?**, **?**].

Chapter 2

Function and Product Types

2.1 Introduction

In this chapter we consider the language $\mathcal{L}^{1,\times,\rightarrow}$ with product and function types.

2.2 Statics

The abstract syntax of $\mathcal{L}^{1,\times,\rightarrow}$ is given by the following grammar:

$$\begin{array}{lcll} \textit{Types} & \tau & ::= & \mathtt{Unit} \mid \tau_1{\times}\tau_2 \mid \tau_1{\rightarrow}\tau_2 \\ \textit{Expressions} & e & ::= & x \mid * \mid {<}e_1{,}e_2{>}_{\tau_1,\tau_2} \mid \mathtt{proj}^1_{\tau_1,\tau_2}(e) \mid \mathtt{proj}^2_{\tau_1,\tau_2}(e) \mid \\ & & & \mathtt{fun}\,(x{:}\tau_1){:}\tau_2\;\mathtt{in}\,e \mid \mathtt{app}_{\tau_1,\tau_2}(e_1{,}e_2) \end{array}$$

In the expression $\mathtt{fun}\,(x{:}\tau_1){:}\tau_2\;\mathtt{in}\,e$ the variable x is bound in e. Capture-avoiding substitution of an expression e for free occurrences of x in an expression e', written $\{e/x\}e'$, is defined as usual.

A *typing judgement* for $\mathcal{L}^{1,\times,\rightarrow}$ is a triple of the form $\Gamma \vdash e : \tau$, where Γ is a *type assignment*, a finite partial function mapping variables to types. We write $\Gamma[x:\tau]$, where $x \notin \mathrm{dom}(\Gamma)$, to denote the extension of Γ mapping x to τ and y to $\Gamma(y)$.

The rules for deriving typing judgements are as follows:

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

$$\Gamma \vdash * : \mathtt{Unit} \qquad \text{(T-UNIT)}$$

$$\frac{\Gamma \vdash e_1 : \tau_1 \qquad \Gamma \vdash e_2 : \tau_2}{\Gamma \vdash \texttt{<}e_1,e_2\texttt{>}_{\tau_1,\tau_2} : \tau_1 \times \tau_2} \tag{T-PAIR}$$

$$\frac{\Gamma \vdash e : \tau_1 \times \tau_2}{\Gamma \vdash \texttt{proj}^i_{\tau_1,\tau_2}(e) : \tau_i} \quad (i = 1, 2) \tag{T-PROJ}$$

$$\frac{\Gamma[x{:}\tau_1] \vdash e : \tau_2}{\Gamma \vdash \texttt{fun}\,(x{:}\tau_1){:}\tau_2\;\texttt{in}\,e : \tau_1 \rightarrow \tau_2} \quad (x \notin \mathrm{dom}(\Gamma)) \tag{T-ABS}$$

$$\frac{\Gamma \vdash e_1 : \tau_2 \rightarrow \tau \qquad \Gamma \vdash e_2 : \tau_2}{\Gamma \vdash \texttt{app}_{\tau_1,\tau_2}(e_1,e_2) : \tau} \tag{T-APP}$$

2.3 Dynamics

The contextual semantics for $\mathcal{L}^{1,\times,\rightarrow}$ is specified by the set of *values*, the set of *evaluation contexts*, and the set of *primitive reductions*.

Values and evaluation contexts are defined by the following grammar:

$$\begin{array}{llll}
\textit{Values} & v & ::= & x \mid * \mid \texttt{<}v_1,v_2\texttt{>}_{\tau_1,\tau_2} \mid \texttt{fun}\,(x{:}\tau_1){:}\tau_2\;\texttt{in}\,e \\
\textit{Eval Ctxts} & E & ::= & \bullet \mid \texttt{<}E,e\texttt{>}_{\tau_1,\tau_2} \mid \texttt{<}v,E\texttt{>}_{\tau_1,\tau_2} \mid \texttt{proj}^1_{\tau_1,\tau_2}(E) \mid \\
 & & & \texttt{proj}^2_{\tau_1,\tau_2}(E) \mid \texttt{app}_{\tau_1,\tau_2}(E,e) \mid \texttt{app}_{\tau_1,\tau_2}(v,E)
\end{array}$$

A *closed* value is a value with no free variables.

The primitive reduction relation, $e \leadsto e'$, for $\mathcal{L}^{1,\times,\rightarrow}$ is defined by the following conditions:

$$\begin{array}{rcll}
\texttt{proj}^i_{\tau_1,\tau_2}(\texttt{<}v_1,v_2\texttt{>}_{\tau_1,\tau_2}) & \leadsto & v_i & (i = 1, 2) \\
\texttt{app}_{\tau_1,\tau_2}((\texttt{fun}\,(x{:}\tau_1){:}\tau_2\;\texttt{in}\,e),v) & \leadsto & \{v/x\}e &
\end{array}$$

The *one-step evaluation* relation, $e \mapsto e'$, is defined to hold iff e and e' are closed expressions such that $e = E[e_1]$, $e' = E[e_2]$, and $e_1 \leadsto e_2$.

Remark 2.1
Two common variations on the semantics of $\mathcal{L}^{1,\times,\rightarrow}$ are commonly considered. In the *lazy pairing*[1] variant all pairs of the form $\texttt{<}e_1,e_2\texttt{>}_{\tau_1,\tau_2}$ are regarded as a value, without restriction on e_1 or e_2. In the *lazy evaluation*,[2] or *call-by-name*, variant the primitive reduction step for application is

$$\texttt{app}_{\tau_1,\tau_2}((\texttt{fun}\,(x{:}\tau_1){:}\tau_2\;\texttt{in}\,e),e') \leadsto \{e'/x\}e.$$

[1] This is a misnomer since it does not account for memoization.
[2] Also a misnomer.

Exercise 2.2
Formulate the dynamic semantics of $\mathcal{L}^{1,\times,\rightarrow}$ *using evaluation semantics, and prove that it is equivalent to the contextual semantics.*

Lemma 2.3

1. *If* v *is a value, then* $v = E[e]$ *iff* $E = \bullet$ *and* $e = v$.

2. *If* e *is not a value, then there is at most one* E *such that* $e = E[r]$ *and* r *is a redex.*

Theorem 2.4 (Determinacy)
For any closed expression e *there is at most one value* v *such that* $e \mapsto^* v$.

2.4 Type Soundness

Lemma 2.5 (Substitution)
If $\Gamma[x{:}\tau] \vdash e : \tau'$ *and* $\Gamma \vdash v : \tau$, *then* $\Gamma \vdash \{v/x\}e : \tau'$.

Proof: By induction on typing derivations. The limitation to values is not essential here, but is all that is needed for the development. ■

Lemma 2.6 (Replacement)
If $\vdash E[e] : \tau$ *then there exists* τ_e *such that* $\vdash e : \tau_e$ *and* $\vdash E[e'] : \tau$ *for every* e' *such that* $\vdash e' : \tau_e$.

Lemma 2.7 (Subject Reduction)
If $\vdash e : \tau$ *and* $e \leadsto e'$, *then* $\vdash e' : \tau$.

Theorem 2.8 (Preservation)
If $\vdash e : \tau$ *and* $e \mapsto e'$, *then* $\vdash e' : \tau$.

Proof: By Lemmas 2.5, 2.6, and 2.7. ■

Lemma 2.9 (Canonical Forms)
Suppose that $\vdash v : \tau$. *If* $\tau = \texttt{Unit}$, *then* $v = *$; *if* $\tau = \tau_1 \times \tau_2$, *then* $v = \texttt{<}v_1,v_2\texttt{>}_{\tau_1,\tau_2}$ *with* $\vdash v_i : \tau_i$ $(i = 1, 2)$; *if* $\tau = \tau_1 \rightarrow \tau_2$, *then* $v = \texttt{fun}\,(x{:}\tau_1){:}\tau_2\ \texttt{in}\,e$ *with* $x{:}\tau_1 \vdash e : \tau_2$.

Theorem 2.10 (Progress)
If $\vdash e : \tau$, *then either* e *is a value or there exists* e' *such that* $e \mapsto e'$.

Proof: We consider here the case that e has the form $\texttt{app}_{\tau_1,\tau_2}(e_1,e_2)$ for some closed expressions e_1 and e_2; the other cases follow a similar pattern. Since $\vdash e : \tau$, there exists τ_2 such that $\vdash e_1 : \tau_2 \rightarrow \tau$ and $\vdash e_2 : \tau_2$. By induction either e_1 is a value or there exists e_1' such that $e_1 \mapsto e_1'$, and similarly for e_2. Suppose that e_1 is a value v_1. Then by Lemma 2.9 $v_1 = \texttt{fun}\,(x{:}\tau_1){:}\tau_2\ \texttt{in}\,e$ for some x and e. If e_2

is a value v_2, then $\mathtt{app}_{\tau_1,\tau_2}(e_1,e_2) = \mathtt{app}_{\tau_1,\tau_2}((\mathtt{fun}\,(x{:}\tau_1){:}\tau_2\ \mathtt{in}\,e),v_2) \mapsto \{v_2/x\}e$. If e_2 is not a value, then $e = \mathtt{app}_{\tau_1,\tau_2}(v_1,e_2) \mapsto \mathtt{app}_{\tau_1,\tau_2}(v_1,e_2')$. Finally, if e_1 is not a value, then $e = \mathtt{app}_{\tau_1,\tau_2}(e_1,e_2) \mapsto \mathtt{app}_{\tau_1,\tau_2}(e_1',e_2)$. ■

2.5 Termination

While it may seem intuitively obvious, it is not at all straightforward to prove that all computations in $\mathcal{L}^{1,\times,\to}$ terminate.

Exercise 2.11
Try to prove directly by induction on the structure of typing derivations that if $\vdash e : \tau$, then there exists v such that $e \mapsto^* v$. Where do you run into problems?

The termination property of $\mathcal{L}^{1,\times,\to}$ may be proved using Tait's *computability method* in which types are interpreted as predicates.[3]

Definition 2.12

1. *The predicate $Comp_\tau(e)$ holds iff there exists v such that $e \Downarrow v$ and $Val_\tau(v)$.*

2. *The predicate $Val_\tau(v)$ is defined by induction on the structure of τ as follows:*

   (a) *If $\tau = \mathtt{Unit}$, then $Val_\tau(v)$ iff $v = *$.*

   (b) *If $\tau = \tau_1{\times}\tau_2$, then $Val_\tau(v)$ iff $v = \lt v_1,v_2\gt_{\tau_1,\tau_2}$ with $Val_{\tau_1}(v_1)$ and $Val_{\tau_2}(v_2)$.*

   (c) *If $\tau = \tau_1{\to}\tau_2$, then $Val_\tau(v)$ iff $v = \mathtt{fun}\,(x{:}\tau_1){:}\tau_2\ \mathtt{in}\,e$ and $Comp_{\tau_2}(\{v_1/x\}e)$ for every v_1 such that $Val_{\tau_1}(v_1)$.*

The predicate $Comp_\tau(-)$ may be thought of as describing the "computations" of type τ, and the predicate $Val_\tau(-)$ as describing "values" of type τ. It is immediately obvious that the predicate $Comp_\tau(-)$ is closed under head expansion in the sense that if $Comp_\tau(e)$ and $e' \mapsto e$, then $Comp_\tau(e')$. It is also obvious that if $Val_\tau(v)$, then $Comp_\tau(v)$ since $v \Downarrow v$.

The value predicate is extended to type assignments as follows. An *environment*, γ, is a finite function assigning closed values to variables. The substitution induced by an environment γ is written $\hat{\gamma}$, and we define $\hat{\gamma}(e)$ by induction on the structure of e in the obvious way. The predicate $Val_\Gamma(\gamma)$ holds iff $\mathrm{dom}(\gamma) = \mathrm{dom}(\Gamma)$ and $Val_{\Gamma(x)}(\gamma(x))$ holds for every $x \in \mathrm{dom}(\Gamma)$.

Theorem 2.13
If $\Gamma \vdash e : \tau$ and $Val_\Gamma(\gamma)$, then $Comp_\tau(\hat{\gamma}(e))$.

[3] Don't make too much of the term "computability" here. It's chosen for historical, rather than descriptive, reasons.

Proof: By induction on typing derivations. We consider some illustrative cases.

t-var Follows directly from the assumption that $\mathit{Val}_\Gamma(\gamma)$.

t-app By induction we have that $\mathit{Comp}_{\tau_2\to\tau}(\hat{\gamma}(e_1))$ and $\mathit{Comp}_{\tau_2}(\hat{\gamma}(e_2))$, and we are to show that $\mathit{Comp}_{\tau}(\hat{\gamma}(\mathtt{app}_{\tau_1,\tau_2}(e_1,e_2)))$. It follows from the definitions that $\hat{\gamma}(e_1) \Downarrow \mathtt{fun}\,(x{:}\tau_2){:}\tau\;\mathtt{in}\,e$, $\hat{\gamma}(e_2) \Downarrow v_2$, and $\mathit{Comp}_{\tau}(\{v_2/x\}e)$, which together suffice for the result.

t-abs We are to show that $\mathit{Comp}_{\tau_1\to\tau_2}(\hat{\gamma}(\mathtt{fun}\,(x{:}\tau_1){:}\tau_2\;\mathtt{in}\,e))$. It suffices to show that $\mathit{Comp}_{\tau_2}(\{v_1/x\}\hat{\gamma}(e))$ for any v_1 such that $\mathit{Val}_{\tau_1}(v_1)$. But $\{v_1/x\}\hat{\gamma}(e) = \widehat{\gamma[x \mapsto v_1]}(e)$, and clearly $\mathit{Val}_{\Gamma[x:\tau_1]}(\gamma[x \mapsto v_1])$. So the result follows by the inductive hypothesis.

■

Corollary 2.14 (Termination)
If $\vdash e : \tau$, then $e \Downarrow v$ for some value v.

Exercise 2.15
Complete the proof of Theorem 2.13.

Remark 2.16
In this chapter we have used what might be called a "fully explicit" syntax in which all primitive expression constructors are labelled with sufficient type information to determine precisely the type of the constructed expression. In practice we omit this information when it is clear from context. For example, we often write $e_1(e_2)$ for $\mathtt{app}_{\tau_1,\tau_2}(e_1,e_2)$ when τ_1 and τ_2 may be determined from e_1 and e_2. Similarly we write $<e_1,e_2>$ for $<e_1,e_2>_{\tau_1,\tau_2}$ and $\mathtt{fun}\,(x{:}\tau)\;\mathtt{in}\,e$ or just $\mathtt{fun}\,x\;\mathtt{in}\,e$ for $\mathtt{fun}\,(x{:}\tau_1){:}\tau_2\;\mathtt{in}\,e$ when the omitted types are clear from context. These informal conventions will be made precise in Chapter 19 when we discuss *type reconstruction*, the process of determining the omitted type information from context whenever it can be uniquely determined.

2.6 References

A good general introduction to the typed λ-calculus is Girard's monograph [?].

The method of logical relations is fundamental to the study of the typed λ-calculus, as emphasized by Friedman [?], Statman [?], and Plotkin [?].

The metaphor of "computations" (versus "values") was made explicit by Constable and Smith in their partial object type theory [?, ?] and by Moggi in his monadic treatment of computational effects [?].

Chapter 3

Sums

3.1 Introduction

In this chapter we briefly summarize the extension of $\mathcal{L}^{1,\times,\rightarrow}$ with *sum* types, otherwise known as *disjoint union* types. This addition presents few complications beyond those considered in Chapter 2.

3.2 Sum Types

The syntax of $\mathcal{L}^{0,1,+,\times,\rightarrow}$ is defined by the following extension to the syntax of $\mathcal{L}^{1,\times,\rightarrow}$:

$$\begin{array}{lrcl} \textit{Types} & \tau & ::= & \ldots \mid \texttt{Void} \mid \tau_1 + \tau_2 \\ \textit{Expressions} & e & ::= & \texttt{any}_\tau(e) \mid \texttt{inl}_{\tau_1,\tau_2}(e) \mid \texttt{inr}_{\tau_1,\tau_2}(e) \mid \\ & & & \texttt{case}_\tau\, e\, \texttt{of}\, \texttt{inl}(x_1{:}\tau_1) \texttt{=>} e_1 \mid \texttt{inr}(x_2{:}\tau_2) \texttt{=>} e_2\, \texttt{esac} \end{array}$$

Informally, `Void` is the empty type (with no members) and $\tau_1+\tau_2$ is the disjoint union of the types τ_1 and τ_2. In the expression

$$\texttt{case}_\tau\, e\, \texttt{of}\, \texttt{inl}(x_1{:}\tau_1) \texttt{=>} e_1 \mid \texttt{inr}(x_2{:}\tau_2) \texttt{=>} e_2\, \texttt{esac}$$

the variable x_1 is bound in e_1 and the variable x_2 is bound in e_2.

The static semantics of these additional constructs is given by the following rules:

$$\frac{\Gamma \vdash e : \texttt{Void}}{\Gamma \vdash \texttt{any}_\tau(e) : \tau} \qquad (\text{T-ANY})$$

$$\frac{\Gamma \vdash e : \tau_1}{\Gamma \vdash \texttt{inl}_{\tau_1,\tau_2}(e) : \tau_1+\tau_2} \qquad (\text{T-INL})$$

$$\frac{\Gamma \vdash e : \tau_2}{\Gamma \vdash \texttt{inr}_{\tau_1,\tau_2}(e) : \tau_1+\tau_2} \qquad (\text{T-INR})$$

$$\frac{\Gamma \vdash e : \tau_1{+}\tau_2 \qquad \Gamma[x_1{:}\tau_1] \vdash e_1 : \tau \qquad \Gamma[x_2{:}\tau_2] \vdash e_2 : \tau}{\Gamma \vdash \mathtt{case}_\tau\, e\, \mathtt{of}\; \mathtt{inl}(x_1{:}\tau_1) \mathtt{=>} e_1 \mid \mathtt{inr}(x_2{:}\tau_2) \mathtt{=>} e_2\; \mathtt{esac} : \tau}\quad (x_1, x_2 \notin \mathrm{dom}(\Gamma))$$

(T-CASE)

The dynamic semantics is given by the following data. First, the languages of values and evaluation contexts are enriched as follows:

$$\begin{array}{llll} \textit{Values} & v & ::= & \ldots \mid \mathtt{inl}_{\tau_1,\tau_2}(v) \mid \mathtt{inr}_{\tau_1,\tau_2}(v) \\ \textit{EvalCtxts} & E & ::= & \ldots \mid \mathtt{any}_\tau(E) \mid \mathtt{case}_\tau\, E\, \mathtt{of}\; \mathtt{inl}(x_1{:}\tau_1) \mathtt{=>} e_1 \mid \mathtt{inr}(x_2{:}\tau_2) \mathtt{=>} e_2\; \mathtt{esac} \end{array}$$

The primitive instructions are as follows:

$$\begin{array}{lll} \mathtt{case}_\tau\; \mathtt{inl}_{\tau_1,\tau_2}(v_1)\; \mathtt{of}\; \mathtt{inl}(x_1{:}\tau_1) \mathtt{=>} e_1 \mid \mathtt{inr}(x_2{:}\tau_2) \mathtt{=>} e_2\; \mathtt{esac} & \mapsto & \{v_1/x_1\}e_1 \\ \mathtt{case}_\tau\; \mathtt{inr}_{\tau_1,\tau_2}(v_2)\; \mathtt{of}\; \mathtt{inl}(x_1{:}\tau_1) \mathtt{=>} e_1 \mid \mathtt{inr}(x_2{:}\tau_2) \mathtt{=>} e_2\; \mathtt{esac} & \mapsto & \{v_2/x_2\}e_2 \end{array}$$

Exercise 3.1
State and prove soundness for $\mathcal{L}^{0,1,+,\times,\rightarrow}$.

Exercise 3.2
Extend the interpretation of types as predicates given in Chapter 2 to establish termination for $\mathcal{L}^{0,1,+,\times,\rightarrow}$.

Exercise 3.3
Show that the type `Bool` *is definable in the language* $\mathcal{L}^{0,1,+,\times,\rightarrow}$ *by taking* `Bool` *to be the type* `Unit+Unit`. *What are the interpretations of* `true` *and* `false`*? What is the interpretation of* $\mathtt{if}_\tau\, e\, \mathtt{then}\, e_1\, \mathtt{else}\, e_2\, \mathtt{fi}$*? Show that both typing and evaluation are preserved by these interpretations.*

Remark 3.4
We often omit the type subscripts on the primitive constructs when they are clear from context.

3.3 References

Chapter 4

Subtyping

4.1 Introduction

Subtyping is a relationship between types validating the *subsumption principle*: if σ is a subtype of τ, then a value of type σ may be provided whenever a value of type τ is required. In this chapter we introduce the fundamental mechanisms of subtyping, and consider three typical situations in which it arises, namely primitive subtyping, tuple subtyping, and record subtyping. To simplify the discussion we work with the extension $\mathcal{L}^{\rightarrow}_{<:}$ of $\mathcal{L}^{\rightarrow}$ with subtyping.

4.2 Subtyping

A *subtyping* relation, $\sigma <: \tau$, is a binary relation between types that is closed under the following rules of inference:

$$\sigma <: \sigma \qquad \text{(S-REFL)}$$

$$\frac{\rho <: \sigma \qquad \sigma <: \tau}{\rho <: \tau} \qquad \text{(S-TRANS)}$$

$$\frac{\tau_1 <: \sigma_1 \qquad \sigma_2 <: \tau_2}{\sigma_1 \rightarrow \sigma_2 <: \tau_1 \rightarrow \tau_2} \qquad \text{(S-ARROW)}$$

These rules require that subtyping be a pre-order (reflexive and transitive) and specify that the function space constructor is *contravariant* in the domain position (*i.e.*, the subtype ordering is reversed) and *covariant* in the co-domain position (*i.e.*, the subtype ordering is preserved).

We will be concerned with various forms of subtyping for various extensions of $\mathcal{L}^{\rightarrow}_{<:}$. These will all have the property that subtyping is structural in the sense that primitive types are subtypes only of primitive types, and function types are subtypes only of function types. More precisely, a subtype relation is *normal* iff it satisfies the following two properties:

1. $\sigma_1 \to \sigma_2 <: \tau_1 \to \tau_2$ iff $\tau_1 <: \sigma_1$ and $\sigma_2 <: \tau_2$, and

2. if $\sigma <: \tau_1 \to \tau_2$, then $\sigma = \sigma_1 \to \sigma_2$, and conversely if $\sigma_1 \to \sigma_2 <: \tau$, then $\tau = \tau_1 \to \tau_2$.

4.2.1 Subsumption

Not just any pre-order on types validating the co- and contra-variance of the function space constructor is a subtype relation. Only those pre-orders that validate the principle of *subsumption* qualify:

$$\frac{\Gamma \vdash e : \tau \qquad \tau <: \tau'}{\Gamma \vdash e : \tau'} \tag{T-SUB}$$

This rule may be paraphrased as follows: *wherever a value of type τ' is required, a value of type τ may be provided.*

Since the rule T-SUB can be applied to any expression at any time, the typing rules for $\mathcal{L}^{\to}_{<:}$ are not syntax-directed. This can present problems for type checking, and requires that we take special care in formulating its meta-theory.

4.3 Primitive Subtyping

One familiar form of subtyping arises from the standard mathematical practice of regarding the integers to be a subset of the reals. Strictly speaking, the inclusion $\mathbb{Z} \subseteq \mathbb{R}$ fails for the standard definitions of these sets — but of course there is an insertion $\mathbb{Z} \hookrightarrow \mathbb{R}$ that we use implicitly when regarding an integer as a real number.

The analogous situation for a programming language arises in an extension $\mathcal{L}^{\mathtt{Int},\mathtt{Float},\to}_{<:}$ of $\mathcal{L}^{\to}$ with base types `Int` and `Float`. It is convenient to regard an integer as a floating point number, even though the typical machine representations of integers and floats are completely different. We will explore the implications of postulating `Int` $<:$ `Float` in a modest extension of $\mathcal{L}^{\to}$ with integer and floating point operations:

$$\begin{array}{llll} \textit{Expressions} & e & ::= & \overline{n} \mid e_1 +_{\mathtt{Int}} e_2 \mid e_1 -_{\mathtt{Int}} e_2 \mid \\ & & & \overline{q} \mid e_1 +_{\mathtt{Float}} e_2 \mid e_1 -_{\mathtt{Float}} e_2 \end{array}$$

where n is an integer and q is a rational number.[1] The typing rule for these expressions are as expected.

The operational semantics for $\mathcal{L}^{\mathtt{Int},\mathtt{Float},\to}_{<:}$ is defined as follows. We take as values the integer literals, $\overline{n}$, and the floating point literals, $\overline{q}$. What are the closed values of type `Int` and `Float`? For the type `Int`, these are clearly the

[1]We will ignore matters of precision in this discussion. Obviously not every integer is representable in a computer, nor is every rational representable as a floating point number.

numerals $\overline{n}$, where $n \in \mathbb{Z}$. The closed values of type `Float` include not only the numerals $\overline{q}$, where $q \in \mathbb{Q}$, but also the integer numerals $\overline{n}$, where $n \in \mathbb{Z}$!

The evaluation contexts are defined in the obvious way, corresponding to a left-to-right evaluation order for the primitive expression forms. The primitive evaluation steps include these basic steps:

$$\begin{array}{rcl}
\overline{m} +_{\mathtt{Int}} \overline{n} & \leadsto & \overline{m+n} \\
\overline{m} -_{\mathtt{Int}} \overline{n} & \leadsto & \overline{m-n} \\
\overline{q} +_{\mathtt{Float}} \overline{r} & \leadsto & \overline{q+r} \\
\overline{q} -_{\mathtt{Float}} \overline{r} & \leadsto & \overline{q-r}
\end{array}$$

However, these instructions are not enough to ensure that well-typed programs make progress! We must also define the behavior of the floating point operations on integer arguments. This is achieved by augmenting the operational semantics with these rules (together with a similar set of rules for the other floating point operations):

$$\begin{array}{rcl}
\overline{m} +_{\mathtt{Float}} \overline{r} & \leadsto & \overline{(m/1)+r} \\
\overline{q} +_{\mathtt{Float}} \overline{n} & \leadsto & \overline{q+(n/1)} \\
\overline{m} +_{\mathtt{Float}} \overline{n} & \leadsto & \overline{(m/1)+(n/1)}
\end{array}$$

Exercise 4.1
Prove progress and preservation for $\mathcal{L}_{<:}^{\mathtt{Int},\mathtt{Float},\rightarrow}$. Note that if the "extra" rules were omitted, then progress would fail.

One thing to notice about this operational semantics is that it relies on the ability to distinguish floating point values from integer values *at run time*. This corresponds to a "tagging" scheme whereby values come with type information on which we may dynamically dispatch. While dynamically-typed languages take this as a design principle, it is unreasonable to impose such a scheme on a statically-typed language. In Chapter 20 we will describe methods for avoiding run-time type dispatch in cases such as this by introducing coercions at compile-time that change the representations of values whenever subtyping is used.

Exercise 4.2
Suppose that we define a type `Nat` *such that* `Nat` <: `Int` <: `Float`. *What adjustments must be made to the operational semantics and to its meta-theory?*

4.4 Tuple Subtyping

An important form of subtyping is derived from tupling. Let us consider the extension $\mathcal{L}_{<:}^{<>,\rightarrow}$ of $\mathcal{L}^{\rightarrow}$ with the variadic tuple type, $<\tau_1,\ldots,\tau_n>$, where $n \geq 0$. Informally, this is the type of n-tuples whose ith component has type τ_i for each $1 \leq i \leq n$. When $n = 0$ this is just the `Unit` type; when $n \geq 1$, these are just familiar n-tuples of values.[2]

[2]We permit the case $n = 1$ for uniformity, but do not identify $<\tau_1,\ldots,\tau_n>$ with τ.

The syntax of the distinctive expressions of this extension is as follows:

$$\textit{Expressions} \quad e \quad ::= \quad <e_1,\ldots,e_n>_{\tau_1,\ldots,\tau_n} \mid \mathtt{proj}^i_{\tau_1,\ldots,\tau_n}(e)$$

The typing rules governing these expressions are as follows:

$$\frac{\Gamma \vdash e_1 : \tau_1 \qquad \cdots \qquad \Gamma \vdash e_n : \tau_n}{\Gamma \vdash <e_1,\ldots,e_n>_{\tau_1,\ldots,\tau_n} : <\tau_1,\ldots,\tau_n>} \qquad (\text{T-TUPLE})$$

$$\frac{\Gamma \vdash e : <\tau_1,\ldots,\tau_n>}{\Gamma \vdash \mathtt{proj}^i_{\tau_1,\ldots,\tau_n}(e) : \tau_i} \quad (1 \leq i \leq n) \qquad (\text{T-PROJ})$$

Informally, we use the abbreviated syntax $<e_1,\ldots,e_n>$ for tuples and $\mathtt{proj}^i(e)$ for projections.

The operational semantics is given as follows. The closed values of type $<\tau_1,\ldots,\tau_n>$ are expressions of the form $<\tau_1,\ldots,\tau_n>v$, where v_i is a closed value of type τ_i for each $1 \leq i \leq n$. Evaluation contexts are defined as follows:

$$\textit{EvalCtxt's} \quad E \quad ::= \quad <v_1,\ldots,v_{i-1},E,e_{i+1},\ldots,e_n>_{\tau_1\ldots\tau_n} \mid \mathtt{proj}^i_{\tau_1,\ldots,\tau_n}(E)$$

The primitive reduction steps are defined as follows:

$$\mathtt{proj}^i_{\tau_1,\ldots,\tau_n}(<v_1,\ldots,v_n>_{\tau_1,\ldots,\tau_n}) \leadsto v_i$$

where $1 \leq i \leq n$. Notice that we require the types of the fields ascribed to the projection to coincide with those ascribed to the tuple itself. In the presence of subtyping this requirement will be relaxed, as we shall see shortly.

Two forms of subtyping arise in connection with tuple types:

1. *Width subtyping*: a "wider" tuple type is regarded as a subtype of a "narrower" tuple type. This is captured by the subtyping axioms

   $$<\tau_1,\ldots,\tau_{m+n}> <: <\tau_1,\ldots,\tau_m> \qquad (\text{S-TUPLE-WIDTH})$$

   where $m,n \geq 0$.

2. *Depth subtyping*: one tuple type is a subtype of another with the same width if every field of the former is a subtype of the latter. That is,

   $$\frac{\tau_1 <: \tau_1' \qquad \cdots \qquad \tau_n <: \tau_n'}{<\tau_1,\ldots,\tau_n> <: <\tau'_1,\ldots,\tau'_n>} \qquad (\text{S-TUPLE-DEPTH})$$

Thus width subtyping allows us to neglect "extra" components at the end of a tuple, and depth subtyping allows us to apply subtyping "recursively" through components of a tuple.

How does the operational semantics change in the presence of tuple subtyping? To accommodate width subtyping, we must change the primitive operations as follows:

$$\mathtt{proj}^i_{\tau_1,\ldots,\tau_m}(<v_1,\ldots,v_{m+n}>_{\tau_1,\ldots,\tau_{m+n}}) \leadsto v_i$$

This accounts for the fact that the closed values of type $<\tau_1,\ldots,\tau_m>$ are tuples of the form $<v_1,\ldots,v_{m+n}>_{\tau_1,\ldots,\tau_{m+n}}$. To account for depth subtyping we further generalize this instruction to

$$\mathtt{proj}^i_{\tau_1,\ldots,\tau_m}(<v_1,\ldots,v_{m+n}>_{\tau'_1,\ldots,\tau'_{m+n}}) \leadsto v_i$$

(taking $n = 0$ if width subtyping is to be disallowed).

Exercise 4.3
Check soundness for width and depth tuple subtyping.

To understand the implications of these rules for implementation, it is essential to pay close attention to the fully-explicit syntax. Let us consider first pure width subtyping (no depth subtyping). A value of type $<\tau_1,\ldots,\tau_n>$ may be thought of as a pointer to a heap-allocated structure consisting of n consecutive values of type $\tau_1, \ldots, \tau_n$.[3] For example, if values of type `Int` are words and values of type `Float` are doublewords, then a tuple of type $<$`Float`,`Int`$>$ is represented by a pointer to a doubleword followed by a word, and a tuple of type $<$`Int`,`Float`$>$ is represented by a pointer to a word followed by a doubleword. *The type determines the layout of the data.* Given this, the behavior of a projection $\mathtt{proj}^i_{\tau_1,\ldots,\tau_n}(-)$ is fully determined by the index i and the types $\tau_1,\ldots,\tau_{i-1}$ — the ith component is an object of size determined by τ_i at offset determined by $\tau_1, \ldots, \tau_{i-1}$. All of this data (apart from the pointer itself) is determined at compile-time, as is made clear in the fully-explicit syntax.

Width subtyping alone presents no significant complications. Since the behavior of a projection is determined by the index and the types of the *preceding* components of the tuple, the presence or absence of *succeeding* components is of no consequence. Thus width subtyping can be had "for free" in the sense that no additional overhead is required to support it. However, depth subtyping introduces complications since the type $<\tau_1,\ldots,\tau_n>$ of a tuple no longer determines the offset or size of any given component of that tuple! If a subtyping relation implies a change of representation, then applying subtyping at position i of a tuple type leads to a loss of information about the size of that component and the offsets of all succeeding components in the tuple. For example, we may think of a value of type $<$`Int`,`Int`$>$ as a value of type $<$`Float`,`Float`$>$. But the second component lies at offset 1, not 2, and is of size 1, not 2, despite what the type may suggest!

What can we do about this? The fully-explicit syntax suggests one solution: the value itself carries enough information to determine its "true" type, regardless of its "apparent" type as the argument to a projection. Continuing the example, the tuple $<v_1,v_2>_{\mathtt{Int},\mathtt{Int}}$ has the true type $<$`Int`,`Int`$>$, which is a subtype of its apparent type $<$`Float`,`Float`$>$. Thus a projection operation must *ignore* the apparent type determined by the explicit syntax for the projection, and instead make use of the true type attached to the tuple itself. This has two implications:

[3]Similar remarks apply to schemes for allocating tuples in registers.

1. Type information must be explicitly attached to the tuple at run-time, at the expense of both space (for the type information) and time (to maintain it).

2. Projections must analyze the "true" type of their tuple argument at run-time to determine the size and offset of its components. This is costly, and difficult, in general, to optimize away.

Is there any alternative? One choice is to introduce a universal representation assumption whereby we ensure that every value has unit size so that projections may be implemented independently of *any* type information: the ith component is always one word at offset i from the start of the tuple. This precludes "flattening" tuples by allocating them as consecutive data items in their natural representation, at least for those types having subtypes with a different natural representation. (Integers and floating point numbers provide one example, as do tuples themselves, in the presence of width subptying. Such values must be heap-allocated to ensure that their representation is predictable at compile time.) A third choice is to adopt a "coercive" interpretation of subtyping whereby a use of depth subtyping causes the tuple to be *reconstructed* as a value of the supertype, ensuring that the type once again fully determines the layout. We will discuss this approach in Chapter 20.

4.5 Record Subtyping

Records are a form of tuple in which the components are labelled to allow more convenient access. Record types have the form $\{\, l_1{:}\tau_1, \ldots, l_n{:}\tau_n \,\}$, where each l_i is a distinct label, drawn from some countable alphabet disjoint from variables. Records are constructed using record-formation expressions, and accessed using labelled selection expressions. These are defined by the following grammar:

$$\textit{Expressions} \quad e \quad ::= \quad \{\, l_1{=}e_1, \ldots, l_n{=}e_n \,\}_{\tau_1,\ldots,\tau_n} \mid \mathtt{sel}^{l}_{l_1:\tau_1,\ldots,l_n:\tau_n}(e)$$

We often use the abbreviated syntax $\{\, l_1{=}e_1, \ldots, l_n{=}e_n \,\}$ and $e.l$ when it is not important to emphasize the type information associated with the fully-explicit forms given above.

The typing rules for these expressions are as follows:

$$\frac{\Gamma \vdash e_1 : \tau_1 \qquad \cdots \qquad \Gamma \vdash e_n : \tau_n}{\Gamma \vdash \{\, l_1{=}e_1, \ldots, l_n{=}e_n \,\}_{\tau_1,\ldots,\tau_n} : \{\, l_1{:}\tau_1, \ldots, l_n{:}\tau_n \,\}} \quad (n \geq 0,\ i \neq j \supset l_i \neq l_j) \qquad \text{(T-RECORD)}$$

$$\frac{\Gamma \vdash e : \{\, l_1{:}\tau_1, \ldots, l_n{:}\tau_n \,\}}{\Gamma \vdash \mathtt{sel}^{l}_{l_1:\tau_1,\ldots,l_n:\tau_n}(e) : \tau_i} \quad (l = l_i,\ 1 \leq i \leq n) \qquad \text{(T-SELECT)}$$

There are two interpretations of record types, according to whether or not the order of fields is considered significant. Under the *ordered* interpretation the types $\{ l{:}\tau, l'{:}\tau' \}$ and $\{ l'{:}\tau', l{:}\tau \}$ are distinct types since the fields occur in a different order. This view is typical of languages such as C that attempt to match their data representations with externally-given constraints (such as network packet layouts). Under the *unordered* interpretation the above two types are identified, as are any two record types differing only in the order of their fields. The unordered interpretation affords more flexibility to the programmer, but precludes the satisfaction of externally-given layout constraints on records.

The dynamic semantics of record operations is similar to that for tuples. Closed values of type $\{ l_1{:}\tau_1, \ldots, l_m{:}\tau_m \}$ are records of the form

$$\{ \tau_1{=}v_1, \ldots, \tau_{m+n}{=}v_{m+n} \},$$

where v_i is a closed value of type τ_i for each $1 \leq i \leq m$. Evaluation contexts are defined as follows:

$$\begin{array}{lll} \mathit{EvalCtxt's} \quad E & ::= & \{ l_1{=}v_1, \ldots, l_{i-1}{=}v_{i-1}, l_i{=}E, l_{i+1}{=}e_{i+1}, \ldots, l_n{=}e_n \} \mid \\ & & \mathtt{sel}^{l}_{l_1:\tau_1,\ldots,l_n:\tau_n}(E) \end{array}$$

Field selection is given by the following instruction:

$$\mathtt{sel}^{l}_{l_1:\tau_1,\ldots,l_m:\tau_m}(\{ l_1{=}v_1, \ldots, l_{m+n}{=}v_{m+n} \}_{\tau_1,\ldots,\tau_{m+n}}) \leadsto v_i$$

where $l = l_i$ for some $1 \leq i \leq m$. This instruction is formulated with subtyping in mind; without subtyping, we would take $n = 0$ in the above instruction.

Remark 4.4
Tuples may be regarded as records whose fields are labelled by the integers $1, \ldots, n$, so that the ith projection $\mathtt{proj}_i(e)$ stands for the selection $i.e$ of the field labelled i.

Just as for tuples we may consider both width and depth subtyping for record types:

$$\{ l_1{:}\tau_1, \ldots, l_{m+n}{:}\tau_{m+n} \} <: \{ l_1{:}\tau_1, \ldots, l_m{:}\tau_m \} \qquad \text{(S-RECORD-WIDTH)}$$

$$\frac{\tau_1 <: \tau_1' \qquad \cdots \qquad \tau_n <: \tau_n'}{\{ l_1{:}\tau_1, \ldots, l_n{:}\tau_n \} <: \{ l_1{:}\tau'_1, \ldots, l_n{:}\tau'_n \}} \qquad \text{(S-RECORD-DEPTH)}$$

The interpretation of the width subtyping rule differs according to whether or not we admit reordering of fields. If not, then the rule S-RECORD-WIDTH specifies that we may neglect fields occurring at the end of a record, but nowhere else. If so, then the rule S-RECORD-WIDTH specifies that we may neglect any field of a record when passing to a supertype. The depth subtyping rule has the same interpretation, irrespective of whether we admit reordering of fields.

What are the implementation implications for the various choices of re-ordering and subtyping principles? If re-ordering is precluded, then the trade-off's are precisely as for tuple types: width subtyping is "free", but depth sub-typing incurs some overhead. If re-ordering is admitted, then the situation is more complex. Re-ordering in the absence of width subtyping (but possibly in the presence of depth subtyping) may be implemented by imposing a standard ordering of fields (*e.g.*, alphabetical ordering of labels). In the presence of width subtyping field selection must involve search since the apparent type of the record does not reveal the position of that field in the underlying record value. Record values must carry their "true" type, revealing the position and type of each field, and selection must refer to this type to retrieve a component.

An alternative is to employ a "coercive" interpretation of subtyping in which a use of record subtyping induces the creation of a fresh record whose principal type is the supertype. This ensures that the apparent type is always the true type of the record value so that selection operations may be fully determined at compile-time, without the need for run-time type analysis. A variant is based on the idea of a *dictionary* (or *vtable*) that mediates access to a fixed underlying record. We will discuss these interpretations in more detail in Chapter 20.

4.6 References

[?, ?]

Chapter 5

Variable Types

5.1 Introduction

This chapter is concerned with the concept of *variable types*, fundamental to a rigorous treatment of separate compilation, data abstraction, polymorphism, and modularity. We consider the extension of $\mathcal{L}^{1,\times,\rightarrow}$ with variable types, designated $\mathcal{L}_{\Omega}^{1,\times,\rightarrow}$, and re-consider the problem of separate compilation in this larger context. A rudimentary form of data abstraction arises naturally in this setting. Finally, we consider *type definitions* for introducing concrete (*i.e.*, non-abstract) types.

5.2 Variable Types

The language $\mathcal{L}_{\Omega}^{1,\times,\rightarrow}$ is the enrichment of $\mathcal{L}^{1,\times,\rightarrow}$ with a new form of type expression, the *type variable*:

$$\textit{Types} \quad \tau \quad ::= \quad \ldots \mid t$$

We let t range over an unspecified countably infinite set of type variables. The enrichment of a language with variable types is of course not dependent on the presence of any particular type constructors; we consider $\mathcal{L}^{1,\times,\rightarrow}$ as the basis for our discussion for the sake of specificity.

The set of *type contexts* is defined as follows:

$$\textit{TypeContexts} \quad \Delta \quad ::= \quad \emptyset \mid \Delta[t]$$

We abuse notation slightly and regard Δ as a finite set of type variables whenever convenient.

The judgement $\Delta \vdash \tau$ means that the type variables occurring in τ are drawn from the set Δ. Similarly, the judgement $\Delta \vdash \Gamma$ means that $\Delta \vdash \Gamma(x)$ for every $x \in \mathrm{dom}(\Gamma)$. Both of these judgements could be defined an an explicit set of derivation rules, but we will refrain from doing so at this stage.

Typing judgements for $\mathcal{L}_{\Omega}^{1,\times,\rightarrow}$ have the form $\Gamma \vdash_{\Delta} \tau$, where $\Delta \vdash \Gamma$ and $\Delta \vdash \tau$. The rules for deriving these judgements are the same as for $\mathcal{L}^{1,\times,\rightarrow}$, except for the presence of the type context Δ, which determines the set of type variables that may be used in the judgement. At this stage the set Δ of type variables does not vary; it is a fixed parameter for all judgements of the system.

Lemma 5.1
The following rules are admissible:

$$\frac{\Gamma \vdash_{\Delta[t]} e : \tau}{\{t'/t\}\Gamma \vdash_{\Delta[t']} \{t'/t\}e : \{t'/t\}\tau} \quad (t' \notin \Delta) \qquad (\text{T-TYP-RENAME})$$

$$\frac{\Gamma \vdash_{\Delta} e : \tau}{\Gamma \vdash_{\Delta'} e : \tau} \quad (\Delta' \supseteq \Delta) \qquad (\text{T-TYP-WEAKEN})$$

$$\frac{\Gamma \vdash_{\Delta} e : \tau}{\Gamma \vdash_{\Delta'} e : \tau} \quad (*) \qquad (\text{T-TYP-STRENGTHEN})$$

provided that $\mathrm{FTV}(\Delta') \cup \mathrm{FTV}(e) \cup \mathrm{FTV}(\tau) \subseteq \Delta' \subseteq \Delta$.

$$\frac{\Gamma \vdash_{\Delta[t]} e : \tau' \qquad \Delta \vdash \tau}{\{\tau/t\}\Gamma \vdash_{\Delta} \{\tau/t\}e : \{\tau/t\}\tau'} \quad (t \notin \Delta) \qquad (\text{T-SUBST})$$

Exercise 5.2
Prove Lemma 5.1. Observe that the rules T-TYP-RENAME *and* T-TYP-STRENGTHEN *are derivable from the other two.*

5.3 Type Definitions

The type variables in Δ are *opaque* in the sense that their ultimate definition (provided by a substitution) is not available during type-checking. Type variables behave like "new" types, distinct from one another and from any other types in the system. You might reasonably wonder whether such opaque types are of any use. At this stage we do not have sufficient machinery available to exploit them to any significant extent, but this will be remedied in subsequent chapters.

Of more immediate use is the ability to introduce *type definitions* in a program using what are called *transparent* type bindings. These are strictly an abbreviatory construct introduced for the convenience of the programmer. The grammar of expressions is extended as follows:

$$\textit{Expressions} \quad e \quad ::= \quad \ldots \mid \texttt{type}\, t\, \texttt{is}\, \tau\, \texttt{in}\, e\, \texttt{end}$$

Informally, this construct binds the type variable t to the type expression τ within the expression e. It is a binding operator, and is subject to the usual rules of α-conversion.

There are two approaches to the static semantics of type definitions. The most obvious is to use the following typing rule:

$$\frac{\Delta \vdash \tau \qquad \Gamma \vdash_\Delta \{\tau/t\}e : \tau'}{\Delta \vdash \texttt{type}\, t\, \texttt{is}\, \tau\, \texttt{in}\, e\, \texttt{end} : \tau'} \quad (t \notin \Delta) \tag{T-TYPDEF}$$

We require that the variable t not occur in Δ to avoid accidental clashes during substitution.

The dynamic semantics of type definitions is defined similarly:

$$\texttt{type}\, t\, \texttt{is}\, \tau\, \texttt{in}\, e\, \texttt{end} \leadsto \{\tau/t\}e$$

It is a simple matter to check that this extension is sound.

A slightly more sophisticated approach is to extend type contexts with *type definitions* as follows:

$$\textit{Type Contexts} \quad \Delta \quad ::= \quad \emptyset \mid \Delta[t] \mid \Delta[t{=}\tau]$$

Notice that a typical type context Δ contains both opaque and transparent type definitions. Such type contexts are sometimes dubbed *translucent* for this reason.

Introduction of type definitions into type contexts complicates the formalism somewhat. In particular, we must now explicitly define well-formedness of type contexts, and, more importantly, we must introduce the notion of *definitional equality* between types to take account of type definitions in contexts.

Exercise 5.3
Give rules for well-formedness of translucent type contexts.

Definitional equality between types, $\Delta \vdash \tau_1 \equiv \tau_2$, is generated by the following "lookup" rule

$$\Delta[t{=}\tau] \vdash t \equiv \tau \tag{E-TYP-DEF}$$

together with rules ensuring that it is an equivalence relation and is compatible with all type-forming constructs.

Exercise 5.4
Give a precise formulation of definitional equality of types relative to a translucent type context.

The force of definitional equality is captured by the following rule stating that typing respects definitional equality of types:

$$\frac{\Gamma \vdash_\Delta e : \tau \qquad \Delta \vdash \tau \equiv \tau'}{\Gamma \vdash_\Delta e : \tau'} \tag{T-DEF-EQ}$$

Using definitional contexts we may give the rule for type definitions as follows:

$$\frac{\Delta \vdash \tau \qquad \Gamma \vdash_{\Delta[t=\tau]} e : \tau'}{\Gamma \vdash \texttt{type}\, t\, \texttt{is}\, \tau\, \texttt{in}\, e\, \texttt{end} : \{\tau/t\}\tau'} \quad (t \notin \mathrm{dom}(\Delta)) \qquad (\text{T-TYPE-DEF'})$$

Notice that we must substitute τ for t in τ' in the conclusion of this rule! Otherwise the type variable t, which might occur in τ', would escape its scope of significance, which is nonsensical.

Exercise 5.5
State and prove soundness of typing for type definitions presented using translucent type contexts.

5.4 Constructors, Types, and Kinds

At this point it is useful to introduce some machinery. The main idea is to introduce a careful distinction between "types as data" and "types as classifiers", with a view towards extensions to the language that we will consider shortly. As data types may be bound to type variables (and, as we shall see in Chapters 7 and 8, passed as parameters and stored in data structures). As classifiers types are used to govern the formation of ordinary expressions. In many situations it is perfectly sensible, and convenient, to gloss over this distinction. Nevertheless it is important to know what is being glossed over whenever this is done.

To make explicit the distinction between types as classifiers and types as data, we consider the following formulation of $\mathcal{L}_{\Omega}^{1,\times,\rightarrow}$:

Kinds	κ	$::=$	Ω
Constructors	τ	$::=$	$t \mid \texttt{int} \mid \texttt{bool} \mid \tau_1 \rightarrow \tau_2$
Types	σ	$::=$	$T(\tau) \mid \texttt{Int} \mid \texttt{Bool} \mid \sigma_1 \rightarrow \sigma_2$
Term Contexts	Γ	$::=$	$\emptyset \mid \Gamma[x{:}\sigma]$
Type Contexts	Δ	$::=$	$\emptyset \mid \Delta[t{::}\kappa]$

There is, at present, only one kind, Ω, which classifies well-formed type constructors. The type-forming operation $T(-)$ is an *inclusion* of constructors into types. Thinking of constructors (of kind Ω) as *names* for types, the type $T(\tau)$ is the type named by τ. This correspondence will be made precise shortly. The language of expressions remains unchanged, apart from the enrichment of types to include type variables.

The judgement forms are as follows:

$\Delta \vdash \tau :: \kappa$	*τ has kind κ over Δ*
$\Delta \vdash \tau_1 \equiv \tau_2 :: \kappa$	*τ_1 and τ_2 are equivalent constructors of kind κ*
$\vdash_\Delta \sigma$	*σ is a valid type over Δ*
$\vdash_\Delta \sigma_1 \equiv \sigma_2$	*σ_1 and σ_2 are equivalent types*
$\Gamma \vdash_\Delta e : \sigma$	*e has type σ relative to Γ and Δ*

The judgement $\Delta \vdash \tau :: \kappa$ expresses that τ is a well-formed type constructor. This is just the judgement $\Delta \vdash \tau$ given above, but written using the more general notation. The judgement $\Delta \vdash \tau_1 \equiv \tau_2 :: \kappa$ expresses definitional equality of type constructors. At present we take it to be the identity relation, but later less trivial notions of definitional equivalence will be considered.

The judgement $\vdash_\Delta \sigma$ expresses that σ is a well-formed type relative to the type context Δ. It is defined by the following rules:

$$\frac{\Delta \vdash \tau :: \Omega}{\vdash_\Delta T(\tau)} \qquad \text{(T-COERCE)}$$

$$\vdash_\Delta \texttt{Int} \qquad \text{(T-INT)}$$

$$\vdash_\Delta \texttt{Bool} \qquad \text{(T-BOOL)}$$

$$\frac{\vdash_\Delta \sigma_1 \qquad \vdash_\Delta \sigma_2}{\vdash_\Delta \sigma_1 \to \sigma_2} \qquad \text{(T-ARROW)}$$

The judgement $\vdash_\Delta \sigma_1 \equiv \sigma_2$ expresses definitional equality of the types σ_1 and σ_2. (We assume that $\vdash_\Delta \sigma_1$ and $\vdash_\Delta \sigma_2$.) It is defined to be the least congruence relation closed under the following rules:

$$\vdash_\Delta T(\texttt{int}) \equiv \texttt{Int} \qquad \text{(E-T-INT)}$$

$$\vdash_\Delta T(\texttt{bool}) \equiv \texttt{Bool} \qquad \text{(E-T-BOOL)}$$

$$\vdash_\Delta T(\tau_1 \to \tau_2) \equiv T(\tau_1) \to T(\tau_2) \qquad \text{(E-T-ARROW)}$$

$$\frac{\Delta \vdash \tau_1 \equiv \tau_2 :: \Omega}{\vdash_\Delta T(\tau_1) \equiv T(\tau_2)} \qquad \text{(E-T-CONG)}$$

Typing respects type equivalence:

$$\frac{\Gamma \vdash_\Delta e : \sigma \qquad \vdash_\Delta \sigma \equiv \sigma'}{\Gamma \vdash_\Delta e : \sigma'} \qquad \text{(T-EQUIV)}$$

Exercise 5.6
Definitional equality may be extended to contexts in the obvious way: $\vdash_\Delta \Gamma \equiv \Gamma'$ *iff* $\mathrm{dom}(\Gamma) = \mathrm{dom}(\Gamma')$ *and* $\vdash_\Delta \Gamma(x) \equiv \Gamma'(x)$ *for every* $x \in \mathrm{dom}(\Gamma)$*. Prove that if* $\Gamma \vdash_\Delta e : \sigma$ *and* $\vdash_\Delta \Gamma \equiv \Gamma'$*, then* $\Gamma' \vdash_\Delta e : \sigma$*.*

It is easy to see that every type is equivalent to one in which the only occurrences of $\mathcal{T}(-)$ are applied to type variables. By keeping types in simplest form, all uses of the rule T-EQUIV may be eliminated. For this reason we often ellide mention of $\mathcal{T}(-)$, and simply regard every constructor of kind Ω as a type. This is called the *implicit* formulation of the system, in contrast to the *explicit* formulation in which all coercions are supplied. At this point the distinction between the implicit and explicit formulations may seem like hair-splitting[1]. We will see later on, however, that it is useful to sort things out this way.

Exercise 5.7
State and prove the equivalence of the implicit and explicit presentations of $\mathcal{L}_\Omega^{1,\times,\rightarrow}$.

A further simplification is possible in some cases. Often the distinction between types as data and types as classifiers is dropped entirely by conflating the level of constructors and types. Then types = constructors are classified by kinds, and expressions are classified by types. This formulation often streamlines the syntax, but at the expense of precluding the possibility of subjecting the constructor and type levels to different closure conditions. In Chapter 7 we will exploit this distinction to distingish the notions of *predicative* and *impredicative* quantification.

5.5 References

The treatment of variable types separately from polymorphism was inspired by Girard [**?**] and the recent categorial accounts of type theory [**?**].

[1]That would only be because it is.

Chapter 6

Recursive Types

6.1 Introduction

In this chapter we consider *recursive* types. First, we consider two important special cases, one the natural numbers treated as an inductive type, the other streams of natural numbers treated as a co-inductive type. Second, we generalize these two examples to arbitrary recursive types, including so-called *reflexive* types that (in a sense) contain their own function spaces.

6.2 Gödel's T

Gödel's system **T** is the extension $\mathcal{L}^{\mathtt{nat},\rightarrow}$ of $\mathcal{L}^{\rightarrow}$ with a type of natural numbers, equipped with primitive recursion at all types. This language is sufficiently expressive that every function on the naturals that is provably total in first-order Peano arithmetic is definable in system **T**. (The proof, which we do not give here, is based on the so-called *Dialectica* interpretation of arithmetic outlined in Gödel's seminal paper on the consistency of arithmetic.) Our purpose in introducing system **T** is as a particular case of a language with an *inductive* type. The methods that we use to analyze system **T** generalize to the case of arbitrary inductive types.

6.2.1 Statics

The syntax of system **T** is as follows:

$$
\begin{array}{llll}
\textit{Types} & \tau & ::= & \mathtt{Nat} \mid \tau_1 \rightarrow \tau_2 \\
\textit{Expressions} & e & ::= & x \mid \mathtt{fun}\,(x{:}\tau)\,\mathtt{in}\,e \mid e_1(e_2) \mid \mathtt{0} \mid \mathtt{succ}(e) \mid \\
 & & & \mathtt{natrec}_\tau\, e\, \mathtt{of}\, \mathtt{0} \mathtt{=>} e_0 \mid \mathtt{succ}(x{:}\tau) \mathtt{=>} e_s\, \mathtt{end}
\end{array}
$$

The typing rules are those of $\mathcal{L}^{\rightarrow}$, extended with the following rules:

$$\Gamma \vdash \mathtt{0} : \mathtt{Nat} \qquad \text{(T-ZERO)}$$

$$\frac{\Gamma \vdash e : \texttt{Nat}}{\Gamma \vdash \texttt{succ}(e) : \texttt{Nat}} \qquad \text{(T-SUCC)}$$

$$\frac{\Gamma \vdash e : \texttt{Nat} \quad \Gamma \vdash e_0 : \tau \quad \Gamma[x{:}\tau] \vdash e_s : \tau}{\Gamma \vdash \texttt{natrec}_\tau\, e\, \texttt{of}\, 0 \texttt{=>} e_0 \mid \texttt{succ}(x{:}\tau) \texttt{=>} e_s\, \texttt{end} : \tau} \quad (x \notin \mathrm{dom}(\Gamma)) \qquad \text{(T-NATREC)}$$

6.2.2 Dynamics

The syntax of values and evaluation contexts is as follows:

$$\begin{array}{llll} \textit{Values} & v & ::= & x \mid \texttt{fun}\,(x{:}\tau)\,\texttt{in}\,e \mid 0 \mid \texttt{succ}(v) \\ \textit{EvalCtxt's} & E & ::= & \bullet \mid E(e_2) \mid v_1(E) \mid \texttt{succ}(E) \\ & & & \mid \texttt{natrec}_\tau\, E\, \texttt{of}\, 0 \texttt{=>} e_0 \mid \texttt{succ}(x{:}\tau) \texttt{=>} e_s\, \texttt{end} \end{array}$$

The primitive reduction steps for `natrec` are these:[1]

$$\begin{array}{rcl} \texttt{natrec}_\tau\, 0\, \texttt{of}\, 0 \texttt{=>} e_0 \mid \texttt{succ}(x{:}\tau) \texttt{=>} e_s\, \texttt{end} & \leadsto & e_0 \\ \texttt{natrec}_\tau\, \texttt{succ}(v)\, \texttt{of}\, 0 \texttt{=>} e_0 \mid \texttt{succ}(x{:}\tau) \texttt{=>} e_s\, \texttt{end} & \leadsto & \\ \texttt{let}\, x{:}\tau\, \texttt{be}\, \texttt{natrec}_\tau\, v\, \texttt{of}\, 0 \texttt{=>} e_0 \mid \texttt{succ}(x{:}\tau) \texttt{=>} e_s\, \texttt{end}\, \texttt{in}\, e_3 & & \end{array}$$

Exercise 6.1
Check type soundness (progress and preservation) for system $\boldsymbol{T}$.

6.2.3 Termination

We prove termination by constructing a family of sets Val_τ for each type τ as follows:

$$\begin{array}{rcl} \mathit{Val}_{\texttt{Nat}} & = & \{\, 0 \,\} \cup \{\, \texttt{succ}(v) \mid v \in \mathit{Val}_{\texttt{Nat}} \,\} \\ \mathit{Val}_{\tau_1 \to \tau_2} & = & \{\, v : \tau_1 \to \tau_2 \mid \forall v_1 \in \mathit{Val}_{\tau_1}\;\; v(v_1) \in \mathit{Comp}_{\tau_2}, \} \end{array}$$

We then define $\mathit{Comp}_\tau = \{\, e : \tau \mid e \mapsto^* v \in \mathit{Val}_\tau \,\}$.

Why is the family of sets Val_τ well-defined? Let $\Phi_{\texttt{Nat}}$ be defined by the equation

$$\Phi_{\texttt{Nat}}(X) = \{\, 0 \,\} \cup \{\, \texttt{succ}(v) \mid v \in X \,\}.$$

Clearly $\Phi_{\texttt{Nat}}$ is monotone, and we seek $\mathit{Val}_{\texttt{Nat}}$ such that $\mathit{Val}_{\texttt{Nat}} = \Phi_{\texttt{Nat}}(\mathit{Val}_{\texttt{Nat}})$. We will take $\mathit{Val}_{\texttt{Nat}}$ to be the least fixed point, $\mu(\Phi_{\texttt{Nat}})$, given by Theorem **??**.

Exercise 6.2
Check that the operator $\Phi_{\texttt{Nat}}$ *is monotone, i.e., if* $X \subseteq Y$*, then* $\Phi_{\texttt{Nat}}(X) \subseteq \Phi_{\texttt{Nat}}(Y)$.

[1] We may either take $\texttt{let}\, x{:}\tau\, \texttt{be}\, e_1\, \texttt{in}\, e_2$ as a primitive sequencing form, or as an abbreviation for the obvious function application. This is a technical device to ensure that we only ever substitute values for variables.

We are now in a position to prove the termination theorem for system **T**.

Theorem 6.3
Suppose that $\Gamma \vdash e : \tau$. *If* $\gamma \in \mathit{Val}_\Gamma$, *then* $\hat{\gamma}(e) \in \mathit{Comp}_\tau$

Proof: We consider only the numerals and the recursor. The other cases follow as before. We write $\hat{e}$ for $\hat{\gamma}(e)$ wherever the meaning is clear from context.

- $\Gamma \vdash \mathtt{0} : \mathtt{Nat}$. Clearly $\mathtt{0} \in \mathit{Val}_{\mathtt{Nat}} \subseteq \mathit{Comp}_{\mathtt{Nat}}$.
- $\Gamma \vdash \mathtt{succ}(e) : \mathtt{Nat}$ by $\Gamma \vdash e : \mathtt{Nat}$. By induction we have that $\hat{e} \in \mathit{Comp}_{\mathtt{Nat}}$, so $\hat{e} \mapsto^* v \in \mathit{Val}_{\mathtt{Nat}}$. Hence $\mathtt{succ}(\hat{e}) \mapsto^* \mathtt{succ}(v) \in \mathit{Val}_{\mathtt{Nat}}$, as required.
- $\Gamma \vdash \mathtt{natrec}_\tau\, e\, \mathtt{of}\, \mathtt{0} \,\mathtt{=>}\, e_0 \,|\, \mathtt{succ}(x{:}\tau) \,\mathtt{=>}\, e_s\, \mathtt{end} : \tau$ by $\Gamma \vdash e : \mathtt{Nat}$, $\Gamma \vdash e_0 : \tau$, and $\Gamma[x{:}\tau] \vdash e_s : \tau$. By inductive hypothesis we have that $\hat{e} \in \mathit{Comp}_{\mathtt{Nat}}$, $\hat{e}_0 \in \mathit{Comp}_\tau$, and $\{v/x\}\hat{e}_s \in \mathit{Comp}_\tau$ whenever $v \in \mathit{Val}_{\mathtt{Nat}}$. We are to show that
$$\mathtt{natrec}_\tau\, \hat{e}\, \mathtt{of}\, \mathtt{0} \,\mathtt{=>}\, \hat{e}_0 \,|\, \mathtt{succ}(x{:}\tau) \,\mathtt{=>}\, \hat{e}_s\, \mathtt{end} \in \mathit{Comp}_\tau\,.$$
Since $\hat{e} \in \mathit{Comp}_{\mathtt{Nat}}$, there exists a value $v \in \mathit{Val}_{\mathtt{Nat}}$ such that $\hat{e} \mapsto^* v$. We show that for every $v \in \mathit{Val}_{\mathtt{Nat}}$, the property $P(v)$ given by
$$\mathtt{natrec}_\tau\, v\, \mathtt{of}\, \mathtt{0} \,\mathtt{=>}\, \hat{e}_0 \,|\, \mathtt{succ}(x{:}\tau) \,\mathtt{=>}\, \hat{e}_s\, \mathtt{end} \in \mathit{Comp}_\tau$$
holds, from which the result follows immediately.
Since $\mathit{Val}_{\mathtt{Nat}} = \mu(\Phi_{\mathtt{Nat}})$, it suffices to show P is $\Phi_{\mathtt{Nat}}$-closed, which is to say that $\Phi_{\mathtt{Nat}}(P) \subseteq P$ (regarding P as a subset of $\mathit{Val}_{\mathtt{Nat}}$). This means that we must show that $P(\mathtt{0})$ and if $P(v)$, then $P(\mathtt{succ}(v))$. For the former, observe that
$$\mathtt{natrec}_\tau\, \mathtt{0}\, \mathtt{of}\, \mathtt{0} \,\mathtt{=>}\, \hat{e}_0 \,|\, \mathtt{succ}(x{:}\tau) \,\mathtt{=>}\, \hat{e}_s\, \mathtt{end} \mapsto \hat{e}_0$$
from which the result follows by the inductive hypothesis. Turning to the latter, we assume that
$$\mathtt{natrec}_\tau\, v\, \mathtt{of}\, \mathtt{0} \,\mathtt{=>}\, \hat{e}_0 \,|\, \mathtt{succ}(x{:}\tau) \,\mathtt{=>}\, \hat{e}_s\, \mathtt{end} \in \mathit{Comp}_\tau$$
and argue that
$$\mathtt{natrec}_\tau\, \mathtt{succ}(v)\, \mathtt{of}\, \mathtt{0} \,\mathtt{=>}\, \hat{e}_0 \,|\, \mathtt{succ}(x{:}\tau) \,\mathtt{=>}\, \hat{e}_s\, \mathtt{end} \in \mathit{Comp}_\tau\,.$$
The latter expression steps to
$$\mathtt{let}\, x{:}\tau\, \mathtt{be}\, \mathtt{natrec}_\tau\, v\, \mathtt{of}\, \mathtt{0} \,\mathtt{=>}\, \hat{e}_0 \,|\, \mathtt{succ}(x{:}\tau) \,\mathtt{=>}\, \hat{e}_s\, \mathtt{end}\, \mathtt{in}\, \hat{e}_s.$$
By our assumption $P(v)$, there exists $w \in \mathit{Val}_{\mathtt{Nat}}$ such that
$$\mathtt{natrec}_\tau\, v'\, \mathtt{of}\, \mathtt{0} \,\mathtt{=>}\, \hat{e}_0 \,|\, \mathtt{succ}(x{:}\tau) \,\mathtt{=>}\, \hat{e}_s\, \mathtt{end} \mapsto^* w.$$
Therefore
$$\mathtt{let}\, x{:}\tau\, \mathtt{be}\, \mathtt{natrec}_\tau\, v'\, \mathtt{of}\, \mathtt{0} \,\mathtt{=>}\, \hat{e}_0 \,|\, \mathtt{succ}(x{:}\tau) \,\mathtt{=>}\, \hat{e}_s\, \mathtt{end}\, \mathtt{in}\, \hat{e}_s \mapsto^* \{w/x\}\hat{e}_s.$$
But by the outer inductive hypothesis $\{w/x\}\hat{e}_s \in \mathit{Comp}_\tau$, from which the result follows by closure under head expansion.

∎

Exercise 6.4
Generalize the foregoing to the type of Brouwer ordinals, $\mathtt{Ord}$, *which have as introductory forms* $\mathtt{0}$, $\mathtt{succ}(e)$, *where* $e : \mathtt{Ord}$, *and* $\mathtt{sup}(e)$, *where* $e : \mathtt{Nat}{\rightarrow}\mathtt{Ord}$. *Ordinal recursion is expressed by the form*

$$\mathtt{ordrec}_\tau\, e\, \mathtt{of}\; 0 \mathtt{=>} e_0 \mid \mathtt{succ}(x{:}\tau) \mathtt{=>} e_s \mid \mathtt{sup}(y{:}\mathtt{Nat}{\rightarrow}\tau) \mathtt{=>} e_l\; \mathtt{end} : \tau$$

where $e : \mathtt{Ord}$, $e_0 : \tau$, $e_s : \tau$, *assuming* $x : \tau$, *and* $e_l : \tau$, *assuming* $y : \mathtt{Nat}{\rightarrow}\tau$. *Writing* $\mathcal{O}(e)$ *for*

$$\mathtt{ordrec}_\tau\, e\, \mathtt{of}\; 0 \mathtt{=>} e_0 \mid \mathtt{succ}(x{:}\tau) \mathtt{=>} e_s \mid \mathtt{sup}(y{:}\mathtt{Nat}{\rightarrow}\tau) \mathtt{=>} e_l\; \mathtt{end},$$

the primitive reduction steps are these:

$$\begin{array}{rcl} \mathcal{O}(\mathtt{0}) & \rightsquigarrow & e_0 \\ \mathcal{O}(\mathtt{succ}(v)) & \rightsquigarrow & \mathtt{let}\, x{:}\tau\, \mathtt{be}\, \mathcal{O}(v)\, \mathtt{in}\, e_s \\ \mathcal{O}(\mathtt{sup}(v)) & \rightsquigarrow & \mathtt{let}\, y{:}\mathtt{Nat}{\rightarrow}\tau\, \mathtt{be}\, \mathtt{fun}\,(n{:}\mathtt{Nat}){:}\tau\; \mathtt{in}\, \mathcal{O}(v(n))\, \mathtt{in}\, e_l \end{array}$$

Prove termination for the language of Brouwer ordinals.

6.3 Mendler's S

Mendler's system $\mathbf{S}$[2] is the extension $\mathcal{L}^{\mathtt{nat},\mathtt{stream},\rightarrow}$ of $\mathcal{L}^{\rightarrow}$ with a type of streams of natural numbers, equipped with operations for creating and manipulating streams. System $\mathbf{S}$ is an example of a language with a *coinductive* type.

The syntax of $\mathbf{S}$ is given by the following grammar:

$$\begin{array}{llll} \textit{Types} & \tau & ::= & \ldots \mid \mathtt{Stream} \\ \textit{Expressions} & e & ::= & \ldots \mid \mathtt{hd}(e) \mid \mathtt{tl}(e) \mid \\ & & & \mathtt{stream}_\tau\, \mathtt{hd}(x{:}\tau) \mathtt{=>} e_h \,\&\, \mathtt{tl}(y{:}\tau) \mathtt{=>} e_t\; \mathtt{from}\, e\; \mathtt{end} \end{array}$$

In the expression $\mathtt{stream}_\tau\, \mathtt{hd}(x{:}\tau) \mathtt{=>} e_h \,\&\, \mathtt{tl}(y{:}\tau) \mathtt{=>} e_t\; \mathtt{from}\, e\; \mathtt{end}$ the variable x is bound in e_h and the variable y is bound in e_t.

The meaning of these constructs is made precise by the following primitive instructions:

$$\begin{array}{l} \mathtt{hd}(\mathtt{stream}_\tau\, \mathtt{hd}(x{:}\tau) \mathtt{=>} e_h \,\&\, \mathtt{tl}(y{:}\tau) \mathtt{=>} e_t\; \mathtt{from}\, v\; \mathtt{end}) \;\rightsquigarrow\; \{v/x\}e_h \\ \mathtt{tl}(\mathtt{stream}_\tau\, \mathtt{hd}(x{:}\tau) \mathtt{=>} e_h \,\&\, \mathtt{tl}(y{:}\tau) \mathtt{=>} e_t\; \mathtt{from}\, v\; \mathtt{end}) \;\rightsquigarrow \\ \qquad\qquad \mathtt{stream}_\tau\, \mathtt{hd}(x{:}\tau) \mathtt{=>} e_h \,\&\, \mathtt{tl}(y{:}\tau) \mathtt{=>} e_t\; \mathtt{from}\, \{v/y\}e_t\; \mathtt{end} \end{array}$$

where evaluation contexts and values are defined as follows:

$$\begin{array}{llll} \textit{Values} & v & ::= & \ldots \mid \mathtt{stream}_\tau\, \mathtt{hd}(x{:}\tau) \mathtt{=>} e_h \,\&\, \mathtt{tl}(y{:}\tau) \mathtt{=>} e_t\; \mathtt{from}\, v\; \mathtt{end} \\ \textit{EvalCtxt's} & E & ::= & \ldots \mid \mathtt{hd}(E) \mid \mathtt{tl}(E) \mid \mathtt{stream}_\tau\, \mathtt{hd}(x{:}\tau) \mathtt{=>} e_h \,\&\, \mathtt{tl}(y{:}\tau) \mathtt{=>} e_t\; \mathtt{from}\, E\; \mathtt{end} \end{array}$$

[2] Not so-called by Mendler, but in light of Section 6.2 the terminology is irresistable.

Exercise 6.5
State and prove soundness for the the language **S***.*

Perhaps surprisingly, the combination of **T** and **S** is terminating.[3] The proof relies on the construction of the set $\mathit{Val}_{\texttt{Stream}}$ as the *largest* set of closed values v of type `Stream` such that $\texttt{hd}(v) \in \mathit{Comp}_{\texttt{Nat}}$ and $\texttt{tl}(v) \in \mathit{Comp}_{\texttt{Stream}}$. Written as a formula:

$$\mathit{Val}_{\texttt{Stream}} = \{\, v : \texttt{Stream} \mid \texttt{hd}(v) \in \mathit{Comp}_{\texttt{Nat}} \wedge \texttt{tl}(v) \Downarrow v' \in \mathit{Comp}_{\texttt{Stream}} \,\}$$

As with $\mathit{Val}_{\texttt{Nat}}$, we regard this as a recursion equation whose solution is a fixed point of the monotone operator

$$\Phi_{\texttt{Stream}}(X) = \{\, v : \texttt{Stream} \mid \texttt{hd}(v) \in \mathit{Comp}_{\texttt{Nat}} \wedge \texttt{tl}(v) \Downarrow v' \in X \,\}.$$

We take $\mathit{Val}_{\texttt{Stream}}$ to be the *greatest* fixed point of $\Phi_{\texttt{Stream}}$ given by Theorem **??**. (The least fixed point is the empty set!)

Exercise 6.6
Check that $\Phi_{\texttt{Stream}}$ *is monotone.*

Theorem 6.7
If $\Gamma \vdash e : \tau$*, and* $\gamma \in \mathit{Val}_\Gamma$*, then* $\hat{\gamma}(e) \in \mathit{Comp}_\tau$*.*

Proof: hWe write $\hat{e}$ for $\hat{\gamma}(e)$. We consider here only the interesting cases.

- If $\hat{e} = \texttt{hd}(\hat{e}_1)$, then we have by IH that $\hat{e}_1 \in \mathit{Comp}_{\texttt{Stream}}$, so $\texttt{hd}(\hat{e}_1) \in \mathit{Comp}_{\texttt{Nat}}$, and similarly for $\hat{e} = \texttt{tl}(\hat{e}_1)$.
- Suppose that $\hat{e} = \texttt{stream}_\tau\, \texttt{hd}(x{:}\tau) \texttt{=>} \hat{e}_h \,\&\, \texttt{tl}(y{:}\tau) \texttt{=>} \hat{e}_t\, \texttt{from}\, \hat{e}_1\, \texttt{end}$. We are to show that $\hat{e} \in \mathit{Comp}_{\texttt{Stream}}$. We prove by coinduction that it is consistent to add to $\mathit{Val}_{\texttt{Stream}}$ all values of the form

  $$\texttt{stream}_\tau\, \texttt{hd}(x{:}\tau) \texttt{=>} \hat{e}_h \,\&\, \texttt{tl}(y{:}\tau) \texttt{=>} \hat{e}_t\, \texttt{from}\, v_1\, \texttt{end},$$

  where $e_1 \Downarrow v_1$ for some $e_1 \in \mathit{Comp}_\tau$. That is, if X is $\mathit{Val}_{\texttt{Stream}}$ unioned with all such values, then $X \subseteq \Phi_{\texttt{Stream}}(X)$. If so, then $X \subseteq \mathit{Val}_{\texttt{Stream}}$, and hence $\hat{e} \in \mathit{Comp}_{\texttt{Stream}}$, as required. To see this, observe that $\hat{e}_1 \in \mathit{Comp}_\tau$ by the inductive hypothesis, and hence $\hat{e}_1 \Downarrow v_1$ for some $v_1 \in \mathit{Val}_\tau$, and so $\hat{e}$ evaluates to

  $$\texttt{stream}_\tau\, \texttt{hd}(x{:}\tau) \texttt{=>} \hat{e}_h \,\&\, \texttt{tl}(y{:}\tau) \texttt{=>} \hat{e}_t\, \texttt{from}\, v_1\, \texttt{end} \in X.$$

  To establish consistency of X, it suffices to show that if

  $$v = \texttt{stream}_\tau\, \texttt{hd}(x{:}\tau) \texttt{=>} \hat{e}_h \,\&\, \texttt{tl}(y{:}\tau) \texttt{=>} \hat{e}_t\, \texttt{from}\, v_1\, \texttt{end},$$

[3] It is sometimes thought that streams are inherently "non-terminating" and that lazy evaluation is required to manipulate them. Mendler's work, outlined in this section, demonstrates that this is nonsense.

then $\mathtt{hd}(v) \in \mathit{Comp}_{\mathtt{Nat}}$ and $\mathtt{tl}(v) \Downarrow v' \in X$. The former follows from the outer inductive hypothesis, since $v_1 \in \mathit{Val}_\tau$. As for the latter, observe that

$$\mathtt{tl}(v) \mapsto \mathtt{stream}_\tau\, \mathtt{hd}(x{:}\tau) \mathrel{=>} \hat{e}_h \,\&\, \mathtt{tl}(y{:}\tau) \mathrel{=>} \hat{e}_t\, \mathtt{from}\, \{v_1/y\}\hat{e}_t\, \mathtt{end}.$$

By inductive hypothesis $\{v_1/y\}\hat{e}_t \in \mathit{Comp}_\tau$, and hence $\{v_1/y\}\hat{e}_t \Downarrow v_1'$ for some $v_1' \in \mathit{Val}_\tau$. But then

$$\mathtt{stream}_\tau\, \mathtt{hd}(x{:}\tau) \mathrel{=>} \hat{e}_h \,\&\, \mathtt{tl}(y{:}\tau) \mathrel{=>} \hat{e}_t\, \mathtt{from}\, \{v_1/y\}\hat{e}_t\, \mathtt{end}$$

evaluates to

$$\mathtt{stream}_\tau\, \mathtt{hd}(x{:}\tau) \mathrel{=>} \hat{e}_h \,\&\, \mathtt{tl}(y{:}\tau) \mathrel{=>} \hat{e}_t\, \mathtt{from}\, v_1'\, \mathtt{end} \in X,$$

which is sufficient for the result.

■

Corollary 6.8
If $e : \tau$, then $e \Downarrow$.

Remark 6.9
Gödel's **T** and Mendler's **S** may be generalized to a language with (respectively) *inductive* and *co-inductive* types, for which `Nat` and `Stream` are but special cases. The idea is simple — extend the language with recursive types subject to a positivity condition to ensure that the associated operator is monotone — but the technical details are surprisingly involved. We refer the reader to Mendler's dissertation [**?**] for a general treatment of inductive and co-inductive types in the context of the NuPRL type theory.

6.4 General Recursive Types

Both the type of natural numbers and the type of streams of natural numbers are examples of *recursive types* since their semantic definitions are "self-referential" and hence involve fixed points. The syntax may be seen as establishing the isomorphisms

$$\mathtt{Nat} \cong \mathtt{Unit}{+}\mathtt{Nat}$$

and

$$\mathtt{Stream} \cong \mathtt{Nat}{\times}\mathtt{Stream}.$$

Exercise 6.10
Write down the required isomorphism pairs and argue informally that they are indeed mutually inverse to one another.

This motivates a more general approach to recursive types based on the requirement that a recursive type should be isomorphic to its "unrolling". This approach generalizes the inductive and co-inductive types described above since it licenses *arbitrary* recursive types, rather than just those that induce monotone operators on value sets.

The syntax of this extension $\mathcal{L}^{\rightarrow,\mathtt{rec}}$ of $\mathcal{L}^{\rightarrow}$ is as follows:

$$\begin{array}{llll} \textit{Types} & \tau & ::= & \mathtt{rec}\,t\,\mathtt{is}\,\tau \\ \textit{Expressions} & e & ::= & \mathtt{in}_{\mathtt{rec}\,t\,\mathtt{is}\,\tau}(e) \mid \mathtt{out}_{\mathtt{rec}\,t\,\mathtt{is}\,\tau}(e) \end{array}$$

The static semantics is as follows:

$$\frac{\Delta[t] \vdash \tau}{\Delta \vdash \mathtt{rec}\,t\,\mathtt{is}\,\tau} \quad (t \notin \Delta) \tag{T-REC}$$

$$\frac{\Gamma \vdash_\Delta e : \{\mathtt{rec}\,t\,\mathtt{is}\,\tau/t\}\tau}{\Gamma \vdash_\Delta \mathtt{in}_{\mathtt{rec}\,t\,\mathtt{is}\,\tau}(e) : \mathtt{rec}\,t\,\mathtt{is}\,\tau} \tag{T-IN}$$

$$\frac{\Gamma \vdash_\Delta e : \mathtt{rec}\,t\,\mathtt{is}\,\tau}{\Gamma \vdash_\Delta \mathtt{out}_{\mathtt{rec}\,t\,\mathtt{is}\,\tau}(e) : \{\mathtt{rec}\,t\,\mathtt{is}\,\tau/t\}\tau} \tag{T-OUT}$$

The dynamic semantics is given by the following grammar of values and evaluation contexts:

$$\begin{array}{llll} \textit{Values} & v & ::= & \mathtt{in}_{\mathtt{rec}\,t\,\mathtt{is}\,\tau}(v) \\ \textit{EvalCtxt's} & E & ::= & \mathtt{out}_{\mathtt{rec}\,t\,\mathtt{is}\,\tau}(E) \end{array}$$

and the following primitive instruction:

$$\mathtt{out}_{\mathtt{rec}\,t\,\mathtt{is}\,\tau}(\mathtt{in}_{\mathtt{rec}\,t\,\mathtt{is}\,\tau}(v)) \leadsto v$$

Exercise 6.11
Check the soundness of this extension.

There is no question of termination for this extension: if we add general recursive types to $\mathcal{L}^{\rightarrow}$ we introduce non-terminating computations into the language. Thus recursive types may only be properly added to $\mathcal{L}^{\rightharpoonup}$ (or its extensions) with partial function types and the possibility of non-termination.

Exercise 6.12
Let τ *be the type* $\mathtt{rec}\,t\,\mathtt{is}\,t{\rightarrow}t$*. Let* Ω *be the expression* $\mathtt{fun}\,(x{:}\tau){:}\tau\,\mathtt{in}\,\mathtt{out}(x)(x)$*. Check that* Ω *has type* $\tau{\rightarrow}\tau$*, so that* $\mathtt{in}(\Omega)$ *has type* τ*. Show that the expression* $\Omega(\mathtt{in}(\Omega))$ *does not terminate.*

6.5 References

[?, ?, ?, ?]

Chapter 7

Polymorphism

7.1 Introduction

Polymorphism is the ability to abstract expressions over types (more generally, type constructors). Such expressions have *universally quantified*, or *polymorphic*, types. There are two main varieties of polymorphism. The *predicative* variant is characterized by the limitation of quantifiers to range only over unquantified types; the *impredicative* has no such restriction. We begin with the predicative variant, in the context of the "explicit" formulation of variable types described in Section 5.4.

7.2 Statics

The language $\mathcal{L}_{\Omega}^{\rightarrow,\forall_{\mathsf{p}}}$ is the extension of $\mathcal{L}^{\rightarrow}$ with predicative type abstraction. We work in the "explicit" framework of Chapter 5.

Kinds	κ	$::=$	Ω
Constructors	μ	$::=$	$t \mid \mu_1 \rightarrow \mu_2$
Types	σ	$::=$	$T(\sigma) \mid \sigma_1 \rightarrow \sigma_2 \mid \forall(t)\sigma$
Expressions	e	$::=$	$x \mid \mathtt{fun}\,(x{:}\tau)\,\mathtt{in}\,e \mid e_1(e_2) \mid \mathtt{Fun}\,(t)\,\mathtt{in}\,e \mid e[\mu]$
Type Contexts	Δ	$::=$	$\emptyset \mid \Delta[t]$
Term Contexts	Γ	$::=$	$\emptyset \mid \Gamma[x{:}\sigma]$

The types consist of the inclusions of constructors of kind Ω, and are closed under formation of function spaces and universal quantification. The expressions are those of $\mathcal{L}^{\rightarrow}$, extended with *polymorphic abstractions*, $\mathtt{Fun}\,(t)\,\mathtt{in}\,e$, and *polymorphic applications* (or *instantiations*), $e[\mu]$.

It is also possible to enrich other base languages with polymorphism; we consider $\mathcal{L}^{\rightarrow}$ as the base language for the sake of specificity.

The judgement forms of $\mathcal{L}_{\Omega}^{\rightarrow,\forall_{\mathrm{p}}}$ are as follows:

$\Delta \vdash \mu :: \kappa$	*μ has kind κ over Δ*
$\Delta \vdash \mu_1 \equiv \mu_2 :: \kappa$	*μ_1 and μ_2 are equivalent constructors of kind κ*
$\vdash_{\Delta} \sigma$	*σ is a valid type over Δ*
$\vdash_{\Delta} \sigma_1 \equiv \sigma_2$	*σ_1 and σ_2 are equivalent types*
$\Gamma \vdash_{\Delta} e : \sigma$	*e has type σ relative to Γ and Δ*

In addition we define $\vdash_{\Delta} \Gamma$ to mean $\vdash_{\Delta} \Gamma(x)$ for every $x \in \mathrm{dom}\,\Gamma$.

The rules for constructor formation and equivalence are as in Section 5.4. The rules for type formation and equivalence are those of Section 5.4, extended with the following rules:

$$\frac{\vdash_{\Delta[t]} \sigma}{\vdash_{\Delta} \forall(t)\sigma} \quad (t \notin \Delta) \tag{T-ALL}$$

$$\frac{\vdash_{\Delta[t]} \sigma \equiv \sigma'}{\vdash_{\Delta} \forall(t)\sigma \equiv \forall(t)\sigma'} \tag{E-ALL}$$

The typing rules for the distinctive expressions of $\mathcal{L}_{\Omega}^{\rightarrow,\forall_{\mathrm{p}}}$ are defined as follows:

$$\frac{\Gamma \vdash_{\Delta[t]} e : \sigma}{\Gamma \vdash_{\Delta} \mathtt{Fun}\,(t)\,\mathtt{in}\,e : \forall(t)\sigma} \quad (t \notin \mathrm{dom}(\Delta)) \tag{T-TABS}$$

$$\frac{\Gamma \vdash_{\Delta} e : \forall(t)\sigma \qquad \Delta \vdash \mu}{\Gamma \vdash_{\Delta} e[\mu] : \{\mu/t\}\sigma} \tag{T-TAPP}$$

We also include the rule t-equiv from Chapter 5 to ensure that typing respects equivalence of types.

7.3 Dynamics

The operational semantics of $\mathcal{L}_{\Omega}^{\rightarrow,\forall_{\mathrm{p}}}$ is defined as follows:

$$\begin{array}{llcl} \textit{Values} & v & ::= & x \mid \mathtt{fun}\,(x{:}\sigma)\,\mathtt{in}\,e \mid \mathtt{Fun}\,(t)\,\mathtt{in}\,e \\ \textit{Eval Ctxts} & E & ::= & \bullet \mid E(e) \mid v(E) \mid E[\mu] \end{array}$$

The evaluation relation $e \mapsto e'$ is defined for closed expressions e and e' iff $e = E[r]$ and $e' = E[c]$, where $r \leadsto c$ according to the following rules:

$$\begin{array}{rcl} (\mathtt{fun}\,(x{:}\sigma)\,\mathtt{in}\,e)(v) & \leadsto & \{v/x\}e \\ (\mathtt{Fun}\,(t)\,\mathtt{in}\,e)[\mu] & \leadsto & \{\mu/t\}e \end{array}$$

Exercise 7.1
Prove the soundness of the operational semantics for $\mathcal{L}_{\Omega}^{\rightarrow,\forall_{\mathrm{p}}}$ given above by stating and proving progress and preservation theorems for this interpretation.

A useful technical property of types in $\mathcal{L}_{\Omega}^{\rightarrow,\forall_\mathsf{p}}$ is that $\{\mu/t\}\sigma$ is always "smaller" than $\forall(t)\sigma$, regardless of how often (if at all) t occurs in σ. Specifically, substitution into the matrix of a quantified type strictly reduces the number of quantifiers in the resulting type, perhaps at the expense of increasing its overall length. This property is not shared by the impredicative variant of $\mathcal{L}_{\Omega}^{\rightarrow,\forall_\mathsf{p}}$ (presented below), and may be understood as the distinguishing feature of predicative polymorphism.

With this in mind, the termination proof for $\mathcal{L}^{\rightarrow}$ given in Chapter 2 may be extended to $\mathcal{L}_{\Omega}^{\rightarrow,\forall_\mathsf{p}}$ by associating predicates Val_σ and $Comp_\sigma$ to each closed type σ in such a way that definitional equivalence is respected. In the present case respect for definitional equivalence can be achieved by considering types in $T(-)$ normal form in which $T(\tau)$ occurs only when τ is a type variable. We may then define the interpretation of normal form quantified types by the equation

$$Val_{\forall(t)\sigma}(v) \textit{ iff } \mu :: \Omega \textit{ implies } Comp_{\{\mu/t\}\sigma}(v[\mu]).$$

It is easy to check that the system of predicates obtained by extending the definition in Chapter 2 with this clause is well-defined.

To prove termination, let us define a *closed type substitution* for a type context Δ to be a finite function δ assigning a closed type expression to each $t \in \mathrm{dom}(\Delta)$.

Theorem 7.2
If $\Gamma \vdash_\Delta e : \sigma$, then $Comp_{\hat{\delta}(\sigma)}(\hat{\gamma}(\hat{\delta}(e)))$ for every closed type substitution δ for Δ and every closed value substitution γ such that $Val_{\hat{\delta}(\Gamma)}(\gamma)$.

Exercise 7.3
Prove Theorem 7.2 by induction on typing derivations. Be sure to take account of the rule t-equiv!

7.4 Impredicative Polymorphism

The distinction between predicative and impredicative polymorphism may be located in the separation between constructors and types. While every constructor μ (of kind Ω) "names" a type $T(\mu)$, not every type arises in this way. Since type variables range only over constructors, quantified types are excluded from the domain of quantification. By ensuring that every type is named by some constructor — or, what is the same, doing away with the distinction entirely — we arrive at the so-called *impredicative* variant of the system, called $\mathcal{L}_{\Omega}^{\rightarrow,\forall}$. This system is known variously as Girard's *System F*, or Reynolds' *polymorphic λ-calculus*, or the *second-order typed λ-calculus*.[1]

The passage to impredicative quantification may be formalized by enriching the constructor level of $\mathcal{L}_{\Omega}^{\rightarrow,\forall_\mathsf{p}}$ with a universal quantifier, say $\texttt{all}\, t\, \texttt{in}\, \mu$. Its

[1]The qualifier "second order" stems from the close connection between $\mathcal{L}_{\Omega}^{\rightarrow,\forall}$ and second-order arithmetic.

interpretation is determine by the definitional equivalence

$$\Delta \vdash\ T\,(\texttt{all}\,t\,\texttt{in}\,\mu) \equiv \forall(t)\,T\,(\mu).$$

With this addition every type is equivalent to a type of the form $T\,(\mu)$ for some constructor μ.

A more streamlined presentation of $\mathcal{L}_{\Omega}^{\rightarrow,\forall}$ may be obtained by eliminating altogether the distinction between constructors and types. This leads to the following grammar for $\mathcal{L}_{\Omega}^{\rightarrow,\forall}$:

Kinds	κ	$::=$	Ω
Types	σ	$::=$	$t \mid \sigma_1 \rightarrow \sigma_2 \mid \forall(t)\sigma$
Expressions	e	$::=$	$x \mid \texttt{fun}\,(x{:}\sigma)\,\texttt{in}\,e \mid e_1(e_2) \mid \texttt{Fun}\,(t)\,\texttt{in}\,e \mid e[\sigma]$

In this system types are classified by kinds, and expressions are classified by types; in effect, we identify types and constructors. The judgement forms are similar to those of the predicative variant, with the change that type formation and equivalence are written $\Delta \vdash \sigma :: \kappa$ and $\Delta \vdash \sigma_1 \equiv \sigma_2 :: \kappa$, respectively, and the judgements governing constructors are eliminated.

Exercise 7.4
Prove that the streamlined presentation is equivalent to the presentation based on an explicit inclusion of constructors as types.

For the remainder of this section we will work with the streamlined presentation of $\mathcal{L}_{\Omega}^{\rightarrow,\forall}$.

7.4.1 Termination

Although the identification of constructors and types leads to a *syntactically* simpler system, it is by no means *semantically* simpler. In particular quantified types may now be substituted for free type variables, and so it is no longer the case that $\{\sigma/t\}\sigma'$ is "simpler" than $\forall(t)\sigma'$. For example, if $\sigma = \forall(t)t$ and $\sigma' = t$, then $\{\sigma/t\}t = \sigma$ — a quantified type can have itself as instance. Worse, a quantified type such as $\sigma = \forall(t)t \rightarrow t$ has as instance $\sigma \rightarrow \sigma$, which has *more* occurrences of quantifiers than σ itself. These phenomena can be traced to a kind of "circularity" inherent in the notion of impredicative quantification: the quantifier ranges over *all* types, including the quantified type itself.

One consequence is that the termination proof for $\mathcal{L}_{\Omega}^{\rightarrow,\forall_{\mathsf{p}}}$ cannot easily be extended to $\mathcal{L}_{\Omega}^{\rightarrow,\forall}$ — since quantifier instantiation does not reduce the complexity of a type, some other treatment of the quanifier is required. A solution to this problem — due to Girard — is to broaden the interpretation of the quantifier to range over a wider class of predicates than are denotable by closed type expressions. In other words, the quantifier is interpreted "semantically", rather than "syntactically", thereby avoiding the problem of circularity inherent in the impredicative framework. This technique is known as *Girard's candidates method.*

Following Girard, we employ a technical device to avoid complications with empty types.[2] We extend the language $\mathcal{L}_{\Omega}^{\rightarrow,\forall}$ with constants $\mathbf{0}_\sigma$ of type σ for every type σ. These constants are considered to be values of the appropriate type. At higher type they are subject to the following reduction rules:

$$\begin{array}{rcl} \mathbf{0}_{\sigma_1\rightarrow\sigma_2}(v) & \leadsto & \mathbf{0}_{\sigma_2} \\ \mathbf{0}_{\forall(t)\sigma}[\sigma_0] & \leadsto & \mathbf{0}_{\{\sigma_0/t\}\sigma} \end{array}$$

Thus we may think of $\mathbf{0}_{\sigma_1\rightarrow\sigma_2}$ as the function $\mathtt{fun}\,(x{:}\sigma_1)\,\mathtt{in}\,\mathbf{0}_{\sigma_2}$, and of $\mathbf{0}_{\forall(t)\sigma}$ as $\mathtt{Fun}\,(t)\,\mathtt{in}\,\mathbf{0}_\sigma$. We will prove termination for this extension of $\mathcal{L}_{\Omega}^{\rightarrow,\forall}$, from which termination of the language without these constants follows immediately.

A *candidate* for a closed type σ is a (non-empty) set of closed values of type σ such that $\mathbf{0}_\sigma \in T$ for every $T \in \mathit{Cand}_\sigma$.[3] Let Cand_σ be the set of all candidates for type σ.

We will need three operations on candidates to model computations, function spaces, and quantification.

Definition 7.5

1. *If $T \in \mathit{Cand}_\sigma$, then $[\overline{T}]_\sigma = \{\, e : \sigma \mid e \Downarrow v \in T \,\}$.*

2. *If $T_1 \in \mathit{Cand}_{\sigma_1}$ and $T_2 \in \mathit{Cand}_{\sigma_2}$, then*

$$[T_1 \rightarrow T_2]_{\sigma_1\rightarrow\sigma_2} = \{\, v : \sigma_1\rightarrow\sigma_2 \mid \forall v_1 \in T_1,\ v(v_1) \Downarrow v_2 \in T_2 \,\}.$$

3. *If $C_\sigma(T) \in \mathit{Cand}_{\{\sigma/t\}\sigma'}$ for every closed type σ and every $T \in \mathit{Cand}_\sigma$, then*

$$\begin{array}{l} [\prod_{\sigma\ \mathit{closed}} \prod_{T\in \mathit{Cand}_\sigma} C_\sigma(T)]_{\forall(t)\sigma'} = \\ \{\, v : \forall(t)\sigma' \mid \forall\sigma\ \mathit{closed},\ v[\sigma] \Downarrow v \in \bigcap_{T\in \mathit{Cand}_\sigma} C_\sigma(T) \,\}. \end{array}$$

Lemma 7.6

1. *If $T_1 \in \mathit{Cand}_{\sigma_1}$ and $T_2 \in \mathit{Cand}_{\sigma_2}$, then $[T_1 \rightarrow T_2]_{\sigma_1\rightarrow\sigma_2} \in \mathit{Cand}_{\sigma_1\rightarrow\sigma_2}$.*

2. *If $C_\sigma(T) \in \mathit{Cand}_{\{\sigma/t\}\sigma'}$ for every closed type σ and every $T \in \mathit{Cand}_\sigma$, then $[\prod_\sigma \prod_{T\in \mathit{Cand}_\sigma} C_\sigma(T)]_{\forall(t)\sigma'} \in \mathit{Cand}_{\forall(t)\sigma'}$.*

Proof:

1. Clearly $[T_1 \rightarrow T_2]_{\sigma_1\rightarrow\sigma_2}$ is a set of values of type $\sigma_1\rightarrow\sigma_2$. To show that it is non-empty, we show that $\mathbf{0}_{\sigma_1\rightarrow\sigma_2}$ is a member of this set. To this end suppose that $v_1 \in T_1$. We are to show that $\mathbf{0}_{\sigma_1\rightarrow\sigma_2}(v_1) \in [\overline{T_2}]_{\sigma_2}$. By definition of the reduction rules governing the "zero" elements, $\mathbf{0}_{\sigma_1\rightarrow\sigma_2}(v_1) \Downarrow \mathbf{0}_{\sigma_2}$. But $\mathbf{0}_{\sigma_2} \in T_2$ since T_2 is a candidate for σ_2.

2. (Left as an exercise.)

[2]This device is not strictly necessary, but it simplifies the argument in a few spots.

[3]If we were to add base types such as `Int` or `Bool` to the language, we would further assume that the constants of a type are elements of each candidate for that type.

■

Exercise 7.7
Complete the proof of Lemma 7.6.

Definition 7.8
The interpretations $|\Delta \vdash \sigma|\delta\xi$ *and* $||\Delta \vdash \sigma||\delta\xi$*, where* δ *is a closed type substitution for* Δ *and* ξ *is an assignment of a candidate for* $\delta(t)$ *to each variable* t *in* Δ*, are defined by induction on the structure of types as follows:*

$$\begin{array}{rcl} |\Delta \vdash \sigma|\delta\xi & = & [\overline{T}]_{\hat{\delta}(\sigma)}, \textit{ where} \\ T & = & ||\Delta \vdash \sigma||\delta\xi \\ \\ ||\Delta \vdash t||\delta\xi & = & \xi(t) \\ \\ ||\Delta \vdash \sigma_1 {\rightarrow} \sigma_2||\delta\xi & = & [T_1 \rightarrow T_2]_{\hat{\delta}(\sigma_1) \rightarrow \hat{\delta}(\sigma_2)}, \textit{ where} \\ T_1 & = & ||\Delta \vdash \sigma_1||\delta\xi \textit{ and} \\ T_2 & = & ||\Delta \vdash \sigma_1||\delta\xi \\ \\ ||\Delta \vdash \forall(t)\sigma'||\delta\xi & = & [\prod_\sigma \prod_{T \in Cand_\sigma} C_\sigma(T)]_{\forall(t)\hat{\delta}(\sigma')}, \textit{ where} \\ C_\sigma(T) & = & ||\Delta[t] \vdash \sigma'||\delta[t \mapsto \sigma]\xi[t \mapsto T] \ (\sigma \textit{ closed}, \ T \in Cand_\sigma) \end{array}$$

Lemma 7.9
If $\Delta \vdash \sigma$*, then* $||\Delta \vdash \sigma||\delta\xi \in Cand_{\hat{\delta}(\sigma)}$ *for every closed type substitution* δ *for* Δ *and every candidate assignment* ξ *for* δ*.*

Proof: By induction on the structure of types. The result follows immediately from Definition 7.8 and Lemma 7.6. ■

In effect $||\Delta \vdash \sigma||\delta\xi$ picks out one candidate among many as the "meaning" of the type σ, relative to the meanings of the types in Δ.

Exercise 7.10
Prove Lemma 7.9 by induction on the structure of types.

Note that by Definition 7.8 and Lemma 7.9, if $v \in ||\Delta \vdash \forall(t)\sigma'||\delta\xi$, then for every closed type σ,

$$v[\sigma] \in |\Delta[t] \vdash \sigma'|\delta[t \mapsto \sigma]\xi[t \mapsto ||\sigma||\emptyset\emptyset],$$

but this condition is not reversible.

Lemma 7.11
The interpretation of types is compositional:

$$||\Delta \vdash \{\sigma/t\}\sigma'||\delta\xi = ||\Delta[t] \vdash \sigma'||\delta[t \mapsto \hat{\delta}(\sigma)]\xi[t \mapsto ||\Delta \vdash \sigma||\delta\xi]$$

Proof: Straightforward induction on the structure of types. ■

Exercise 7.12
Prove Lemma 7.11.

Theorem 7.13
If $\Gamma \vdash_\Delta e : \sigma$*, then* $\hat{\gamma}(\hat{\delta}(e)) \in |\Delta \vdash \sigma|\delta\xi$ *for every closed type substitution* δ *for* Δ*, every candidate assignment* ξ *for* δ*, and every closed value substitution* γ *such that* $\gamma(x) \in ||\Delta \vdash \Gamma(x)||\delta\xi$ *for every* $x \in \mathrm{dom}(\Gamma)$*.*

Proof (sketch): By induction on typing derivations. The case of an ordinary variable is covered by the assumption on γ. Ordinary and type abstractions are handled by closure under head expansion. Ordinary and type applications are handled by applying Lemma 7.6. Respect for type equivalence is assured because definitional equality of types is trivial in the present setting. ■

Exercise 7.14
Complete the proof of Theorem 7.13.

Corollary 7.15
If $\vdash e : \sigma$*, then there exists* v *such that* $e \Downarrow v$*.*

Proof: By considering the empty assignment of types and candidates to type variables and the empty assignment of values to expression variables, and applying Theorem 7.13, we determine that $e \in |\Delta \vdash \sigma|\emptyset\emptyset$, from which it follows that $e \Downarrow v$ for some value v. ■

Exercise 7.16
Consider extending $\mathcal{L}_\Omega^{\rightarrow,\forall}$ *with a constant* $\mathbf{J} : \forall(t)\forall(u)t{\rightarrow}u$ *with the following evaluation rules:*

$$\begin{array}{lcll} \mathbf{J}[\sigma_1][\sigma_2] & \leadsto & \mathtt{fun}\,(x{:}\sigma_1)\,\mathtt{in}\,x & \textit{if } \sigma_1 = \sigma_2 \\ \mathbf{J}[\sigma_1][\sigma_2] & \leadsto & \mathtt{fun}\,(x{:}\sigma_1)\,\mathtt{in}\,\mathbf{0}_{\sigma_2} & \textit{otherwise} \end{array}$$

Consider the term

$$v = \mathtt{Fun}\,(t)\,\mathtt{in}\,\mathbf{J}[\varepsilon{\rightarrow}\varepsilon][t](\mathtt{fun}\,(x{:}\varepsilon)\,\mathtt{in}\,x[\varepsilon{\rightarrow}\varepsilon](x))$$

of type $\varepsilon = \forall(t)t$ *(!).*

1. *Prove that this extension of* $\mathcal{L}_\Omega^{\rightarrow,\forall}$ *is not terminating by considering the evaluation of* $v[\varepsilon{\rightarrow}\varepsilon](v)$*.*

2. *Show that* $\mathbf{J} \notin ||\emptyset \vdash \forall(t)\forall(u)t{\rightarrow}u||\emptyset\emptyset$ *by arguing that otherwise* Cand_σ *would be a singleton for all closed types* σ*, which is a contradiction.*

7.4.2 Definability of Types

An intriguing property of $\mathcal{L}_\Omega^{\rightarrow,\forall}$ is that an impressive variety of types are definable within it. By a type being *definable* we mean that there is an encoding of the type and its introduction and elimination forms in $\mathcal{L}_\Omega^{\rightarrow,\forall}$ in such a way that the operational semantics is preserved.[4] We will illustrate here a few examples, and suggest a few exercises to give a flavor of what is involved.

We begin with product types. The product type is defined in $\mathcal{L}_\Omega^{\rightarrow,\forall}$ by the following equation:

$$\tau_1 \times \tau_2 = \forall(t)(\tau_1 \rightarrow \tau_2 \rightarrow t) \rightarrow t.$$

The idea is to think of an element of the product type as a polymorphic operation for computing a value of a given "answer" type in terms of a "continuation" that computes the answer in terms of the components of the product. Thus we define the pairing and projection operations as follows:

$$\begin{array}{rcl} <e_1,e_2>_{\tau_1,\tau_2} & = & \texttt{Fun}\,(t)\,\texttt{in}\,\texttt{fun}\,(f{:}\tau_1 \rightarrow \tau_2 \rightarrow t)\,\texttt{in}\,f(e_1)(e_2) \\ \texttt{proj}^i_{\tau_1,\tau_2}(e) & = & e[\tau_i](\texttt{fun}\,(x_1{:}\tau_1)\,\texttt{in}\,\texttt{fun}\,(x_2{:}\tau_2)\,\texttt{in}\,x_i) \end{array}$$

Observe that $\texttt{proj}^i_{\tau_1,\tau_2}(<v_1,v_2>_{\tau_1,\tau_2}) \rightsquigarrow^* v_i$ according to the rules of $\mathcal{L}_\Omega^{\rightarrow,\forall}$ — in other words, the operational semantics of product types is preserved by these definitions.

Exercise 7.17
Show that the `Unit` *type is definable in* $\mathcal{L}_\Omega^{\rightarrow,\forall}$.

By a similar pattern of reasoning we arrive at the following definition of the sum type in $\mathcal{L}_\Omega^{\rightarrow,\forall}$:

$$\begin{array}{rcl} \tau_1 + \tau_2 & = & \forall(t)(t_1 \rightarrow t) \rightarrow (t_2 \rightarrow t) \rightarrow t \\ \texttt{inl}_{\tau_1,\tau_2}(e) & = & \texttt{Fun}\,(t)\,\texttt{in}\,\texttt{fun}\,(f_1{:}\tau_1 \rightarrow t)\,\texttt{in}\,\texttt{fun}\,(f_2{:}\tau_2 \rightarrow t)\,\texttt{in}\,f_1(e) \\ \texttt{inr}_{\tau_1,\tau_2}(e) & = & \texttt{Fun}\,(t)\,\texttt{in}\,\texttt{fun}\,(f_1{:}\tau_1 \rightarrow t)\,\texttt{in}\,\texttt{fun}\,(f_2{:}\tau_2 \rightarrow t)\,\texttt{in}\,f_2(e) \end{array}$$

$$\begin{array}{l} \texttt{case}_\tau\;e\;\texttt{of}\;\texttt{inl}(x_1{:}\tau_1) \texttt{=>} e_1 \mid \texttt{inr}(x_2{:}\tau_2) \texttt{=>} e_2\;\texttt{esac} \\ \qquad = \\ \qquad\qquad e[\tau](\texttt{fun}\,(x_1{:}\tau_1)\,\texttt{in}\,e_1)(\texttt{fun}\,(x_2{:}\tau_2)\,\texttt{in}\,e_2) \end{array}$$

Exercise 7.18
Check that the operational semantics of sums is properly captured by these definitions. Show that the `Void` *type is definable in* $\mathcal{L}_\Omega^{\rightarrow,\forall}$.

[4] In many situations more is required than mere preservation of operational semantics. In many situations equational theories are considered, and it is required that the equations governing the defined type be derivable in the equational theory of $\mathcal{L}_\Omega^{\rightarrow,\forall}$.

The natural numbers are also definable in $\mathcal{L}_{\Omega}^{\rightarrow,\forall}$:

$$
\begin{aligned}
\texttt{Nat} &= \forall(t)t\to(t\to t)\to t \\
\texttt{0} &= \\
&\texttt{Fun}\,(t)\,\texttt{in}\,\texttt{fun}\,(z{:}t)\,\texttt{in}\,\texttt{fun}\,(s{:}t\to t)\,\texttt{in}\,z \\
\texttt{succ}(n) &= \\
&\texttt{Fun}\,(t)\,\texttt{in}\,\texttt{fun}\,(z{:}t)\,\texttt{in}\,\texttt{fun}\,(s{:}t\to t)\,\texttt{in}\,n[t](s(z))(s) \\
\texttt{natrec}_{\tau}\,n\,\texttt{of}\,\texttt{0}\,\texttt{=>}\,b\,|\,\texttt{succ}(x{:}\tau)\,\texttt{=>}\,s\,\texttt{end} &= t[n](b)(\texttt{fun}\,(x{:}\tau)\,\texttt{in}\,s(x))
\end{aligned}
$$

Exercise 7.19
Check that the operational semantics of the recursor is accurately mimicked by these definitions. Show that the type `Bool` *is definable in* $\mathcal{L}_{\Omega}^{\rightarrow,\forall}$ *by giving definitions for* `true`, `false`, *and* `if − then − else − fi`.

Exercise 7.20
Show that streams are definable in $\mathcal{L}_{\Omega}^{\rightarrow,\forall}$.

The question arises: which number-theoretic functions are definable in $\mathcal{L}_{\Omega}^{\rightarrow,\forall}$? In view of the termination theorem only total functions are definable, and it should be clear that all such functions are recursive (by an arithmetization of the operational semantics). A simple diagonal argument shows that not every total recursive function is definable in $\mathcal{L}_{\Omega}^{\rightarrow,\forall}$. But can we say more? Girard proved that the number-theoretic functions definable in $\mathcal{L}_{\Omega}^{\rightarrow,\forall}$ are precisely those that can be proved total in second-order arithmetic. The proof breaks into two parts. The termination proof for $\mathcal{L}_{\Omega}^{\rightarrow,\forall}$ can be formalized in second-order arithmetic, so the definable functions of $\mathcal{L}_{\Omega}^{\rightarrow,\forall}$ are provably total in this system. In the other direction we may define an "extraction" function that associates a term of $\mathcal{L}_{\Omega}^{\rightarrow,\forall}$ with proofs in second-order arithmetic which ensures that every provably total function is in fact represented in $\mathcal{L}_{\Omega}^{\rightarrow,\forall}$.

Impressive though it is, this fact must be kept in proper perspective. In particular, Girard's theorem says nothing about efficiency of encoding, neither in terms of size of program nor the complexity of the encoding. In fact, Parigot has established a linear time (!) lower bound for the predecessor operation in $\mathcal{L}_{\Omega}^{\rightarrow,\forall}$.

7.5 References

The polymorphic typed λ-calculus was introduced by Girard [**?**] and Reynolds [**?**]. Our presentation of the termination proof for $\mathcal{L}_{\Omega}^{\rightarrow,\forall}$ is based on Gallier's paper in the Odifreddi volume [**?**].

Chapter 8

Abstract Types

Support for data abstraction is provided by *existential types*, which are "dual" to the universal types associated with polymorphism.

8.1 Statics

The language $\mathcal{L}_{\Omega}^{\rightarrow,\forall_{\mathsf{p}},\exists_{\mathsf{p}}}$ is defined to be an enrichment of $\mathcal{L}_{\Omega}^{\rightarrow,\forall_{\mathsf{p}}}$ with the following additional constructs:

$$\begin{array}{lll} \textit{Types} & \sigma & ::= \quad \exists u{::}\kappa(\sigma) \\ \textit{Expressions} & e & ::= \quad \mathtt{pack}\,\mu\,\mathtt{with}\,e\,\mathtt{as}\,\sigma \mid \mathtt{open}_{\sigma'}\,e\,\mathtt{as}\,u{::}\kappa\,\mathtt{with}\,x{:}\sigma\,\mathtt{in}\,e' \end{array}$$

The variables u and x are bound in e', and, as usual, we identify expressions that differ only in the names of the bound variables.

The expression

$$\mathtt{open}_{\sigma}\,(\mathtt{pack}\,\mu\,\mathtt{with}\,e'\,\mathtt{as}\,\exists u{::}\kappa(\sigma'))\,\mathtt{as}\,u{::}\kappa\,\mathtt{with}\,x{:}\sigma'\,\mathtt{in}\,e$$

is sometimes written

$$\mathtt{abstype}\,u{::}\kappa\,\mathtt{is}\,\mu\,\mathtt{with}\,x{:}\sigma'{=}e'\,\mathtt{in}\,e\,\mathtt{end}$$

to emphasize the connection with the data abstraction mechanisms found in ML and CLU where the only expressions of existential type are explicit packages.

The formation rules for these constructs are as follows:

$$\frac{\vdash_{\Delta[u::\kappa]} \sigma}{\vdash_{\Delta} \exists u{::}\kappa(\sigma)} \quad (u \notin \mathrm{dom}(\Delta)) \qquad \text{(T-EXIST)}$$

$$\frac{\Delta \vdash \mu :: \kappa \qquad \Gamma \vdash_{\Delta} e : \{\mu/u\}\sigma}{\Gamma \vdash_{\Delta} \mathtt{pack}\,\mu\,\mathtt{with}\,e\,\mathtt{as}\,\exists u{::}\kappa(\sigma) : \exists u{::}\kappa(\sigma)} \qquad \text{(T-PACK)}$$

$$\frac{\Delta \vdash \sigma' \qquad \Gamma \vdash_\Delta e : \exists u{::}\kappa(\sigma) \qquad \Gamma[x{:}\sigma] \vdash_{\Delta[u::\kappa]} e' : \sigma'}{\Gamma \vdash_\Delta \mathtt{open}_{\sigma'}\, e\, \mathtt{as}\, u{::}\kappa\, \mathtt{with}\, x{:}\sigma\, \mathtt{in}\, e' : \sigma'} \quad (u \notin \mathrm{dom}(\Delta); x \notin \mathrm{dom}(\Gamma))$$

(T-OPEN)

In the rule T-PACK the type of e depends on the choice of μ. The idea is that μ is the *implementation type* (more generally, *implementation constructor*) of the abstraction, and that e is the implementation of its operations, which are defined over the implementation type. In the rule T-OPEN the constructor variable u is required to be "new" (*i.e.*, not be already declared in Δ), reflecting the informal idea that an abstract type is distinct from all other types for the purposes of type checking. The requirement that $u \notin \mathrm{dom}(\Delta)$ can always be met by appropriate use of α-conversion prior to type checking. This approach to data abstraction is sometimes called a *closed scope* or a *client-enforced* abstraction mechanism — when opening a package the client holds the implementation type abstract in a given body of code. If the client opens the same package twice, the abstract types introduced by the two occurrences of `open` are, by α-conversion, distinct from one another, and their associated operations cannot be intermixed. In a manner of speaking, in the existential type framework there are no *intrinsic*, or *implementation-enforced*, abstract types, rather only *abstract uses* of types.

8.2 Dynamics

The operational semantics of these constructs is given as follows.

$$\begin{array}{lcl} \textit{Values} & v & ::= & \mathtt{pack}\,\mu\,\mathtt{with}\,v\,\mathtt{as}\,\sigma \\ \textit{EvalCtxt's} & E & ::= & \mathtt{pack}\,\mu\,\mathtt{with}\,E\,\mathtt{as}\,\sigma \mid \mathtt{open}_{\sigma'}\,E\,\mathtt{as}\,u{::}\kappa\,\mathtt{with}\,x{:}\sigma\,\mathtt{in}\,e' \end{array}$$

The evaluation relation is generated by the following primitive reduction step:

$$\mathtt{open}_{\sigma'}\,(\mathtt{pack}\,\mu\,\mathtt{with}\,v\,\mathtt{as}\,\exists u{::}\kappa(\sigma))\,\mathtt{as}\,u{::}\kappa\,\mathtt{with}\,x{:}\sigma\,\mathtt{in}\,e \leadsto \{v/x\}\{\mu/u\}e.$$

Note that during evaluation of an `open` expression the constructor μ replaces the variable u before evaluation of its body commences: *there are no "new" types at run time.* We emphasize that data abstraction is purely a matter of type checking, and has no significance at run time.

Exercise 8.1
State and prove progress and preservation theorems for $\mathcal{L}_\Omega^{\rightarrow,\forall_p,\exists_p}$.

Exercise 8.2
Prove termination for $\mathcal{L}_\Omega^{\rightarrow,\forall_p,\exists_p}$ *by extending the termination argument given for* $\mathcal{L}_\Omega^{\rightarrow,\forall_p}$.

8.3 Impredicativity

An impredicative variant, $\mathcal{L}_\Omega^{\rightarrow,\forall,\exists}$, of $\mathcal{L}_\Omega^{\rightarrow,\forall_p,\exists_p}$ may be defined analogously to the polymorphic case, either by introducing a constructor corresponding to the existential type, or by simply collapsing the constructor and type levels into one.

Existential types are definable in $\mathcal{L}_{\Omega}^{\rightarrow,\forall}$ using the following definitions:

$$\exists u{::}\kappa(\sigma) \quad := \quad \forall t{::}\Omega((\forall u{::}\kappa(\sigma \to t)) \to t)$$

$$\begin{array}{rcl}
\mathtt{pack}\,\mu\,\mathtt{with}\,e\,\mathtt{as}\,\exists u{::}\kappa(\sigma) & := & \mathtt{Fun}\,(t{::}\Omega)\,\mathtt{in}\,\mathtt{fun}\,(f{:}\forall u{::}\kappa(\sigma \to t))\,\mathtt{in}\,f[\mu](e) \\
\mathtt{open}_{\sigma'}\,e\,\mathtt{as}\,u{::}\kappa\,\mathtt{with}\,x{:}\sigma\,\mathtt{in}\,e' & := & e[\sigma'](\mathtt{Fun}\,(u{::}\kappa)\,\mathtt{in}\,\mathtt{fun}\,(x{:}\sigma)\,\mathtt{in}\,e')
\end{array}$$

Exercise 8.3
Check that the typing and evaluation rules for $\mathcal{L}_{\Omega}^{\rightarrow,\forall,\exists}$ are derivable under the above definitions. Conclude that $\mathcal{L}_{\Omega}^{\rightarrow,\forall,\exists}$ enjoys the progress, preservation, and termination properties.

8.4 Dot Notation

The `open` construct is somewhat awkward to use in practice because of the requirement that the "client" of the abstraction be specified at the time that the package is opened. For example, if the package is an implementation of a widely-used concept such as a priority queue, then the programmer must open the package at a very early stage to ensure that it is accessible to the entire program. Thus abstract types must be given very wide scope, which tends to obstruct the goal to decompose programs into separable components. It would be useful to admit a more flexible means of handling packages that is nevertheless equivalent to the original language in the sense that programs using the more flexible notation can be re-written automatically into uses of `open`.

A natural proposal is to augment the language with a pair of operations, $\mathtt{Typ}(e)$ and $\mathtt{ops}(e)$, that extract the type part and the code part of the package, respectively. This allows us to refer directly to the operations of the package without having to delineate the complete scope of the abstraction. In view of the dependency of the type of the code part of a package on the type part of that package, we expect to have typing rules of roughly the following form:

$$\frac{\Gamma \vdash_\Delta e : \exists u{::}\kappa(\sigma)}{\Gamma \vdash_\Delta \mathtt{Typ}(e) :: \kappa} \qquad (\textsc{t-wit?})$$

$$\frac{\Gamma \vdash_\Delta e : \exists u{::}\kappa(\sigma)}{\Gamma \vdash_\Delta \mathtt{ops}(e) : \{\mathtt{Typ}(e)/u\}\sigma} \qquad (\textsc{t-opns?})$$

We can see immediately a technical problem with the rule T-WIT?: the *phase distinction* between types and terms is not preserved. Specifically, a type expression can now involve ordinary expressions (including free ordinary variables) as sub-expressions, so that types now depend on run-time values. This is reflected formally in the use of the judgement $\Gamma \vdash_\Delta \mathtt{Typ}(e) :: \kappa$ in the rule T-WIT?, where now both Γ and Δ must be considered in defining the well-formed types.

Furthermore, the potential replication of the expression e in forming the substitution $\{\texttt{Typ}(e)/u\}\sigma$ is suspect in the case that e can have an effect — two separate occurrences of e can have distinct meanings.

But are these serious problems? Yes, unless further restrictions are imposed. Specifically, the type expression $\texttt{Typ}(e)$ appears to stand for a specific type ("the type part of e"), even though e may not have a single, determinate type part! For example, e may be the expression $\texttt{if}_{\exists u::\kappa(\sigma)}\,\texttt{flip}(*)\,\texttt{then}\,e_1\,\texttt{else}\,e_2\,\texttt{fi}$, where e_1 and e_2 are packages of type $\exists u::\kappa(\sigma)$, each with distinct type components, and `flip` alternately yields `true` and `false` when applied to $*$. Roughly speaking, $\texttt{Typ}(e)$ changes its meaning every time you look at it! This can be turned into a counterexample to type soundness.

Exercise 8.4
Turn this example into a proof of unsoundness of $\mathcal{L}_\Omega^{\rightarrow,\forall_\mathsf{p},\exists_\mathsf{p}}$ *augmented with* $\texttt{Typ}(-)$ *and* $\texttt{ops}(-)$ *as described above.*

A natural restriction is to limit the application of $\texttt{Typ}(-)$ to values so that indeterminacies of the kind just described are avoided. This ensures that $\texttt{Typ}(v)$ is well-defined (stands for a specific type). As we will see in Chapter 10, this restriction leads to a sensible language, but it is interesting to note that it is not equivalent to $\mathcal{L}_\Omega^{\rightarrow,\forall_\mathsf{p},\exists_\mathsf{p}}$. That is, programs written in this language cannot be translated back into pure $\mathcal{L}_\Omega^{\rightarrow,\forall_\mathsf{p},\exists_\mathsf{p}}$ by suitable introduction of `open` expressions. Consider the expression $\texttt{fun}\,(x{:}\texttt{Typ}(v))\,\texttt{in}\,x$ of type $\texttt{Typ}(v) \rightarrow \texttt{Typ}(v)$, where $v : \exists u::\kappa(\sigma)$ is some package value. To paraphrase this using `open`, the only possibility is to consider the expression

$$\texttt{open}_{u\rightarrow u}\,v\,\texttt{as}\,u::\kappa\,\texttt{with}\,x{:}\sigma\,\texttt{in}\,\texttt{fun}\,(x{:}u)\,\texttt{in}\,x.$$

But this is clearly nonsense because the type $u\rightarrow u$ of the `open` expression involves a *bound* type variable, u, whose scope is limited to the body of the *open* expression.

By imposing a further restriction it is possible admit the operations $\texttt{Typ}(-)$ and $\texttt{ops}(-)$ while preserving equivalence with $\mathcal{L}_\Omega^{\rightarrow,\forall_\mathsf{p},\exists_\mathsf{p}}$. By restricting occurrences of $\texttt{Typ}(-)$ and $\texttt{ops}(-)$ to *variables*, we can identify the scope of the package with the scope of the variable to which it is bound. This is achieved by introducing an `open` immediately after the introduction of a variable of existential type. Notice that if, say, $\texttt{fun}\,(x{:}\exists u::\kappa(\sigma))\,\texttt{in}\,e$ is well-formed, then it has a type of the form $(\exists u::\kappa(\sigma)) \rightarrow \sigma'$, where $e : \sigma'$, assuming $x : \exists u::\kappa(\sigma)$. Now suppose that e has the form $\{\texttt{ops}(x)/y\}\{\texttt{Typ}(x)/u\}e'$, and consider the expression

$$\texttt{fun}\,(x{:}\exists u::\kappa(\sigma))\,\texttt{in}\,\texttt{open}_{\sigma'}\,x\,\texttt{as}\,u::\Omega\,\texttt{with}\,y{:}\sigma\,\texttt{in}\,e'.$$

This expression is well-formed because the type of e' is σ', which does not involve the bound variable of the `open` expression.

Exercise 8.5
Make this argument precise by giving a translation from a language with `Typ( − )` *and* `ops( − )` *restricted to variables to the language* $\mathcal{L}_{\Omega}^{\rightarrow,\forall_{\mathsf{p}},\exists_{\mathsf{p}}}$ *based on the preceding sketch. Prove that your translation preserves typing.*

It is important to note that the restriction of `Typ( − )` and `ops( − )` to variables is somewhat unnatural because the restriction is not preserved under evaluation. More precisely, the steps of evaluation involve the substitution of values for variables in expressions. In other words the restriction to variables is not preserved by evaluation — the expected type preservation property does not hold, for a somewhat unsatisfying reason. We will see in Chapter 10 how to avoid these difficulties.

8.5 References

The connection between data abstraction and existential types was established by Mitchell and Plotkin [**?**]. The "dot notation" was studied by Leroy and Cardelli [**?**].

Chapter 9

Higher Kinds

9.1 Introduction

This chapter is under construction. Most of the material here is cut from other chapters and needs to be re-worked.

9.2 Functional Kinds

The languages $\mathcal{L}_{\Omega}^{\rightarrow,\forall_{\mathsf{p}}}$ and $\mathcal{L}_{\Omega}^{\rightarrow,\forall}$ have only one kind, the kind of types. To support type operators we enrich the language with functional kinds, as described by the following grammar:

$$
\begin{array}{llll}
\textit{Kinds} & \kappa & ::= & \Omega \mid \kappa_1 \Rightarrow \kappa_2 \\
\textit{Constructors} & \mu & ::= & t \mid \rightarrow \mid \mathtt{fun}\,(u{:}\kappa)\,\mathtt{in}\,\mu \mid \mu_1(\mu_2) \\
\textit{Type Contexts} & \Delta & ::= & \emptyset \mid \Delta[t :: \kappa]
\end{array}
$$

The extension of $\mathcal{L}_{\Omega}^{\rightarrow,\forall_{\mathsf{p}}}$ with higher kinds is written $\mathcal{L}_{\Omega,\Rightarrow}^{\rightarrow,\forall_{\mathsf{p}}}$, and similarly $\mathcal{L}_{\Omega,\Rightarrow}^{\rightarrow,\forall}$ is the extension of $\mathcal{L}_{\Omega}^{\rightarrow,\forall}$ with higher kinds.

The rules for constructor formation are readily derived from those given in Chapter 2 — the kind and constructor level of $\mathcal{L}_{\Omega}^{\rightarrow,\forall_{\mathsf{p}}}$ is nothing more than a typed λ-calculus. The only non-obvious rule is the formation rule for the arrow constructor:

$$\Delta \vdash \rightarrow : \Omega \Rightarrow \Omega \Rightarrow \Omega$$

This is the function space operator as a constructor of higher kind. When we write $\mu_1 {\rightarrow} \mu_2$, it is shorthand for $\rightarrow(\mu_1)(\mu_2)$.

The relation $\Delta \vdash \mu_1 \equiv \mu_2 :: \kappa$ expresses definitional equivalence of constructors. It is defined as follows:

$$\frac{\Delta[u{::}\kappa_2] \vdash \mu :: \kappa \qquad \Delta \vdash \mu_2 :: \kappa_2}{\Delta \vdash (\mathtt{fun}\,(u{:}\kappa_2)\,\mathtt{in}\,\mu)(\mu_2) \equiv \{\mu_2/u\}\mu :: \kappa} \quad (u \notin \mathrm{dom}(\Delta)) \qquad \text{(E-BETA)}$$

$$\frac{\Delta \vdash \mu :: \kappa}{\Delta \vdash \mu \equiv \mu :: \kappa} \qquad \text{(E-REFL)}$$

$$\frac{\Delta \vdash \mu \equiv \mu' :: \kappa}{\Delta \vdash \mu' \equiv \mu :: \kappa} \qquad \text{(E-SYM)}$$

$$\frac{\Delta \vdash \mu \equiv \mu' :: \kappa \qquad \Delta \vdash \mu' \equiv \mu'' :: \kappa}{\Delta \vdash \mu \equiv \mu'' :: \kappa} \qquad \text{(E-TRANS)}$$

$$\frac{\Delta \vdash \mu_1 \equiv \mu_1' :: \kappa_2 \Rightarrow \kappa \qquad \Delta \vdash \mu_2 \equiv \mu_2' :: \kappa_2}{\Delta \vdash \mu_1(\mu_2) \equiv \mu_1'(\mu_2') :: \kappa} \qquad \text{(E-CAPP)}$$

$$\frac{\Delta[u{::}\kappa_1] \vdash \mu \equiv \mu' :: \kappa_2}{\Delta \vdash \mathtt{fun}\,(u{:}\kappa_1)\,\mathtt{in}\,\mu \equiv \mathtt{fun}\,(u{:}\kappa_1)\,\mathtt{in}\,\mu' :: \kappa_1 \Rightarrow \kappa_2} \quad (u \notin \mathrm{dom}(\Delta)) \qquad \text{(E-CABS)}$$

Exercise 9.1
Prove that if $\Delta \vdash \mu \equiv \mu' :: \kappa$, then $\Delta \vdash \mu :: \kappa$ and $\Delta \vdash \mu' :: \kappa$.

The rules for expression formation are essentially as before, generalized to quantifiers of the form $\forall t{::}\kappa(\sigma)$, where κ is an arbitrary kind.

An impredicative version of the system with higher kinds is obtained by identifying constructors of kind Ω with types, just as for the second-order fragment. In this language quantification is represented by constants $\forall_\kappa$ of kind $(\kappa \Rightarrow \Omega) \Rightarrow \Omega$ for each kind κ. The quantified type $\forall t{::}\kappa(\sigma)$ is regarded as an abbreviation for $\forall_\kappa(()\mathtt{fun}\,(t{:}\kappa)\,\mathtt{in}\,\sigma)$. This language is called Girard's System F_ω in the literature.

Exercise 9.2 (Difficult)
Extend the proof of termination to F_ω by interpreting the kind Ω as the set of all candidates for all closed types, and the kind $\kappa_1 \Rightarrow \kappa_2$ as the set of all (set-theoretic) functions from the set assigned to κ_1 to the set assigned to κ_2.

9.3 Subtyping and Higher Kinds

The framework of variable types and higher kinds may be extended to subtyping by introducing a pre-order on constructors (at least at kind Ω, and possibly also at higher kinds) as well as on types. The judgement forms are as follows:

$\Delta \vdash \mu_1 <: \mu_2 :: \kappa$	*sub-constructor relation*
$\vdash_\Delta \sigma_1 <: \sigma_2$	*sub-type relation*

The rules governing the sub-type relation are essentially as for $\mathcal{L}^{\rightarrow}_{<:}$, augmented with the following rule for the inclusion of constructors:

$$\frac{\Delta \vdash \mu_1 <: \mu_2 :: \Omega}{\vdash_\Delta T(\mu_1) <: T(\mu_2)} \qquad \text{(S-INCL)}$$

In words, sub-constructors of kind Ω determine sub-types under the explicit inclusion.

The subsumption rule is essentially as in $\mathcal{L}^{\rightarrow}_{<:}$, but relativized to the type context:

$$\frac{\Gamma \vdash_\Delta e : \sigma \qquad \vdash_\Delta \sigma <: \sigma'}{\Gamma \vdash_\Delta e : \sigma'} \tag{T-SUB-POLY}$$

The sub-constructor relation at each kind κ is defined to be the least pre-order (reflexive and transitive relation) respecting definitional equivalence of constructors and closed under a given set of kind-specific rules. At kind Ω we typically consider axioms such as

$$\frac{\Delta \vdash \mu_1' <: \mu_1 :: \Omega \qquad \Delta \vdash \mu_2 <: \mu_2' :: \Omega}{\Delta \vdash \mu_1 \rightarrow \mu_2 <: \mu_1' \rightarrow \mu_2' :: \Omega} \tag{S-ARROW-CON}$$

and similar rules governing the formation of constructors such as products or sums.

At higher kinds there are two typical choices. One is to take no additional axioms so that the sub-constructor relation coincides with definitional equivalence — in other words, the sub-constructor relation is trivial and hence may be neglected entirely. An alternative, called *operator subtyping*, is to extend the sub-constructor relation "pointwise" to constructors of functional kind. Specifically,

$$\frac{\Delta[t{::}\kappa] \vdash \mu <: \mu' :: \kappa'}{\Delta \vdash \mathtt{Fun}\, t{::}\,\kappa\, \mathtt{in}\, \mu <: \mathtt{Fun}\, t{::}\,\kappa\, \mathtt{in}\, \mu' :: \kappa \Rightarrow \kappa'} \tag{S-OPER}$$

In either case we also require that constructor application respect the pre-order:

$$\frac{\Delta \vdash \mu_1 <: \mu_1' :: \kappa_2 \Rightarrow \kappa \qquad \Delta \vdash \mu_2 <: \mu_2' :: \kappa_2}{\Delta \vdash \mu_1(\mu_2) <: \mu_1'(\mu_2') :: \kappa} \tag{S-APPCON}$$

9.4 Bounded Quantification and Power Kinds

The extension of $\mathcal{L}^{\rightarrow,\forall_\mathsf{p}}_{\Omega}$ with subtyping, $\mathcal{L}^{\rightarrow,\forall}_{\Omega,<:}$, includes not only the subtyping mechanisms of $\mathcal{L}^{\rightarrow}_{<:}$, but also extends quantification to *bounded quantification*, $\forall t{<:}\mu{::}\kappa(\sigma)$. The idea is that the quantification is restricted to all subtypes of a given type. The syntax is as follows:

$$\begin{array}{llcl} \textit{Types} & \sigma & ::= & \forall t{<:}\mu{::}\kappa(\sigma) \\ \textit{Expressions} & e & ::= & \mathtt{Fun}\, t{<:}\mu{::}\kappa(e) \mid e[\mu] \end{array}$$

We often abbreviate $\forall t{<:}\tau{::}\Omega(\sigma)$ to $\forall t{<:}\tau(\sigma)$.

Type contexts are extended as follows:

$$\begin{array}{llcl} \textit{Type Ctxt's} & \Delta & ::= & \emptyset \mid \Delta[t] \mid \Delta[t{<:}\mu{::}\kappa] \end{array}$$

We restrict attention to type contexts with no repeated declarations of a constructor variable.

The sub-constructor relation is defined to include the axiom

$$\Delta[t{<:}\mu{::}\kappa] \vdash t <: \mu :: \kappa \tag{S-VAR}$$

The subtyping relation is extended to quantified types as follows:

$$\frac{\vdash_{\Delta[t<:\tau']} \sigma <: \sigma' \qquad \Delta \vdash \tau' <: \tau}{\vdash_{\Delta} \forall t{<:}\tau(\sigma) <: \forall t{<:}\tau'(\sigma')} \quad (t \notin \mathrm{dom}(\Delta)) \tag{S-ALL}$$

In addition we have the expected rule for function types

$$\frac{\vdash_{\Delta} \sigma_1' <: \sigma_1 \qquad \vdash_{\Delta} \sigma_2 <: \sigma_2'}{\vdash_{\Delta} \sigma_1 \to \sigma_2 <: \sigma_1' \to \sigma_2'} \tag{S-ARROW-POLY}$$

Exercise 9.3
Give an informal justification for the "contravariance" of the subtyping relation with respect to the bounds on quantified types.

Typing judgements have the form $\Gamma \vdash_{\Delta} e : \sigma$. The characteristic expressions of $\mathcal{L}_{\Omega,<:}^{\to,\forall}$ have the following typing rules:

$$\frac{\Gamma[t{<:}\mu{::}\kappa] \vdash_{\Delta} e : \sigma}{\Gamma \vdash_{\Delta} \mathtt{Fun}\, t{<:}\mu{::}\kappa(e) : \forall t{<:}\mu{::}\kappa(\sigma)} \quad (t \notin \mathrm{dom}(\Delta)) \tag{T-STFN}$$

$$\frac{\Gamma \vdash_{\Delta} e : \forall t{<:}\mu{::}\kappa(\sigma) \qquad \Delta \vdash \mu' <: \mu :: \kappa}{\Gamma \vdash_{\Delta} e[\mu'] : \{\mu'/t\}\sigma} \tag{T-STAPP}$$

These are essentially the rules for higher-kind quantification, restricted according to the pre-order on constructors. The ordinary quantifiers can be recovered by postulating a "largest" constructor, $\mathtt{Top}_{\kappa}$ of kind κ, and defining $\forall t{::}\kappa(\sigma)$ to mean $\forall t{<:}\mathtt{Top}_{\kappa}{::}\kappa(\sigma)$.

Exercise 9.4
How should $\mathtt{Top}_{\kappa_1 \Rightarrow \kappa_2}$ *behave?* Hint*: what if we apply a constructor of this kind to an argument?*

Exercise 9.5
Define an operational semantics for $\mathcal{L}_{\Omega,<:}^{\to,\forall}$ *and prove progress and preservation for it.*

9.5 Singleton Kinds, Dependent Kinds, and Subkinding

It is also possible to regard a transparent type definition as a special form of opaque type definition (!). This may be achieved through the use of *singleton kinds* in the formalism of Section 5.4. Specifically, we consider the following language of kinds:

$$\textit{Kinds} \quad \kappa \quad ::= \quad \Omega \mid \mathcal{E}(\tau)$$

The kind $\mathcal{E}(\tau)$ is the kind of type constructors that are definitionally equivalent to the constructor τ. This is captured by the following rules:

$$\frac{\Gamma \vdash \tau \equiv \tau'}{\Gamma \vdash \tau :: \mathcal{E}(\tau')} \qquad (\textsc{t-sgl})$$

$$\frac{\Gamma \vdash \tau :: \mathcal{E}(\tau')}{\Gamma \vdash \tau \equiv \tau'} \qquad (\textsc{e-sgl})$$

In the presence of singleton kinds, we may make an opaque definition transparent by simply declaring the type variable to have a singleton kind:

$$\texttt{abstype}\, t{::}\,\mathcal{E}(\rho)\ \texttt{is}\ \rho\ \texttt{with}\, x{:}\tau{=}v\ \texttt{in}\, e'\ \texttt{end}$$

As before, the type variable t is "abstract" in e' in the sense that only its kind is revealed. But now the kind fully determines the definition of t, rendering the declaration completely transparent! This idea will be used again in our discussion of modularity.

Exercise 9.6
It has been proposed to take the opposite tack of treating opaque type bindings as a form of transparent type binding, but in which the type has been "opaqified" (e.g., by wrapping it with a special constructor). Discuss the merits and drawbacks of this approach. (Hint: consider definitional equality.)

9.6 References

60

Chapter 10

Modularity

10.1 Introduction

To motivate the development it is useful to outline some properties that may be expected of a flexible module system.

A *module* is a program fragment treated in isolation. In the case of typed languages such as those that we are considering in these notes a program fragment consists of a collection of definitions of types and values. These definitions are sometimes said to have "open scope" or "indeterminate scope" because the range of significance of the defined identifier (type or value) is not specified as part of the definition. The scope is unspecified precisely because we seek to treat modules in isolation from one another, insofar as that is possible.

A module is described by an *interface* that constrains the visibility of the components of the module. The notion of visibility of components plays a central role in the design of a module system. There are two aspects of visibility. One aspect is *namespace management*, the determination of which components of a module are accessible to clients — that is, which defined identifiers of a module are considered to be "in scope" for a client of that module. Another aspect is *sharing management*, the exposure of the definitions of type identifiers in a module — should a type identifier have its definition hidden, or propagated to the client?

We expect a given module to satisfy many interfaces. This allows the programmer to establish different *views* of a module appropriate for different contexts. For example, we may choose to neglect the fact that a module has certain components, or the fact that a type component is defined a certain way. *Component visibility* rules determine which components of a module are accessible to clients of the module; *type visibility* rules determine the extent to which type definitions are propagated from the module to its clients. Most languages provide control over component visibility through some form of *export* mechanism. In contrast, little control is provided over type visibility — typical policies include making all type definitions visible (as in C) or no types visible (as in

Modula-2).

It is useful to provide a many-to-many relationship between interfaces and modules. A given interface to be satisfied by many modules. For example, there may be several distinct implementations of priority queues, all of which satisfy the same interface. A given module may have many interfaces. For example, we may choose to export one set of components of a module into one context and another set into another context. Or we may limit the visibility of types to clients of an abstraction, but not to implementors of derived abstractions.

There is a fundamental tension inherent in modular programming between the desire to treat modules in isolation from one another and the need to combine them into a coherent whole to form a working system. A critical issue in the design of a module system is management of *dependence* between modules. We distinguish two forms of dependence:

1. *Essential*, or *specification*, dependence.

2. *Accidental*, or *implementation*, dependence.

Implementation dependences arise when one *module* makes reference to another *module*. That is, if the code of one module refers to the code of another, then the former is said to *depend accidentally*, or to have an *implementation dependence*, on the latter. The word "accidental" is meant to suggest that the dependence is entirely a matter of programming; another implementation of the same behavior may not exhibit such a dependence.

Specification dependences arise when one *interface* makes reference to another *module*.[1] That is, the description of the components of one module makes reference to types defined in another module. The word "essential" is meant to suggest a semantic dependence in the sense that the interface of one module cannot be given without reference to the implementation of another. Whether a specification dependence is truly essential is sensitive to the type visibility rules of the language. If the interface of a module M makes reference to types defined in module N, but N makes its types visible, then M can be used independently of N. The interesting case, however, occurs when this dependence cannot so easily be broken, namely when N holds its types abstract so that any manipulations of values of this type must involve the operations that N defines.

In order to treat modules in isolation from one another it is important to have *self-contained* interfaces for them. But in the presence of essential dependences between modules, how can this be achieved? There are two main methods for achieving independent interfaces:

1. *Parameterized interfaces.* Essential dependents are abstracted as *parameters* of the interface that are to be *instantiated* into specific contexts.

2. *Hierarchical interfaces.* Essential dependents are included as *constituents* of the interface that are to be *provided* by the context.

[1] A third possibility, dependence of one *interface* on another *interface* raises no significant technical issues.

In practice not all dependences need be eliminated. Often dependences on a common library of commonly-used modules are not suppressed. It is a matter of judgement to determine which dependences to admit in a signature, and which to suppress.

Regardless of however interface independence is achieved, there is a concomitant problem of *coherence* — how can the dependencies be restored so that two modules with these interfaces can be usefully combined? In the case of parameterized interfaces, the main technique is *consistent instantiation*. The interface parameters of several interfaces are instantiated by the same type to ensure that the component types are compatible by construction. In the case of hierarchical interfaces, the main technique is the *sharing constraint*. The constituents of several modules are constrained to coincide on one more components to ensure that they may be coherently combined. We will see an example of these two methods below.

Implementation dependences may be handled in one of two ways. In many situations it is perfectly reasonable to admit inter-module references without restriction. If two modules are to be separately compiled, some means of "breaking" and "restoring" the references between them must be provided, using standard program linking techniques. In some situations the implementation dependencies are *generic* — for example, the implementation of a dictionary may rely on a comparison operation on the elements. Most often, the dictionary code is insensitive to the particular element type, and in fact may be instantiated at several different element types in a given situation. In this case it is usual to use *parameterized modules* to define the dictionary independently of the element to, and to *instantiate* this module with a specific element types for specific situations. It is important to note that this form of parameterization is at the *module* level, and is a distinct concept from parameterization at the *interface* level!

To illustrate these ideas, consider the example program in Figure 10.1. It is written using an informal pseudo-notation that it is hoped will be self-evident. In the example, there is an essential dependence between `Rect` and `Circ` on `Pt` evidenced by the occurrences of the type `Pt.pt` in the interfaces `RECT` and `CIRC`. Consequently, the interfaces `RECT` and `CIRC` may only be understood in the presence of the module `Pt`, which implements the interface `PT`. A positive consequence of this is that the operations on rectangles and circles are compatible; we may, for example, create a unit circle centered on the lower-left corner of a rectangle r by writing `Circ.mk (Rect.lower_left r, 1.0)`. On the other hand, we may not separate `Circ` and `Rect` from `Pt`; any changes to `Pt` requires that `Circ` and `Rect` be reconstructed to reflect these changes.

To achieve the independence of `Circ` and `Rect` from `Pt`, we may employ either parameterized or hierarchical interfaces. We consider first the use of parameterization. In Figure 10.2 we define `RECT_WRT` and `CIRC_WRT` to be parameterized interfaces taking the type `pt` of points as a parameter. These are instantiated by the type `Pt.pt` to form the interfaces of the modules `Rect` and `Circ`. These modules are compatible in the sense described above since both

```
interface PT is type pt fun mk (x:float, y:float):pt fun x_coord (p:pt):float
  fun y_coord (p:pt):float end
module Pt implements PT is
  type pt is {x:float,y:float}
  fun mk (x:float, y:float):pt = {x=x,y=y}
  fun x_coord (p:pt):float = p.x
  fun y_coord (p:pt):float = p.y
end
interface RECT is
  type rect
  fun mk (p:Pt.pt, q:Pt.pt):rect
  fun lower_left (r:rect):Pt.pt
  fun upper_right (r:rect):Pt.pt
end
module Rect implements RECT is
  type rect is {x:float,y:float}
  fun mk(ll:Pt.pt, ur:Pt.pt):rect = {ll=ll,ur=ur}
  fun lower_left (r:rect):Pt.pt = r.ll
  fun upper_right (r:rect):Pt.pt = r.ur
end
interface CIRC is
  type circ
  fun mk (c:Pt.pt, r:float):circ
  fun center (c:circ):float
  fun radius (c:circ):float
end
module Circ implements CIRC is
  type circ is {c:Pt.pt, r:float}
  fun mk (c:Pt.pt, r:float):circ = {c=c, r=r}
  fun center (c:circ):float = c.c
  fun radius (c:circ):float = c.r
end
```

Figure 10.1: Modularity Example

```
interface RECT_WRT (type pt) is
  type rect
  fun mk (ll:pt, ur:pt):rect
  fun lower_left (r:rect):pt
  fun upper_right (r:rect):pt
end

interface RECT is RECT_WRT (type pt is Pt.pt)

module Rect implements RECT is (see Figure~10.1)

interface CIRC_WRT (type pt) is
  type circ
  fun mk (c:pt, r:float):circ
  fun center (c:circ):float
  fun radius (c:circ):float
end

interface CIRC is CIRC_WRT (type pt is Pt.pt)

module Circ implements CIRC is (see Figure~10.1)

module Pt' implements PT is (another representation of points)

interface RECT' is RECT_WRT (type pt is Pt'.pt)

module Rect' implements RECT' is (similar to Rect)

interface CIRC' is CIRC_WRT (type pt is Pt'.pt)

module Circ' implements CIRC' is (similar to Circ)
```

Figure 10.2: Modularity Example Using Parameterization

```
module RectBuilder (module Pt implementing PT)
  implements RECT (type pt is Pt.pt) is (see Figure~10.1)
module CircBuilder (module Pt implementing PT)
  implements RECT (type pt is Pt.pt) is (see Figure~10.1)
module Rect implements RECT is RectBuilder (module Pt = Pt)
module Rect' implements RECT' is RectBuilder (module Pt = Pt')
module Circ implements CIRC is CircBuilder (module Pt = Pt)
module Circ' implements CIRC' is CircBuilder (module Pt = Pt')
```

Figure 10.3: Parameterized Modules and Parameterized Interfaces

are built on the same notion of point. We then introduce a second module, `Pt'`, that also implements the interface `PT`, and build modules `Rect'` implementing `RECT_WRT (type pt is Pt'.pt)` and `Circ'` implementing `CIRC_WRT (type pt is Pt'.pt)`. Now `Rect'` is compatible with `Circ'`, but neither `Rect` nor `Circ` are compatible with either `Rect'` or `Circ'`! Here we are assuming that the type system segregates the types `Pt.pt` and `Pt.pt'`. If not, then we *may* compose `Circ'.mk` with `Rect.lower_left`, but the result will, in general, be erroneous since the underlying implementations of points are not compatible.

It is unfortunate that the construction of the modules `Rect'` and `Circ'` requires repetition of code. In fact, the code is "generic" in the specific implementation of points. This suggests that we may provide a generic implementation of rectangles and circles that is parametric in the implementation of points. See figure 10.3.

Compatibility of components in the parameterized setting is achieved by "threading" the appropriate types through the interfaces to build compatible instances of the parameterized interfaces. An alternative is to use hierarchical structuring of interfaces to ensure that modules are self-contained. This is illustrated in Figure 10.4. In this example we include an implementation of points as part of the implementation of rectangles and circles. This means that the rectangle and circle modules must be parameterized in the implementation of points (otherwise code duplication would result, as before). It is important to note, however, that `Rect.lower_left` is *incompatible* with `Circ.center`, even though they happen to have been built using the same `Pt` module! Put another way, since each geometric object module comes with its own notion of point, we must assume that each such module comes with a *distinct* notion of point — since they *can* be different, we must *assume* they are different.

But suppose that we wish to build a unit circle centered at the lower-left corner of a given rectangle. How are we to achieve this? It can *only* be sensible if `Rect` and `Circ` are built with the same notion of `Point`, but how can this constraint be expressed? In the parameterized case we "thread" a common implementation of points through the interface to ensure compatibility. In the hierarchical case we avoid the need to repeatedly instantiate interfaces, but,

```
interface RECT is
  module Pt implements PT
  type rect
  fun mk (ll:Pt.pt, ur:Pt.pt):rect
  fun lower_left (r:rect):Pt.pt
  fun upper_right (r:rect):Pt.pt
end
interface CIRC is
  module Pt implements PT
  type circ
  fun mk (c:Pt.pt, r:float):circ
  fun center (c:circ):Pt.pt
  fun radius (c:circ):float
end
module RectBuilder (module Pt:PT) implements RECT is (see Figure~10.1)
module CircBuilder (module Pt:PT) implements CIRC is (see Figure~10.1)
module Rect implements RECT is RectBuilder (module Pt is Pt)
module Rect' implements RECT is RectBuilder (module Pt is Pt')
module Circ implements CIRC is CircBuilder (module Pt is Pt)
module Circ' implements CIRC is CircBuilder (module Pt is Pt')
```

Figure 10.4: Modules Example Using Hierarchies and Parameterized Modules

```
module Rect implements RECT is RectBuilder (module Pt is Pt)

module Circ implements CIRC sharing type Self.Pt.pt = Rect.Pt.pt
  is CircBuilder (module Pt is Rect.Pt)

module Rect' implements RECT is RectBuilder (module Pt is Pt')

module Circ' implements CIRC sharing type Selt.Pt.pt = Rect.Pt.pt
  is CircBuilder (module Pt is Rect'.Pt)

module Main
  (module Circ implements CIRC and Rect implements RECT
   sharing type Circ.Pt.pt = Rect.Pt.pt)
  is ... Circ.mk (Rect.lower_left r, 1.0) ...
```

Figure 10.5: Module Example Using Hierarchies and Sharing Constraints

in compensation, some other mechanism for ensuring compatibility is required. Here we arrive at the notion of a *sharing constraint*. See Figure 10.5. The essential idea is to make explicit in the interface of `Circ` that it is built with the same notion of points as is `Rect`, and similarly for `Circ'` and `Rect'`. Then the required composition is sensible. An alternative is to build a parameterized module that, when given two compatible implementations of circules and rectangles, yields code that builds the required circle (among other things).

10.2 A Critique of Some Modularity Mechanisms

In this section we discuss the strengths and weaknesses of two well-known formalisms for expressing modularity, *existential types* (discussed in Chapter 8) and *sigma types* (described below).

Existential types have a number of attributes that are desirable in a modularity mechanism:

1. Data abstraction. The implementation of an abstraction is "hidden" from clients, allowing the package to be modified without affecting the behavior of the client.

2. Separate compilation. The client of an abstraction need know only the type of the package, and never its contents. Consequently libraries may be separately compiled and maintained.

3. Flexibility. Packages are "first class" values. Implementations of abstractions may be selected on the basis of run-time considerations. Packages may be stored in data structures and passed as arguments to functions.

However existential types also have some drawbacks:

1. Closed-scope abstraction. Packages can only be opened for use in a specific scope, and this scope must be chosen wide enough to cover all uses of the

package. This leads to an inversion of program structure in which the most basic constructs must be given the widest possible scope. Different uses of the same abstraction introduce different types, complicating the construction of large systems from independent components.

2. Poor support for hierarchies. The hierarchical approach to managing dependencies is useless in the existential formalism because there is no possibility to capture the fact that two hierarchies share a common component. For example, if we `open` the packages `Rect` and `Circ` defined in Figure 10.4 (regarded as having existential type), then there will be two distinct notions of point, one used by the operations in `Rect`, the other by those in `Circ`. Consequently, the operations of `Rect` and `Circ` will be incompatible with one another.

3. Poor support for parameterized modules. It is impossible to track the dependence of the result of a parameterized module on its input. Consider the following pseudo-code:

```
interface POSET is
  type elt
  fun le(x:elt,y:elt:bool
end

module LexOrd (module X implements POSET) implements POSET is ...

module Int implements POSET is ...

module IntList implements POSET is LexOrd (module X is Int)
```

   The module `IntList` is useless because, when opened, a new type `elt` is introduced supporting *only* the operation `le`, which cannot be applied to any value!

An alternative to the existential formalism that avoids some of these difficulties is the *sigma*, or *strong sum*, formalism. These types have the form $\Sigma u{::}\kappa(\tau)$. The introductory form $\texttt{pack}\,\mu\,\texttt{with}\,e\,\texttt{as}\,\Sigma u{::}\kappa(\tau)$ is the same as for existentials: $\Sigma u{::}\kappa(\tau)$ if $\mu :: \kappa$ and $e : \{\mu/u\}\tau$. The eliminatory forms are the "projections" $\texttt{Typ}(e)$ and $\texttt{ops}(e)$ discussed in Chapter 8, but subject to the following *transparency principle* of type equivalence:

$$\texttt{Typ}(\texttt{pack}\,\mu\,\texttt{with}\,v\,\texttt{as}\,\Sigma u{::}\kappa(\tau)) \equiv \mu :: \kappa.$$

This axiom ensures that the type component of a package is "visible" during type checking. This clearly violates data abstraction, but, on the other hand, it admits a more flexible typing system for modules.

If we regard modules as packages of strong sum type, then we have the following equivalences

$$\texttt{Typ(Fst(Rect))} = \texttt{Typ(Fst(Circ))} = \{\texttt{x:float,y:float}\}.$$

This is so because `Rect` and `Circ` are built on the same `Pt` module, which implements `pt` as the type {`x:float,y:float`}. For this to work, we must emply a *substitution semantics* for module definitions — defined module identifiers are replaced by their definitions *before* type checking. This ensures that the packages themselves are available during type checking so that the transparency principle may be applied. Thus, type checking depends on the *implementation*, and not just the *interface*, of a module.

By augmenting the formalism with additional equations corresponding to β-reduction of functor applications, we may deduce type sharing properties of functor applications. For example, returning to the lexicographic ordering example above, we may deduce that

$$\mathtt{Typ(IntList)} = \mathtt{Typ(Int)\,list}$$

by "running" the parameterized module `LexOrd` on the argument `IntList`. This reliance on "execution" of parameterized modules during type checking is problematic in the presence of computational effects, even non-termination. For this (and other reasons) the strong sum formalism is restricted to "second class" module systems, as described briefly in Section 10.4 below.

The strong sum formalism solves the problem of "over-abstraction" associated with the opaque sum formalism, but at the expense of introducing problems of its own. Specifically,

1. No data abstraction. To achieve flexibility the underlying implementation type of a module is propagated to the client.

2. No separate compilation. The utility of the transparent sum formalism relies on the "substitution semantics" for module definitions. This precludes separately compiling modules from one another.

3. Second-class modules. Even under the "value restriction" on the $\mathtt{Typ}(-)$ operation, it is possible to encode a "type of all types" using transparent sums, specifically $\Sigma t{::}\Omega(\mathtt{Unit})$. This implies that types are themselves "first class", which precludes compile-time type checking. Consequently strong sums may be safely used only in a *stratified* language in which module types are separated from ordinary types.

It is worth noting that a form of data abstraction is available in the transparent sum formalism by introducing a *non-substitutive* module definition mechanism.[2] However, this is a "closed scope" abstraction construct, which suffers from the same problems as the `open` construct of the opaque sum formalism. In particular, we cannot define a functor whose result is "intrinsically abstract".

The discussion of the opaque and transparent sum formalisms isolates several key ideas that are critical to a flexible module system:

1. Controlled abstraction. It should be possible to selectively propagate or obscure the type components of a module under programmer control.

[2]This is called an `abstraction` binding in MacQueen's original modules proposal.

2. Separate compilation. It should be possible to compile modules independently of one another.

3. Computational effects. The module system should be compatible with the full range of control and store effects.

4. Flexible combinators. It should be possible to build module hierarchies and generic modules without undue circumlocution.

5. Views. It should be possible to consider a module to implement a variety of interfaces, and a given interface should admit many implementations.

As we will see below, the key is to enrich the expressiveness of the type system so that type sharing relationships can be tracked at compile time through the type, and not the contents, of a package.

10.3 A Modules Language

There are two different approaches to formalizing a language with modules according to whether or not modules are "first-class" values or not. By "first-class", we mean that modules are ordinary values that may be passed to and returned from functions, occur in data structures, be stored in reference cells, *etc.*. There are two forms of first-class module system, predicative and impredicative, according to whether or not type variables range over module types. The alternative is a "second-class" module system in which modules are segregated from the rest of the language, with only limited forms of module expression (*e.g.*, no conditional module expressions). We begin by considering a language with first-class modules since the type theory is fully compatible with this decision. Second-class module systems raise interesting type-theoretic questions that we defer to a later section.

The language $\mathcal{L}_{\Omega,\mathcal{E}}^{\mathtt{sig},\Sigma,\Pi}$ of predicative, first-class modules is based on the higher-kind, variable type formalism considered in Chapters 5 and 4. For the sake of simplicity we do no consider a pre-order on constructors, but rather only on types and kinds. We omit explicit consideration of "generic" constructs such as record kinds and associated sub-kinding, and of expressions and types other than those associated with modules. The impredicative variant of $\mathcal{L}_{\Omega,\mathcal{E}}^{\mathtt{sig},\Sigma,\Pi}$ is obtained by collapsing the constructor and type levels of the predicative system, and dispensing with the explicit inclusion of constructors into types.

10.3.1 Structures

The basic form of module in $\mathcal{L}^{\mathtt{sig},\Sigma,\Pi}_{\Omega,\mathcal{E}}$ is the *structure*. The syntactic forms associated with structures are as follows:

$$
\begin{array}{llll}
\textit{Kinds} & \kappa & ::= & \Omega \mid \mathcal{E}(\mu) \\
\textit{Constr's} & \mu & ::= & \mathtt{Typ}(v) \\
\textit{Types} & \sigma & ::= & \mathtt{sig}\, u{::}\kappa\, \mathtt{with}\, \sigma \\
\textit{Expr's} & e & ::= & \mathtt{ops}(e) \mid \mathtt{struct}\, \mu\, \mathtt{with}\, e\, \mathtt{as}\, \sigma \\
\textit{Values} & v & ::= & \mathtt{ops}(v) \mid \mathtt{struct}\, \mu\, \mathtt{with}\, v\, \mathtt{as}\, \sigma
\end{array}
$$

Types of the form $\mathtt{sig}\, u{::}\kappa\, \mathtt{with}\, \sigma$ are called *structure types*; expressions of the form $\mathtt{struct}\, \mu\, \mathtt{with}\, e\, \mathtt{as}\, \sigma$ are called *structures*. We emphasize the importance of singleton kinds for the selective propagation of constructor sharing information.

Since constructors may involve ordinary variables, it is necessary to consolidate the type and kind contexts into a single, unified context consisting of declarations for both expression variables and constructor variables.

$$
\textit{Contexts} \quad \Theta \quad ::= \quad \emptyset \mid \Theta[u{::}\kappa] \mid \Theta[x{:}\sigma]
$$

As usual we restrict attention to contexts that declare a given variable at most once.

The type system consists of rules for deriving judgements of the following forms:

$$
\begin{array}{rl}
\vdash \Theta & \Theta \text{ is a valid context} \\
\Theta \vdash \kappa & \kappa \textit{ is a valid kind} \\
\Theta \vdash \kappa_1 <:: \kappa_2 & \kappa_1 \textit{ is a sub-kind of } \kappa_2 \\
\Theta \vdash \kappa_1 \equiv \kappa_2 & \kappa_1 \textit{ and } \kappa_2 \textit{ are equivalent kinds} \\
\Theta \vdash \mu :: \kappa & \mu \textit{ is a valid constructor of kind } \kappa \\
\Theta \vdash \mu_1 <: \mu_2 :: \kappa & \mu_1 \textit{ is a sub-constructor of } \mu_2 \textit{ of kind } \kappa \\
\Theta \vdash \mu_1 \equiv \mu_2 :: \kappa & \mu_1 \textit{ and } \mu_2 \textit{ are equivalent constructors} \\
\Theta \vdash \sigma_1 <: \sigma_2 & \sigma_1 \textit{ is a subtype of } \sigma_2 \\
\Theta \vdash \sigma_1 \equiv \sigma_2 & \sigma_1 \textit{ and } \sigma_2 \textit{ are equivalent types} \\
\Theta \vdash e : \sigma & e \textit{ has type } \sigma
\end{array}
$$

Kinds

To track type sharing properties we employ the formalism of singleton kinds. Roughly speaking, the kind $\mathcal{E}(\mu)$ is the kind of types that are definitionally equivalent to the type μ. Type sharing information may be neglected through the use of subsumption at the constructor level for a natural pre-ordering of kinds generated by regarding singletons as sub-kinds of Ω.

The formation rules for kinds are as follows:

$$
\Theta \vdash \Omega \qquad \text{(K-TYPE)}
$$

$$
\frac{\Theta \vdash \mu :: \Omega}{\Theta \vdash \mathcal{E}(\mu)} \qquad \text{(K-SGL)}
$$

The sub-kinding rules include the following rule:

$$\frac{\Theta \vdash \mu :: \Omega}{\Theta \vdash \mathcal{E}(\mu) <:: \Omega} \qquad (\textsc{s-sgl})$$

Singleton kinds respect definitional equivalence of constructors:

$$\frac{\Theta \vdash \mu_1 \equiv \mu_2 :: \Omega}{\Theta \vdash \mathcal{E}(\mu_1) \equiv \mathcal{E}(\mu_2)} \qquad (\textsc{e-sgl})$$

Note, however, that $\mathcal{E}(-)$ does *not* respect the sub-constructor relation!

Exercise 10.1
Give an informal argument as to why $\mathcal{E}(-)$ should not respect the sub-constructor relation based on the "re-use" interpretation of the sub-kind relation.

Neglecting type constructor components of a module is achieved by augmenting the kind structure with record kinds. As with record types, we consider a sub-kinding relation induced by "forgetting" record components. Record kinds respect kind equivalence and sub-kinding in each field.

$$\frac{\Theta \vdash \kappa_1 \quad \ldots \quad \Theta \vdash \kappa_n}{\Theta \vdash \{L_1{:}\kappa_1, \ldots, L_n{:}\kappa_n\}} \qquad (\textsc{k-recd})$$

$$\frac{\Theta \vdash \kappa_1 \equiv \kappa_1' \quad \ldots \quad \Theta \vdash \kappa_n \equiv \kappa_n'}{\Theta \vdash \{L_1{:}\kappa_1, \ldots, L_n{:}\kappa_n\} \equiv \{L_1{:}\kappa'{}_1, \ldots, L_n{:}\kappa'{}_n\}} \qquad (\textsc{e-recd})$$

$$\frac{\begin{array}{c}\Theta \vdash \kappa_1 <:: \kappa_1' \quad \ldots \quad \Theta \vdash \kappa_m <:: \kappa_m' \\ \Theta \vdash \kappa_{m+1} \quad \ldots \Theta \vdash \kappa_{m+n}\end{array}}{\Theta \vdash \{L_1{:}\kappa_1, \ldots, L_{m+n}{:}\kappa_{m+n}\} <:: \{L_1{:}\kappa'{}_1, \ldots, L_m{:}\kappa'{}_m\}} \qquad (\textsc{s-recd})$$

Singletons may be extended to record kinds by defining

$$\mathcal{E}(\mu :: \{L_1{:}\kappa_1, \ldots, L_n{:}\kappa_n\}) \equiv \{L_1{:}\mathcal{E}(\mu.L)_1, \ldots, L_n{:}\mathcal{E}(\mu.L)_n\}$$

(where $\mu.L$ is the record field selection operation described below).

As an aside, we note that the extension of singletons to function kinds requires dependent function spaces:

$$\mathcal{E}(\mu :: \Pi\, u{::}\kappa_1\, (\kappa_2)) \equiv \Pi\, u{::}\kappa_1\, (\,\mathcal{E}(\mu(u) :: \kappa_2)).$$

Constructors

The constructor formation rules are largely inherited from the ambient programming language. The sole new construct is the type projection from a module:

$$\frac{\Theta \vdash v : \mathtt{sig}\, u{::}\kappa\, \mathtt{with}\, \sigma}{\Theta \vdash \mathtt{Typ}(v) :: \kappa} \qquad (\text{T-TYP})$$

Singleton kinds induce equivalences between constructors, and conversely:

$$\frac{\Theta \vdash \mu \equiv \mu' :: \Omega}{\Theta \vdash \mu :: \mathcal{E}(\mu')} \qquad (\text{K-SGL-INTRO})$$

$$\frac{\Theta \vdash \mu :: \mathcal{E}(\mu')}{\Theta \vdash \mu \equiv \mu' :: \Omega} \qquad (\text{K-SGL-ELIM})$$

We postulate a principle of subsumption for constructors corresponding to the subkinding relation:

$$\frac{\Theta \vdash \mu :: \kappa \qquad \Theta \vdash \kappa <:: \kappa'}{\Theta \vdash \mu :: \kappa'} \qquad (\text{K-SUB})$$

This ensures that we may "forget" the identity of a constructor or the components of a record constructor by using the rule K-SUB in conjunction with rules S-SGL and S-RECD.

In the presence of record kinds we also have record constructors and record selections:

$$\frac{\Theta \vdash \mu_1 :: \kappa_1 \ldots \quad \ldots \Theta \vdash \mu_n :: \kappa_n}{\Theta \vdash \{L_1{=}\mu_1, \ldots, L_n{=}\mu_n\} :: \{L_1{:}\kappa_1, \ldots, L_n{:}\kappa_n\}} \qquad (\text{T-CON-RECD})$$

$$\frac{\Theta \vdash \mu :: \{L_1{:}\kappa_1, \ldots, L_n{:}\kappa_n\}}{\Theta \vdash \mu.L_i :: \kappa_i} \qquad (\text{T-CON-SEL})$$

Record types and associated ordering are as described in Chapter 4.

Types

The formation, subtyping, and equivalence rules for structure types are as follows:

$$\frac{\Theta[u{::}\kappa] \vdash \sigma :: \Omega}{\Theta \vdash \mathtt{sig}\, u{::}\kappa\, \mathtt{with}\, \sigma :: \Omega} \qquad (\text{T-SIG})$$

$$\frac{\Theta \vdash \kappa <:: \kappa' \qquad \Theta[u{::}\kappa] \vdash \sigma <: \sigma'}{\Theta \vdash \mathtt{sig}\, u{::}\kappa\, \mathtt{with}\, \sigma <: \mathtt{sig}\, u{::}\kappa'\, \mathtt{with}\, \sigma'} \qquad (\text{S-SIG})$$

$$\frac{\Theta \vdash \kappa \equiv \kappa' \qquad \Theta[u{::}\kappa] \vdash \sigma \equiv \sigma'}{\Theta \vdash \mathtt{sig}\, u{::}\kappa\, \mathtt{with}\, \sigma \equiv \mathtt{sig}\, u{::}\kappa'\, \mathtt{with}\, \sigma'} \qquad (\text{E-SIG})$$

Rule S-SIG allows sharing information, constructor components, and value components to be neglected. Rule E-SIG allows propagation of type sharing information governing the constructor variable u to the type σ.

Terms

The formation rules for structures is as follows:

$$\frac{\Theta \vdash \mu :: \kappa \qquad \Theta \vdash e :: \{\mu/u\}\sigma}{\Theta \vdash \mathtt{struct}\,\mu\,\mathtt{with}\,e\,\mathtt{as}\,\mathtt{sig}\,u{::}\kappa\,\mathtt{with}\,\sigma : \mathtt{sig}\,u{::}\kappa\,\mathtt{with}\,\sigma} \qquad (\text{T-STRUCT})$$

This rule is exactly the same as the introductory rule for existential types or sigma types.

The run-time component of a structure is selected as follows:

$$\frac{\Theta \vdash e : \mathtt{sig}\,u{::}\kappa\,\mathtt{with}\,\sigma}{\Theta \vdash \mathtt{ops}(e) : \sigma} \quad (u \notin \mathrm{FV}(\sigma)) \qquad (\text{T-OPNS})$$

Notice that this rule imposes the requirement that u not occur freely in σ: the dependence of σ on u must be eliminated by equational reasoning before application of this rule. In the case that e is a value, this is no restriction, but in general we cannot select the operations part of a structure without first determining that the type part is well-defined.

The following "self-recognition" rules are critical to propagation of well-definedness of type components:

$$\frac{\Theta \vdash v : \mathtt{sig}\,u{::}\kappa\,\mathtt{with}\,\sigma \qquad \Theta \vdash \mathtt{Typ}(v) :: \kappa'}{\Theta \vdash v : \mathtt{sig}\,u{::}\kappa'\,\mathtt{with}\,\sigma} \qquad (\text{T-SELF-TYP})$$

$$\frac{\Theta \vdash v : \mathtt{sig}\,u{::}\kappa\,\mathtt{with}\,\sigma \qquad \Theta \vdash \mathtt{ops}(v) :: \sigma'}{\Theta \vdash v : \mathtt{sig}\,u{::}\kappa\,\mathtt{with}\,\sigma'} \quad (u \notin \mathrm{FV}(\sigma)) \qquad (\text{T-SELF-OPS})$$

In practice the rule T-SELF-TYP is used to replace the kind Ω by the kind $\mathcal{E}(\mathtt{Typ}(v))$, expressing the fact that the type component of a structure value v is known to be $\mathtt{Typ}(v)$. One use of this arises in connection with variables: if x is a variable of type $\mathtt{sig}\,u{::}\Omega\,\mathtt{with}\,\sigma$, then x also has type $\mathtt{sig}\,u{::}\mathcal{E}(\mathtt{Typ}(x))\,\mathtt{with}\,\sigma$, hence if y is defined to be x, then y also has the latter type, and so $\mathtt{Typ}(y) \equiv \mathtt{Typ}(x)$. In the absence of rule T-SELF-TYP the types $\mathtt{Typ}(x)$ and $\mathtt{Typ}(y)$ would be distinct, even though y and x are the same structure. The rule T-SELF-OPS is used to propagate the rule T-SELF-TYP into σ, the "operations" part of the structure. We rely on $\mathtt{ops}(v)$ being a value whenever v is a value, otherwise the rule T-SELF-TYP would not apply to $\mathtt{ops}(v)$.

Exercise 10.2
Prove that the following are derived rules:

1. *The type components of structure values are visible:*

$$\frac{\Theta \vdash \mathtt{struct}\,\mu\,\mathtt{with}\,v\,\mathtt{as}\,\mathtt{sig}\,u{::}\kappa\,\mathtt{with}\,\sigma : \mathtt{sig}\,u{::}\kappa\,\mathtt{with}\,\sigma}{\Theta \vdash \mathtt{Typ}(\mathtt{struct}\,\mu\,\mathtt{with}\,v\,\mathtt{as}\,\mathtt{sig}\,u{::}\kappa\,\mathtt{with}\,\sigma) \equiv \mu :: \kappa} \qquad (\text{E-TRANSPARENT})$$

2. *Type sharing information may be propagated and forgotten:*

$$\frac{\Theta \vdash \mathtt{sig}\, u{::}\mathcal{E}(\mu)\,\mathtt{with}\,\sigma :: \Omega}{\Theta \vdash \mathtt{sig}\, u{::}\mathcal{E}(\mu)\,\mathtt{with}\,\sigma <: \mathtt{sig}\, u{::}\Omega\,\mathtt{with}\,\{\mu/u\}\sigma} \qquad (\text{E-PROPAGATE})$$

3. *The operations part of a value may always be extracted:*

$$\frac{\Theta \vdash v : \mathtt{sig}\, u{::}\kappa\,\mathtt{with}\,\sigma}{\Theta \vdash \mathtt{ops}(v) : \{\mathtt{Typ}(v)/u\}\sigma} \qquad (\text{T-OPNS-VAL})$$

Signatures mediate the propagation of type sharing information. The preceding exercise demonstrates that a structure value is "transparent" in the sense that the identity of its type component may be propagated to its type. But what if we wish to hide this information? The *sealing* operation $e{:}\sigma$ provides the means for doing so. Its typing rule is

$$\frac{\Theta \vdash e : \sigma}{\Theta \vdash e{:}\sigma : \sigma} \qquad (\text{T-SEAL})$$

The expression $e{:}\sigma$ is, by design, not a value. This ensures that the seal cannot be broken using the rule T-SELF-TYP. In typical cases the type σ has opaque constructor components in positions where the "natural" type of e is transparent (*e.g.*, when e is a structure value).

10.3.2 Module Hierarchies

To support hierarchical configurations of modules, we extend the grammar of $\mathcal{L}_{\Omega,\mathcal{E}}^{\mathtt{sig},\Sigma,\Pi}$ with the following clauses:

$$\begin{array}{llll} \textit{Types} & \sigma & ::= & \Sigma\, x{:}\sigma_1(\sigma_2) \\ \textit{Expr's} & e & ::= & \mathtt{structure}\, e_1\,; e_2 \mid \mathtt{FST}(e) \mid \mathtt{SND}(e) \end{array}$$

A type of the form $\Sigma\, x{:}\sigma_1(\sigma_2)$ is a *substructure type* since it allows one to define a constituent structure (of type σ_1) of an encompassing structure (of type σ_2). An expression of the form $\mathtt{structure}\, e_1\,; e_2$ is a *substructure expression*; expressions of the form $\mathtt{FST}(e)$ and $\mathtt{SND}(e)$ are *substructure projections.*

The formation, subtyping, and equivalence rules for the substructure types are these:

$$\frac{\Theta \vdash \sigma_1 \qquad \Theta[x{:}\sigma_1] \vdash \sigma_2}{\Theta \vdash \Sigma\, x{:}\sigma_1(\sigma_2)} \quad (x \notin \mathrm{dom}(\Theta)) \qquad (\text{T-SUBSIG})$$

$$\frac{\Theta \vdash \sigma_1 <: \sigma_1' \qquad \Theta[x{:}\sigma_1] \vdash \sigma_2 <: \sigma_2'}{\Theta \vdash \Sigma\, x{:}\sigma_1(\sigma_2) <: \Sigma\, x{:}\sigma_1'(\sigma_2')} \quad (x \notin \mathrm{dom}(\Theta)) \qquad (\text{S-SUBSIG})$$

$$\frac{\Theta \vdash \sigma_1 \equiv \sigma_1' \qquad \Theta[x{:}\sigma_1] \vdash \sigma_2 \equiv \sigma_2'}{\Theta \vdash \Sigma\, x{:}\sigma_1(\sigma_2) \equiv \Sigma\, x{:}\sigma_1'(\sigma_2')} \qquad (\text{E-SUBSIG})$$

The formation rules for substructures and projections are as follows:

$$\frac{\Theta \vdash e_1 : \sigma_1 \qquad \Theta[x{:}\sigma_1] \vdash e_2 : \sigma_2}{\Theta \vdash \texttt{structure}\, e_1\, ; e_2 : \Sigma\, x{:}\sigma_1(\sigma_2)} \qquad (\text{T-SUBSTR})$$

$$\frac{\Theta \vdash e : \Sigma\, x{:}\sigma_1(\sigma_2)}{\Theta \vdash \texttt{FST}(e) : \sigma_1} \qquad (\text{T-FIRST})$$

$$\frac{\Theta \vdash e : \Sigma\, x{:}\sigma_1(\sigma_2)}{\Theta \vdash \texttt{SND}(e) : \sigma_2} \quad (x \notin \mathrm{FV}(\sigma_2)) \qquad (\text{T-SECOND})$$

Notice that in the rule T-SUBSTR only the type σ_1 of the substructure e_1 is propagated, rather than the substructure itself as in the case of transparent sum types. This ensures the enforceability of abstraction, and avoids complications in the case that the substructure expression has a computational effect.

The rule T-SECOND is reminiscent of the rule T-OPNS for basic modules. All dependences of the second component on the first must be resolved (by sharing propagation) prior to projection. If the second component cannot be given a type that is independent of the first component, then the second projection is not well-formed. In the case that e is a value, we can always form the second projection by using the following rules:

$$\frac{\Theta \vdash v : \Sigma\, x{:}\sigma_1(\sigma_2) \qquad \Theta \vdash \texttt{FST}(v) : \sigma_1'}{\Theta \vdash v : \Sigma\, x{:}\sigma_1'(\sigma_2)} \qquad (\text{T-SELF-SUBSIG})$$

$$\frac{\Theta \vdash v : \Sigma\, x{:}\sigma_1(\sigma_2) \qquad \Theta[x{:}\sigma_1] \vdash \texttt{SND}(v) : \sigma_2'}{\Theta \vdash v : \Sigma\, x{:}\sigma_1(\sigma_2')} \qquad (\text{T-SELF-SUBSIG'})$$

We regard $\texttt{FST}(v)$ and $\texttt{SND}(v)$ to be values to ensure that these "self" rules are applicable.

Exercise 10.3
Prove that if $v : \Sigma\, x{:}\sigma_1(\sigma_2)$, then there exists σ_1' and σ_2' such that $x \notin \mathrm{FV}(\sigma_2')$ and $v : \Sigma\, x{:}\sigma_1'(\sigma_2')$.

Exercise 10.4
Define a call-by-value operational semantics for substructures and verify progress and preservation for this semantics.

10.3.3 Parameterized Modules

To support generic modules, the grammar of $\mathcal{L}_{\Omega,\mathcal{E}}^{\texttt{sig},\Sigma,\Pi}$ is enriched as follows:

$$\begin{array}{lll} \textit{Types} & \sigma & ::= \quad \Pi\, x{:}\sigma_1(\sigma_2) \\ \textit{Terms} & \sigma & ::= \quad \texttt{functor}\, x{:}\sigma_1(e) \mid \texttt{inst}\, e_1(e_2) \end{array}$$

A type of the form $\Pi\, x{:}\sigma_1(\sigma_2)$ is a *functor type*. An expression of the form $\texttt{functor}\, x{:}\sigma_1(e)$ is a *functor*, and an expression of the form $\texttt{inst}\, e_1(e_2)$ is a *functor application* or *functor instantiation*.

It is interesting to note that polymorphic types are definable in this system by regarding polymorphic operations as functors. Specifically, we may define $\forall u{::}\kappa(\sigma)$ to stand for

$$\Pi\, x{:}\texttt{sig}\, u{::}\kappa\, \texttt{with}\, \texttt{Unit}(\{\texttt{FST}(x)/u\}\sigma).$$

Corresponding definitions may be given for polymorphic functions and instantiations.

The formation, subtyping, and equivalence rules for functor types are as follows:

$$\frac{\Theta \vdash \sigma_1 \qquad \Theta[x{:}\sigma_1] \vdash \sigma_2}{\Theta \vdash \Pi\, x{:}\sigma_1(\sigma_2)} \quad (x \notin \mathrm{dom}(\Theta)) \qquad (\text{T-FUNSIG})$$

$$\frac{\Theta \vdash \sigma_1' <: \sigma_1 \qquad \Theta[x{:}\sigma_1'] \vdash \sigma_2 <: \sigma_2'}{\Theta \vdash \Pi\, x{:}\sigma_1(\sigma_2) <: \Pi\, x{:}\sigma_1'(\sigma_2')} \quad (x \notin \mathrm{dom}(\Theta)) \qquad (\text{S-FUNSIG})$$

$$\frac{\Theta \vdash \sigma_1 \equiv \sigma_1' \qquad \Theta[x{:}\sigma_1'] \vdash \sigma_2 \equiv \sigma_2'}{\Theta \vdash \Pi\, x{:}\sigma_1(\sigma_2) \equiv \Pi\, x{:}\sigma_1'(\sigma_2')} \quad (x \notin \mathrm{dom}(\Theta)) \qquad (\text{E-FUNSIG})$$

The subtyping rule for functor signatures expresses the contravariance of functor types in the domain position, as might be expected. An important instance arises when σ_1' contains more type sharing information than σ_1 — the functor type is *less general* when the domain type is *more specific*. The equivalence rule for functor signatures allows type sharing information to be propagated from the argument to the result type of the functor.

The formation rules for functors and functor instantiation are as follows:

$$\frac{\Theta[x{:}\sigma_1] \vdash e : \sigma_2}{\Theta \vdash \texttt{functor}\, x{:}\sigma_1(e) : \Pi\, x{:}\sigma_1(\sigma_2)} \quad (x \notin \mathrm{dom}(\Theta)) \qquad (\text{T-FUNCTOR})$$

$$\frac{\Theta \vdash e : \Pi\, x{:}\sigma_2(\sigma) \qquad \Theta \vdash e_2 : \sigma_2}{\Theta \vdash \texttt{inst}\, e(e_2) : \sigma} \quad (x \notin \mathrm{FV}(\sigma)) \qquad (\text{T-FUNAPP})$$

The application rule is limited to functors whose type can be expressed without dependences. As before, we rely on sharing propagation to eliminate dependences prior to application.

Exercise 10.5
Show that if $e : \Pi\, x{:}\sigma_1(\sigma_2)$, $v : \sigma_1'$, and $\sigma_1' \leq \sigma_1$, then there exists σ_2' such that $\texttt{inst}\, e(v) : \sigma_2'$. Thus applications of functors to values are always well-formed, provided that the types match up properly.

10.4 Second-Class Modules

A second-class module system is one in which the modularity constructs are segregated from the ordinary expression constructs. This allows the module language to avoid some of the difficulties associated with first-class modules, principally conditional module expressions. The two levels are linked by a construct for introducing a module declaration in the scope of an ordinary expression. The pay-off for this restriction is that the type system can be strengthened to take account of the limitations of the module language.

The syntax of a representative second-class module language is as follows:

$$
\begin{array}{llcl}
\textit{Signatures} & \Sigma & ::= & \mathtt{sig}\,u{::}\kappa\,\mathtt{with}\,\sigma \mid \Sigma\,x{:}\Sigma_1(\Sigma_2) \mid \Pi\,x{:}\sigma_1(\sigma_2) \\
\textit{Modules} & M & ::= & \mathtt{struct}\,\mu\,\mathtt{with}\,e\,\mathtt{as}\,\Sigma \mid M{:}\Sigma \\
 & & \mid & \mathtt{structure}\,M_1\,;M_2 \mid \mathtt{FST}(M) \mid \mathtt{SND}(M) \\
 & & \mid & \mathtt{functor}\,x{:}\Sigma(M) \mid \mathtt{inst}\,M_1(M_2)
\end{array}
$$

We retain the constructor form $\mathtt{Typ}(M)$, where M is any module expression other than a "sealed" module — the restriction to values is no longer necessary since the type component of a module is always well-defined, regardless of whether or not any constituent ordinary expressions are well-defined. We also retain the expression form $\mathtt{ops}(M)$, where M is a module expression, without restriction. Finally we add the expression form $\mathtt{module}\,x{:}\Sigma{=}M\;\mathtt{in}\,e$. This form defines a module variable x with signature Σ bound to module M in the scope of expression e. This construct provides a link between the module language and the core language.

The formation, subsignature, and signature equivalence rules may all be readily derived from those given above in the first-class case. The restriction to values in the "self" rules can be relaxed to include any module expression except a "sealed" module.

10.5 Higher-Order Modules

1. *Discussion of functor generativity.*
2. *Higher-order functor typing.*

10.6 References

The type-theoretic approach to modularity in programming languages was pioneered by MacQueen [?] and Burstall and Lampson [?]. MacQueen's approach was subsequently developed by Harper and Mitchell [?] who suggested a comprehensive type theory to encompass all of Standard ML. This suggestion was further refined by Harper, Mitchell, and Moggi [?] to account for the *phase distinction* in type theory [?]. These calculi did not, however, properly account for computational effects or separate compilation, nor did they provide an adequate treatment of sharing specifications. This was rectified by Harper and

Lillibridge [?] and Leroy [?]. The account given here is based on the latter two papers.

Chapter 11

Objects

11.1 Introduction

11.2 Primitive Objects

1. Methods and fields.
2. Self-reference.
3. Persistent case only; no mutation.
4. Records and objects.

11.3 Object Subtyping

1. Depth OK in fields and methods (what is the rule for methods?).
2. Width unsound in the absence of dictionaries.

11.4 Second-Order Object Types

1. Self-types.
2. Relation to recursive types.

11.5 References

82

Chapter 12

Dynamic Types

12.1 Type Dynamic

1. Basic primitives.

2. Absence of abstraction.

12.2 Hierarchical Tagging

1. SML `exn` type. Extensible sums.

2. Hierarchical, extensible sums.

Chapter 13

Classes

13.1 Introduction

13.2 Public, Private, and Protected

1. Public: type of generated objects.
2. Protected: interface of the class (for subclasses).
3. Private: via subsumption.

13.3 Constructors

1. Behavior under inheritance.
2. Overloading?

13.4 Subtyping and Inheritance

Subclasses may or may not induce subtypes.

13.5 Dynamic Dispatch

Hierarchical tagging for dynamic dispatch.

13.6 References

86

Part II

Computational Effects

Chapter 14

Recursive Functions

14.1 Introduction

In view of the termination theorem **??** for the simply-typed λ-calculus (with base types `Int` and `Bool`), something more is needed to achieve Turing equivalence. The simplest approach is to enrich $\mathcal{L}^{1,\times,\to}$ with the ability to define recursive functions. We call this language $\mathcal{L}^{1,\times,\rightharpoonup}$, using the "partial arrow" notation to indicate the possibility of defining partial functions. If we include the base types `Int` and `Bool`, the language is a call-by-value variant of Plotkin's language PCF, which we call $\mathbf{PCF}_v$.

14.2 Statics

The syntax of $\mathcal{L}^{1,\times,\rightharpoonup}$ is a minor variation of $\mathcal{L}^{1,\times,\to}$:

$$\begin{array}{llcl} \textit{Types} & \tau & ::= & \texttt{Unit} \mid \tau_1 \times \tau_2 \mid \tau_1 \rightharpoonup \tau_2 \\ \textit{Expressions} & e & ::= & x \mid * \mid <e_1,e_2>_{\tau_1,\tau_2} \mid \texttt{proj}^1_{\tau_1,\tau_2}(e) \mid \texttt{proj}^2_{\tau_1,\tau_2}(e) \mid \\ & & & \texttt{fun}\, f(x{:}\tau_1){:}\tau_2 \;\texttt{is}\; e \mid \texttt{app}_{\tau_1,\tau_2}(e_1,e_2) \end{array}$$

In the function expression $\texttt{fun}\, f(x{:}\tau_1){:}\tau_2 \;\texttt{is}\; e$, the variables f and x are bound in e, and may be consistently renamed without specific mention.

The only differences compared to $\mathcal{L}^{1,\times,\to}$ are the replacement of the type expression $\tau_1 \to \tau_2$ by the type expression $\tau_1 \rightharpoonup \tau_2$, and the replacement of $\texttt{fun}\,(x{:}\tau_1){:}\tau_2 \;\texttt{in}\; e$ by the "self-referential" form $\texttt{fun}\, f(x{:}\tau_1){:}\tau_2 \;\texttt{is}\; e$. The ordinary (non-recursive) function expression $\texttt{fun}\,(x{:}\tau_1){:}\tau_2 \;\texttt{in}\; e$ is defined to be the recursive function expression $\texttt{fun}\, f(x{:}\tau_1){:}\tau_2 \;\texttt{is}\; e$, where f is chosen differently from x and so as not to otherwise occur freely in e. In other words, a non-recursive function is a recursive function that happens to not be recursive.

The typing rules for $\mathcal{L}^{1,\times,\rightharpoonup}$ are essentially the same as for $\mathcal{L}^{1,\times,\to}$, with the modification that the rule T-ABS is replaced by the following rule:

$$\frac{\Gamma[f{:}\tau_1 \rightharpoonup \tau_2][x{:}\tau_1] \vdash e : \tau_2}{\Gamma \vdash \texttt{fun}\, f(x{:}\tau_1){:}\tau_2\, \texttt{is}\, e : \tau_1 \rightharpoonup \tau_2} \quad (f, x \notin \mathrm{dom}(\Gamma)) \qquad \text{(T-REC)}$$

Notice the use of "self-reference" in the above typing rule: to show that $\texttt{fun}\, f(x{:}\tau_1){:}\tau_2\, \texttt{is}\, e$ has type $\tau_1 \rightharpoonup \tau_2$, we assume that f has type $\tau_1 \rightharpoonup \tau_2$ and that the argument has type τ_1, and show that the body has type τ_2 under these assumptions.

14.3 Dynamics

The contextual semantics of $\mathcal{L}^{1,\times,\rightharpoonup}$ is obtained from that of $\mathcal{L}^{1,\times,\rightarrow}$ by generalizing the primitive reduction step for non-recursive functions to the following reduction for recursive functions:

$$\texttt{app}_{\tau_1,\tau_2}((\texttt{fun}\, f(x{:}\tau_1){:}\tau_2\, \texttt{is}\, e), v) \rightsquigarrow \{\texttt{fun}\, f(x{:}\tau_1){:}\tau_2\, \texttt{is}\, e/f\}\{v/x\}e$$

Note that the recursion is "unwound" by replacing f by the recursive function itself whenever it is applied. It is in this sense that recursive functions are "self-referential".

Exercise 14.1
Give an evaluation semantics for $\mathcal{L}^{1,\times,\rightharpoonup}$ and prove that it is equivalent to the reduction semantics just given.

Exercise 14.2
Extend the language to account for mutual recursion *by generalizing the recursive function form to (a) define $n \geq 1$ mutually recursive functions, and (b) project the ith (where $1 \leq i \leq n$) function from the n so defined. This expression form should be regarded as a value. Give suitable typing and evaluation rules for this extension.*

14.4 Type Soundness

The basic syntactic properties of typing (preservation and progress) may be readily extended to the case of recursively-defined functions. In particular, note that if $\vdash \texttt{app}_{\tau_2,\tau}((\texttt{fun}\, f(x{:}\tau_2){:}\tau\, \texttt{is}\, e), v) : \tau$, then $f{:}\tau_2 \rightharpoonup \tau, x{:}\tau_2 \vdash e : \tau$ and $\vdash v : \tau_2$, and consequently $\vdash \{\texttt{fun}\, f(x{:}\tau_2){:}\tau\, \texttt{is}\, e/f\}\{v/x\}e : \tau$.

Theorem 14.3
If $\vdash e : \tau$, then either e is a value or there exists e' such that $\vdash e' : \tau$ and $e \mapsto e'$.

Exercise 14.4
Prove Theorem 14.3.

14.5 Compactness

A fundamental property of $\mathcal{L}^{1,\times,\rightharpoonup}$ is the *compactness* of evaluation. Roughly speaking, this property states that for any given complete evaluation of an expression to a value, at most finitely many "recursive calls" of any given recursive function occurring in that expression are needed to complete the evaluation. While intuitively clear, this property is remarkably tricky to state and prove, mostly because of the presence of higher-order functions.

To illustrate the problems that arise, and to establish a framework for the proof of the compactness property, we extend the language of expressions of $\mathcal{L}^{1,\times,\rightharpoonup}$ to include *labelled recursive functions.*

$$\textit{Expressions} \quad e \quad ::= \quad \texttt{fun}^{(n)}\, f(x{:}\tau_1){:}\tau_2\,\texttt{is}\,e$$

Intuitively, the labelled function $\texttt{fun}^{(n)}\, f(x{:}\tau_1){:}\tau_2\,\texttt{is}\,e$ may make up to $n \geq 0$ recursive calls to itself, after which it diverges when applied.

The typing rule for labelled recursive functions is precisely the same as that for unlabelled recursive functions. The primitive reduction steps are as follows:

$$\begin{array}{rcl}\texttt{app}_{\tau_1,\tau_2}((\texttt{fun}^{(0)}\, f(x{:}\tau_1){:}\tau_2\,\texttt{is}\,e),v) & \leadsto & \texttt{app}_{\tau_1,\tau_2}((\texttt{fun}^{(0)}\, f(x{:}\tau_1){:}\tau_2\,\texttt{is}\,e),v) \\ \texttt{app}_{\tau_1,\tau_2}((\texttt{fun}^{(n+1)}\, f(x{:}\tau_1){:}\tau_2\,\texttt{is}\,e),v) & \leadsto & \{\texttt{fun}^{(n)}\, f(x{:}\tau_1){:}\tau_2\,\texttt{is}\,e, v/f, x\}e\end{array}$$

Notice that in the second rule the label on the recursive function is reduced by one on the recursive call.

Exercise 14.5
Check that the type soundness property holds for this extension of $\mathcal{L}^{1,\times,\rightharpoonup}$.

In the following exercise we explore the behavior of labelled recursive functions in $\mathbf{PCF}_v$.

Exercise 14.6

1. *Let fact be the obvious representation of the factorial function in $\mathcal{L}^{1,\times,\rightharpoonup}$, and let $\mathit{fact}^{(k)}$ be the labelling of that function with the number k. Show that for $m \geq 0$, $\mathit{fact}(\overline{m}) \mapsto^* \overline{n}$ iff for every $k \geq m+1$, $\mathit{fact}^{(k)}(\overline{m}) \mapsto^* \overline{n}$. Moreover, show that if $k \leq m$, then $\mathit{fact}^{(k)}(\overline{m})$ fails to terminate.*

2. *Let id_τ be the identity function at type τ, and let $id_\tau^{(k)}$ be its labelled form with label k[1]. Clearly $id_{\texttt{Int}\rightharpoonup\texttt{Int}}(\mathit{fact}) \mapsto^* \mathit{fact}$. Show that for $k \geq 1$, $id^{(k)}_{\texttt{Int}\rightharpoonup\texttt{Int}}(\mathit{fact}) \mapsto^* \mathit{fact}$ and, moreover, for $k \geq 1$ and $l \geq 0$, $id^{(k)}_{\texttt{Int}\rightharpoonup\texttt{Int}}(\mathit{fact}^{(l)}) \mapsto^* \mathit{fact}^{(l)}$.*

3. *Let $\mathit{apply}_{\tau_1,\tau_2}$ be $\texttt{fun}\,(f{:}\tau_1\rightharpoonup\tau_2)\,\texttt{in}\,\texttt{fun}\,(x{:}\tau_1)\,\texttt{in}\,f(x)$. Check the following assertions:*

   (a) *For any $k \geq 1$, $\mathit{apply}^{(k)}_{\texttt{Int},\texttt{Int}}(\mathit{fact}) \mapsto^* \texttt{fun}\,(x{:}\texttt{Int})\,\texttt{in}\,\mathit{fact}(x)$.*

[1] Remember that even the identity is (vacuously) a recursive function!

(b) *For any* $k \geq 1$ *and* $l \geq 0$, $\mathit{apply}^{(k)}_{\mathtt{Int},\mathtt{Int}}(\mathit{fact}^{(l)}) \mapsto^* \mathtt{fun}\,(x{:}\mathtt{Int})\,\mathtt{in}\,\mathit{fact}^{(l)}(x)$.

(c) *For any* $k \geq 1$, $m \geq 0$, $l \geq m+1$, $\mathit{apply}^{(k)}_{\mathtt{Int},\mathtt{Int}}(\mathit{fact}^{(l)})(\overline{m}) \mapsto^* \overline{m!}$.

As can be seen from the preceding exercise a difficulty with stating compactness is that labels can appear in the result of evaluation. Consequently we cannot expect to get the same result from a labelled evaluation as we do from an unlabelled one, even if we choose sufficiently large labels for the recursive functions involved.

To give a concise formulation of compactness we introduce the following notation. For a closed expression e, define $\mathit{Lab}(e)$ to be the set of *labellings* of e — expressions derived from e by labelling *every* recursive function occurring within e with a natural number. Note that from a labelling $e^* \in \mathit{Lab}(e)$ we can recover e by simply *erasing* the labels from the recursive functions. It is easy to check that labelling commutes with substitution and replacement, and that the labelled form of a value is again a value. Finally, define $\mathit{Lab}^{\geq k}(e) \subseteq \mathit{Lab}(e)$ to the the set of labellings of e with all labels at least k. Note that $\mathit{Lab}^{\geq l}(e) \subseteq \mathit{Lab}^{\geq k}(e)$ whenever $l \geq k$.

Theorem 14.7 (Compactness)

1. *If* $e \mapsto^* v$, *then there exists* $k \geq 0$ *such that for every* $e^* \in \mathit{Lab}^{\geq k}(e)$, *there exists* $v^* \in \mathit{Lab}(v)$ *such that* $e^* \mapsto^* v^*$.

2. *If* $e^* \mapsto^* v^*$ *with* $e^* \in \mathit{Lab}(e)$ *and* $v^* \in \mathit{Lab}(v)$, *then* $e \mapsto^* v$.

Proof: Both parts are proved by induction on the length of evaluation sequences.

1. For a length 0 computation, we have $e = v$. Take $k = 0$ and observe that $v^* \mapsto^* v^*$. Otherwise, let e_1 be such that $e \mapsto e_1 \mapsto^* v$. By inductive hypothesis there exists $k_1 \geq 0$ such that for every $e_1^* \in \mathit{Lab}^{\geq k_1}(e_1)$ there exists $v^* \in \mathit{Lab}(v)$ such that $e_1^* \mapsto^* v^*$. Since $e \mapsto e_1$, there exists an evaluation context E and a redex r such that $e = E[r]$, $e_1 = E[c]$, and $r \leadsto c$. We proceed by cases on the form of r. Suppose that $r = \mathtt{app}_{\tau,\tau'}(\mathtt{fun}\,f(x{:}\tau){:}\tau'\,\mathtt{is}\,e_2{,}v_2)$ and $c = \{\mathtt{fun}\,f(x{:}\tau){:}\tau'\,\mathtt{is}\,e_2, v_2/f, x\}e_2$. Take $k = k_1 + 1$, and let $e^* \in \mathit{Lab}^{\geq k}(e)$. Notice that

$$\begin{array}{rcll} e^* & = & E^*[r^*] & \\ & = & E^*[\mathtt{app}_{\tau,\tau'}(\mathtt{fun}^{(i)}\,f(x{:}\tau){:}\tau'\,\mathtt{is}\,e_2^*{,}v_2^*)] & (\exists i \geq k = k_1 + 1) \\ & \mapsto & E^*[\{\mathtt{fun}^{(i-1)}\,f(x{:}\tau){:}\tau'\,\mathtt{is}\,e_2^*, v_2^*/f, x\}e_2^*] & \\ & \in & \mathit{Lab}^{\geq k_1}(e_1). & \end{array}$$

   Applying the inductive hypothesis we obtain the required v^* such that $e^* \mapsto^* v^*$. The other cases for r follow similarly, taking $k = k_1$ since no recursive unrollings are involved in the reduction.

2. For a length 0 computation we have that $e^* = v^*$, and so $e = v$, from which it follows that $e \mapsto^* v$. Otherwise let e_1^* be such that $e^* \mapsto e_1^* \mapsto v^*$.

By the inductive hypothesis we have that $e_1 \mapsto^* v$. Since $e^* \mapsto e_1^*$, there exists an evaluation context E^*, a redex r^* and contractum c^* such that $e^* = E^*[r^*]$, $e_1^* = E^*[c^*]$ and $r^* \leadsto c^*$. We proceed by cases on the form of r^*. Suppose that $r^* = \mathtt{app}_{\tau,\tau'}(\mathtt{fun}^{(k)}\, f(x{:}\tau){:}\tau' \,\mathtt{is}\, e_2^*, v_2^*)$ (where $k > 0$) and $c = \{\mathtt{fun}^{(k-1)}\, f(x{:}\tau){:}\tau' \,\mathtt{is}\, e_2^*, v_2^*/f, x\}e_2^*$. By erasing labels we obtain $e = E[r]$, $e_1 = E[c]$, and $r \leadsto c$, hence $e \mapsto e_1$. The result follows by an application of the inductive hypothesis. The treatment of the other primitive reduction steps follows a similar pattern of reasoning. Note that the $k = 0$ case of reduction of a labelled recursive function cannot occur since we are considering only terminating reduction sequences.

■

Exercise 14.8
Complete the proof of Theorem 14.7.

14.6 References

Plotkin's study of PCF [?] is a landmark in the study of programming languages based on the typed λ-calculus. The suggested treatment of mutual recursion is adapted from the SML/NJ intermediate language.

Chapter 15

Continuations

15.1 Introduction

The call-by-value evaluation strategy has the virtue that the order of evaluation of expressions is predictable, in contrast to the call-by-need strategy in which expressions are evaluated only upon demand. Although sometimes considered a stricture, the deterministic character of the call-by-value strategy admits introduction of language constructs that, while mathematically sensible, are all but useless in a demand-driven language. In particular we may add the ability to dynamically change the flow of control during evaluation through the use of continuation-passing primitives.

15.2 Statics

The language $\mathcal{L}^{\rightarrow,\mathtt{Cont}}$ is an extension of $\mathcal{L}^{\rightarrow}$ whose syntax is defined as follows:[1]

Types	τ	$::=$	$b \mid \tau_1 \rightarrow \tau_2 \mid \tau\,\mathtt{Cont}$
Expressions	e	$::=$	$v \mid e_1(e_2) \mid \mathtt{letcc}\,x{:}\tau\,\mathtt{Cont}\,\mathtt{in}\,e \mid \mathtt{throw}\,e_1\,\mathtt{to}\,e_2$
Values	v	$::=$	$x \mid c \mid \mathtt{fun}\,(x{:}\tau)\,\mathtt{in}\,e \mid E$
Continuations	E	$::=$	$\bullet \mid E(e) \mid v(E) \mid \mathtt{throw}\,e\,\mathtt{to}\,E$

Note that continuations are evaluation contexts; we tend to use the word "continuation" when thinking of an evaluation context as a value to be passed in a program. This dual usage results in a syntactic ambiguity: $\mathtt{throw}\,e\,\mathtt{to}\,E$ may be read either as an expression in which E occurs as a value, as as a continuation in which the hole occurs as a sub-expression of the second argument of $\mathtt{throw}$. We rely on context to disambiguate.

The typing rules for the continuation primitives are defined as follows:

$$\frac{\Gamma[x{:}\tau\,\mathtt{Cont}] \vdash e : \tau}{\Gamma \vdash \mathtt{letcc}\,x{:}\tau\,\mathtt{Cont}\,\mathtt{in}\,e : \tau} \quad (x \notin \mathrm{dom}(\Gamma)) \qquad \text{(T-LETCC)}$$

[1] We restrict attention to non-recursive functions for the sake of simplicity, but there is no difficulty in extending $\mathcal{L}^{\rightharpoonup}$ with continuation primitives.

$$\frac{\Gamma \vdash e_1 : \tau \qquad \Gamma \vdash e_2 : \tau\,\texttt{Cont}}{\Gamma \vdash \texttt{throw}\,e_1\,\texttt{to}\,e_2 : \tau'} \tag{T-THROW}$$

Note well that the type of a `throw` expression is arbitrary!

It remains to define the typing rules for continuations. To do so we fix as a parameter of the type system a base type `ans` of "answers", the ultimate value of a complete program. (In a language with side effects `ans` might be chosen to be `Unit`, the one-element type; in our simplified setting it would be more natural to choose an "interesting" base type such as `Nat`.)

$$\frac{\Gamma[x{:}\tau] \vdash E[x] : \texttt{ans}}{\Gamma \vdash E : \tau\,\texttt{Cont}} \quad (x \notin \mathrm{dom}(\Gamma)) \tag{T-CONT}$$

Thus a continuation of type $\tau\,\texttt{Cont}$ may be thought of as a "function" from τ to `ans`. We distinguish continuations from functions in order to preserve the abstraction boundary; in particular, we wish to preserve the canonical forms property. The reason to establish a fixed answer type will become clearer once the evaluation rules for continuations are presented.

Evaluation contexts enjoy a "replacement" property that is stronger than the substitution property of arbitrary expressions:

Lemma 15.1 (Replacement)
If $\Gamma[x{:}\tau] \vdash E[x] : \tau'$ and $\Gamma \vdash e : \tau$, then $\Gamma \vdash E[e] : \tau'$.

Although the restriction to evaluation contexts is not necessary at present, we prefer to state only the form that we need for the development.

15.3 Dynamics

To describe the evaluation of $\mathcal{L}^{\rightarrow,\texttt{Cont}}$ expressions using reduction semantics we must generalize the "local" rewriting method considered earlier to a "global" method that keeps track of the context of evaluation. This is motivated by the informal understanding of the evalation of $\texttt{letcc}\,x{:}\tau\,\texttt{Cont}\,\texttt{in}\,e$: the "current continuation" (evaluation context) is substituted for x during evaluation of e. Thus the evaluation context must be explicitly represented as a value, and the rewriting rules must be defined in such a way that the context of evaluation is apparent. We therefore abandon the separation between "instructions" and "evaluation steps" considered earlier, in favor of a direct specification of the one-step evaluation relation.

Definition 15.2
The one-step evaluation relation $e \mapsto e'$ for $\mathcal{L}^{\rightarrow,\texttt{Cont}}$ is defined as follows:

$$\begin{array}{rcl}
E[(\texttt{fun}\,(x{:}\tau)\,\texttt{in}\,e)(v)] & \mapsto & E[\{v/x\}e] \\
E[\texttt{letcc}\,x{:}\tau\,\texttt{Cont}\,\texttt{in}\,e] & \mapsto & E[\{E/x\}e] \\
E[\texttt{throw}\,e\,\texttt{to}\,E'] & \mapsto & E'[e]
\end{array}$$

The reduction step for `throw` applies for arbitrary e. We could equivalently require that e be a value, and extend the set of evaluation contexts to include those of the form $\texttt{throw}\, E\, \texttt{to}\, e$. However, it is easy to see that insertion of an expression into an evaluation context ensures that that expression is evaluated before computation proceeds. We adopt the above definition to simplify the meta-theory below.

Notice that, as before, the application of a function to a value occurs "in place" in the sense that the evaluation context is not disturbed. Evaluation of a `letcc` expression also occurs in place, but in addition the evaluation context is *duplicated* and bound to x in e. Evaluation of a `throw` expression causes the evaluation context to be *abandoned* and replaced by the target of the `throw`. It is immediately obvious that the direct definition of the one-step evaluation relation generalizes the earlier approach in which the basic instructions are defined separately from their context of use.

Exercise 15.3
Define a function `ccwf` *that takes a continuation* k *of type* $\tau\,\texttt{Cont}$ *and a function* f *of type* $\tau' \to \tau$ *and yields a continuation of type* $\tau'\,\texttt{Cont}$ *that, when passed a value* v' *of type* τ'*, passes* $f(x')$ *to* k.

It is interesting to consider an evaluation semantics in the presence of continuations.

Definition 15.4
The relation $E \vdash e \Downarrow v$ *is inductively defined by the following rules:*

$$\bullet \vdash v \Downarrow v \qquad (\text{E-STOP})$$

$$\frac{\bullet \vdash E[v] \Downarrow v'}{E \vdash v \Downarrow v'} \quad (E \neq \bullet) \qquad (\text{E-VALUE})$$

$$\frac{E[\bullet(e_2)] \vdash e_1 \Downarrow v}{E \vdash e_1(e_2) \Downarrow v} \qquad (\text{E-APP-FN})$$

$$\frac{E[v_1(\,\bullet\,)] \vdash e_2 \Downarrow v}{E \vdash v_1(e_2) \Downarrow v} \qquad (\text{E-APP-ARG})$$

$$\frac{E \vdash \{v/x\}e \Downarrow v'}{E \vdash (\texttt{fun}\,(x{:}\tau)\,\texttt{in}\,e)(v) \Downarrow v'} \qquad (\text{E-APP})$$

$$\frac{E \vdash \{E/x\}e \Downarrow v}{E \vdash \texttt{letcc}\, x{:}\tau\,\texttt{Cont}\,\texttt{in}\, e \Downarrow v} \qquad (\text{E-LETCC})$$

$$\frac{E[\texttt{throw}\, e_1\, \texttt{to}\, \bullet] \vdash e_2 \Downarrow v}{E \vdash \texttt{throw}\, e_1\, \texttt{to}\, e_2 \Downarrow v} \qquad (\text{E-THROW-CONT})$$

$$\frac{E' \vdash e \Downarrow v}{E \vdash \texttt{throw}\, e\, \texttt{to}\, E' \Downarrow v} \qquad (\text{E-THROW})$$

Note that the process of "search" for the next evaluation step is explicit in the evaluation semantics.

Theorem 15.5

1. *If* $E \vdash e \Downarrow v$*, then* $E[e] \mapsto^* v$.
2. *If* $E[e] \mapsto E'[e']$ *and* $E' \vdash e' \Downarrow v$*, then* $E \vdash e \Downarrow v$.
3. $E \vdash e \Downarrow v$ *iff* $\bullet \vdash E[e] \Downarrow v$.
4. *If* $e \mapsto^* v$*, then* $\bullet \vdash e \Downarrow v$.

Exercise 15.6
Prove Theorem 15.5.

Exercise 15.7
The evaluation semantics given above builds the evaluation context "inside out" by extending E with an evaluation frame *representing the suspension of a primitive operation during evaluation of a constituent expression. This suggests making explicit the operation of extending an evaluation frame in the semantics. For the purposes of this exercise let us work with the following definition of evaluation context:*

$$\begin{array}{lcll} \mathit{EvalContext} & E & ::= & \bullet \mid F;E \\ \mathit{EvalFrame} & F & ::= & \bullet(e) \mid v(\bullet) \mid \texttt{throw}\, e\, \texttt{to}\, \bullet \end{array}$$

1. *Re-formulate the evaluation semantics given in Definition 15.4 by simultaneously defining the relations $E \vdash e \Downarrow v$ and $v \vdash E \Downarrow v'$, where the latter relation expresses that the continuation E, when thrown the value v, yields the value v'. (Hint: Define $v \vdash E \Downarrow v'$ by induction on the structure of E; in the case that $E = F;E'$, proceed by cases on F. Modify the definition $E \vdash e \Downarrow v$ so that values are passed to continuations using this relation.)*
2. *State and prove the relationship between this version of the evaluation semantics and the one given in Definition 15.4.*

15.4 Soundness

The preservation theorem is stated for complete programs, *i.e.*closed expressions of type `ans`.

Theorem 15.8 (Preservation)
If $\vdash e : \texttt{ans}$ *and* $e \mapsto e'$*, then* $\vdash e' : \texttt{ans}$.

Proof: We proceed by induction on the structure of e. Since $e \mapsto e'$, there exists E and e_i such that $e = E[e_i]$ and $E[e_i]$ is reducible according to the definition of one-step evaluation. Since $\vdash e : \texttt{ans}$, there exists a unique type τ_i

such that $\vdash e_i : \tau_i$ and $\vdash E : \tau_i\,\texttt{Cont}$. We show that $\vdash e' : \texttt{ans}$ by case analysis on e_i.

If $e_i = (\texttt{fun}\,(x{:}\tau)\,\texttt{in}\,e_i')(v)$, then $e' = E[\{v/x\}e_i']$, and it is a simple matter to check that $\vdash e' : \texttt{ans}$.

Suppose that $e_i = \texttt{letcc}\,x{:}\tau\,\texttt{Cont}\,\texttt{in}\,e$. Then $e' = E[\{E/x\}e]$, and it suffices to show that $\vdash \{E/x\}e : \tau_i$. This follows from the fact that $x{:}\tau\,\texttt{Cont} \vdash e : \tau$ and $\vdash E : \tau\,\texttt{Cont}$ by the substitution lemma.

Suppose that $e_i = \texttt{throw}\,e_i'\,\texttt{to}\,E_i'$ so that $e' = E_i'[e_i']$. Since $\vdash \texttt{throw}\,e_i'\,\texttt{to}\,E_i' : \tau_i$, it follows that there exists τ_i' such that $\vdash e_i' : \tau_i'$ and $\vdash E_i' : \tau_i\,\texttt{Cont}$, which suffices for the result by the replacement lemma. ■

Lemma 15.9 (Canonical Forms)
Suppose that v is a closed value of type τ. If $\tau = b$, then $v = c$ for some constant c of type b; if $\tau = \tau_1 \to \tau_2$, then $v = \texttt{fun}\,(x{:}\tau_1)\,\texttt{in}\,e$ for some x and e; if $\tau = \tau\,\texttt{Cont}$, then $v = E$ for some continuation E.

Theorem 15.10 (Progress)
If $\vdash e : \tau$, then either e is a value, or there exists e' such that $e \mapsto e'$.

Proof: We proceed by induction on the structure of e. If e is not a value, then either it is an application, a `letcc`, or a `throw`. The `letcc` case is immediate since $\texttt{letcc}\,x{:}\tau\,\texttt{Cont}\,\texttt{in}\,e \mapsto \{\bullet/x\}e$. Suppose that $e = \texttt{throw}\,e_1\,\texttt{to}\,e_2$. If e_2 is a value, then by the canonical forms lemma it must be a continuation E, and hence $e \mapsto E[e_1]$. If e_2 is not a value, then by inductive hypothesis there exists e_2' such that $e_2 \mapsto e_2'$. It follows that $e_2 = E_2[e_i]$ for some E_2 and e_i. Taking $E = \texttt{throw}\,e_1\,\texttt{to}\,E_2$, we note that $e = E[e_i]$; we proceed by cases on e_i. If $e_i = \texttt{fun}\,(x{:}\tau)\,\texttt{in}\,e_i'$, then $e = E[e_i] \mapsto E[\{v/x\}e_i']$. If $e_i = \texttt{letcc}\,x{:}\tau_i\,\texttt{Cont}\,\texttt{in}\,e_i'$, then $e = E[e_i] \mapsto E[\{E/x\}e_i']$. If $e_i = \texttt{throw}\,e_i'\,\texttt{to}\,E_i'$, then $e = E[\texttt{throw}\,e_i'\,\texttt{to}\,E_i'] \mapsto E_i'[e_i']$. ■
Note that e' is not necessarily well-typed (unless $\tau = \texttt{ans}$) because of the typing restrictions on continuations.

Exercise 15.11
Complete the proof of the progress theorem for $\mathcal{L}^{\to,\texttt{Cont}}$.

The continuation-passing primitives `letcc` and `throw` are the "goto's" of functional programming. In particular, `letcc` establishes a "label" as the target for a future jump instruction, and `throw` executes the jump, providing a value to use as the value of the labelled expression. Although "goto's" ordinarily may be used to define infinite loops, it is a remarkable fact that the typing conditions imposed here ensure that $\mathcal{L}^{\to,\texttt{Cont}}$ programs terminate. This fact will follow from our work in Chapter 22.

15.5 References

For historical context Reynolds's article [?] on the history of continuations in programming languages is highly recommended. Early studies of particular importance were carried out by Fischer [?], Reynolds [?], and Plotkin [?].

Chapter 16

References

16.1 Introduction

A unique aspect of ML is the strict segregation of mutable from immutable data structures through the use of *reference* types. A value of type $\tau\,\mathtt{Ref}$ is a *pointer* or *reference* to a cell containing a value of type τ. The contents of the cell may be retrieved by explicitly *de-referencing* the pointer; the contents may be changed by *assigning* a value to the cell given by the pointer. References may occur as components of complex data structures; sharing is achieved by replication of the reference to the cell. References may be tested for equality, with the intention that two references are equal iff they determine the same memory cell. In contrast to LISP pointer equality is defined *only* for reference types, and not for other data structures (which may or may not be implemented using pointers, and which may have different sharing properties in different implementations of the language).

16.2 Statics

The language $\mathcal{L}^{\rightarrow,\mathtt{Ref}}$ is defined by the following grammar:

$$
\begin{array}{lcll}
\textit{Expressions} & e & ::= & v \mid e_1(e_2) \mid \mathtt{letref}\, d\, \mathtt{in}\, e \mid \mathtt{!}\, e \mid e_1 := e_2 \\
\textit{Memories} & d & ::= & x_1{:}\tau_1\ \mathtt{is}\, v_1\ \mathtt{and}\ \cdots\ \mathtt{and}\, x_n{:}\tau_n\ \mathtt{is}\, v_n \\
\textit{Values} & v & ::= & c \mid x \mid \mathtt{fun}\,(x{:}\tau)\,\mathtt{in}\, e \\
\textit{Programs} & p & ::= & \mathtt{letref}\, d\, \mathtt{in}\, e \\
\textit{Answers} & a & ::= & \mathtt{letref}\, d\, \mathtt{in}\, v
\end{array}
$$

Note that cells may *only* be bound to values, and not to arbitrary expressions. The binding conventions for `letref` expressions are essentially the same as those governing `letrec` expressions given in Exercise 14.2.

The expression $\mathtt{ref}\,(e) : \tau$ is defined to be $\mathtt{let}\, x{:}\tau\ \mathtt{be}\, e\ \mathtt{in}\, \mathtt{letref}\, y\ \mathtt{in}\, \tau\,\mathtt{Ref} xy$; the motivation for this definition will become apparent once the operational semantics has been given.

16.3 Dynamics

A reduction semantics for $\mathcal{L}^{\rightarrow,\texttt{Ref}}$ may be given by the method of program rewriting used in previous chapters. Evaluation contexts are defined as follows:

$$E \quad ::= \quad \bullet \mid E(e) \mid v(E) \mid\, !\,E \mid E := e \mid v := E$$

The relation $p \mapsto p'$ is defined for closed programs p as follows:

$$\begin{array}{rcl}
\texttt{letref}\, d \,\texttt{in}\, E[\texttt{fun}\,(x{:}\tau)\,\texttt{in}\, e(v)] & \mapsto & \texttt{letref}\, d \,\texttt{in}\, E[\{v/x\}e] \\
\texttt{letref}\, d \,\texttt{in}\, E[\texttt{letref}\, d' \,\texttt{in}\, e] & \mapsto & \texttt{letref}\, d \,\texttt{and}\, d' \,\texttt{in}\, E[e] \\
\texttt{letref}\ \cdots x{:}\tau \,\texttt{is}\, v \cdots \,\texttt{in}\, E[!\,x] & \mapsto & \texttt{letref}\ \cdots x{:}\tau \,\texttt{is}\, v \cdots \,\texttt{in}\, E[v] \\
\texttt{letref}\ \cdots x{:}\tau \,\texttt{is}\, v \cdots \,\texttt{in}\, E[x := v'] & \mapsto & \texttt{letref}\ \cdots x{:}\tau \,\texttt{is}\, v' \cdots \,\texttt{in}\, E[v']
\end{array}$$

The scope of locally-allocated references is extended to encompass the entire program, relying on α-conversion to avoid conflicts.

Evaluation of a closed expression e is performed by considering the program $\texttt{letref in}\, e$ — evaluation commences with an "empty" memory.

Exercise 16.1
Evaluate `ref` (9) *using the definition given above. What answer is returned?*

Exercise 16.2
Give an evaluation semantics for $\mathcal{L}^{\rightarrow,\texttt{Ref}}$ *in which the evaluation relation has the form* $(d, e) \Downarrow (d', v)$.

16.4 Type System

The syntax of type expressions is defined as follows:

$$\tau \quad ::= \quad b \mid \tau_1 \rightarrow \tau_2 \mid \tau\,\texttt{Ref}$$

Typing judgements have the form $\Gamma \vdash e : \tau$, where Γ is a type context mapping variable to types.

The typing rules governing the new constructs are as follows:

$$\frac{\Gamma \vdash d : \Gamma_d \qquad \Gamma + \Gamma_d \vdash e : \tau}{\Gamma \vdash \texttt{letref}\, d \,\texttt{in}\, e : \tau} \quad (\mathrm{dom}(\Gamma) \cap \mathrm{dom}(\Gamma_d) = \emptyset) \qquad (\text{T-LETREF})$$

$$\frac{\Gamma \vdash e : \tau\,\texttt{Ref}}{\Gamma \vdash\, !\,e : \tau} \qquad (\text{T-DEREF})$$

$$\frac{\Gamma \vdash e_1 : \tau\,\texttt{Ref} \qquad \Gamma \vdash e_2 : \tau}{\Gamma \vdash e_1 := e_2 : \tau} \qquad (\text{T-ASSIGN})$$

$$\frac{\Gamma + \Gamma' \vdash v_i : \tau_i \quad \Gamma'(x_i) = \tau_i\,\texttt{Ref} \quad (1 \leq i \leq n)}{\Gamma \vdash x_1{:}\tau_1 \,\texttt{is}\, v_1 \,\texttt{and}\, \cdots \,\texttt{and}\, x_n{:}\tau_n \,\texttt{is}\, v_n : \Gamma'} \quad (\mathrm{dom}(\Gamma) \cap \mathrm{dom}(\Gamma') = \emptyset)$$

(T-MEM)

The context Γ_d in the rule T-LETREF is sometimes called a "store typing". This rule is similar to that for `letrec` declaration lists; in particular, reference cells can be mutually recursive due to assignment statements.

Exercise 16.3
*Show that single recursion can be encoded using assignment. (*Hint: *use "back-patching".) Prove your encoding correct by arguing that your encoding simulates the operational semantics of single recursion. How would you extend your encoding to deal with mutual recursion?*

Assume that `ans` is a fixed, but unspecified, base type of answers. Note that the answer $a = \texttt{letref}\,d\,\texttt{in}\,v$ has type `ans` iff $\vdash d : \Gamma_d$ and $\Gamma_d \vdash v : \texttt{ans}$ for some context Γ_d. Thus the memory must be well-typed and the returned value must be a constant of type `ans`.

Theorem 16.4 (Soundness)
If $\vdash p : \texttt{ans}$*, then either* p *is an answer, or there exists* p' *such that* $\vdash p' : \texttt{ans}$ *and* $p \mapsto p'$*.*

Corollary 16.5
If $\vdash p : \texttt{ans}$ *and* $p \Downarrow \texttt{letref}\,d\,\texttt{in}\,v$*, then* $v = c : \texttt{ans}$*.*

Exercise 16.6
Prove the soundness theorem for reference types. For preservation argue that if $\texttt{letref}\,d\,\texttt{in}\,e \mapsto \texttt{letref}\,d'\,\texttt{in}\,e'$*, and* Γ_d *is such that* $\vdash d : \Gamma_d$ *and* $\Gamma_d \vdash e : \texttt{ans}$*, then there exists* Γ'_d *extending* Γ_d *such that* $\vdash d' : \Gamma'_d$ *and* $\Gamma'_d \vdash e' : \texttt{ans}$*. For progress analyze the canonical forms of each type, and argue that some evaluation rule applies to any non-answer.*

Exercise 16.7
Suppose that the language is modified to admit declarations of the form $\texttt{letref}\,x_1{:}\tau_1\,\texttt{is}\,e_1\,\texttt{and}\,\cdots\,\texttt{and}\,x_n{:}\tau_n\,\texttt{is}\,e_n\,\texttt{in}\,e$*, where the bindings to the variables need not be values. What difficulties do you encounter when attempting to give a reduction semantics for this extension?*

16.5 Allocation and Collection

The reduction semantics for $\mathcal{L}^{\rightarrow,\texttt{Ref}}$ is defined in such a way that all allocated cells are accumulated at top level, and never de-allocated. While this may not be problematic for most theoretical purposes, it is clear that for practical purposes cells some form of de-allocation of unused storage must be provided.

Let p be the closed program $\texttt{letref}\,d\,\texttt{in}\,e$ of basic type `ans`, where

$$d = x_1{:}\tau_1\,\texttt{is}\,v_1\,\texttt{and}\,\cdots\,\texttt{and}\,x_n{:}\tau_n\,\texttt{is}\,v_n.$$

A variable x defined in d is said to be *garbage* (relative to p) iff the outcome of the computation of p is independent of x. More precisely, $p \Downarrow \texttt{letref}\,d'\,\texttt{in}\,v$ iff $p_x \Downarrow \texttt{letref}\,d''\,\texttt{in}\,v$, where p_x is obtained from p by dropping the binding for

x from d. (Consequently p_x may fail to be a closed program since e or the v_i's may have a free occurrence of x; evaluation is still sensible for such programs, provided that we recognize that we may get "stuck" if an unbound variable is to be evaluated.)

Exercise 16.8
Consider the extension of $\mathcal{L}^{\rightarrow,\texttt{Ref}}$ with recursive functions and integers so that all partial recursive functions may be programmed. Prove that it is recursively undecidable to determine whether or not a given cell is garbage relative to a given program.

A *collection* of the program p is obtained by removing zero or more garbage cells from p's memory. In view of the undecidability of determining whether or not a cell is garbage, it is impossible to effectively compute a *complete* collection of a program, *i.e.*, one for which all garbage cells have been eliminated. In practice conservative approximations are used instead. The most common approximation is based on the *accessibility* criterion — a cell is accessible iff it occurs freely in the program or in the contents of a cell accessible from a cell occurring freely in the program. It is clear that inaccessible cells must be garbage (why?). Typical collectors proceed by determining the set of accessible cells in a given program, disposing of those that are inaccessible.

Exercise 16.9
Suppose that we enrich the reduction semantics with the following free variable *rule:*

$$\texttt{letref}\, d \,\texttt{and}\, d' \,\texttt{in}\, e \mapsto_{fv} \texttt{letref}\, d \,\texttt{in}\, e$$

provided that no variable bound in d' occurrs free in $\texttt{letref}\, d \,\texttt{in}\, e$ (i.e., no dangling references are introduced by the removal of d').

1. *Prove that the fv rule commutes with all other evaluation rules, i.e., if $p_1 \mapsto_{fv} p_2 \mapsto_r p_3$, then there exists p_2' such that $p_1 \mapsto_r p_2' \mapsto_{fv} p_3$.*

2. *Conclude that the outcome of a computation is not affected by garbage collection according to the free variable rule.*

Exercise 16.10
A copying garbage collector *computes a collection of a program p by eliminating all inaccessible locations from p's memory. Give an algorithm for copying garbage collection in the form of a transition system whose states are triples (d, s, d'), where d and d' are memories and s is a set of variables. The set s is called the* scan set, *d is called* from space, *and d' is called* to space. *The initial state of the collector for the program $p = \texttt{letref}\, d \,\texttt{in}\, e$ is the triple $(d, \mathrm{FV}(e), \emptyset)$, where $\mathrm{FV}(e)$ is the set of free variables of e, and the final state is the triple $(d, \emptyset, d')$. Transitions consist of a systematic exploration of* from space, *copying accessible locations into* to space. *Argue that a copying collector is an implementation of the free variable rule for garbage collection, and hence does not affect the outcome of a computation.*

Exercise 16.11
In practice allocation of memory is not restricted to reference cells. Give an operational semantics for $\mathcal{L}^{\rightarrow}$ extended with a base type of integers such that numbers and functions are allocated in memory, with implicit retrieval during evaluation. For example, if x_1 is bound to $\mathtt{fun}\,(x{:}\mathtt{Int})\,\mathtt{in}\,e$ *and x_2 is bound to 3, then the application $x_1(x_2)$ proceeds by evaluating $\{3/x\}e$. You will need to distinguish* heap values *(which occur in cells) from* program values *(which are restricted to variables) in order to give the semantics.*

To do:

1. Arrays.
2. Mutable fields in records and objects.

16.6 References

The treatment of references given here was inspired by the work of Felleisen and Wright [?, ?]. The approach to proving soundness through a reduction semantics is adapted from Felleisen and Wright and Harper [?].

Chapter 17

Exceptions

17.1 Introduction

An important feature of ML is the *exception* mechanism which allows programs to effect non-local "jumps" in the flow of control by setting a *handler* during evaluation of an expression that may be invoked by *raising* an exception. Exceptions are value-carrying in the sense that one may pass a value to the exception handler when the exception is raised. Because of the dynamic nature of exception handlers, it is required that exception values have a single type, `Exn`, which is shared by all exception handlers. Although this seems on the surface to be restrictive, Standard ML compensates for this restriction by defining `Exn` to be an *extensible data type* — new constructors may be introduced using the `exception` declaration, with no restriction on the type of value that may be associated with the constructor. Since extensible data types are separable from the exception mechanism itself, we will simply assume that there is a distinguished base type `Exn` of exception values, and not concern ourselves here with how values of this type are created or used. (But see Chapter 12 for a detailed discussion.)

17.2 Statics

The language $\mathcal{L}^{\rightarrow,\texttt{Exn}}$ is the enrichment of $\mathcal{L}^{\rightarrow}$ defined by the following grammar:

$$\begin{array}{llll} \textit{Expressions} & e & ::= & \ldots \mid \texttt{try}\, e_1 \,\texttt{ow}\, e_2 \mid \texttt{raise}\, e \\ \textit{Answers} & a & ::= & v \mid \texttt{raise}\, v \end{array}$$

Values remain as for $\mathcal{L}^{\rightarrow}$. An *answer* is either a value or an *uncaught exception*.

Let `Exn` stand for a fixed, but unspecified, base type of exception values.[1]

[1]The restriction to base type is not strictly necessary, but is made to simplify the development.

The typing rules governing the primitives of $\mathcal{L}^{\rightarrow,\texttt{Exn}}$ are as follows:

$$\frac{\Gamma \vdash e_1 : \tau \qquad \Gamma \vdash e_2 : \texttt{Exn} \to \tau}{\Gamma \vdash \texttt{try}\, e_1 \,\texttt{ow}\, e_2 : \tau} \quad (\text{T-TRY})$$

$$\frac{\Gamma \vdash e : \texttt{Exn}}{\Gamma \vdash \texttt{raise}\, e : \tau} \quad (\text{T-RAISE})$$

17.3 Dynamics

The main difficulty with giving a reduction semantics for exceptions is to take account of the dynamic binding of handlers. When evaluating a `try` expression we must "install" a handler to catch all uncaught exceptions raised during evaluation of the expression we are trying to evaluate. Of course, inner handlers take priority over outer handlers, so there is a dynamic call chain associated with exception handlers that must be maintained. The trick to doing this is to define a suitable notion of evaluation context to ensure that we correctly isolate the innermost handler (if any) during evaluation.

Evaluation contexts are defined as follows.

$$\mathit{EvalCtxt} \quad E \quad ::= \quad \bullet \mid E(e) \mid v(E) \mid \texttt{raise}\, E \mid \texttt{try}\, E \,\texttt{ow}\, e$$

To capture propagation of exceptions we also define a class of "raise frames" as follows:

$$\mathit{RaiseFrame} \quad R \quad ::= \quad \bullet(e) \mid v(\,\bullet\,) \mid \texttt{raise}\,\bullet$$

The idea is that exceptions propagate through raise frames.

The reduction semantics for programs is defined by the following rules:

$$\begin{array}{rcl} E[(\texttt{fun}\,(x{:}\tau)\,\texttt{in}\,e)(v)] & \mapsto & E[\{v/x\}e] \\ E[\texttt{try}\,\texttt{raise}\,v\,\texttt{ow}\,e] & \mapsto & E[e(v)] \\ E[\texttt{try}\,v\,\texttt{ow}\,e] & \mapsto & E[v] \\ E[R[\texttt{raise}\,v]] & \mapsto & E[\texttt{raise}\,v] \end{array}$$

Application steps may occur anywhere, and are evaluated as usual. A `raise` expression occurring within the scope of a handler passes the raised value to that handler. A successful evaluation of an expression with an associated handler causes the handler to be abandoned and the value to be returned. Finally, exceptions propagate through raise frames.

Exercise 17.1
Suppose that evaluation contexts of the form $\texttt{raise}\, E$ *are dropped, and the reduction rule for catching exceptions is replaced by the following rule:*

$$E[\texttt{try}\,\texttt{raise}\,e\,\texttt{ow}\,e'] \mapsto E[e'(e)].$$

Suppose further that the propagation rule for exceptions is generalized to

$$E[R[\texttt{raise}\,e]] \mapsto E[\texttt{raise}\,e]$$

so that exceptions are propagated in un-evaluated form.

1. *What happens to uncaught exceptions arising from e? Is this reasonable?*
2. *Give an example of a program which evaluates differently under the two semantics. Discuss the merits of the two approaches.*

Exercise 17.2
Give a natural semantics for exceptions based on the idea that an answer *is either a value v or an "exception packet" $\texttt{raise}\,v$.*

Theorem 17.3 (Determinacy)
For every e there is at most one answer a such that $e \mapsto^* a$.

Exercise 17.4
Give a proof of determinacy by arguing that if e is not an answer, then at most one reduction rule applies to e.

Since the evaluation relation is defined with respect to an evaluation context, it becomes important to consider the type of answers and the initial continuation. To prove preservation we need make no assumptions about the "answer" type.

Theorem 17.5 (Preservation)
If $\vdash e : \texttt{ans}$ and $e \mapsto e'$, then $\vdash e' : \texttt{ans}$.

Corollary 17.6
If $\vdash e : \texttt{ans}$ and $e \mapsto^* a$, then either $a = c : \texttt{ans}$ or $a = \texttt{raise}\,v$ for some v.

Exercise 17.7
Give a proof of the soundness theorem.

Well-typed programs do not "get stuck". They may, however, raise an exception.

Theorem 17.8 (Progress)
If $\vdash e : \tau$, then either e is an answer, or there exists e' such that $e \mapsto e'$.

(We need consider expressions of arbitrary type τ in order to get the induction to go through.)

Exercise 17.9
Give a proof of the progress theorem.

In practice the answer type is a fixed base type, and the initial context is chosen so that both "normal" and "error" returns yield a value of this type. For example, we may choose $\texttt{ans} = \texttt{String}$ and define the initial context for an expression of type τ to be

$$E_\tau := \texttt{try}\,\texttt{makestring}_\tau(\bullet)\,\texttt{ow}\,\texttt{fun}\,(x{:}\texttt{Exn})\,\texttt{in}\,\texttt{"uncaught exception"}.$$

Evaluation of $e : \tau$ commences by considering the expression $E_\tau[e]$, terminating with a string representation of the result of evaluation (if there is any result at all).

17.4 Exceptions and First-Class Continuations

It is interesting to consider the interaction between exceptions and first-class continuations. Let us suppose that we naïvely combine $\mathcal{L}^{\rightarrow,\texttt{Exn}}$ with $\mathcal{L}^{\rightarrow,\texttt{Cont}}$ by simply merging their syntax, type rules, and evaluation rules. Although the typing and evaluation rules are well-defined, the situation we find ourselves in is far from clear.

The following example explores the semantics of `throw` in the presence of exceptions.

```
letcc k0 : int cont in let k : int cont be letcc r : int cont cont in throw
  (try (letcc k : int cont in throw k to r) ow λ x:exn.17) to k0 in
  try (throw raise 0 to k) ow λ x:exn.1
```

This expression evaluates to `17`, not `1`! The reason is that the evaluation rule for `throw`

$$E[\texttt{throw}\,e\,\texttt{to}\,E'] \mapsto E'[e]$$

passes e unevaluated to its continuation. Thus, in particular,

$$E[\texttt{throw}\,\texttt{raise}\,0\,\texttt{to}\,E'] \mapsto E'[\texttt{raise}\,0]$$

so that E' has the opportunity (seized in the above example) to handle the exception. This is arguably a mistake (and is certainly not the behavior we get in SML). The "correct" (or at any rate preferred) rule is

$$E[\texttt{throw}\,v\,\texttt{to}\,E'] \mapsto E'[v],$$

where the set of evaluation contexts is enriched to include those of the form $\texttt{throw}\,E\,\texttt{to}\,e$ and is modified to replace $\texttt{throw}\,e\,\texttt{to}\,E$ by $\texttt{throw}\,v\,\texttt{to}\,E$. Under this definition of evaluation, the above expression evaluates to `1`, as might have been expected.

The second example explores the behavior of uncaught exceptions raised by a continuation.

```
letcc k0 : int cont in let k : int cont be letcc r : int cont cont in throw if
  (letcc k : int cont in throw k to r) = 0 then 1 else raise 0 to k0 in try
  (throw 1 to k) ow λx:exn.17
```

This expression evaluates to `raise 0`, rather than `17` — throwing to a continuation is "irrevocable" in the sense that computation cannot be resumed at the point of the `throw`, even if the continuation raises an exception. Formally, this behavior arises because the seized evaluation context bound to the variable `k` is

```
let k : int cont be if •=0 then 1 else raise 0 in try (throw 1 to k) ow
  λx:exn.17
```

If the hole is filled with any number other than `0`, then an exception is raised, which aborts the `let` and yields `raise 0` as final answer.

17.5 References

The treatment of exceptions is inspired by the definition of Standard ML [?] and Felleisen and Wright's account [?].

Part III

Implementation

Chapter 18

Type Checking

18.1 Introduction

18.2 Type Synthesis

The typing rules for $\mathcal{L}_{\Omega}^{\rightarrow,\forall_p}$ are not syntax-directed. The rule T-EQUIV, expressing invariance of typing under definitional equality of types, applies to an arbitrary expression e. As a result there are infinitely many derivations of $\Gamma \vdash_{\Delta} e : \sigma$, if there are any at all. This raises the question of whether type checking for $\mathcal{L}_{\Omega}^{\rightarrow,\forall_p}$ is decidable: given Δ, Γ, e, σ, can we effectively determine whether or not $\Gamma \vdash_{\Delta} e : \sigma$?

We proceed by reducing the type checking problem to the type equivalence problem by introducing the technique of *type synthesis*. The main observation is that typing derivations can be put into a standard form in which the rule T-EQUIV is used only at function applications. The standardized typing derivations are defined by a syntax-directed *type synthesis* relation, $\Gamma \vdash_{\Delta} e \Rightarrow \sigma_e$, which determines a *canonical* type σ_e for e, if e has a type at all. We then prove that $\Gamma \vdash_{\Delta} e : \sigma$ iff $\Delta \vdash \sigma \equiv \sigma_e$.

Exercise 18.1

1. *Define a type synthesis relation $\Gamma \vdash_{\Delta} e \Rightarrow \sigma_e$ by induction on the structure of e with the property that*

$$\Gamma \vdash_{\Delta} e : \sigma \textit{ iff } \Gamma \vdash_{\Delta} e \Rightarrow \sigma_e \textit{ and } \Delta \vdash \sigma \equiv \sigma_e.$$

   Prove that the type synthesis relation you define has this property.

2. *Argue that type checking is reducible to type equivalence checking.*

For the language $\mathcal{L}^{\rightarrow,\mathtt{Exn}}$ the type equivalence relation is trivial, hence decidable. It follows from the preceding exercise that type checking is decidable.

Theorem 18.2
Type checking for $\mathcal{L}^{\rightarrow,\mathtt{Exn}}$ is decidable.

18.3 Definitional Equality

The question arises once again as to whether or not the typing relation is decidable. We may as before reduce the question to the decidability of constructor equivalence: can we effectively determine whether or not $\Delta \vdash \mu_1 \equiv \mu_2 :: \kappa$? We will prove that this is so by a logical relations argument similar to the ones we've encountered before, except that the interpretation of a type will be a predicate on *open*, rather than *closed*, expressions. The idea will be to prove that if $\Delta \vdash \mu :: \kappa$, then μ is *strongly normalizable (SN)* and *locally confluent (LC)* with respect to a *reduction relation* derived from the axioms of definitional equality of constructors. We will then prove that $\Delta \vdash \mu_1 \equiv \mu_2 :: \kappa$ iff $\Delta \vdash \mu_1 :: \kappa$, $\Delta \vdash \mu_2 :: \kappa$ and μ_1 and μ_2 have the same normal form. This provides an effective (but not an especially efficient) procedure for testing definitional equality.

Put into the context of term rewriting systems (defined in Chapter **??**), the relation of interest will be *β-reduction* on "raw" constructors (not necessarily closed or well-formed). We define $\mu_1 \to_\beta \mu_2$ iff μ_2 can be obtained from μ_1 by replacing a single occurrence of $(\texttt{fun}\,(u{:}\kappa)\,\texttt{in}\,\mu')(\mu)$ in μ_1 by $\{\mu/u\}\mu'$ to get μ_2.

Exercise 18.3
[1] *Give a rigorous definition of β-reduction on constructors.*

Exercise 18.4
Prove the following facts about β-reduction.

1. *If $\mu \to_\beta \mu'$, then $\{\mu_2/u\}\mu \to_\beta \{\mu_2/u\}\mu'$.*
2. *If $\mu_2 \to_\beta \mu_2'$, then $\{\mu_2/u\}\mu \to_\beta^* \{\mu_2'/u\}\mu$.*
3. *β-reduction is locally confluent.*

Exercise 18.5
Prove the following facts relating reduction, typing, and definitional equality:

1. *The* subject reduction *property for β-reduction: if $\Delta \vdash \mu :: \kappa$ and $\mu \to_\beta \mu'$, then $\Delta \vdash \mu' :: \kappa$.*
2. *If $\Delta \vdash \mu :: \kappa$ and $\mu \to_\beta \mu'$, then $\Delta \vdash \mu \equiv \mu' :: \kappa$.*
3. *If $\Delta \vdash \mu :: \kappa$, $\Delta \vdash \mu' :: \kappa$, and $\mu \downarrow_\beta \mu'$, then $\Delta \vdash \mu \equiv \mu' :: \kappa$.*
4. *If $\Delta \vdash \mu_1 \equiv \mu_2 :: \kappa$, then $\mu_1 \leftrightarrow_\beta^* \mu_2$.*

It remains to prove strong normalization for β-reduction of well-formed constructors. With this in place it follows that the relation $\mu_1 \downarrow_\beta \mu_2$ is decidable for well-formed constructors (simply reduce both sides to normal form and check equality up to renaming of bound variables). Moreover, $\to_\beta$ is confluent on well-typed terms, and hence if $\Delta \vdash \mu_1 :: \kappa$, $\Delta \vdash \mu_2 :: \kappa$, and $\mu_1 \leftrightarrow_\beta^* \mu_2$, then $\mu_1 \downarrow_\beta \mu_2$, and hence $\Delta \vdash \mu_1 \equiv \mu_2 :: \kappa$. This yields a sound and complete

[1] For the author.

decision procedure for definitional equality of constructors: to decide whether or not $\Delta \vdash \mu_1 \equiv \mu_2 :: \kappa$, check that $\Delta \vdash \mu_i :: \kappa$ and $\mu_1 \downarrow_\beta \mu_2$.

Strong normalization for β-reduction of well-formed constructors is proved using logical relations. In contrast to the preceding applications of the method, we must consider predicates on *open*, as well as closed, terms to get the desired result. Free variables are elegantly handled using Kripke's method of *possible worlds*. The main idea is to define the interpretation of kinds relative to a "world", a kind context providing a set of variables and their kinds. Worlds "evolve" by extension (addition of new variables). The definition of the function kind accounts for the evolution of worlds.

Definition 18.6
The interpretation $||\kappa||_\Delta$ of a kind κ relative to a world Δ is defined by induction on the structure of κ as follows:

$$\begin{array}{rcl} ||\Omega||_\Delta & = & \{\, \mu \mid \Delta \vdash \mu :: \Omega \textit{ and } \mathrm{SN}_\beta(\mu)\,\} \\ ||\kappa_1 \Rightarrow \kappa_2||\Delta & = & \{\, \mu \mid \Delta \vdash \mu :: \kappa_1 \Rightarrow \kappa_2 \textit{ and } \forall \Delta' \supset \Delta,\ \forall \mu_1 \in ||\kappa_1||_{\Delta'},\ \mu(\mu_1) \in ||\kappa_2||_{\Delta'}\,\} \end{array}$$

Lemma 18.7

1. *If $\mu \in ||\kappa||_\Delta$, then $\Delta \vdash \mu :: \kappa$.*
2. *If $\mu \in ||\kappa||_\Delta$ and $\Delta' \supseteq \Delta$, then $\mu \in ||\kappa||_{\Delta'}$.*

Exercise 18.8
Prove Lemma 18.7 by induction on the structure of κ.

Lemma 18.9

1. *If $\mu \in ||\kappa||_\Delta$, then $\mathrm{SN}_\beta(\mu)$.*
2. *Suppose that $\Delta \vdash u :: \kappa_1 \Rightarrow \cdots \kappa_n \Rightarrow \kappa$, and $\Delta' \vdash \mu_i :: \kappa_i$ $(1 \le i \le n)$ for some $\Delta' \supseteq \Delta$. If $\mathrm{SN}_\beta(\mu_i)$ for each $1 \le i \le n$, then $u(\mu_1)\cdots(\mu_n) \in ||\kappa||_{\Delta'}$.*

Proof: Simultaneously, by induction on the structure of κ.

For $\kappa = \Omega$, part (1) is immediate from the definition of $||\kappa||_\Delta$, and part (2) holds because $\mathrm{SN}_\beta(u(\mu_1)\cdots(\mu_n) \in ||\kappa||_{\Delta'})$ whenever $\mathrm{SN}_\beta(\mu_i)$ $(1 \le i \le n)$.

Suppose $\kappa = \kappa' \Rightarrow \kappa''$. For part (1), suppose that $\mu \in ||\kappa||_\Delta$, and consider the kind context $\Delta' = \Delta[u{::}\kappa'] \supseteq \Delta$. By inductive hypothesis, part (2), we know that $u \in ||\kappa'||_{\Delta'}$, and hence $\mu(u) \in ||\kappa''||_{\Delta'}$, and so by inductive hypothesis, part (1), it follows that $\mathrm{SN}_\beta(\mu(u))$. But then $\mathrm{SN}_\beta(\mu)$ since any infinite reduction sequence in μ can be mimicked in $\mu(u)$, contradicting $\mathrm{SN}_\beta(\mu(u))$. For part (2), let $\mu \in ||\kappa'||_{\Delta'}$; we are to show that $u(\mu_1)\cdots(\mu_n)(\mu) \in ||\kappa''||_{\Delta'}$. By inductive hypothesis, part (1), $\mathrm{SN}_\beta(\mu)$, and hence the result follows by inductive hypothesis, part (2). ■

Lemma 18.10
Suppose that $\Delta[u{::}\kappa_0] \vdash \mu :: \kappa_1 \Rightarrow \cdots \kappa_n \Rightarrow \kappa$, $\Delta \vdash \mu_i :: \kappa_i$ $(0 \le i \le n)$, and $\mathrm{SN}_\beta(\mu_0)$. If $\{\mu_0/u\}\mu(\mu_1)\cdots(\mu_n) \in ||\kappa||_\Delta$, then

$$(\texttt{fun}\,(u{:}\kappa_0)\,\texttt{in}\,\mu)(\mu_0)(\mu_1)\cdots(\mu_n) \in ||\kappa||_\Delta.$$

Exercise 18.11
Prove Lemma 18.10 by induction on the structure of κ. Use the fact that the strongly normalizable constructors are closed under head-β-expansion.

Let δ range over substitutions of constructors for constructor variables. The relation $\Delta' \Vdash \delta :: \Delta$ holds iff $\delta(u) \in ||\Delta(u)||_{\Delta'}$ for every $u \in \mathrm{dom}(\Delta)$. The relation $\Delta' \Vdash \mu :: \kappa\,[\delta]$ holds iff $\hat{\delta}(\mu) \in ||\kappa||_{\Delta'}$.

Theorem 18.12
If $\Delta \vdash \mu :: \kappa$, and $\Delta' \Vdash \delta :: \Delta$, then $\Delta' \Vdash \mu :: \kappa\,[\delta]$.

Proof: By induction on typing derivations, using Lemma 18.10 to deal with λ-abstractions. ■

Exercise 18.13
Prove Theorem 18.12.

Corollary 18.14
If $\Delta \vdash \mu :: \kappa$, then $\mathrm{SN}_\beta(\mu)$.

Proof: Note that $\Delta \Vdash id :: \Delta$, where *id* is the identity substitution. ■

Corollary 18.15
Definitional equality of constructors is decidable.

18.4 Subtyping

Fortunately we can restrict attention to typing derivations in *standard form* that determine the *minimal*, or *principal*, type of an expression (relative to a context). This process is called *type synthesis* since it synthesizes the principal type of an expression from the expression itself. Standard-form typing derivations are defined as the least relation closed under the following rules for deriving judgements of the form $\Gamma \vdash e \Rightarrow \tau$:

$$\Gamma \vdash x \Rightarrow \tau \quad (\Gamma(x) = \tau) \tag{T-S-VAR}$$

$$\frac{\Gamma[x{:}\tau_1] \vdash e_2 \Rightarrow \tau_2}{\Gamma \vdash \texttt{fun}\,(x{:}\tau_1)\,\texttt{in}\,e_2 : \tau_1 \to \tau_2} \quad (x \notin \mathrm{dom}(\Gamma)) \tag{T-S-ABS}$$

$$\frac{\Gamma \vdash e_1 : \tau_2' \to \tau \qquad \Gamma \vdash e_2 : \tau_2 \qquad \tau_2 <: \tau_2'}{\Gamma \vdash e_1(e_2) \Rightarrow \tau} \tag{T-S-APP}$$

Note in particular that there is no longer an explicit rule of subsumption, but that in compensation the application rule enforces a subtyping constraint.

Standard form derivations are sound and complete for typing in $\mathcal{L}^{\to,\texttt{Exn}}$ in the sense that a term is typable iff it has a standard-form typing derivation, and, moreover, every type for that term is a supertype of its minimal type.

Theorem 18.16 (Standardization)
$\Gamma \vdash e : \tau$ in $\mathcal{L}^{\rightarrow,\mathtt{Exn}}$ iff $\Gamma \vdash e \Rightarrow \tau'$ for some τ' such that $\tau' <: \tau$.

Proof: The "if" direction is obvious: every standard-form typing derivation determines a typing derivation. More precisely, it suffices to show that the typing relation $\Gamma \vdash e : \tau$ is closed under the rules defining the standard-form typing relation $\Gamma \vdash e \Rightarrow \tau$, from which the result follows from the minimality of the standard-form relation among relations closed under these rules.

The "only if" direction is proved similarly: we show that the relation "there exists $\tau' <: \tau$ such that $\Gamma \vdash e \Rightarrow \tau'$" is closed under the typing rules for $\mathcal{L}^{\rightarrow,\mathtt{Exn}}$.

t-var $\Gamma \vdash x : \Gamma(x) <: \Gamma(x)$.

t-abs By induction we have that $\Gamma[x{:}\tau_1] \vdash e_2 \Rightarrow \tau_2' <: \tau_2$. It follows that

$$\Gamma \vdash \mathtt{fun}\,(x{:}\tau_1)\,\mathtt{in}\,e_2 \Rightarrow \tau_1{\rightarrow}\tau_2' <: \tau_1{\rightarrow}\tau_2.$$

t-app By induction we have $\Gamma \vdash e_1 \Rightarrow \tau_1' <: \tau_2{\rightarrow}\tau$ and $\Gamma \vdash e_2 \Rightarrow \tau_2' <: \tau_2$. Since the subtyping relation is normal, $\tau_1' = \tau_2'{\rightarrow}\tau'$ with $\tau_2 <: \tau_2'$ and $\tau' <: \tau$. Hence $\Gamma \vdash e_1(e_2) \Rightarrow \tau' <: \tau$.

■

Exercise 18.17
Exhibit a non-normal subtyping system for which standardization fails. Hint: *Postulate that some base type b is a subtype of some function type, then construct a valid typing derivation that cannot be put into standard form.*

Remark 18.18
If we use a fully explicit syntax, then the standard-form rule for function abstractions must change to the following:

$$\frac{\Gamma[x{:}\tau_1] \vdash e_2 \Rightarrow \tau_2' \qquad \tau_2' <: \tau_2}{\Gamma \vdash \mathtt{fun}\,(x{:}\tau_1){:}\tau_2\,\mathtt{in}\,e_2 \Rightarrow \tau_1{\rightarrow}\tau_2} \quad (x \notin \mathrm{dom}(\Gamma)) \qquad (\textsc{t-s-abs-x})$$

The ascription of a specified result type to the function prohibits inferring anything other than the ascribed type to that function. Consequently, the result type forms a constraint that must be checked during type synthesis.

Exercise 18.19
Check that Theorem 18.16 goes through for the fully-explicit syntax.

18.5 Subtyping

1. Minimal types.
2. Subtype checking.
3. Undecidabiltiy of subtyping for $\mathcal{L}^{\rightarrow,\forall}_{\Omega,<:}$ with contravariant rule.

18.6 References

Chapter 19

Type Reconstruction

19.1 Introduction

All of the language we have considered so far are based on an "explicitly-typed" syntax in which the primitive operations are decorated with sufficient type information to ensure that a type-sensitive semantics for the language can be given.[1] For example, the "fully explicit" syntax for application includes both the domain and range types, admitting the possibility of using type information to determine the call- and return-sequence of a function. Of course a given implementation is free to ignore types, and implement the primitive operations uniformly. Since we cannot recover what was not there in the first place, the explicit syntax is more general than an implicit syntax in which this information has been omitted.

From a programmer's point of view, however, the explicitly-typed syntax is rather burdensome. For example, we have routinely written $e_1(e_2)$, rather than $\mathtt{app}_{\tau_1,\tau_2}(e_1,e_2)$, because otherwise the notation would be too cumbersome to be conveniently manageable. The situation is worse in the case of polymorphism, for in that we must explicitly apply polymorphic values to constructors to select a suitable "instance". It quickly becomes apparent that it is necessary to provide an *external language*, which is written by the programmer, together with a mapping to an *internal language*, the fully explicit syntax. This mapping is called *type reconstruction.*

19.2 Type Reconstruction for $\mathcal{L}_{\Omega}^{\rightarrow,\forall}$

We will work with $\mathcal{L}_{\Omega}^{\rightarrow,\forall}$ as the internal language, presented in the "streamlined" form discussed in Chapter 7. The fully explicit syntax of $\mathcal{L}_{\Omega}^{\rightarrow,\forall}$ is given by the

[1]This criterion is admittedly unclear, but I have been unable to formulate a precise definition of what we might mean by "explicitly-typed" syntax.

following grammar:

$$\textit{InternalExpr's} \quad e \quad ::= \quad x \mid c \mid \texttt{fun}\,(\tau_1{:}\tau_2){:}x\;\texttt{in}\,e \mid \texttt{app}_{\tau_1,\tau_2}(e_1,e_2) \mid \texttt{Fun}_\phi\,(t)\;\texttt{in}\,e \mid \texttt{App}_\phi(e,\tau)$$

The type system consists of rules for deriving judgements of the following forms:

$\Delta \vdash \tau :: \kappa$	τ *has kind* κ
$\Delta \vdash \tau_1 \equiv \tau_2 :: \kappa$	τ_1 *is definitionally equivalent to* τ_2 *at kind* κ
$\Gamma \vdash_\Delta e : \tau$	e *has type* τ

The derivation rules for these judgements are easily derived from those given in Chapter 7. However, as a technical convenience, we consider not just the kind Ω, but also the kind $\Omega \Rightarrow \Omega$, even though polymorphic abstraction and application are limited to the kind Ω. One reason for this is that the fully explicit syntax for polymorphic abstraction and application is indexed by a constructor of kind $\Omega \Rightarrow \Omega$ in order to determine the quantified type in question. Another is that we shall have need of variables of both kinds, and hence we admit kind contexts in which constructor variables are ascribed either the kind Ω or the kind $\Omega \Rightarrow \Omega$. The constructor formation and equality rules for this presentation of $\mathcal{L}_\Omega^{\rightarrow,\forall}$ are readily derived from the "higher kinds" extension of $\mathcal{L}_\Omega^{\rightarrow,\forall}$ by restricting attention to the two kinds considered here.. The meta-variable ϕ will be used to range over constructors of kind $\Omega \Rightarrow \Omega$.

We shall work with an implicit syntax that allows the omission of a substantial amount of type information on terms. Primitive operations need not be indexed by their types (although we may specify the domain type of a function). Moreover, uses of polymorphic instantiation may omit explicit mention of the type argument, indicating only that an instantiation operation is to occur at that point. The following grammar defines an implicit syntax for $\mathcal{L}_\Omega^{\rightarrow,\forall}$:

$$\textit{ExternalExpr's} \quad u \quad ::= \quad x \mid c \mid \texttt{fun}\,(x{:}\tau)\;\texttt{in}\,u \mid \texttt{fun}\,x\;\texttt{in}\,u \mid u_1(u_2) \mid \texttt{Fun}\,(t)\;\texttt{in}\,u \mid u[\tau] \mid u[]$$

Expressions of the external language are sometimes called "partially typed" since they may contain some, but not all, type information associated with fully typed terms.

19.3 Type Reconstruction as Static Semantics

The well-formedness of external language expressions may be defined by a static semantics consisting of a set of elaboration rules that describe a non-deterministic procedure mapping external to internal expressions. Elaboration judgements have the form $\Gamma \vdash_\Delta u : \tau \Rightarrow e$, where Δ is a kind context, Γ is a type context over Δ, τ is a type over Δ, u is an expression of the external language, and e is an expression of the internal language. We say that e is a *reconstruction* of u (equivalently, that u is an *abbreviation* of e whenever such a judgement is derivable in accordance with the following rules.

$$\frac{\Delta \vdash \Gamma(x) \equiv \tau :: \Omega}{\Gamma \vdash_\Delta x : \tau \Rightarrow x} \qquad (\text{RS-VAR})$$

$$\frac{\Delta \vdash \tau \equiv b :: \Omega}{\Gamma \vdash_\Delta c : \tau \Rightarrow c} \quad (c : b) \qquad \text{(RS-CONST)}$$

$$\frac{\Gamma[x{:}\tau_1] \vdash_\Delta u : \tau_2 \Rightarrow e \qquad \Delta \vdash \tau_1 :: \Omega \qquad \Delta \vdash \tau \equiv \tau_1 \to \tau_2 :: \Omega}{\Gamma \vdash_\Delta \texttt{fun}\,(x{:}\tau_1)\,\texttt{in}\,u : \tau \Rightarrow \texttt{fun}\,(\tau_1{:}\tau_2){:}x\,\texttt{in}\,e} \quad (x \notin \mathrm{dom}(\Gamma)) \qquad \text{(RS-FN)}$$

$$\frac{\Gamma[x{:}\tau_1] \vdash_\Delta u : \tau_2 \Rightarrow e \qquad \Delta \vdash \tau \equiv \tau_1 \to \tau_2 :: \Omega}{\Gamma \vdash_\Delta \texttt{fun}\,x\,\texttt{in}\,u : \tau \Rightarrow \texttt{fun}\,(\tau_1{:}\tau_2){:}x\,\texttt{in}\,e} \quad (x \notin \mathrm{dom}(\Gamma)) \qquad \text{(RS-UFN)}$$

$$\frac{\Gamma \vdash_\Delta u_1 : \tau_1 \Rightarrow e_1 \qquad \Gamma \vdash_\Delta u_2 : \tau_2 \Rightarrow e_2 \qquad \Delta \vdash \tau_1 \equiv \tau_2 \to \tau :: \Omega}{\Gamma \vdash_\Delta u_1(u_2) : \tau \Rightarrow \texttt{app}_{\tau_2,\tau}(e_1,e_2)} \qquad \text{(RS-APP)}$$

$$\frac{\Gamma \vdash_{\Delta[t::\Omega]} u : \phi(t) \Rightarrow e \qquad \Delta \vdash \tau \equiv \forall(t)\phi(t) :: \Omega}{\Gamma \vdash_\Delta \texttt{Fun}\,(t)\,\texttt{in}\,u : \tau \Rightarrow \texttt{Fun}_\phi\,(t)\,\texttt{in}\,e} \quad (t \notin \Delta) \qquad \text{(RS-TFN)}$$

$$\frac{\Gamma \vdash_\Delta u : \tau_1 \Rightarrow e \qquad \Delta \vdash \tau :: \Omega \qquad \Delta \vdash \tau' \equiv \phi(\tau) :: \Omega \qquad \Delta \vdash \tau_1 \equiv \forall(t)\phi(t) :: \Omega}{\Gamma \vdash_\Delta u[\tau] : \tau' \Rightarrow \texttt{App}_\phi(e,\tau)} \qquad \text{(RS-TAPP)}$$

$$\frac{\Gamma \vdash_\Delta u : \tau_1 \Rightarrow e \qquad \Delta \vdash \tau :: \Omega \qquad \Delta \vdash \tau' \equiv \phi(\tau) :: \Omega \qquad \Delta \vdash \tau_1 \equiv \forall(t)\phi(t) :: \Omega}{\Gamma \vdash_\Delta u[\,] : \tau' \Rightarrow \texttt{App}_\phi(e,\tau)} \qquad \text{(RS-UTAPP)}$$

We shall have need of the following properties of the elaboration relation.

Lemma 19.1
If $\Gamma \vdash_{\Delta[t::\kappa]} u : \tau \Rightarrow e$ *by a derivation of height* h *and* $\Delta \vdash \tau' :: \kappa$*, then*

$$\{\tau'/t\}\Gamma \vdash_\Delta \{\tau'/t\}u : \{\tau'/t\}\tau \Rightarrow \{\tau'/t\}e$$

by a derivation of height h*.*

Lemma 19.2
If $\Gamma \vdash_\Delta u : \tau \Rightarrow e$*,* $\Delta \vdash \tau \equiv \tau' :: \Omega$*, and for every* $x \in \mathrm{dom}(\Gamma)$*,* $\Delta \vdash \Gamma(x) \equiv \Gamma'(x) :: \Omega$*, then* $\Gamma' \vdash_\Delta u : \tau' \Rightarrow e$*.*

Lemma 19.3
If $\Gamma \vdash_\Delta u : \tau \Rightarrow e$*, then* $\Gamma \vdash_\Delta e : \tau$*.*

Reconstructions are not unique, even if we fix Γ, Δ, and τ. For example, $(\texttt{fun}\,y\,\texttt{in}\,3)(\texttt{fun}\,x\,\texttt{in}\,x)$: `Int` has infinitely many reconstructions as an explicitly-typed term!

19.4 Reconstruction as Constraint Solving

The description of type reconstruction as static semantics is intuitively appealing, but is not directly amenable to implementation. How is the indeterminacy in the rules to be captured in a reconstruction algorithm? A natural approach is to associate with each expression of the internal language a set of constraints governing the choices to be made by the non-deterministic static semantics in such a way that every reconstruction arises as a solution of the constraints.

19.4.1 Unification Logic

Constraints are expressed as formulae of *(second-order) unification logic* whose syntax is defined by the following grammar:

$$\Phi \quad ::= \quad \tau_1 \doteq \tau_2 \mid \Phi_1 \wedge \Phi_2 \mid \exists t{::}\kappa.\Phi \mid \forall t{::}\kappa.\Phi$$

Here κ is either Ω or $\Omega \Rightarrow \Omega$; higher-order unification logic is defined by relaxing this restriction to admit arbitrary kinds.

A formula Φ is *well-formed* over the kind context Δ iff the involved equations are well-formed in the extension of Δ appropriate to their occurrence. Specifically, the equation $\tau_1 \doteq \tau_2$ is well-formed over Δ iff $\Delta \vdash \tau_i :: \Omega$ $(i = 1, 2)$; the conjunction $\Phi_1 \wedge \Phi_2$ is well-formed over Δ just in case both Φ_1 and Φ_2 are well-formed over Δ; the quantified formulae $\forall t{::}\kappa.\Phi$ and $\exists t{::}\kappa.\Phi$ are well-formed over Δ just in case Φ is well-formed over $\Delta[t{::}\kappa]$.

The *provability* property, $\Vdash_\Delta \Phi$, for Φ a well-formed formula over kind context Δ, is defined by the following rules:

$$\frac{\Delta \vdash \tau_1 \equiv \tau_2 :: \Omega}{\Vdash_\Delta \tau_1 \doteq \tau_2} \tag{UL-EQ}$$

$$\frac{\Vdash_\Delta \Phi_1 \qquad \Vdash_\Delta \Phi_2}{\Vdash_\Delta \Phi_1 \wedge \Phi_2} \tag{UL-AND}$$

$$\frac{\Vdash_\Delta \{\tau/t\}\Phi \qquad \Delta \vdash \tau :: \kappa}{\Vdash_\Delta \exists t{::}\kappa.\Phi} \tag{UL-SOME}$$

$$\frac{\Vdash_{\Delta[t::\kappa]} \Phi}{\Vdash_\Delta \forall t{::}\kappa.\Phi} \quad (t \notin \Delta) \tag{UL-ALL}$$

Lemma 19.4

1. *If* $\Vdash_{\Delta[t::\kappa]} \Phi$ *and* $\Delta \vdash \tau :: \kappa$, *then* $\Vdash_\Delta \{\tau/t\}\Phi$.

2. *Suppose that* $\Delta[t{::}\kappa] \vdash \Phi$. *If* $\Vdash_\Delta \{\tau/t\}\Phi$ *and* $\Delta \vdash \tau \equiv \tau' :: \kappa$, *then* $\Vdash_\Delta \{\tau'/t\}\Phi$.

A formula Φ is in Σ_2-*form* iff it has the shape

$$\exists t_1{::}\kappa_1.\cdots\exists t_m{::}\kappa_m.\forall t'_1{::}\kappa'_1.\cdots\forall t'_n{::}\kappa'_n.\tau_1 \doteq \tau'_1 \wedge \cdots \wedge \tau_p \doteq \tau'_p.$$

That is, all quantifiers appear at the front, and all existential quantifiers precede all universal quantifiers.

Theorem 19.5
For every formula Φ of second-order unification logic there is a formula Φ' of higher-order unification logic in Σ_2-form such that $\Vdash_\Delta \Phi$ iff $\Vdash_\Delta \Phi'$.

Proof: The formula Φ' is obtained by application of the following *prenex operations*:

1. $(\exists t{::}\kappa.\Phi_1) \wedge \Phi_2 \mapsto \exists t{::}\kappa.(\Phi_1 \wedge \Phi_2)$ provided that t is not free in Φ_2, and similarly for the universal quantifier and for quantifiers in the right-hand conjunct.

2. $\forall t{::}\kappa.\exists t'{::}\kappa'.\Phi \mapsto \exists u{::}\kappa \Rightarrow \kappa'.\forall t{::}\kappa.\{u(t)/t'\}\Phi$.

∎

Exercise 19.6
Prove Theorem 19.5.

From a proof of a Σ_2 formula Φ in higher-order unification logic we may read off a substitution ρ for the existentially-quantified variables of Φ such that the inner universally-quantified system of equations is provable under this substitution. Such a substitution is called a *unifying substitution*, or *unifier*, for Φ. In view of Theorem 19.5 we can reduce the problem of provability in second-order unification logic to the problem of finding a unifier for the associated Σ_2-form of Φ. Huet's higher-order unification algorithm[2] may be used to find a unifier, if one exists (but it may diverge looking for a non-existent unifier). This is the best we can do: provability in second- (and higher-) order unification logic is undecidable.

A *principal unifier* for Φ is a unifier for Φ through which all other unifiers for Φ factor — that is, ρ_0 is a principal unifier for Φ iff whenever ρ is a unifier for Φ, there is a substitution ρ' such that $\rho = \rho' \circ \rho_0$. For the special case of a Σ_2-form all of whose existential variables range over the kind Ω, if any unifier exists, then there is a principal unifier, but this is not true in general for second- or higher-order problems. As will become clear below, it is important for type reconstruction to determine whether or not a given unification problem has a principal solution. There is an algorithm derived from Huet's procedure with the property that it always terminates, classifying the input formula Φ as either unprovable, provable with a specified principal unifier, or ambiguous.

[2] By a careful analysis of the prenex operations it is possible to prove that only second-order unification is required, but this does not materially affect the discussion.

The algorithm is incomplete in two senses: it can fail to classify a formula Φ that admits a principal unifier as such, and it can fail to identify an unprovable formula as such. It is, however, sound in that affirmative decisions are always correct, and it is "practically complete" in the informal sense that the class of problems for which it renders a decision appears to be sufficiently large to be useful in practice.

19.4.2 Constraint Generation

Type reconstruction may be described by associating a formula Φ of second-order unification logic with a partially typed term u that, in an appropriate sense, captures all possible reconstructions of u. The relation $\Gamma \vdash_\Delta u : \tau \Rightarrow \Phi$ is defined for kind contexts Δ (as above), well-formed type contexts Γ over Δ, well-formed type τ over Δ, and expressions u of the external language by the following set of rules:

$$\Gamma \vdash_\Delta x : \tau \Rightarrow (\tau' \doteq \tau) \quad (\Gamma(x) = \tau') \tag{RC-VAR}$$

$$\Gamma \vdash_\Delta c : \tau \Rightarrow (\tau \doteq b) \quad (c : b) \tag{RC-CONST}$$

$$\frac{\Gamma[x{:}\tau_1] \vdash_{\Delta[t_2::\Omega]} u : t_2 \Rightarrow \Phi}{\Gamma \vdash_\Delta \mathtt{fun}\,(x{:}\tau_1)\,\mathtt{in}\,u : \tau \Rightarrow \exists t_2{::}\Omega.(\Phi \wedge \tau \doteq \tau_1 {\to} t_2)} \quad (x \notin \mathrm{dom}(\Gamma); t_2 \notin \mathrm{dom}(\Delta)) \tag{RC-FN}$$

$$\frac{\Gamma[x{:}t_1] \vdash_{\Delta[t_1::\Omega,t_2::\Omega]} u : t_2 \Rightarrow \Phi}{\Gamma \vdash_\Delta \mathtt{fun}\,x\,\mathtt{in}\,u : \tau \Rightarrow \exists t_1{::}\Omega.\exists t_2{::}\Omega.(\Phi \wedge \tau \doteq t_1 {\to} t_2)} \quad (x \notin \mathrm{dom}(\Gamma); t_1, t_2 \notin \mathrm{dom}(\Delta)) \tag{RC-UFN}$$

$$\frac{\Gamma \vdash_{\Delta[t_1::\Omega]} u_1 : t_1 \Rightarrow \Phi_1 \qquad \Gamma \vdash_{\Delta[t_2::\Omega]} u_2 : t_2 \Rightarrow \Phi_2}{\Gamma \vdash_\Delta u_1(u_2) : \tau \Rightarrow \exists t_1{::}\Omega.\exists t_2{::}\Omega.(\Phi_1 \wedge \Phi_2 \wedge t_1 \doteq t_2 {\to} \tau)} \quad (t_1, t_2 \notin \mathrm{dom}(\Delta)) \tag{RC-APP}$$

$$\frac{\Gamma \vdash_{\Delta[s::\Omega\Rightarrow\Omega,t::\Omega]} u : s(t) \Rightarrow \Phi}{\Gamma \vdash_\Delta \mathtt{Fun}\,(t)\,\mathtt{in}\,u : \tau \Rightarrow \exists s{::}\Omega \Rightarrow \Omega.(\tau \doteq \forall(t)s(t) \wedge \forall t{::}\Omega.\Phi)} \quad (s, t \notin \mathrm{dom}\,\Delta) \tag{RC-TFN}$$

$$\frac{\Gamma \vdash_{\Delta[s::\Omega\Rightarrow\Omega,t_1::\Omega]} u : t_1 \Rightarrow \Phi \qquad \Delta \vdash \tau :: \Omega}{\Gamma \vdash_\Delta u[\tau] : \tau' \Rightarrow \exists s{::}\Omega \Rightarrow \Omega.\exists t_1{::}\Omega.(\Phi \wedge \tau' \doteq s(\tau) \wedge t_1 \doteq \forall(t)s(t))} \quad (s, t_1 \notin \mathrm{dom}(\Delta)) \tag{RC-TAPP}$$

$$\frac{\Gamma \vdash_{\Delta[s::\Omega\Rightarrow\Omega,t_1::\Omega,t::\Omega]} u : t_1 \Rightarrow \Phi}{\Gamma \vdash_\Delta u[] : \tau' \Rightarrow \exists s{::}\Omega \Rightarrow \Omega.\exists t_1{::}\Omega.\exists t{::}\Omega.(\Phi \wedge \tau' \doteq s(t) \wedge t_1 \doteq \forall(t)s(t))} \quad (s, t_1, t \notin \mathrm{dom}(\Delta)) \tag{RC-UTAPP}$$

The constraint associated with a partially-typed term captures the context-sensitive conditions that are implicit in the elaboration rules given in Section 19.3. The correspondence between the two systems is easily established.

Every term determines a well-formed set of constraints.

Lemma 19.7
For any term u of the external language there exists Φ such that $\Gamma \vdash_{\Delta} u : \tau \Rightarrow \Phi$.

Lemma 19.8
If $\Gamma \vdash_{\Delta[t::\kappa]} u : \tau \Rightarrow \Phi$ by a derivation of height h and $\Delta \vdash \tau' :: \kappa$, then

$$\{\tau'/t\}\Gamma \vdash_{\Delta} \{\tau'/t\}u : \{\tau'/t\}\tau \Rightarrow \{\tau/t\}\Phi$$

by a derivation of height h.

Theorem 19.9
Suppose that $\Gamma \vdash_{\Delta} u : \tau \Rightarrow \Phi$.

1. *If $\Vdash_{\Delta} \Phi$, then there exists e such $\Gamma \vdash_{\Delta} u : \tau \Rightarrow e$.*
2. *If $\Gamma \vdash_{\Delta} u : \tau \Rightarrow e$, then $\Vdash_{\Delta} \Phi$.*

Proof:

1. The term e is determined by the derivations of $\Gamma \vdash_{\Delta} u : \tau \Rightarrow \Phi$ and $\Vdash_{\Delta} \Phi$.
2. The proof of Φ is determined from the derivation of $\Gamma \vdash_{\Delta} u : \tau \Rightarrow e$.

■

Exercise 19.10
Prove Theorem 19.9.

Exercise 19.11
In the following exercises you are asked to consider a closed partially-typed expression u and to exhibit the formula $\Phi(u)$ such that $\emptyset \vdash_{t::\Omega} u : t \Rightarrow \Phi(u)$.

1. *Consider the term $u = \texttt{fun}\,x\,\texttt{in}\,x(x)$. Argue that $\Phi(u)$ is not provable in second-order unification logic.*
2. *Consider the term $u = \texttt{fun}\,x\,\texttt{in}\,x[](x)$. Argue that $\Phi(u)$ has no principal solutions by exhibiting two incomparable (with respect to the factorization ordering) unifiers for $\Phi(u)$.*
3. *Consider the term $u = \texttt{Fun}\,(t)\,\texttt{in}\,\texttt{fun}\,x\,\texttt{in}\,x$. Find a principal unifier for $\Phi(u)$.*

19.5 Reconstruction and Definitions

The interaction between type reconstruction and definitions raises a number of important issues. Let us first consider "closed scope" definitions of the form $\texttt{def}\,x\,\texttt{is}\,u_1\,\texttt{in}\,u_2$. Since the scope of the definition is explicit, we may regard this as a derived form in one of two ways:

1. "Definition by value": $\texttt{def}\,x\,\texttt{is}\,u_1\,\texttt{in}\,u_2$ is an abbreviation for $(\texttt{fun}\,x\,\texttt{in}\,u_2)(u_1)$.

2. "Definition by name": $\texttt{def}\,x\,\texttt{is}\,u_1\,\texttt{in}\,u_2$ is an abbrevation for $\{u_1/x\}u_2$ (since x might not occur in u_2, we may instead use $\mathbf{K}(\{u_1/x\}u_2)(u_1)$, where $\mathbf{K} = \texttt{fun}\,x\,\texttt{in}\,\texttt{fun}\,y\,\texttt{in}\,x$, to ensure that u_1 actually occurs in the expansion).

Under either of these interpretations the reconstruction of the defining term, u_1, is sensitive to the context of its use. Under the "by value" interpretation a single type τ must be derived for all occurrences of x in u_2, and this type must be correctly ascribable to u_1. Under the "by name" interpretation each occurrence of x may be assigned a different type, but each such type must be ascribable to u_1.

Either interpretation of $\texttt{def}\,x\,\texttt{is}\,u_1\,\texttt{in}\,u_2$ is semantically sensible, provided that we consistently interpret it as a derived form for the purposes of *both* type checking and evaluation. In ML the "by name" interpretation is used for type checking purposes, but the "by value" interpretation is used for evaluation. *This is a fundamental error in the design of ML. It leads to unsoundness in any circumstance in which the "by value" and "by name" interpretations are not operationally equivalent.* For example, if u_1 is $\texttt{ref}[](\texttt{nil}[])$, then it is only rarely the case that the two interpretations coincide because replication of u_1 leads to allocation of a fresh location each time it is evaluated. This observation can be readily turned into a counterexample to soundness.

In an interactive system or in the presence of modules or separate compilation definitions have "open" scope — there is no fixed context of use of the definition. The way is not open to use to regard an open scope definition as a derived form in any obvious way (short of deferring interpretation until the complete program is available). There are two cases to consider. In the case that an intended type for the defined variable is given by an explicit ascription, then to elaborate $\texttt{def}\,x{:}\tau\,\texttt{is}\,u$, determine the constraint Φ_u such that $\Gamma \vdash_\Delta u : \tau \Rightarrow \Phi_u$ and check whether or not Φ_u is provable. This will, in effect, impose the constraint that the type of u be τ, and thereby force resolution of any undetermined existentially-quantified variables in Φ_u.

Elaboration of the "bare" definition $\texttt{def}\,x\,\texttt{is}\,u$ is more difficult. Consider the constraint Φ_u such that $\Gamma \vdash_{\Delta[t::\Omega]} u : t \Rightarrow \Phi_u$, where $t \notin \mathrm{dom}(\Delta)$. If $\exists t{::}\Omega.\Phi_u$ is unprovable, then the definition must be rejected — there is no type that may be correctly ascribed to u. If it is ambiguous (according to the classification algorithm mentioned above), the definition must also be rejected, with the comment that more type information is required to disambiguate the definition. If it is provable with a principal unifier ρ_0, and there are no free variables of higher kind occurring in ρ_0, then we may *implicitly quantify* the free variables, assigning x the type $\forall t_1, \ldots, t_n{::}\Omega.\rho_0(t)$, and binding x to $\Lambda t_1, \ldots, t_n{::}\Omega.e_0$, where e_0 is the reconstruction of u determined by ρ_0. If free variables of higher kind remain, it is impossible to express the principal type within $\mathcal{L}_{\Omega}^{\to,\forall}$ because quantifiers range only over the kind Ω, and hence the expression must be rejected. (This outcome can be avoided by passing to the extension of $\mathcal{L}_{\Omega}^{\to,\forall}$ with higher kinds so that all free constructor variables may be quantified, but this extension has problems of its own ....)

Note, however, that implicit quantification may change the operational meaning of a definition in the case that we adopt a "by value" interpretation! For example, if $u = \texttt{ref}[\,](\texttt{nil}[\,])$, then implicit quantification yields the expression $e = \texttt{Fun}\,(t)\,\texttt{in}\,\texttt{ref}[t\,\texttt{list}](\texttt{nil}[t])$. The reconstruction e is a value under the operational semantics given in Chapter 7, but u itself is not. In particular evaluation of u (using the semantics given in Chapter 5 to account for free constructor variables) causes memory to be allocated, whereas evaluation of e terminates immediately. Since the semantics of memory allocation involves "scope extrusion", it is *incorrect* to quantify the type variable t after evaluation of u, for in that case the variable t occurs in the type of a location bound by the outermost `letref`, and hence is not dischargeable. This is the source of the unsoundness of the ML type system in the presence of references (similar problems arise with exceptions and continuations). Thus implicit quantification may only be correctly employed under either a "by name" interpretation of definitions or when u is restricted to be a value (in which case it is possible to prove that the quantified variable may be discharged after evaluation).

19.6 References

The first systematic account of type inference in programming languages was developed by Robin Milner [**?**]. This approach has since been extended by numerous researchers. The generalization considered here was introduced by Frank Pfenning [**?**, **?**] and implemented by him for the language LEAP [**?**].

Chapter 20

Coercion Interpretation of Subtyping

A variant of the coercion interpretation is based on the idea of a q*dictionary*, or *vtable*. This approach makes width subtyping less costly than the coercive approach, but does not affect the cost of depth subtyping. We will therefore outline this approach for width-only record subtyping, with implicit re-ordering of fields. The leading idea is that each use of width subtyping is witnessed by an *injection* (1:1 mapping) of the supertype into the subtype that merely determines the position of each field of the supertype in the subtype. This suggests a simple implementation strategy. Rather than create a new record whenever width subtyping is applied, we instead associate with record values a dictionary representing the implicit injection of fields from the supertype to the subtype. Each use of width subtyping leads to the creation of a new dictionary, but the underlying record value is shared among all views of it as a value of a supertype. Since the composition of two injections is itself an injection, we may collapse successive uses of width subtyping into a single dictionary; there is never a need for more than one level of dictionary governing access to a record. Finally, since the identity mapping is injective, records start out life with a dictionary that sends each label to itself.

Chapter 21

Named Form

21.1 Introduction

This should be reworked to use partial functions.

The distinction between values and computations in the semantic interpretation of types can be made explicit in the syntax. Specifically, we introduce the type $[\tau]$ representing the type of "computations" of type τ, and re-formulate the system to take account of the distinction. In particular we introduce an explicit sequencing construct into the language that evaluates a computation to extract its value, and passes this value to another computation.

The language $\mathcal{L}^{\rightarrow[]}$ is defined to be a "computational" analogue of $\mathcal{L}^{\rightarrow}$ in which the value/computation distinction is made explicit. The syntax of $\mathcal{L}^{\rightarrow[]}$ is defined as follows:

$$
\begin{array}{lcl}
\tau & ::= & b \mid \tau_1 \rightarrow \tau_2 \mid [\tau] \\
e & ::= & v \mid \texttt{let}\, x{:}\tau_1\, \texttt{be}\, e_1\, \texttt{in}\, e_2 \mid e_1(e_2) \\
v & ::= & x \mid c \mid [v] \mid \texttt{fun}\,(x{:}\tau)\,\texttt{in}\, e
\end{array}
$$

The typing rules are as for $\mathcal{L}^{\rightarrow}$, with the following additions:

$$
\frac{\Gamma \vdash v : \tau}{\Gamma \vdash [v] : [\tau]} \qquad (\textsc{t-comp})
$$

$$
\frac{\Gamma \vdash e_1 : [\tau_1] \qquad \Gamma[x{:}\tau_1] \vdash e_2 : \tau_2}{\Gamma \vdash \texttt{let}\, x{:}\tau_1\, \texttt{be}\, e_1\, \texttt{in}\, e_2 : \tau_2} \quad (x \notin \mathrm{dom}(\Gamma)) \qquad (\textsc{t-let})
$$

Evaluation is defined as for $\mathcal{L}^{\rightarrow}$, with the addition of an evaluation context of the form $\texttt{let}\, x{:}\tau\, \texttt{be}\, E\, \texttt{in}\, e$, and the following primitive computation step:

$$
\texttt{let}\, x{:}\tau\, \texttt{be}\, [v]\, \texttt{in}\, e \leadsto \{v/x\}e
$$

The translation from $\mathcal{L}^{\rightarrow}$ to $\mathcal{L}^{\rightarrow[\,]}$ may be motivated by considering the following translation of types and contexts:

$$
\begin{array}{rcl}
|\tau| & = & [||\tau||] \\
\\
||b|| & = & b \\
||\tau_1 \rightarrow \tau_2|| & = & ||\tau_1|| \rightarrow |\tau_2| \\
\\
||\Gamma||(x) & = & ||\Gamma(x)||
\end{array}
$$

The type $|\tau|$ represents the "computations" of type τ, and the type $||\tau||$ represents the "values" of type τ.

The translation from $\mathcal{L}^{\rightarrow}$ to $\mathcal{L}^{\rightarrow[\,]}$ is a four place relation $\Gamma \vdash e : \tau \Rightarrow |e|$ defined inductively by the following rules:

$$\Gamma \vdash x : \tau \Rightarrow [x] \quad (\Gamma(x) = \tau) \tag{C-VAR}$$

$$\Gamma \vdash c : b \Rightarrow [c] \quad \textit{(c has type b)} \tag{C-CONST}$$

$$\frac{\Gamma[x{:}\tau_1] \vdash e : \tau_2 \Rightarrow |e|}{\Gamma \vdash \texttt{fun}\,(x{:}\tau_1)\,\texttt{in}\,e : \tau_1 \rightarrow \tau_2 \Rightarrow [\texttt{fun}\,(x{:}||\tau_1||)\,\texttt{in}\,e]} \quad (x \notin \mathrm{dom}(\Gamma)) \tag{C-ABS}$$

$$\frac{\Gamma \vdash e_1 : \tau_2 \rightarrow \tau \Rightarrow |e_1| \qquad \Gamma \vdash e_2 : \tau_2 \Rightarrow |e_2|}{\Gamma \vdash e_1(e_2) : \tau \Rightarrow \texttt{let}\,x_1{:}\tau_2 \rightarrow \tau\,\texttt{be}\,|e_1|\,\texttt{in}\,\texttt{let}\,x_2{:}\tau_2\,\texttt{be}\,|e_2|\,\texttt{in}\,x_1(x_2)} \tag{C-APP}$$

The translation has the following notable properties:

1. Applications only arise in the form $v_1(v_2)$.
2. Intermediate computation steps are named.
3. Sequencing of evaluation is made explicit using `let`.

Moreover, these properties are preserved under evaluation since only values are ever substituted for variables.

Exercise 21.1
Suppose that primitive arithmetic operations are added to the source language. What happens to the target language, and how must the translation be extended to account for these primitives?

Theorem 21.2

1. *Suppose that* $\Gamma \vdash e : \tau \Rightarrow |e|$*. Then* $\Gamma \vdash e : \tau$ *and* $||\Gamma|| \vdash |e| : |\tau|$.
2. *If* $\Gamma \vdash e : \tau$*, then there exists a unique* $|e|$ *such that* $\Gamma \vdash e : \tau \Rightarrow |e|$.

Proof: Both parts are routine inductions based on the definitions. ∎

The source program and its translation are related by establishing a correspondence between computations and values in the source language and computations and values in the target language such that values of basic type correspond only if they are equal.

Definition 21.3

$$e \sim_\tau e' \quad \textit{iff} \quad e \Downarrow v \textit{ iff } e' \Downarrow [v'] \textit{ and } v \approx_\tau v'$$

$$\begin{array}{rcl} v \approx_b v' & \textit{iff} & v = v' = c : b \\ v \approx_{\tau_1 \to \tau_2} v' & \textit{iff} & v_1 \approx_{\tau_1} v_1' \textit{ implies } v(v_1) \sim_{\tau_2} v'(v_1') \end{array}$$

$$\eta \approx_\Gamma \eta' \quad \textit{iff} \quad \eta(x) \approx_{\Gamma(x)} \eta'(x) \quad (\forall x \in \mathrm{dom}(\Gamma))$$

Theorem 21.4
If $\Gamma \vdash e : \tau \Rightarrow |e|$ *and* $\eta \approx_\Gamma \eta'$, *then* $\hat{\eta}(e) \sim_\tau \widehat{\eta'}(|e|)$.

Proof: By induction on the derivation of the compilation relation.

c-var Follows immediately from the assumption that $\eta \approx_\Gamma \eta'$.

c-const Trivial.

c-abs It suffices to show that

$$\texttt{fun}\,(x{:}\tau_1)\,\texttt{in}\,\hat{\eta}(e) \approx_{\tau_1 \to \tau_2} \texttt{fun}\,(x{:}||\tau_1||)\,\texttt{in}\,\widehat{\eta'}(|e|)$$

Suppose that $v_1 \approx_{\tau_1} v_1'$. It suffices to show that

$$\widehat{\eta[x \mapsto v_1]}(e) \sim_{\tau_2} \widehat{\eta'[x \mapsto v_1']}(|e|)$$

But this follows directly from the inductive hypothesis, noting that $\eta[x \mapsto v_1] \approx_{\Gamma[x:\tau_1]} \eta'[x \mapsto v_1']$.

c-app We are to show that $\hat{\eta}(e)) \sim_\tau \widehat{\eta'}(|e|)$ where $e = e_1(e_2)$ and

$$|e| = \texttt{let}\,x_1{:}\tau_2 \to \tau\,\texttt{be}\,|e_1|\,\texttt{in}\,\texttt{let}\,x_2{:}\tau_2\,\texttt{be}\,|e_2|\,\texttt{in}\,x_1(x_2).$$

By induction we have $\hat{\eta}(e_1) \sim_{\tau_2 \to \tau} \widehat{\eta'}(|e_1|)$ and $\hat{\eta}(e_2) \sim_{\tau_2} \widehat{\eta'}(|e_2|)$. It follows that $\hat{\eta}(e_1) \Downarrow v_1$, $\widehat{\eta'}(|e_1|) \Downarrow [v_1']$, and $v_1 \approx_{\tau_2 \to \tau} v_1'$. Similarly, $\hat{\eta}(e_2) \Downarrow v_2$, $\widehat{\eta'}(|e_2|) \Downarrow [v_2']$, and $v_2 \approx_{\tau_2} v_2'$. It follows that $v_1(v_2) \sim_\tau v_1'(v_2')$, from which the result follows by closure under inverse evaluation.

∎

Corollary 21.5
If $\vdash e : b \Rightarrow |e|$, *then* $e \Downarrow c$ *iff* $|e| \Downarrow [c]$.

21.2 References

Chapter 22

Continuation-Passing Style

22.1 Continuation-Passing Style

As the operational semantics given above makes clear, it is essential to the implementation of $\mathcal{L}^{\rightarrow,\texttt{Cont}}$ that evaluation contexts be "reified" as values. Given a complete program one can statically determine a finite set of evaluation context "templates" such that all evaluation contexts that arise during any execution are substitution instances of these templates. This is directly analogous to higher-order functions: while the "code bodies" of all functions that arise dynamically are statically apparent, the bindings of the free variables are only determined at run time. This analogy suggests a compilation strategy in which continuations are reified as λ-abstractions so as to take advantage of this similarity between the two.

The *CPS transformation* is a translation from $\mathcal{L}^{\rightarrow,\texttt{Cont}}$ into $\mathcal{L}^{\rightarrow}$ with the following properties:

1. The program and its translation have the same value at basic type.

2. The result of each basic computation step is bound to a variable.

3. The evaluation order of arguments to primitives (including application) is made explicit.

4. Evaluation contexts are reified as λ-abstractions.

The first three properties are shared with the "explicit computation" translation given in Chapter 2. The main advantage of the CPS transformation is the explicit representation of control context.

The translation of a $\mathcal{L}^{\rightarrow,\texttt{Cont}}$ term is a $\mathcal{L}^{\rightarrow}$ term of a limited form, called *CPS*. The *CPS sub-language* of $\mathcal{L}^{\rightarrow}$ is defined by the following grammar:

$$\begin{array}{llcl} \textit{CPSExpressions} & e_{cps} & ::= & v_{cps} \mid v_{cps}(v'_{cps}) \mid v_{cps}(v'_{cps})(v''_{cps}) \\ \textit{CPSValues} & v_{cps} & ::= & x \mid c \mid \texttt{fun}\,(x{:}\tau)\,\texttt{in}\,e_{cps} \end{array}$$

The typing rules for the CPS sub-language are simply the restriction of the typing rules for $\mathcal{L}^{\rightarrow}$ restricted to the CPS sub-language. One aspect of the CPS sub-language that is not captured by this grammar is that the application $v_{cps}(v'_{cps})$ in $v_m athitcps(v'_{cps})(v''_{cps})$ is "trivial" in the sense that it always terminates. In a language with products this property may be captured by considering instead the application $v_{cps}(<v'_{cps},v''_{cps}>)$.

It is immediately obvious from the definitions that the CPS sub-language is closed under substitution: $\{v_{cps}/x\}e_{cps}$ is a CPS expression and $\{v_{cps}/x\}v'_{cps}$ is a CPS value. With this in mind it is clear that the reduction semantics for $\mathcal{L}^{\rightarrow}$ restricts to the CPS sub-language. Note, however, that fewer evaluation contexts are required: the restrictions on CPS ensure that evaluation contexts of the form $v(E)$ never arise, and consequently that there is no distinction between call-by-name and call-by-value for the CPS sub-language.

It is remarkable that the restrictions on the CPS sub-language do not limit the expressiveness of the language in the sense that every closed expression of basic type has a representation in the CPS sub-language with the same value.

The *call-by-value type translation* associated with the CPS transformation is a translation of types in $\mathcal{L}^{\rightarrow,\mathtt{Cont}}$ to types in $\mathcal{L}^{\rightarrow}$ defined as follows:

Definition 22.1

$$
\begin{array}{rcl}
|\tau| & := & (||\tau|| \rightarrow \mathtt{ans}) \rightarrow \mathtt{ans} \\
\\
||b|| & := & b \\
||\tau_1 \rightarrow \tau_2|| & := & ||\tau_1|| \rightarrow |\tau_2| \\
||\tau\ \mathtt{Cont}|| & := & ||\tau|| {\rightarrow} \mathtt{ans} \\
\\
||\Gamma||(x) & := & ||\Gamma(x)||
\end{array}
$$

Here the "computations" of type τ are functions mapping continuations of type τ to answers, where a continuation of type τ is a function mapping values of type τ to answers. The *call-by-name* type translation is defined similarly, except that we take $||\tau_1 \rightarrow \tau_2|| = |\tau_1| \rightarrow |\tau_2|$.

The *call-by-value CPS transformation* consists of two relations, $\Gamma \vdash v : \tau \Rightarrow_v ||v||$ and $\Gamma \vdash e : \tau \Rightarrow |e|$, that are defined as follows:

Definition 22.2

$$\Gamma \vdash x : \tau \Rightarrow_v x \quad (\Gamma(x) = \tau) \qquad \text{(CPS-VAR)}$$

$$\Gamma \vdash c : b \Rightarrow_v c \quad (c : b) \qquad \text{(CPS-CONST)}$$

$$\frac{\Gamma[x{:}\tau_1] \vdash e : \tau_2 \Rightarrow |e|}{\Gamma \vdash \mathtt{fun}\,(x{:}\tau_1)\,\mathtt{in}\,e \Rightarrow_v \mathtt{fun}\,(x{:}||\tau_1||)\,\mathtt{in}\,|e|} \quad (x \notin \mathrm{dom}(\Gamma)) \qquad \text{(CPS-ABS)}$$

$$\frac{\Gamma[x{:}\tau] \vdash E[x] : \mathtt{ans} \Rightarrow |E[x]|}{\Gamma \vdash E : \tau\,\mathtt{Cont} \Rightarrow_v \mathtt{fun}\,(x{:}||\tau||)\,\mathtt{in}\,|E[x]|} \quad (x \notin \mathrm{dom}(\Gamma)) \qquad (\textsc{cps-cont})$$

$$\frac{\Gamma \vdash v : \tau \Rightarrow_v ||v||}{\Gamma \vdash v : \tau \Rightarrow \lambda x{:}||\tau||{\to}\mathtt{ans}.x(||v||)} \qquad (\textsc{cps-val})$$

$$\frac{\Gamma \vdash e_1 : \tau_2 \to \tau \Rightarrow |e_1| \qquad \Gamma \vdash e_2 : \tau_2 \Rightarrow |e_2|}{\Gamma \vdash e_1(e_2) : \tau \Rightarrow \mathtt{fun}\,(x{:}||\tau||{\to}\mathtt{ans})\,\mathtt{in}\,|e_1|(\mathtt{fun}\,(x_1{:}||\tau_2 \to \tau||)\,\mathtt{in}\,|e_2|(\mathtt{fun}\,(x_2{:}||\tau_2||)\,\mathtt{in}\,x_1(x_2)(x)))} \qquad (\textsc{cps-app})$$

$$\frac{\Gamma[x{:}\tau\,\mathtt{Cont}] \vdash e : \tau \Rightarrow |e|}{\Gamma \vdash \mathtt{letcc}\,x{:}\tau\,\mathtt{Cont}\,\mathtt{in}\,e : \tau \Rightarrow \mathtt{fun}\,(y{:}||\tau||{\to}\mathtt{ans})\,\mathtt{in}\,\{y/x\}|e|(y)} \quad (x \notin \mathrm{dom}(\Gamma)) \qquad (\textsc{cps-letcc})$$

$$\frac{\Gamma \vdash e_1 : \tau \Rightarrow |e_1| \qquad \Gamma \vdash e_2 : \tau\,\mathtt{Cont} \Rightarrow |e_2|}{\Gamma \vdash \mathtt{throw}\,e_1\,\mathtt{to}\,e_2 : \tau' \Rightarrow \mathtt{fun}\,(x{:}||\tau'||{\to}\mathtt{ans})\,\mathtt{in}\,|e_2|(\mathtt{fun}\,(x_2{:}||\tau\,\mathtt{Cont}||)\,\mathtt{in}\,|e_1|(x_2))} \qquad (\textsc{cps-throw})$$

The typing properties of the translation are summarized by the following lemma.

Lemma 22.3

1. *$\Gamma \vdash v : \tau$ in $\mathcal{L}^{\to,\mathtt{Cont}}$ iff there exists a value $||v||$ such that $\Gamma \vdash v : \tau \Rightarrow_v ||v||$ and $||\Gamma|| \vdash ||v|| : ||\tau||$ in $\mathcal{L}^{\to}$.*

2. *$\Gamma \vdash e : \tau$ in $\mathcal{L}^{\to,\mathtt{Cont}}$ iff there exists a value $|e|$ such that $\Gamma \vdash e : \tau \Rightarrow |e|$ and $||\Gamma|| \vdash |e| : |\tau|$ in $\mathcal{L}^{\to}$.*

Exercise 22.4
Prove Lemma 22.3.

Exercise 22.5
Give a call-by-name *CPS transformation and prove a suitable type preservation property for it based on the call-by-name type translation given above.*

Exercise 22.6

1. *Extend the CPS transform to account for integers and booleans, with primitive operations for addition, subtraction, test for zero, and conditional expression.*

2. *Write out the CPS transform of the expression*

$$\mathtt{fun}\,(x{:}\mathtt{Int})\,\mathtt{in}\,\mathtt{if}_{\mathtt{Int}}\,x = 0\,\mathtt{then}\,x + 1\,\mathtt{else}\,x - 1\,\mathtt{fi}$$

Exercise 22.7
Extend the CPS transform to account for product types under both eager and (fully) lazy interpretations.

1. *Give the type transforms appropriate for each interpretation of products.*

2. *Give the CPS transform for pairing and projection under each interpretation, and show that types are preserved (in the appropriate sense).*

It remains to establish the correctness of the CPS transform. To do so we employ an interpretation of types as binary relations between expressions of $\mathcal{L}^{\to,\texttt{Cont}}$ and expressions of $\mathcal{L}^{\to}$ that relates their behavior under evaluation.

If e is an expression of basic type b in $\mathcal{L}^{\to,\texttt{Cont}}$ and e' is an expression of type b in $\mathcal{L}^{\to}$, then we define $e \simeq e'$ to hold iff $e \Downarrow c$ iff $e' \Downarrow c$.

Definition 22.8

$$
\begin{array}{rll}
e \sim_\tau e_{cps} & \textit{iff} & E \approx_{\tau\,\texttt{Cont}} k_{cps} \textit{ implies } E[e] \simeq e_{cps}(k_{cps}) \\
\\
v \approx_b v_{cps} & \textit{iff} & v = v_{cps} = c : b \\
v \approx_{\tau_1 \to \tau_2} v_{cps} & \textit{iff} & v' \approx_{\tau_1} v'_{cps} \textit{ implies } v(v') \sim_{\tau_2} v_{cps}(v'_{cps}) \\
v \approx_{\tau\,\texttt{Cont}} v_{cps} & \textit{iff} & v' \approx_\tau v'_{cps} \textit{ implies } \texttt{throw}\,v'\,\texttt{to}\,v \simeq v_{cps}(v'_{cps}) \\
\\
\eta \approx_\Gamma \eta_{cps} & \textit{iff} & \forall x \in \mathrm{dom}(\Gamma)\ \eta(x) \approx_{\Gamma(x)} \eta_{cps}(x)
\end{array}
$$

It is instructive to observe that $v \approx_{\tau_1 \to \tau_2} v_{cps}$ iff $E[v(v')] \simeq v_{cps}(v'_{cps})(v''_{cps})$ whenever $v' \approx_{\tau_1} v'_{cps}$ and $E \approx_{\tau_2\,\texttt{Cont}} v''_{cps}$.

Exercise 22.9
Convince yourself that the system of relations given above is well-defined.

Theorem 22.10

1. *If* $\Gamma \vdash v : \tau \Rightarrow_v ||v||$ *and* $\eta \approx_\Gamma \eta_{cps}$*, then* $\hat{\eta}(v) \approx_\tau \hat{\eta}_{cps}(||v||)$.

2. *If* $\Gamma \vdash e : \tau \Rightarrow |e|$*, and* $\eta \approx_\Gamma \eta_{cps}$*, then* $\hat{\eta}(e) \sim_\tau \hat{\eta}_{cps}(|e|)$.

Proof: We prove both parts simultaneously by induction on the translation relation.

cps-var Immediate.

cps-const Immediate.

cps-abs We are to show that $\texttt{fun}\,(x{:}\tau_1)\,\texttt{in}\,\hat{\eta}(e) \approx_{\tau_1 \to \tau_2} \texttt{fun}\,(x{:}||\tau_1||)\,\texttt{in}\,\hat{\eta}_{cps}(|e|)$. Suppose that $v' \approx_{\tau_1} v'_{cps}$. It suffices to show that

$$\eta\widehat{[x \mapsto v']}(e) \sim_{\tau_2} \eta_{cps}\widehat{[x \mapsto v'_{cps}]}(|e|).$$

This follows from the inductive hypothesis, since $\eta[x \mapsto v'] \approx_{\Gamma[x:\tau_1]} \eta_{cps}[x \mapsto v'_{cps}]$

cps-cont We are to show that $\hat{\eta}(E) \approx_{\tau\,\texttt{Cont}} \texttt{fun}\,(x{:}||\tau||)\,\texttt{in}\,\hat{\eta}_{cps}(|E[x]|)$. Suppose that $v \approx_\tau v_{cps}$. It suffices to show that

$$\eta\widehat{[x \mapsto v]}(e) \sim_{\texttt{ans}} \eta_{cps}\widehat{[x \mapsto v_{cps}]}(|E[x]|)$$

since

$$e\hat{t}a(E)[v] = \widehat{\eta x \mapsto v}(E[x])$$

and

$$\texttt{fun}\,(x{:}||\tau||)\,\texttt{in}\,\hat{\eta}_{cps}(|E[x]|)(v_{cps}) \simeq \eta_{cps}\widehat{[x \mapsto v_{cps}]}(|E[x]|).$$

But this follows directly from the inductive hypothesis since $\eta[x \mapsto v] \approx_{\Gamma[x:\tau]} \eta_{cps}[x \mapsto v_{cps}]$.

cps-val We are to show that $\hat{\eta}(v) \sim_\tau \texttt{fun}\,(x{:}||\tau||{\rightarrow}\texttt{ans})\,\texttt{in}\,x(\hat{\eta}_{cps}(||v||))$. Suppose that $E \approx_{\tau\,\texttt{Cont}} k_{cps}$; we are to show that

$$E[\hat{\eta}(v)] \simeq (\texttt{fun}\,(x{:}||\tau||{\rightarrow}\texttt{ans})\,\texttt{in}\,x(\hat{\eta}_{cps}(||v||)))(k_{cps}).$$

Note that

$$(\texttt{fun}\,(x{:}||\tau||{\rightarrow}\texttt{ans})\,\texttt{in}\,x(\hat{\eta}_{cps}(||v||)))(k_{cps}) \mapsto k_{cps}(\hat{\eta}_{cps}(||v||))$$

and $\texttt{throw}\,\hat{\eta}(v)\,\texttt{to}\,E \mapsto E[\hat{\eta}(v)]$. Therefore

$$E[\hat{\eta}(v)] \simeq k_{cps}(\hat{\eta}_{cps}(||v||)),$$

since $\hat{\eta}(v) \approx_\tau \hat{\eta}_{cps}(||v||)$ by the inductive hypothesis and $E \approx_{\tau\,\texttt{Cont}} k_{cps}$ by assumption.

cps-app We are to show that $\hat{\eta}(e) \sim_\tau \hat{\eta}_{cps}(|e|)$, where $e = e_1(e_2)$ and

$$|e| = \texttt{fun}\,(x{:}||\tau||{\rightarrow}\texttt{ans})\,\texttt{in}\,|e_1|(\texttt{fun}\,(x_1{:}||\tau_2 \rightarrow \tau||)\,\texttt{in}\,|e_2|(\texttt{fun}\,(x_2{:}||\tau_2||)\,\texttt{in}\,x_1(x_2)(x))).$$

Suppose that $E \approx_{\tau\,\texttt{Cont}} k_{cps}$; we are to show that

$$E[e] \simeq \hat{\eta}_{cps}(|e|)(k_{cps}).$$

It suffices to show that $E' \approx_{\tau_2 \rightarrow \tau\,\texttt{Cont}} k'_{cps}$ where $E' = E[\bullet(e_2)]$ and $k'_{cps} = \texttt{fun}\,(x_1{:}||\tau_2 \rightarrow \tau||)\,\texttt{in}\,\hat{\eta}_{cps}(|e_2|)(\texttt{fun}\,(x_2{:}||\tau_2||)\,\texttt{in}\,x_1(x_2)(k_{cps}))$, for then the result follows by the inductive hypothesis, together with the observation that $E'[e_1] = E[e_1(e_2)] = E[e]$ and

$$\hat{\eta}_{cps}(|e_1|)(k_{cps}) \mapsto \hat{\eta}_{cps}(|e_1|)(k'_{cps}).$$

To this end suppose that $v' \approx_{\tau_2 \rightarrow \tau} v'_{cps}$; we are to show that $\texttt{throw}\,v'\,\texttt{to}\,E' \simeq k'_{cps}(v'_{cps})$. It suffices to show that $E'' \approx_{\tau_2\,\texttt{Cont}} k''_{cps}$ where $E'' = E[v'(\,\bullet\,)]$ and $k''_{cps} = \texttt{fun}\,(x_2{:}||\tau_2||)\,\texttt{in}\,v'_{cps}(x_2)(k_{cps})$ from which the result follows by the inductive hypothesis and the fact that $\texttt{throw}\,v'\,\texttt{to}\,E' \mapsto E'[v'] = E''[e_2]$ and

$$k'_{cps}(v'_{cps}) \mapsto |e_2|(k''_{cps}).$$

Suppose that $v'' \approx_{\tau_2} v''_{cps}$; we are to show that $\texttt{throw}\,v''\,\texttt{to}\,E'' \simeq k''_{cps}(v''_{cps})$. Note that $\texttt{throw}\,v''\,\texttt{to}\,E'' \mapsto E''[v''] = E[v'(v'')]$ and that $k''_{cps}(v''_{cps}) \mapsto v'_{cps}(v''_{cps})(k_{cps})$, from which the result follows from the assumptions that $v' \approx_{\tau_2 \to \tau} v'_{cps}$, $v'' \approx_{\tau_2} v''_{cps}$, and $E \approx_{\tau\,\texttt{Cont}} k_{cps}$.

cps-letcc We are to show that

$$\texttt{letcc}\,x{:}\tau\,\texttt{Cont}\,\texttt{in}\,\hat{\eta}(e) \sim_\tau \texttt{fun}\,(y{:}||\tau||{\to}\texttt{ans})\,\texttt{in}\,\{y/x\}\hat{\eta}_{cps}(|e|)(y).$$

Suppose that $E \approx_{\tau\,\texttt{Cont}} k_{cps}$; it suffices to show that

$$E[\texttt{letcc}\,x{:}\tau\,\texttt{Cont}\,\texttt{in}\,\hat{\eta}(e) \simeq \eta_{cps}\widehat{[x \mapsto k_{cps}]}(|e|)(k_{cps}).$$

Since $E[\texttt{letcc}\,x{:}\tau\,\texttt{Cont}\,\texttt{in}\,\hat{\eta}(e)] \mapsto \eta\widehat{[x \mapsto E]}(e)$, it suffices to argue that

$$\eta[x \mapsto E] \approx_{\Gamma[x:\tau\,\texttt{Cont}]} \eta_{cps}[x \mapsto k_{cps}]$$

which follows directly from the assumptions.

cps-throw We are to show that

$$\texttt{throw}\,\hat{\eta}(e_1)\,\texttt{to}\,\hat{\eta}(e_2) \sim_{\tau'} \texttt{fun}\,(x{:}||\tau'||{\to}\texttt{ans})\,\texttt{in}\,\hat{\eta}_{cps}(|e_2|)(\texttt{fun}\,(x_2{:}||\tau\,\texttt{Cont}||)\,\texttt{in}\,\hat{\eta}_{cps}(|e_1|)(x_2)).$$

Suppose that $E \approx_{\tau'\,\texttt{Cont}} k_{cps}$; we are to show that

$$E[\texttt{throw}\,\hat{\eta}(e_1)\,\texttt{to}\,\hat{\eta}(e_2)] \simeq \hat{\eta}_{cps}(|e_2|)(\texttt{fun}\,(x_2{:}||\tau\,\texttt{Cont}||)\,\texttt{in}\,\hat{\eta}_{cps}(|e_1|)(x_2)).$$

It suffices to show that $E' \approx_{\tau\,\texttt{Cont}\,\texttt{Cont}} k'_{cps}$, where

$$E' = E[\texttt{throw}\,\hat{\eta}(e_1)\,\texttt{to}\,\bullet]$$

and

$$k'_{cps} = \texttt{fun}\,(x_2{:}||\tau\,\texttt{Cont}||)\,\texttt{in}\,\hat{\eta}_{cps}(|e_1|)(x_2).$$

Suppose that $E'' \approx_{\tau\,\texttt{Cont}} k''_{cps}$. Since

$$E'[E''] = E[\texttt{throw}\,\hat{\eta}(e)\,\texttt{to}\,E''] \mapsto E''[\hat{\eta}(e_1)]$$

and

$$k'_{cps}(k''_{cps}) \mapsto \hat{\eta}_{cps}(|e_1|)(k''_{cps})$$

the result follows by an application of the inductive hypothesis.

■

For closed expressions of base type we may relate the "direct" and "cps" semantics by choosing the answer type appropriately.

Corollary 22.11
If $\vdash e : \texttt{ans} \Rightarrow |e|$, *then* $e \simeq |e|(\texttt{fun}\,(x{:}\texttt{ans})\,\texttt{in}\,x)$.

Proof: Observe that $\bullet \approx_{b\,\texttt{Cont}} \texttt{fun}\,(x{:}b)\,\texttt{in}\,x$ provided that we take $\texttt{ans} = b$. The result then follows by Theorem 22.10. ■

Corollary 22.12
If $\vdash e : \texttt{ans}$ *in* $\mathcal{L}^{\rightarrow,\texttt{Cont}}$, *then* $e \Downarrow$.

Proof: By the previous corollary $e \Downarrow$ iff $|e|(\texttt{fun}\,(x{:}\texttt{ans})\,\texttt{in}\,x) \Downarrow$. But all well-typed programs of $\mathcal{L}^{\rightarrow}$ terminate. ■

For closed expressions of non-basic type τ we cannot simply take $\texttt{ans} = \tau$. Instead we must postulate that there is an *initial continuation* mapping values of type τ to values of type ans. In typical cases $\texttt{ans} = \texttt{String}$, and we assume given an operation $\texttt{makestring}_\tau(-)$ of type $\tau \rightarrow \texttt{ans}$. The initial continuation in the "direct" semantics is defined to be

$$E_\tau := \texttt{makestring}_\tau(\bullet),$$

and in the "cps" semantics is defined to be the function

$$k_\tau := \lambda x{:}||\tau||.\texttt{makestring}_{||\tau||}(x).$$

It is easy to check that $E_\tau \approx_{\tau\,\texttt{Cont}} k_\tau$, from which it follows that $E_\tau[e] \simeq |e|(k_\tau)$ by Theorem 22.10.

22.2 References

The typing properties of the cps transform were first studied by Meyer and Wand [?] and subsequently extended to the polymorphic case by Harper and Lillibridge [?]. The extension of typed languages with control operators was studied by Harper, *et al.* [?] and Filinski [?]. The connection between control operators, cps conversion, and classical logic was established by Griffin [?] and Murthy [?].

Chapter 23

Closure-Passing Style

Chapter 24

Separate Compilation

24.1 Introduction

Separate compilation complicates compilation because of unknown, or variable, types. To get at the issues involved, it is helpful to remind ourselves that the "fully explicit" syntax for our languages attaches type information to all of the primitive operations. For example, the primitives of $\mathcal{L}^{1,\times,\rightarrow}$, when written in fully explicit form, are these:

$$\begin{array}{ll} \texttt{fun}\,(\tau_1{:}\tau_2){:}x\;\texttt{in}\,e & \texttt{app}_{\tau_1,\tau_2}(e_1,e_2) \\ <e_1,e_2>_{\tau_1,\tau_2} & \texttt{proj}^{i}_{\tau_1,\tau_2}(e) \end{array}$$

The abbreviated syntax used earlier is chosen to have sufficient information that the fully explicit syntax can be unambiguously recovered.

Now let us re-consider the operational semantics of $\mathcal{L}^{1,\times,\rightarrow}$. When written in terms of the fully explicit syntax the basic instructions are these:

$$\begin{array}{rcl} \texttt{app}_{\tau_1,\tau_2}(\texttt{fun}\,(\tau_1{:}\tau_2){:}x\;\texttt{in}\,e,v) & \leadsto & \{v/x\}e \\ \texttt{proj}^{i}_{\tau_1,\tau_2}(<v_1,v_2>_{\tau_1,\tau_2}) & \leadsto & v_i \end{array}$$

The fact that the types match in the way indicated is *not* something that needs to be checked during evaluation, rather it is a consequence of type preservation that the types *will* match during the reduction of a well-typed term. Thus we may limit attention to this case without loss of generality.

When written without type annotations, the basic instructions rely on the abstract syntax to decompose a pair into its constituent parts. In an implementation this corresponds to a form of run-time tagging: a pair must carry with it sufficient information to (a) identify it as a pair, and (b) determine the constituent components. Similar remarks apply to functions (closures) and the application primitive — the abstract syntax provides enough information for us to decompose a function and perform substitution. The point of explicit syntax is this: by attaching type information to primitive operations we ensure that they are implementable *without* relying on tagging of values.

One such implementation is based on defining the size of a value of each type, and relying on the size information to implement the primitive operations. For example, we might make the following definitions:

$$\begin{array}{rcl}
\texttt{sizeof}[b] & := & 1 \\
\texttt{sizeof}[\tau_1 \times \tau_2] & := & \texttt{sizeof}[\tau_1] + \texttt{sizeof}[\tau_2] \\
\texttt{sizeof}[\tau_1 \to \tau_2] & := & 1
\end{array}$$

Using this information, the implementation of pairing might proceed by allocating a region of memory of size $\texttt{sizeof}[\tau_1] + \texttt{sizeof}[\tau_2]$, copying its arguments into this region, and returning a pointer to it. Correspondingly, second projection would proceed by calculating $s_1 = \texttt{sizeof}[\tau_1]$ and $s_2 = \texttt{sizeof}[\tau_2]$, allocating s_2 words of memory, and copying to it s_2 words starting at offset s_1 words from the region passed as argument. Similarly, application might use $\texttt{sizeof}[\tau_1]$ and $\texttt{sizeof}[\tau_2]$ to effect the call and return sequence for a function of type $\tau_1 \to \tau_2$.

This model of execution yields a simple and natural approach to compilation in the absence of variable types. Specifically, each instance of a family of primitive operations is treated separately so that in general different code is generated for the pairing operation at each product type. Thus $<-,->_{\tau_1,\tau_2}$ is regarded as a distinct primitive operation from $<-,->_{\tau_1',\tau_2'}$, with the generated code determined by the types. (In the case that only size information is used by the primitives there is considerable uniformity among the implementations, but in general this need not be so.) Since type information is completely known at compile time, we can generate code for each of the finitely many instances of a primitive operation that arise in a given program. This property is known as *compile time code selection*, and is enjoyed by all languages without variable types.

The problem is to extend this property to languages with variable types. The essential difficulty is how to compile primitives at variable type. For example, if we are to extend the size-based implementation of pairing sketched above to the case of variable types, we must define $\texttt{sizeof}[t]$, where t is a type variable. But if the rule **??** is to be validated, then we must ensure that $\texttt{sizeof}[-]$ is compatible with substitution in the sense that $\texttt{sizeof}[t]$ must be the same as $\texttt{sizeof}[\tau]$ for *any* type τ. This suggests the *universal representation* strategy for variable types: values of unknown type must be represented uniformly as values of unit size. With this assumption we are ensured that the primitive operations are well-defined, even at variable types. We will explore two strategies for realizing this assumption.

24.2 Full Boxing

One approach is to adopt a uniform representation for all values so that $\texttt{sizeof}[\tau] = 1$ for all types τ. The standard way to achieve this is by *boxing* — all values are heap-allocated and represented by a pointer into the heap. In fact we shall abstract away from any particular implementation and work instead with an abstract notion of boxing. The full boxing representation is specified by a translation from $\mathcal{L}^{1,\times,\rightarrow}$ into the language $\mathcal{L}^{\rightarrow,\times,\texttt{box}}$ in which boxing is made explicit. The syntax of the boxing primitives in the target language is given by the following grammar:

$$\begin{array}{llll} \textit{Types} & \tau & ::= & \ldots \mid \tau\,\texttt{boxed} \\ \textit{Expressions} & e & ::= & \ldots \mid \texttt{box}_\tau(e) \mid \texttt{unbox}_\tau(e) \\ \textit{Values} & e & ::= & \ldots \mid \texttt{box}_\tau(v) \end{array}$$

The typing rules governing these expressions are as follows:

$$\frac{\Gamma \vdash_\Delta e : \tau}{\Gamma \vdash_\Delta \texttt{box}_\tau(e) : \tau\,\texttt{boxed}} \qquad (\text{T-BOX})$$

$$\frac{\Gamma \vdash_\Delta e : \tau\,\texttt{boxed}}{\Gamma \vdash_\Delta \texttt{unbox}_\tau(e) : \tau} \qquad (\text{T-UNBOX})$$

Evaluation is extended to include the instruction

$$\texttt{unbox}_\tau(\texttt{box}_\tau(v)) \leadsto v$$

expressing the obvious interpretation of boxing and unboxing.

Exercise 24.1
Prove that the type system of $\mathcal{L}^{\rightarrow,\times,\texttt{box}}$ is sound by checking preservation and progress.

The type translation associated with the "full boxing" representation is given as follows:

$$|\tau| \quad := \quad ||\tau||\,\texttt{boxed}$$

$$\begin{array}{rcl} ||t|| & := & t \\ ||b|| & := & b \\ ||\tau_1 \times \tau_2|| & := & |\tau_1| \times |\tau_2| \\ ||\tau_1 \rightarrow \tau_2|| & := & |\tau_1| \rightarrow |\tau_2| \end{array}$$

$$|\Gamma|(x) \quad := \quad |\Gamma(x)| \quad (x \in \mathrm{dom}(\Gamma))$$

The type $||\tau||$ may be thought of as the "natural" or "unboxed" representation of values of type τ. Basic types are represented as themselves, pairs are represented as pairs of boxed values, and functions map boxed arguments to boxed results.

Exercise 24.2
Check that $\{||\tau||/t\}||\tau'|| = ||\{\tau/t\}\tau'||$ and that $\{||\tau||/t\}|\tau'| = |\{\tau/t\}\tau'|$.

The term translation associated with the "full boxing" representation is given by a relation $\Gamma \vdash_\Delta e : \tau \Rightarrow |e|$ defined by induction on typing derivations as follows:

$$\Gamma \vdash_\Delta x : \tau \Rightarrow x \quad (\Gamma(x) = \tau) \qquad (\text{C-B-VAR})$$

$$\Gamma \vdash_\Delta c : b \Rightarrow \texttt{box}_b(c) \qquad \text{(C-B-CONST)}$$

$$\frac{\Gamma[x{:}\tau_1] \vdash_\Delta e : \tau_2 \Rightarrow |e|}{\Gamma \vdash_\Delta \texttt{fun}\,(x{:}\tau_1)\,\texttt{in}\, e : \tau_1 \to \tau_2 \Rightarrow \texttt{box}_{|\tau_1| \to |\tau_2|}(\texttt{fun}\,(|\tau_1|{:}|\tau_2|){:}x\;\texttt{in}\,|e|)} \quad (x \notin \text{dom}(\Gamma)) \qquad \text{(C-B-ABS)}$$

$$\frac{\Gamma \vdash_\Delta e_1 : \tau_2 \to \tau \Rightarrow |e_1| \qquad \Gamma \vdash_\Delta e_2 : \tau_2 \Rightarrow |e_2|}{\Gamma \vdash_\Delta e_1(e_2) : \tau \Rightarrow \texttt{app}_{|\tau_2|,|\tau|}(\texttt{unbox}_{|\tau_2| \to |\tau|}(|e_1|),|e_2|)} \qquad \text{(C-B-APP)}$$

We have used explicit syntax on the right-hand side to emphasize the code selection issues involved here. In particular, notice that the primitive operations (other than `box` and `unbox`) are only ever used at boxed types. Assuming that boxed values are manipulated uniformly by the primitives, this means that only one instance of any primitive operation (other than boxing and unboxing) is ever required in the generated code. Furthermore, the boxing and unboxing primitives are never required at variable type. Taken together, this means that we achieve compile time code selection in the presence of variable types, at the expense of representing all values in boxed form.

Exercise 24.3
Give the remaining clauses for the compilation of $\mathcal{L}^{1,\times,\to}$. Check that compile-time code selection is maintained.

Types are preserved under translation in the following sense.

Theorem 24.4
If $\Gamma \vdash_\Delta e : \tau$, then there exists $|e|$ such that $\Gamma \vdash_\Delta e : \tau \Rightarrow |e|$ and $|\Gamma| \vdash_\Delta |e| : |\tau|$.

Exercise 24.5
Prove Theorem 24.4.

Under the full boxing representation linking commutes with compilation.

Theorem 24.6

1. *If $\Gamma \vdash_\Delta v : \tau_v \Rightarrow |v|$ and $\Gamma[x{:}\tau_v] \vdash_\Delta e : \tau_e \Rightarrow |e|$, then $\Gamma \vdash_\Delta \{v/x\}e : \tau_e \Rightarrow \{|v|/x\}|e|$.*

2. *If $\Gamma \vdash_{\Delta[t]} e : \tau_e \Rightarrow |e|$ and $\Delta \vdash \tau$, then $\{\tau/t\}\Gamma \vdash_\Delta \{\tau/t\}e : \{\tau/t\}\tau_e \Rightarrow \{||\tau||/t\}|e|$.*

Exercise 24.7
Prove Theorem 24.6.

The correctness of the translation may be established by establishing a suitable relationship between a term and its translation with the property that if $e : b$, then $e \Downarrow c$ iff $|e| \Downarrow \texttt{box}_b(c)$. We define the system of relations $e \sim_\tau e'$, where $e : \tau$ and $e' : |\tau|$, $v \approx_\tau v'$, where $v : \tau$ and $v' : |\tau|$, and $\eta \approx_\Gamma \eta'$, where $\eta(x) : \Gamma(x)$ and $\eta'(x) : |\Gamma(x)|$ for each $x \in \mathrm{dom}(\Gamma)$, as follows:

$$e \sim_\tau e' \quad \textit{iff} \quad e \Downarrow v \textit{ iff } e' \Downarrow v' \textit{ and } v \approx_\tau v'$$

$$\begin{array}{rll}
v \approx_b v' & \textit{iff} & v = c \textit{ and } v' = \texttt{box}_b(c) \\
v \approx_{\tau_1 \to \tau_2} v' & \textit{iff} & v_1 \approx_{\tau_1} v_1' \textit{ implies } v(v_1) \sim_{\tau_2} \texttt{unbox}_{|\tau_1| \to |\tau_2|}(v')(v_1') \\
v \approx_{\tau_1 \times \tau_2} v' & \textit{iff} & \texttt{proj}_1(v) \sim_{\tau_1} \texttt{proj}_1(\texttt{unbox}_{|\tau_1| \times |\tau_2|}(v')) \textit{ and} \\
& & \texttt{proj}_2(v) \sim_{\tau_2} \texttt{proj}_2(\texttt{unbox}_{|\tau_1| \times |\tau_2|}(v'))
\end{array}$$

$$\eta \approx_\Gamma \eta' \quad \textit{iff} \quad \eta(x) \approx_{\Gamma(x)} \eta'(x) \ (\forall x \in \mathrm{dom}(\Gamma))$$

To obtain the correctness of the translation we prove that a term is related to its translation following the pattern of our previous correctness proofs. To take account of variable types we must consider the notion of a *closing substitution* for a set Δ of type variables, a function ρ assigning closed type expressions to the type variables in Δ. We write $\hat{\rho}(\tau)$ and $\hat{\rho}(e)$ for the application of ρ to types τ and expressions e, respectively. This is extended to contexts variable-wise by defining $\hat{\rho}(\Gamma)(x) = \hat{\rho}(\Gamma(x))$. Type translation is extended to substitutions variable-wise by defining $||\rho||(t) = ||\rho(t)||$.

Theorem 24.8
If $\Gamma \vdash_\Delta e : \tau \Rightarrow |e|$, ρ is a closing substitution for Δ, and $\eta \approx_{\hat{\rho}(\Gamma)} \eta'$, then

$$\hat{\eta}(\hat{\rho}(e)) \sim_{\hat{\rho}(\tau)} \widehat{\eta'}(\widehat{||\rho||}(|e|)).$$

Corollary 24.9
If $\vdash_\emptyset e : b \to |e|$, then $e \Downarrow c$ iff $|e| \Downarrow \texttt{box}_b(c)$.

Exercise 24.10
Prove Theorem 24.8 by induction on typing derivations.

24.3 Partial Boxing

The "full boxing" strategy may be criticized on the grounds that it incurs a time and space performance penalty even if variable types are never used. In particular, the natural representation $||\tau||$ of τ is not equal to τ even if no type variables occur in τ. It is desirable to pay only for those features of a language you actually use. Thus we seek a translation that admits compile-time code selection and is such that $||\tau|| = \tau$ when τ contains no type variables. The following type translation achieves these goals.

$$\begin{array}{rcl}
||t|| & := & t\,\texttt{boxed} \\
||b|| & := & b \\
||\tau_1 \times \tau_2|| & := & ||\tau_1|| \times ||\tau_2|| \\
||\tau_1 \to \tau_2|| & := & ||\tau_1|| \to ||\tau_2||
\end{array}$$

The translation does not, however, commute with substitution: there are types τ and τ' such that

$$\{||\tau||/t\}||\tau'|| \neq ||\{\tau/t\}\tau'||.$$

The mismatch stems from the fact that $||t|| = t\,\texttt{boxed}$ so that $\{||\tau||/t\}||t|| = ||\tau||\,\texttt{boxed} \neq ||\tau||$.

The "partial boxing" translation seeks to maintain values in their natural forms as long as their types are known. In compensation special precaution has to be taken to account for type substitution. The translation relation $\Gamma \vdash_\Delta e : \tau \Rightarrow ||e||$ is defined as follows:

$$\Gamma \vdash_\Delta x : \tau \Rightarrow x \quad (\Gamma(x) = \tau) \qquad \text{(C-PB-VAR)}$$

$$\Gamma \vdash_\Delta c : b \Rightarrow c \qquad \text{(C-PB-CONST)}$$

$$\frac{\Gamma[x{:}\tau_1] \vdash_\Delta e : \tau_2 \Rightarrow ||e||}{\Gamma \vdash_\Delta \texttt{fun}\,(x{:}\tau_1)\,\texttt{in}\,e : \tau_1 \to \tau_2 \Rightarrow \texttt{fun}\,(||\tau_1||{:}||\tau_2||){:}x\;\texttt{in}\,||e||} \quad (x \notin \mathrm{dom}(\Gamma)) \qquad \text{(C-PB-ABS)}$$

$$\frac{\Gamma \vdash_\Delta e_1 : \tau_2 \to \tau \Rightarrow ||e_1|| \qquad \Gamma \vdash_\Delta e_2 : \tau_2 \Rightarrow ||e_2||}{\Gamma \vdash_\Delta e_1(e_2) : \tau \Rightarrow \texttt{app}_{||\tau_2||,||\tau||}(||e_1||,||e_2||)} \qquad \text{(C-PB-APP)}$$

In contrast to the "full boxing" translation, we consider primitive operations at unboxed types, but never at a variable type, ensuring the compile-time code selection property.

Exercise 24.11
Give the "partial boxing" compilation rules for pairing and projection.

Theorem 24.12
If $\Gamma \vdash_\Delta e : \tau$, then there exists $||e||$ such that $\Gamma \vdash_\Delta e : \tau \Rightarrow ||e||$ and $||\Gamma|| \vdash_\Delta ||e|| : ||\tau||$.

Exercise 24.13
Prove Theorem 24.12.

Now let us consider the linking rules, beginning with the rule **??**. Under translation we have

$$\Gamma \vdash_\Delta v : \tau_v \Rightarrow ||v||$$

and

$$\Gamma[x{:}\tau_v] \vdash_\Delta e : \tau_e \Rightarrow ||e||$$

By Theorem 24.12 it follows that

$$||\Gamma|| \vdash_\Delta ||v|| : ||\tau_v||$$

and

$$||\Gamma||[x{:}||\tau_v||] \vdash_\Delta ||e|| : ||\tau_e||$$

from which it follows that

$$||\Gamma|| \vdash_\Delta \{||v||/x\}||e|| : ||\tau_e||.$$

We will argue below that this is a suitable translation of the **??** rule.

Now consider the type specialization rule **??**. Suppose that

$$\Gamma \vdash_{\Delta[t]} e : \tau_e \Rightarrow ||e||.$$

By Theorem 24.12 it follows that

$$||\Gamma|| \vdash_{\Delta[t]} ||e|| : ||\tau_e||.$$

By rule **??** we have

$$\{||\tau||/t\}||\Gamma|| \vdash_\Delta \{||\tau||/t\}||e|| : \{||\tau||/t\}||\tau_e||$$

since clearly $\Delta \vdash ||\tau||$ whenever $\Delta \vdash \tau$. By introducing suitable coercions that mediate between the types $\{||\tau||/t\}||\tau'||$ and $||\{\tau/t\}\tau'||$ we will obtain a term $(*)$ such that

$$||\{\tau/t\}\Gamma|| \vdash_\Delta (*) : ||\{\tau/t\}\tau_e||$$

which is a suitable translation for $\{\tau/t\}e$.

The definition of the coercions make use of the functionals $[e_1 \times e_2] : (\tau_1 \times \tau_2) \to (\tau_1' \times \tau_2')$ and $[e_1 \to e_2] : (\tau_1' \to \tau_2) \to (\tau_1 \to \tau_2')$, where $e_1 : \tau_1 \to \tau_1'$ and $e_2 : \tau_2 \to \tau_2'$. These are defined as follows:

$$[e_1 \times e_2] = \texttt{fun}\,(x{:}\tau_1 \times \tau_2)\,\texttt{in}\,{<}e_1(\texttt{proj}_1(x)),e_2(\texttt{proj}_2(x)){>}$$

and

$$[e_1 \to e_2] = \texttt{fun}\,(f{:}\tau_1' \to \tau_2)\,\texttt{in}\,\texttt{fun}\,(x{:}\tau_1)\,\texttt{in}\,e_2(f(e_1(x))).$$

Note that $[e_1 \times e_2](e)$ and $[e_1 \to e_2](e)$ evaluate to a value whenever e does.

Definition 24.14
The operations

$$\mathsf{S}_{[\tau/t]}[\tau'] : \{||\tau||/t\}||\tau'|| \to ||\{\tau/t\}\tau'||$$

and

$$\mathsf{G}_{[\tau/t]}[\tau'] : ||\{\tau/t\}\tau'|| \to \{||\tau||/t\}||\tau'||$$

are defined simultaneously by induction on the structure of τ' as follows:

$$\begin{array}{rcl}
\mathsf{S}_{[\tau/t]}[t'] & := & \left\{ \begin{array}{ll} \texttt{fun}\,(x{:}||\tau||\,\texttt{boxed})\,\texttt{in}\,\texttt{unbox}_{||\tau||}(x) & \textit{if } t' = t \\ id_{||t'||} & \textit{if } t' \neq t \end{array} \right. \\
\mathsf{S}_{[\tau/t]}[b] & := & id_b \\
\mathsf{S}_{[\tau/t]}[\tau_1 \times \tau_2] & := & [\mathsf{S}_{[\tau/t]}[\tau_1] \times \mathsf{S}_{[\tau/t]}[\tau_2]] \\
\mathsf{S}_{[\tau/t]}[\tau_1 \to \tau_2] & := & [\mathsf{G}_{[\tau/t]}[\tau_1] \to \mathsf{S}_{[\tau/t]}[\tau_2]]
\end{array}$$

$$\begin{array}{rcl}
\mathsf{G}_{[\tau/t]}[t'] & := & \left\{ \begin{array}{ll} \texttt{fun}\,(x{:}||\tau||)\,\texttt{in}\,\texttt{box}_{||\tau||}(x) & \textit{if } t' = t \\ id_{||t'||} & \textit{if } t' \neq t \end{array} \right. \\
\mathsf{G}_{[\tau/t]}[b] & := & id_b \\
\mathsf{G}_{[\tau/t]}[\tau_1 \times \tau_2] & := & [\mathsf{G}_{[\tau/t]}[\tau_1] \times \mathsf{G}_{[\tau/t]}[\tau_2]] \\
\mathsf{G}_{[\tau/t]}[\tau_1 \to \tau_2] & := & [\mathsf{S}_{[\tau/t]}[\tau_1] \to \mathsf{G}_{[\tau/t]}[\tau_2]]
\end{array}$$

Exercise 24.15
Check that

$$\mathsf{S}_{[\tau/t]}[\tau'] : \{||\tau||/t\}||\tau'|| \to ||\{\tau/t\}\tau'||$$

and

$$\mathsf{G}_{[\tau/t]}[\tau'] : ||\{\tau/t\}\tau'|| \to \{||\tau||/t\}||\tau'||.$$

Using these coercions we may define the required term $(*)$ corresponding to type instantiation as follows:

$$\begin{array}{l}
\texttt{let } x_1 \texttt{ be } \mathsf{G}_{[\tau/t]}[\tau_1](x_1) \texttt{ in} \\
\quad \ldots \\
\qquad \texttt{let } x_n \texttt{ be } \mathsf{G}_{[\tau/t]}[\tau_n](x_n) \texttt{ in} \\
\qquad\quad \mathsf{S}_{[\tau/t]}[\tau_e](\{||\tau||/t\}||e||)
\end{array}$$

where $\mathrm{dom}(\Gamma) = \{\, x_1, \ldots, x_n \,\}$ and $\Gamma(x_i) = \tau_i$ $(1 \leq i \leq n)$.

Exercise 24.16

1. *Work out the translation of* $\vdash_\emptyset \{0, \texttt{fun}\,(x{:}\texttt{Nat})\,\texttt{in}\,x + 1/x, f\} f(f(x)) : \texttt{Nat}$ *according to the partial boxing interpretation.*

2. *Work out the translations of* $x{:}t, f{:}t \to t \vdash_{\{\,t\,\}} f(f(x)) : t$, $\vdash_\emptyset 0 : \texttt{Nat}$, *and* $\vdash_\emptyset \texttt{fun}\,(x{:}\texttt{Nat})\,\texttt{in}\,x + 1 : \texttt{Nat} \to \texttt{Nat}$ *according to the partial boxing interpretation.*

3. *Work out the result of linking these two compilations using the type- and value-substitution rules, and compare your result with the compilation of the integrated expression.*

Lemma 24.17

1. *If* $v : ||\{\tau/t\}\tau'||$, *then* $\mathsf{S}_{[\tau/t]}[\tau'](\mathsf{G}_{[\tau/t]}[\tau'](v)) \simeq v$.

2. *If* $v : \{||\tau||/t\}||\tau'||$, *then* $\mathsf{G}_{[\tau/t]}[\tau'](\mathsf{S}_{[\tau/t]}[\tau'](v)) \simeq v$.

Proof: Simultaneously by induction on the structure of τ'. If $\tau' = t$, then we are to show that

$$\mathtt{box}_{||\tau||}(\mathtt{unbox}_{||\tau||}(v)) \simeq v$$

and that

$$\mathtt{unbox}_{||\tau||}(\mathtt{box}_{||\tau||}(v)) \simeq v$$

The latter follows directly from the evaluation rules. The former holds because $v : ||\tau||$ `boxed` and hence v must have the form $\mathtt{box}_{||\tau||}(v')$.

If $\tau' = t' \neq t$ or $\tau' = b$, then the result is immediate.

For $\tau' = \tau_1' \times \tau_2'$, we are to show that

$$[\mathsf{S}_{[\tau/t]}[\tau_1'] \times \mathsf{S}_{[\tau/t]}[\tau_2']]([\mathsf{G}_{[\tau/t]}[\tau_1'] \times \mathsf{G}_{[\tau/t]}[\tau_2']](v)) \simeq v$$

and that

$$[\mathsf{G}_{[\tau/t]}[\tau_1'] \times \mathsf{G}_{[\tau/t]}[\tau_2']]([\mathsf{S}_{[\tau/t]}[\tau_1'] \times \mathsf{S}_{[\tau/t]}[\tau_2']](v)) \simeq v.$$

As to the former, the left-hand side is Kleene equivalent to

$$<\mathsf{S}_{[\tau/t]}[\tau_1'](\mathsf{G}_{[\tau/t]}[\tau_1'](\mathtt{proj}_1(v))),\mathsf{S}_{[\tau/t]}[\tau_2'](\mathsf{G}_{[\tau/t]}[\tau_2'](\mathtt{proj}_2(v)))>$$

which by induction is Kleene equivalent to

$$<\mathtt{proj}_1(v),\mathtt{proj}_2(v)>$$

which is equivalent to v since v must be a pair $<v_1,v_2>$. The other case is handled symmetrically.

The case of $\tau' = \tau_1' \to \tau_2'$ is left to the reader. ■

Exercise 24.18
Complete the proof of Lemma 24.17.

The correctness of the partial boxing translation is proved by defining define type-indexed families of relations $e \sim_\tau e'$, where $e : \tau$ and $e' : ||\tau||$, and $v \approx_\tau v'$, where $v : \tau$ and $v' : ||\tau||$ as follows:

$$\begin{array}{rcl}
e \sim_\tau e' & \mathit{iff} & e \Downarrow v \ \mathit{iff}\ e' \Downarrow v' \ \mathit{and}\ v \approx_\tau v' \\
\\
v \approx_b v' & \mathit{iff} & v = v' = c : b \\
v \approx_{\tau_1 \to \tau_2} v' & \mathit{iff} & v_1 \approx_{\tau_1} v_1' \ \mathit{implies}\ v(v_1) \sim_{\tau_2} v'(v_1') \\
v \approx_{\tau_1 \times \tau_2} v' & \mathit{iff} & \mathtt{proj}_i(v) \sim_{\tau_i} \mathtt{proj}_i(v')\ (i = 1, 2) \\
\\
\eta \approx_\Gamma \eta' & \mathit{iff} & \eta(x) \approx_{\Gamma(x)} \eta'(x)\ (\forall x \in \mathrm{dom}(\Gamma))
\end{array}$$

These relations are quite straightforward, expressing essentially that e and e' are "extensionally equivalent", meaning that they behave the same way when used with the appropriate elimination forms for their type (application for function types, projection for product types).

To state the correctness theorem we generalize the S and G operations to $\mathsf{S}_\rho[\tau]$ and $\mathsf{G}_\rho[\tau]$, where ρ is a type substitution and τ is a type expression, in the obvious way. The correctness theorem for the translation is then stated as follows:

Theorem 24.19
If $\Gamma \vdash_\Delta e : \tau \Rightarrow ||e||$ with $\mathrm{dom}(\Gamma) = \{x_1, \ldots, x_n\}$, then for every closing substitution ρ for Δ and every pair of environments η and η' such that $\eta \approx_{\hat{\rho}(\Gamma)} \eta'$,

$$\hat{\eta}(\hat{\rho}(e)) \sim_{\hat{\rho}(\tau)} \texttt{let}\, x_i{:}\widehat{||\rho||}(||\tau_i||)\, \texttt{be}\, \mathsf{G}_\rho[\tau_i](\eta'(x_i))\, \texttt{in}\, \mathsf{S}_\rho[\tau](\widehat{||\rho||}(||e||)).$$

Corollary 24.20
If $\vdash e : b$, then $e \sim_b ||e||$.

Proof: Consider the empty type substitution and empty environments. ■

Exercise 24.21
Give a proof of Theorem 24.19.

Exercise 24.22
1. *Check that the linking rule corresponding to the rule* **??** *is correct in the sense of Theorem 24.19.*
2. *Prove that the linking rule corresponding to the rule* **??** *is correct in the sense of Theorem 24.19.*

Chapter 25

Data Representation

Chapter 26

Garbage Collection

Part IV

Models and Logics

Chapter 27

Denotational Semantics

27.1 Introduction

27.2 Types as Domains

The language $\mathcal{L}^{\rightarrow}$ admits a natural set-theoretic interpretation in which types are interpreted as sets and terms are interpreted as functions of their free variables. In particular, the type $\tau_1 \rightarrow \tau_2$ is interpreted as the full set-theoretic function space. This interpretation breaks down for $\mathcal{L}^{\rightharpoonup}$ because of the presence of unbounded recursion. One approach to achieving a "set-like" interpretation of $\mathcal{L}^{\rightharpoonup}$ is to interpret types as domains.

27.2.1 Denotational Semantics

A denotational semantics for $\mathcal{L}^{\rightharpoonup}$ consists of an assignment of domains to types together with an assignment of elements of suitable domains to well-typed terms.

The interpretation of types is defined as follows:

Definition 27.1

$$
\begin{array}{rcl}
|\tau| & = & ||\tau||_{\perp} \\
\\
||\mathtt{Nat}|| & = & \mathbb{Z} \\
||\mathtt{Bool}|| & = & \{\, tt, \mathit{ff} \,\} \\
||\tau_1 \rightharpoonup \tau_2|| & = & [\,||\tau_1|| \rightarrow |\tau_2|\,] \\
\\
||\Gamma|| & = & \{\, \rho \mid \forall x \in \mathrm{dom}(\Gamma)\ \rho(x) \in ||\Gamma(x)||\,\}
\end{array}
$$

It is easy to see that $||\Gamma||$ is a dcpo under the pointwise ordering of environments. We may think of the dcpo's $|\tau|$ and $||\tau||$ as providing interpretations for "computations" and "values" of type τ, respectively.

The interpretation of well-typed terms is defined by induction on typing derivations as follows.

Definition 27.2
Given a valid typing $\Gamma \vdash e : \tau$, we assign a function $|\Gamma \vdash e : \tau| : ||\Gamma|| \to |\tau|$ by induction on the structure of e as follows:

$$
\begin{aligned}
|\Gamma \vdash x : \tau|\,\rho &= \lfloor \rho(x) \rfloor \\[1ex]
|\Gamma \vdash \overline{n} : \mathtt{Int}|\,\rho &= \lfloor n \rfloor \\[1ex]
|\Gamma \vdash \mathtt{true} : \mathtt{Bool}|\,\rho &= \lfloor tt \rfloor \\
|\Gamma \vdash \mathtt{false} : \mathtt{Bool}|\,\rho &= \lfloor \mathit{ff} \rfloor \\[1ex]
|\Gamma \vdash e_1 + e_2 : \mathtt{Int}|\,\rho &= \mathit{plus}(|\Gamma \vdash e_1 : \mathtt{Int}|\,\rho, |\Gamma \vdash e_2 : \mathtt{Int}|\,\rho) \\
|\Gamma \vdash e_1 - e_2 : \mathtt{Int}|\,\rho &= \mathit{minus}(|\Gamma \vdash e_1 : \mathtt{Int}|\,\rho, |\Gamma \vdash e_2 : \mathtt{Int}|\,\rho) \\[1ex]
|\Gamma \vdash e_1 = e_2 : \mathtt{Bool}|\,\rho &= \mathit{equal}(|\Gamma \vdash e_1 : \mathtt{Int}|\,\rho, |\Gamma \vdash e_2 : \mathtt{Int}|\,\rho) \\
|\Gamma \vdash \mathtt{if}_\tau\, e\, \mathtt{then}\, e_1\, \mathtt{else}\, e_2\, \mathtt{fi} : \tau|\,\rho &= \mathit{cond}(|\Gamma \vdash e : \mathtt{Bool}|\,\rho, |\Gamma \vdash e_1 : \tau|\,\rho, |\Gamma \vdash e_2 : \tau|\,\rho) \\[1ex]
|\Gamma \vdash \mathtt{fun}\, f(x{:}\tau_1){:}\tau_2\, \mathtt{is}\, e : \tau_1 \rightharpoonup \tau_2|\,\rho &= \mathit{fix}_{||\tau_1 \rightharpoonup \tau_2||}(\Phi), \ \ \mathit{where} \\
\Phi(\psi)(u) &= |\Gamma[f{:}\tau_1 \rightharpoonup \tau_2][x{:}\tau_1] \vdash e : \tau_2|\,\rho[f \mapsto \psi][x \mapsto u]) \\[1ex]
|\Gamma \vdash \mathtt{app}_{\tau_2,\tau}(e_1,e_2) : \tau|\,\rho &= \mathit{apply}(|\Gamma \vdash e_1 : \tau_2 \rightharpoonup \tau|\,\rho, |\Gamma \vdash e_2 : \tau_2|\,\rho)
\end{aligned}
$$

where

$$
\begin{aligned}
\mathit{plus}(d_1, d_2) &= \begin{cases} \lfloor n_1 + n_2 \rfloor & \mathit{if}\ d_1 = \lfloor n_1 \rfloor, d_2 = \lfloor n_2 \rfloor \\ \bot_{|\mathtt{Int}|} & \mathit{otherwise} \end{cases} \\
\mathit{minus}(d_1, d_2) &= \begin{cases} \lfloor n_1 - n_2 \rfloor & \mathit{if}\ d_1 = \lfloor n_1 \rfloor, d_2 = \lfloor n_2 \rfloor \\ \bot_{|\mathtt{Int}|} & \mathit{otherwise} \end{cases} \\[1ex]
\mathit{equal}(d_1, d_2) &= \begin{cases} \lfloor tt \rfloor & \mathit{if}\ d_1 = \lfloor n \rfloor = d_2 \\ \lfloor \mathit{ff} \rfloor & \mathit{if}\ d_1 = \lfloor n_1 \rfloor \neq \lfloor n_2 \rfloor = d_2 \\ \bot_{|\mathtt{Int}|} & \mathit{otherwise} \end{cases} \\
\mathit{cond}(d, d_1, d_2) &= \begin{cases} d_1 & \mathit{if}\ d = \lfloor tt \rfloor \\ d_2 & \mathit{if}\ d = \lfloor \mathit{ff} \rfloor \\ \bot & \mathit{otherwise} \end{cases} \\[1ex]
\mathit{apply}(d_1, d_2) &= \begin{cases} \phi(u) & \mathit{if}\ d_1 = \lfloor \phi \rfloor,\ d_2 = \lfloor u \rfloor \\ \bot & \mathit{otherwise} \end{cases}
\end{aligned}
$$

Note that $|\Gamma \vdash v : \tau|\,\rho = \lfloor x \rfloor \neq \bot$ for syntactic values v; we write $||\Gamma \vdash v : \tau||\rho$ for x in this case.

Exercise 27.3
Extend the denotational semantics to account for the types $\mathtt{Unit}$ *and* $\tau_1 \times \tau_2$. *What is the appropriate interpretation of the product of two dcpo's?*

Lemma 27.4
If $\Gamma \vdash e : \tau$ *is a well-typed term, then* $|\Gamma \vdash e : \tau|$ *is a continuous function* $[||\Gamma|| \to |\tau|]$.

Exercise 27.5
Prove Lemma 27.4. Pay careful attention to the case of recursive functions. (There are a lot of details to work out in this exercise regarding continuity properties of the various constructs used in the semantics.)

Lemma 27.6
Suppose that $\Gamma \vdash v : \tau$ *and* $\Gamma[x{:}\tau] \vdash e' : \tau'$. *The denotational semantics is compositional in the sense that the interpretation commutes with substitution:*

$$|\Gamma \vdash \{v/x\}e' : \tau'|\rho = |\Gamma[x{:}\tau] \vdash e' : \tau'|\rho[x \mapsto ||\Gamma \vdash v : \tau||\rho].$$

Exercise 27.7
Prove Lemma 27.6.

27.2.2 Computational Adequacy

Having made a passage from an operational to a denotational semantics, it is natural to consider the relationship between the two. If we consider the value of an expression to be its "operational meaning", then we might conjecture that its denotational and operational meanings coincide. That is, we might expect that $e \Downarrow v$ iff $|e : \tau|\,\emptyset = |v : \tau|\,\emptyset$. In particular, if τ is the type `Int`, then we would have that $|e : \mathtt{Int}| = \lfloor n \rfloor$ iff $e \Downarrow \overline{n}$.

One direction is relatively straightforward: an expression has the same meaning as its value.

Lemma 27.8
If $\vdash e : \tau$ *and* $e \Downarrow v$, *then* $|e : \tau|\,\emptyset = |v : \tau|\,\emptyset$

Proof: By induction on the evaluation derivation. The result is immediate whenever e is a syntactic value. Consider the case $e = \mathtt{app}_{\tau_2,\tau}(e_1,e_2)$, where $\vdash e_1 : \tau_2 \rightharpoonup \tau$ and $\vdash e_2 : \tau_2$. Since $e \Downarrow v$, it follows that $e_1 \Downarrow v_1 = \mathtt{fun}\, f(x{:}\tau_2){:}\tau\ \mathtt{is}\ e_1'$, $e_2 \Downarrow v_2$, and $\{v_1, v_2/f, x\}e_1' \Downarrow v$. By induction hypothesis we have that $|e_1 : \tau_2 \rightharpoonup \tau|\emptyset = |v_1 : \tau_2 \rightharpoonup \tau|\emptyset$ and that $|e_2 : \tau_2|\emptyset = |v_2 : \tau_2|\emptyset$. Now $||v_1 : \tau_2 \rightharpoonup \tau|| = \phi$, where $\phi = \mathit{fix}(\Phi)$ with $\Phi(\psi)(u) = |f : \tau_2 \rightharpoonup \tau, u : \tau_2 \vdash e_1' : \tau|[f \mapsto \psi][x \mapsto u]$ for any $\psi \in ||\tau_2 \rightharpoonup \tau||$ and $u \in ||\tau_2||$. In particular, $\phi(||v_2 : \tau_2||\emptyset) = |f : \tau_2 \rightharpoonup \tau, u : \tau_2 \vdash e_1' : \tau|[f \mapsto \phi][x \mapsto ||v_2 : \tau_2||\emptyset] = |\{v_1, v_2/f, x\}e_1' : \tau|\emptyset = |v : \tau|\emptyset$. The other cases are handled similarly. ∎

In particular, if $\tau = \mathtt{Int}$ and $e \Downarrow \overline{n}$, then $|e : \mathtt{Int}|\,\emptyset = \lfloor n \rfloor$.

Exercise 27.9
Complete the proof of Lemma 27.8.

The converse of Lemma 27.8 fails because at higher types there are many distinct function expressions with the same denotation. (For example, the functions $\texttt{fun}\,(x{:}\texttt{Int})\,\texttt{in}\,x$ and $\texttt{fun}\,(x{:}\texttt{Int})\,\texttt{in}\,x+0$ are given the same interpretation by the denotational semantics, but each has only itself as value according to the operational semantics.) The best we can hope for is to obtain such a correspondence at base types. The proof is based on an interpretation of types as binary relations between the denotational and the operational semantics.

Definition 27.10
The relations $\lesssim_\tau$ between elements of $|\tau|$ and closed expressions of type τ, $\lessapprox_\tau$ between elements of $||\tau||$ and closed values of type τ, and $\lessapprox_\Gamma$ between environments and substitutions, are defined as follows

$$d \lesssim_\tau e \quad \textit{iff} \quad d = \lfloor x \rfloor \textit{ implies } e \Downarrow v \textit{ and } x \lessapprox_\tau v$$

$$\begin{array}{rcl} x \lessapprox_{\texttt{Int}} v & \textit{iff} & x = n \textit{ and } v = \overline{n} \\ x \lessapprox_{\texttt{Bool}} v & \textit{iff} & x = \textit{tt} \textit{ and } v = \texttt{true} \textit{ or } x = \textit{ff} \textit{ and } v = \texttt{false} \\ x \lessapprox_{\tau_1 \rightharpoonup \tau_2} v & \textit{iff} & x_1 \lessapprox_{\tau_1} v_1 \textit{ implies } x(x_1) \lesssim_{\tau_2} \texttt{app}_{\tau_1,\tau_2}(v,v_1) \end{array}$$

$$\rho \lessapprox_\Gamma \gamma \quad \textit{iff} \quad \rho(x) \lessapprox_{\Gamma(x)} \gamma(x)\ (\forall x \in \mathrm{dom}(\Gamma))$$

Lemma 27.11
Let $e : \tau$ be a closed term of $\mathcal{L}^{\rightharpoonup}$.

1. *Pointedness: $\perp \lesssim_\tau e$.*
2. *Downward closure: if $d \lesssim_\tau e$ and $d' \sqsubseteq d$, then $d' \lesssim_\tau e$.*
3. *Chain completeness: If $d_0 \sqsubseteq d_1 \sqsubseteq \cdots$ is a non-empty chain in $|\tau|$, and $d_i \lesssim_\tau e$ for every $i \geq 0$, then $\bigsqcup_{i\geq 0} d_i \lesssim_\tau e$.*

Proof: Pointedness follows immediately from the definitions. Downward closure and chain completeness are proved by induction on the structure of types.

Downward closure at base types follows from the fact that $|\texttt{Int}|$ and $|\texttt{Bool}|$ are "flat", together with pointedness. At higher types, suppose that $d \lesssim_{\tau_1 \rightharpoonup \tau_2} e$ and $d' \sqsubseteq d$. We are to show that $d' \lesssim_{\tau_1 \rightharpoonup \tau_2} e$. Suppose that $d' = \lfloor x' \rfloor$. Then clearly $d = \lfloor x \rfloor$ with $x' \sqsubseteq x$. But by the assumption on d it follows that $e \Downarrow v$ and $x \lessapprox_{\tau_1 \rightharpoonup \tau_2} v$. It suffices to show that $x'(x_1) \lesssim_{\tau_2} \texttt{app}_{\tau_1,\tau_2}(v,v_1)$ whenever $x_1 \lessapprox_{\tau_1} v_1$. But this follows immediately from the monotonicity of application, since $x(x_1) \lesssim_{\tau_2} \texttt{app}_{\tau_1,\tau_2}(v,v_1)$.

For chain completeness, it suffices to consider non-trivial chains in which at least one element is not $\perp$, for otherwise the supremum is $\perp$ and the result holds by pointnedness.

At base type chain completeness follows from flatness of the interpretations of Int and Bool. For if $d_i \lesssim_{\texttt{Int}} e$ for every $i \geq 0$, then there exists an $i \geq 0$ such that $d_i = \lfloor x_i \rfloor$, in which case $e \Downarrow v$ with $x_i \lessapprox_{\texttt{Int}} v$. By flatness $x_j = n$ for some $n \in \mathbb{Z}$ for every $j \geq i$, and hence the supremum of the x_i's is also n, from which the result follows. Suppose now that $\tau = \tau_1 \rightharpoonup \tau_2$.

Now there exists an $i \geq 0$ such that $d_i = \lfloor \phi_i \rfloor$ for some $\phi_i \in ||\tau_1 \rightharpoonup \tau_2||$. Consequently, $e \Downarrow v$ for some value v of type $\tau_1 \rightharpoonup \tau_2$. Moreover, for every $j \geq i$, we have that $d_j = \lfloor \phi_j \rfloor$ with $\phi_i \lessapprox_{\tau_1 \rightharpoonup \tau_2} \phi_j$. We are to show that if $d \lessapprox_{\tau_1} w$, then $(\bigsqcup_{i\geq 0} \phi_i)(d) \lesssim_{\tau_2} \texttt{app}_{\tau_1,\tau_2}(v,w)$. From the assumptions we have that $\phi_j(d) \lesssim_{\tau_2} \texttt{app}_{\tau_1,\tau_2}(v,w)$ for every $j \geq i$, and by choice of i and downward closure, this holds for every $j \geq 0$. By inductive hypothesis it follows that $\bigsqcup_{j\geq 0} \phi_j(d) \lesssim_{\tau_2} \texttt{app}_{\tau_1,\tau_2}(v,w)$. But $\bigsqcup_{j\geq 0} \phi_j(d) = (\bigsqcup_{i\geq 0} \phi_i)(d)$. This completes the proof. ■

Exercise 27.12
Complete the proof of Lemma 27.11. Extend it to unit and product types.

Exercise 27.13
Show that the approximation relation $\lesssim_\tau$ is directed complete for every type τ. That is, if $S \subseteq |\tau|$ is directed, and $d \lesssim_\tau e$ for every $d \in S$, then $\bigsqcup S \lesssim_\tau e$.

Theorem 27.14
Suppose that $\Gamma \vdash e : \tau$. If $\rho \lessapprox_\Gamma \gamma$, then $|\Gamma \vdash e : \tau|\,\rho \lesssim_\tau \hat{\gamma}(e)$.

Proof: By induction on typing derivations. The most interesting case is for recursively-defined functions. Suppose that $\rho \lessapprox_\Gamma \gamma$. We are to show that

$$|\Gamma \vdash \texttt{fun}\, f(x{:}\tau_1){:}\tau_2\; \texttt{is}\, e : \tau_1 \rightharpoonup \tau_2|\,\rho \lesssim_{\tau_1 \rightharpoonup \tau_2} \hat{\gamma}(\texttt{fun}\, f(x{:}\tau_1){:}\tau_2\; \texttt{is}\, e).$$

Let $v = \texttt{fun}\, f(x{:}\tau_1){:}\tau_2\; \texttt{is}\, \hat{\gamma}(e)$ and let $\phi = \mathit{fix}_{||\tau_1 \rightharpoonup \tau_2||}(\Phi)$, where

$$\Phi(\psi)(u) = |\Gamma[f{:}\tau_1 \rightharpoonup \tau_2][x{:}\tau_1] \vdash e : \tau_2|\,\rho[f \mapsto \psi][x \mapsto u].$$

We are to show that $\phi \lessapprox_{\tau_1 \rightharpoonup \tau_2} v$. Letting $\phi_0 = \bot$ and $\phi_{i+1} = \Phi(\phi_i)$, the sequence $\phi_0 \sqsubseteq \phi_1 \sqsubseteq \cdots$ is a non-empty chain with limit ϕ, and hence by Lemma 27.11 it suffices to show that $\phi_i \lessapprox_{\tau_1 \rightharpoonup \tau_2} v$ for every $i \geq 0$. We proceed by induction on i. The basis is immediate; we have only to note that $\bot_{||\tau_1 \rightharpoonup \tau_2||}(d_1) = \bot_{|\tau_2|}$. Assuming that $\phi_i \lessapprox_{\tau_1 \rightharpoonup \tau_2} v$, we are to show that $\phi_{i+1} \lessapprox_{\tau_1 \rightharpoonup \tau_2} v$. Suppose that $d_1 \lessapprox_{\tau_1} v_1$. By the "outer" induction hypothesis and the definition of Φ, it is sufficient to show that

$$\Phi(\phi_i)(d_1) \lesssim_{\tau_2} \widehat{\gamma[f \mapsto v][x \mapsto v_1]}(e).$$

This follows by induction on typing derivations provided that we can show

$$\rho[f \mapsto \phi_i][x \mapsto d_1] \lessapprox_{\Gamma[f:\tau_1 \rightharpoonup \tau_2][x:\tau_1]} \gamma[f \mapsto v][x \mapsto v_1].$$

But this follows from the assumptions that $\rho \lessapprox_\Gamma \gamma$, $d_1 \lessapprox_{\tau_1} v_1$, and $\phi_i \lessapprox_{\tau_1 \rightharpoonup \tau_2} v$. ■

Exercise 27.15

1. *Complete the proof of Theorem 27.14.*

2. *Extend Theorem 27.14 to the types* `Unit` *and* $\tau_1 \times \tau_2$. *What is an appropriate definition of the approximation relation for these types? What guides your choice?*

Corollary 27.16
If $e : \mathtt{Int}$ *and* $|e : \mathtt{Int}|\,\emptyset = \lfloor n \rfloor$, *then* $e \Downarrow \overline{n}$. *Equivalently,* $|e : \mathtt{Int}|\,\emptyset \neq \bot$ *iff* $e \Downarrow v$ *for some value* v.

27.2.3 Compactness, Revisited

We may derive a form of the compactness theorem from the adequacy of a denotational semantics for $\mathcal{L}^{\rightharpoonup}$ extended with labelled recursive functions. The form of compactness that we obtain in this way states that for complete programs e of observable type (say, `Int`), $e \Downarrow v$ iff $e^* \Downarrow v$ for some $e^* \in \mathit{Lab}(e)$. This form of compactness is sufficient for most purposes.

We define the interpretation of the labelled recursive functions as follows:

Definition 27.17

$$|\Gamma \vdash \mathtt{fun}^{(n)}\, f(x{:}\tau_1){:}\tau_2\, \mathtt{is}\, e : \tau_1 \rightharpoonup \tau_2|\, \rho = \lfloor\, \Phi^{(n)}(\bot)\, \rfloor,$$

where

$$\Phi(\phi)(d) = |\Gamma[f{:}\tau_1 \rightharpoonup \tau_2][x{:}\tau_1] \vdash e : \tau_2|\, \rho[f \mapsto \phi][x \mapsto d].$$

Exercise 27.18
Check that $|\Gamma \vdash e : \tau| : ||\Gamma|| \to |\tau|$.

Observe that the proof of the adequacy theorem already covers the case of bounded recursive functions under the interpretation given in Definition 27.17.

Lemma 27.19

1. *If* $k \leq l$, *then*

$$|\Gamma \vdash \mathtt{fun}^{(k)}\, f(x{:}\tau_1){:}\tau_2\, \mathtt{is}\, e : \tau_1 \rightharpoonup \tau_2|\rho \sqsubseteq |\Gamma \vdash \mathtt{fun}^{(l)}\, f(x{:}\tau_1){:}\tau_2\, \mathtt{is}\, e : \tau_1 \rightharpoonup \tau_2|\rho.$$

2. *For every* $n \geq 0$,

$$|\Gamma \vdash \mathtt{fun}^{(n)}\, f(x{:}\tau_1){:}\tau_2\, \mathtt{is}\, e : \tau_1 \rightharpoonup \tau_2|\, \rho \sqsubseteq |\Gamma \vdash \mathtt{fun}\, f(x{:}\tau_1){:}\tau_2\, \mathtt{is}\, e : \tau_1 \rightharpoonup \tau_2|\, \rho$$

Proof (sketch): The first is proved by induction on l, then second by induction on n, making use of monotonicity and the definition of the interpretation of labelled recursive functions. ■

Lemma 27.20
Suppose that $\Gamma \vdash e : \tau$. *For any* $e^* \in \mathit{Lab}(e)$,

$$|\Gamma \vdash e^* : \tau|\, \rho \sqsubseteq |\Gamma \vdash e : \tau|\, \rho.$$

Proof (sketch): By induction on the structure of e, making use of Lemma 27.19 and monotonicity. ∎

Lemma 27.21
The set $\{\, |\Gamma \vdash e^* : \tau|\,\rho \mid e^* \in \mathit{Lab}(e)\,\}$ *is directed, and has supremum*

$$\bigsqcup_{e^* \in \mathit{Lab}(e)} |\Gamma \vdash e^* : \tau|\,\rho = |\Gamma \vdash e : \tau|\,\rho.$$

Proof: For directedness observe that any two labellings of an expression e have, by Lemma 27.19, an upper bound defined by taking the maximum of labels of corresponding recursive functions. By directed completeness this set has a supremum, which by Lemma 27.19 is less than $|\Gamma \vdash e : \tau|\,\rho$.

To prove equality, we proceed by induction on typing derivations. The key step is the case of a recursive function $e = \mathtt{fun}\, f(x{:}\tau_1){:}\tau_2 \,\mathtt{is}\, e_2 : \tau_1 \rightharpoonup \tau_2$. Let $|\Gamma \vdash e : \tau_1 \rightharpoonup \tau_2|\rho = \lfloor \phi \rfloor$, where $\phi = \mathit{fix}(\Phi)$ and

$$\Phi(\psi)(u) = |\Gamma[f : \tau_1 \rightharpoonup \tau_2][x : \tau_1] \vdash e_2 : \tau_2|\rho[f \mapsto \psi][x \mapsto u].$$

Observe that any labelling e^* of e has the form $\mathtt{fun}^{(k)}\, f(x{:}\tau_1){:}\tau_2 \,\mathtt{is}\, e_2^*$ for some $k \geq 0$ and some labelling e_2^* of e_2. For each $k \geq 0$, the labelling e_k^* of e with outermost label k is assigned the interpretation $|\Gamma \vdash e_k^* : \tau_1 \rightharpoonup \tau_2|\rho = \lfloor \phi_k^* \rfloor$, where $\phi_k^* = \Phi^{*(k)}(\bot)$ and

$$\Phi^*(\psi)(u) = |\Gamma[f : \tau_1 \rightharpoonup \tau_2][x : \tau_1] \vdash e_2^* : \tau_2|\rho[f \mapsto \psi][x \mapsto u].$$

Applying the inductive hypothesis, we observe that the supremum of the set all such Φ^*'s arising from labellings e_2^* of e_2 is directed and has as supremum the function Φ given above. Consequently, the set of functions ϕ_k^* arising from each labelling is directed and has as supremum the function $\phi_k = \Phi^{(k)}(\bot)$. By Lemma 27.19 the sequence of all such ϕ_k's forms a chain. By directed completeness this sequence has a supremum, which by the Kleene fixed point theorem is the fixed point of Φ. Putting it all together, we have that

$$\bigsqcup_{e^* \in \mathit{Lab}(e)} |\Gamma \vdash e^* : \tau_1 \rightharpoonup \tau_2|\rho = \bigsqcup_{k \geq 0} \bigsqcup_{e^* \in \mathit{Lab}(e)} \phi_k^* = \bigsqcup_{k \geq 0} \phi_k = \bigsqcup_{k \geq 0} \Phi^{(k)}(\bot) = \mathit{fix}(\Phi) = \phi.$$

This completes the proof for the case of recursive functions. ∎

Exercise 27.22
Complete the proof of Lemma 27.21.

We may now give an alternate proof of compactness for complete programs. Fix a program $\vdash e : \mathtt{Int}$. Suppose that $e \Downarrow v$. By the canonical forms lemma $v = \overline{n}$ for some $n \in \mathbb{Z}$, and since $|e : \mathtt{Int}|\emptyset = |v : \mathtt{Int}|\emptyset$, it follows that $|e : \mathtt{Int}| = \lfloor n \rfloor$. By Lemma 27.21 and the fact that $|\mathtt{Int}|$ is flat, there exists

$e^* : \texttt{Int}$ such that $|e^*|\emptyset = \lfloor n \rfloor$. By adequacy $e^* \Downarrow v^*$ for some v^* such that $n \lessapprox_{\texttt{Int}} v^*$. But then $v^* = \overline{n}$ — that is, $e^* \Downarrow \overline{n}$. Conversely, suppose that $e^* \Downarrow v^*$. Since $e^* : \texttt{Int}$, it follows that $v^* = \overline{n}$ for some $n \in \mathbb{Z}$. Hence $|e^* : \texttt{Int}|\emptyset = |v^* : \texttt{Int}|\emptyset = \lfloor n \rfloor$. Now $|e^* : \texttt{Int}|\emptyset \sqsubseteq |e : \texttt{Int}|\emptyset$ by Lemma 27.21, so $|e : \texttt{Int}|\emptyset = \lfloor n \rfloor$ as well. But then by adequacy there exists v such that $e \Downarrow v$ with $n \lessapprox_{\texttt{Int}} v$, from which it follows that $v = \overline{n}$. That is, $e \Downarrow \overline{n}$.

27.3 References

Plotkin's lectures notes on denotational semantics are a superb reference [**?**]. The denotational approach used here is adapted from Winskel's book [**?**], which is itself based on Plotkin's notes.

Chapter 28

Inequational Reasoning

28.1 Introduction

In this chapter we study inequalities between expressions for the language $\mathbf{PCF}_v$ defined in Chapter 14.

We shall be concerned with families of binary relations $R = \{ R_{\Gamma,\tau} \}_{\Gamma,\tau}$ indexed by type assignments and types, where

$$R_{\Gamma,\tau} \subseteq \{ (e_1, e_2) \mid \Gamma \vdash e_i : \tau \ (i = 1, 2) \}.$$

We write $\Gamma \vdash e_1 \, R \, e_2 : \tau$ to mean that $(e_1, e_2) \in R_{\Gamma,\tau}$. When R and S are such families of relations, we write $R \subseteq S$ to mean that $R_{\Gamma,\tau} \subseteq S_{\Gamma,\tau}$ for each Γ and τ (and similarly for set equality). We say that R is a *pre-order* iff each $R_{\Gamma,\tau}$ is a pre-order (reflexive and transitive). As a notational convention, we use variants of the symbol $\lesssim$ for pre-orders on terms, and the corresponding variant of the symbol $\cong$ for the smallest equivalence relation containing the pre-order.

The *open extension* of a relation $R = \{ R_{\emptyset,\tau} \}_\tau$ between closed terms is the relation $R^\circ = \{ R^\circ_{\Gamma,\tau} \}$ on open terms defined by $\Gamma \vdash e_1 \, R^\circ \, e_2 : \tau$ iff $\hat{\gamma}(e_1) \, R_\tau \, \hat{\gamma}(e_2)$ for all substitutions $\gamma : \Gamma$ assigning a value $\gamma(x)$ of type $\Gamma(x)$ to each variable $x \in \mathrm{dom}(\Gamma)$. Since R° and R coincide on closed terms, we usually write R instead of R° to avoid heavy notation. We sometimes write $e_1 \, R \, e_2 : \tau$, or even just $e_1 \, R \, e_2$, when the context (and type) are clear from context.

28.2 Operational Orderings

The operational semantics of $\mathbf{PCF}_v$ induces a number of orderings of interest. We will consider here the *Kleene*, *contextual*, *uses of closed instances (uci)*, *applicative*, *simulation*, *logical*, and *denotational* orderings. The Kleene ordering is most useful at base type (`Unit` or `Nat` or `Bool`); an expression, e, of base type approximates another, e', iff whenever e terminates with a value, so e' must terminate with the same value. The contextual ordering is the canonical notion of equality for an operational semantics deriving from Leibniz's principle of identity of indiscernables: e approximates e' in the contextual ordering iff any observable behavior engendered by a use of e in a complete program is also engendered by the same use of e'. The uci, applicative, simulation, and logical orderings are characterizations of the contextual ordering that are more amenable to use in establishing equivalences of programs. The denotational ordering derives from an adequate denotational semantics: e and e' are denotationally equivalent iff they have the same "meaning" (denotation) in the model. For an adequate, but not fully abstract, semantics this is only a sufficient condition for contextual equivalence.

28.2.1 Kleene Ordering

Definition 28.1 (Kleene Ordering)
For closed expressions e_1 and e_2 of type τ, we define the Kleene ordering $e_1 \lesssim^{kl} e_2 : \tau$ *to hold iff $e_1 \Downarrow v$ implies $e_2 \Downarrow v$.*

The Kleene ordering is contained in the termination ordering, with the containment strict for any type with two or more values. At function types Kleene equivalence is extremely fine-grained: any two syntactically distinct function values are distinguished by the Kleene ordering. Consequently the Kleene ordering is only useful at base type.

Fix a base type o of *observations*. A *program* is a closed term of type o. Our results are largely insensitive to the choice of observables; any type whose elements are atomic (have no constituent expressions) will suffice.

Lemma 28.2
The Kleene ordering contains evaluation: if $e : o$ and $e \mapsto e'$, then $e \cong^{kl} e' : o$.

Proof: If $e \mapsto e'$, then $e' \lesssim^{kl} e : o$ since any value of e' is a value of e. Conversely $e \lesssim^{kl} e' : o$ by determinacy of evaluation: the only values of e are those of e'. ■

We write $\mathcal{E} : \tau \rightsquigarrow \tau'$ to mean that $\mathcal{E}[e] : \tau'$ whenever $\vdash e : \tau$.

Lemma 28.3
The Kleene ordering is preseved by evaluation contexts: if $e_1 \lesssim^{kl} e_2 : \tau$, then $\mathcal{E}[e_1] \lesssim^{kl} \mathcal{E}[e_2] : o$ whenever $\mathcal{E} : \tau \rightsquigarrow o$.

Definition 28.4
A pre-order $R = \{ R_{\Gamma,\tau} \}_{\Gamma,\tau}$ is consistent *iff its restriction to closed terms of observable type contains the Kleene ordering. That is, $e_1 \, R \, e_2 : o$ only if $e_1 \lesssim^{kl} e_2 : o$.*

Thus an inconsistent ordering is one that either relates a non-terminating to a terminating program, or that relates a terminating program with value v to a terminating program with a distinct value v'. Obviously the Kleene ordering is consistent by definition, and is clearly a pre-order.

28.2.2 Contextual Ordering

The Kleene ordering is not preserved by the expression constructors of the language: replacing a sub-expression by a larger expression (in the Kleene ordering) does not necessarily result in a larger expression (in the Kleene ordering). For example, $\texttt{fun}\,(x{:}\tau_1)\,\texttt{in}\,e \not\lesssim^{kl} \texttt{fun}\,(x{:}\tau_1)\,\texttt{in}\,e' : \tau_1 \rightharpoonup \tau_2$ unless e is identical to e'. But we may easily choose distinct e and e' such that $x{:}\tau_1 \vdash e \lesssim^{kl} e' : \tau_2$.

A pre-order that is preserved by all expression-forming constructs is called a *pre-congruence*. The preceding argument shows that the Kleene ordering is not a pre-congruence. To make the definition of pre-congruence precise requires the notion of a context.

Definition 28.5
A context *is an expression with zero or more "holes" in it, as defined by the following grammar:*

$$
\begin{array}{lll}
\textit{Contexts} \;\; \mathcal{C} & ::= & \bullet \mid x \mid \overline{n} \mid \mathcal{C}_1 \; op \; \mathcal{C}_2 \mid \\
 & & \texttt{true} \mid \texttt{false} \mid \mathcal{C}_1 = \mathcal{C}_2 \mid \texttt{if}_\tau \, \mathcal{C}_1 \, \texttt{then} \, \mathcal{C}_2 \, \texttt{else} \, \mathcal{C}_3 \, \texttt{fi} \mid \\
 & & * \mid \texttt{<}\mathcal{C}_1,\mathcal{C}_2\texttt{>} \mid \texttt{proj}_1(\mathcal{C}) \mid \texttt{proj}_2(\mathcal{C}) \mid \\
 & & \texttt{fun}\, f(x{:}\tau)\, \texttt{is}\, \mathcal{C} \mid \mathcal{C}_1(\mathcal{C}_2)
\end{array}
$$

We will restrict attention to *unitary* contexts, those with at most one hole. Thus any expression is (degenerately) a unitary context (with no holes). The replacement of the hole (if any) in a context $\mathcal{C}$ by an expression e is written $\mathcal{C}[e]$. Replacement can incur capture of free variables in e by binders surrounding the hole in $\mathcal{C}$. Contexts are closed under composition in that if $\mathcal{C}_1$ and $\mathcal{C}_2$ are contexts, then so is $C_1[C_2[\bullet]]$. Composition is associative, with $\bullet$ as unit element, where two contexts are deemed equal if they result in identical expressions (modulo α-conversion) when filled with the same expression.

A context $\mathcal{C}$ is said to *bind* the type assignment Γ iff the hole in $\mathcal{C}$ occurs in the scope of the declarations in Γ. We write $\mathcal{C} : \Gamma/\tau \rightsquigarrow \Gamma'/\tau'$ to mean that $\Gamma' \vdash \mathcal{C}[e] : \tau'$ whenever $\Gamma \vdash e : \tau$. In the case that Γ' is empty and $\Gamma \vdash e : \tau$, the context $\mathcal{C}$ is said to be a *closing context* for e since $\mathcal{C}[e] : \tau'$ under no typing assumptions. In this case we write $\mathcal{C} : \Gamma/\tau \rightsquigarrow \tau'$. In the case that both Γ and Γ' are empty, we write $\mathcal{C} : \tau \rightsquigarrow \tau'$, as we did for the special case of evaluation contexts (which do not bind any variables).

Definition 28.6
A pre-order R is a pre-congruence *iff it is preserved by all contexts: if $\mathcal{C} : \Gamma/\tau \rightsquigarrow \Gamma'/\tau'$, and $\Gamma \vdash e_1 \, R \, e_2 : \tau$, then $\Gamma' \vdash \mathcal{C}[e_1] \, R \, \mathcal{C}[e_2] : \tau'$. A pre-congruence R is a* congruence *if it is also an equivalence relation.*

A natural notion of equality of expressions is based on Leibniz's principle of *identity of indiscernables*: two expressions are equivalent iff we cannot tell them apart. More precisely, two expressions are deemed equivalent iff no matter how we use them in a complete program, the final result is always the same for both. A "use" of an expression is a unitary closing context of observable type.

Thus two expressions are inequivalent iff there is some closing context that distinguishes them.

Definition 28.7 (Contextual Ordering)
If $\Gamma \vdash e_1 : \tau$ and $\Gamma \vdash e_2 : \tau$, then the contextual approximation *relation $\Gamma \vdash e_1 \lesssim^{ctx} e_2 : \tau$ is defined to hold iff $\mathcal{C}[e_1] \lesssim^{kl} \mathcal{C}[e_2] : o$ for any context $\mathcal{C} : \Gamma/\tau \rightsquigarrow o$.*

Lemma 28.8
Contextual approximation is the largest consistent pre-congruence.

Proof: Contextual approximation is clearly consistent (consider the empty context for closed terms of observable type). It is a pre-congruence because contexts are closed under composition. If R is any other consistent pre-congruence such that $\Gamma \vdash e_1 \, R \, e_2 : \tau$, then $\Gamma \vdash e_1 \lesssim^{ctx} e_2 : \tau$. For if $\mathcal{C} : \Gamma/\tau \rightsquigarrow o$ is a closing context for e_1 and e_2, then $\mathcal{C}[e_1] \, R \, \mathcal{C}[e_2] : o$ since R is a pre-congruence. But then $\mathcal{C}[e_1] \lesssim^{kl} \mathcal{C}[e_2] : o$ since R is consistent. ∎

Thus to show that a pre-ordering is contained in contextual equivalence, it suffices to show that it is a consistent pre-congruence.

It is rather difficult in practice to establish contextual equivalence between terms because we must reason about all possible contexts of use of those terms. One useful technique for simplifying the required reasoning is to provide alternate characterizations of contextual equivalence that restrict the contexts that must be considered.

28.2.3 UCI Ordering

One characterization of contextual equivalence is called the *uses of closed instances (uci)* ordering.

Definition 28.9 (UCI Ordering)
If $\Gamma \vdash e_1 : \tau$ and $\Gamma \vdash e_2 : \tau$ are open terms, we define $\Gamma \vdash e_1 \lesssim^{uci} e_2 : \tau$ to hold iff for all value substitutions $\gamma : \Gamma$ and all evaluation contexts $\mathcal{E} : \tau \rightsquigarrow \tau'$, $\mathcal{E}[\hat{\gamma}(e_1)] \lesssim^{kl} \mathcal{E}[\hat{\gamma}(e_2)]$.

Remark 28.10
Mason and Talcott introduced this ordering under the name "*ciu* ordering", which appears, at first, to be a misnomer. Their terminology may be justified by observing that the uci ordering as defined above is the open extension of the "uses" ordering, the restriction of the uci ordering to closed terms. Thus the uci ordering is seen to be the "closed instances of uses" ordering. Nevertheless we prefer the name "uci" to "ciu".

Lemma 28.11
If $\Gamma \vdash e_1 \lesssim^{ctx} e_2 : \tau$, then $\Gamma \vdash e_1 \lesssim^{uci} e_2 : \tau$.

Proof: Suppose that $\Gamma \vdash e_1 \lesssim^{ctx} e_2 : \tau$ and $\gamma : \Gamma$. We are to show that $\mathcal{E}[\hat{\gamma}(e_1)] \lesssim^{kl} \mathcal{E}[\hat{\gamma}(e_2)]$ for every evaluation context $\mathcal{E} : \tau \rightsquigarrow o$. Let $\mathcal{C}$ be the context

$$\mathcal{E}[(\texttt{fun}\,(x_1{:}\tau_1)\,\texttt{in}\,\ldots\texttt{fun}\,(x_n{:}\tau_n)\,\texttt{in}\,\bullet)(v_1)\cdots(v_n)]$$

where $\mathrm{dom}(\Gamma) = \{\, x_1, \ldots, x_n \,\}$ and for each $1 \leq i \leq n$, $\Gamma(x_i) = \tau_i$ and $\gamma(x_i) = v_i$. By assumption $\mathcal{C}[e_1] \lesssim^{kl} \mathcal{C}[e_2] : o$. But $\mathcal{C}[e_i] \cong^{kl} \mathcal{E}[\hat{\gamma}(e_i)]$ for each $i = 1, 2$, from which the result follows by transitivity of the Kleene ordering. ■

The converse holds as well, but the proof is somewhat more involved.

Lemma 28.12 (Expression Constructors Preserve UCI Ordering)
Each term constructor of $\mathbf{PCF}_v$ *preserves the uci ordering. Specifically:*

1. *If* $\Gamma \vdash e_1 \lesssim^{uci} e_1' : \texttt{Nat}$, *then* $\Gamma \vdash e_1\, op\, e_2 \lesssim^{uci} e_1'\, op\, e_2 : \texttt{Nat}$ *and* $\Gamma \vdash e_1 = e_2 \lesssim^{uci} e_1' = e_2 : \texttt{Bool}$.
2. *If* $\Gamma \vdash e_2 \lesssim^{uci} e_2' : \texttt{Nat}$, *then* $\Gamma \vdash e_1\, op\, e_2 \lesssim^{uci} e_1\, op\, e_2' : \texttt{Nat}$ *and* $\Gamma \vdash e_1 = e_2 \lesssim^{uci} e_1 = e_2' : \texttt{Bool}$.
3. *If* $\Gamma \vdash e \lesssim^{uci} e' : \texttt{Bool}$, *then*
$$\Gamma \vdash \texttt{if}_\tau\, e\, \texttt{then}\, e_1\, \texttt{else}\, e_2\, \texttt{fi} \lesssim^{uci} \texttt{if}_\tau\, e'\, \texttt{then}\, e_1\, \texttt{else}\, e_2\, \texttt{fi} : \tau.$$
4. *If* $\Gamma \vdash e_1 \lesssim^{uci} e_1' : \tau$, *then*
$$\Gamma \vdash \texttt{if}_\tau\, e\, \texttt{then}\, e_1\, \texttt{else}\, e_2\, \texttt{fi} \lesssim^{uci} \texttt{if}_\tau\, e\, \texttt{then}\, e_1'\, \texttt{else}\, e_2\, \texttt{fi} : \tau.$$
5. *If* $\Gamma \vdash e_2 \lesssim^{uci} e_2' : \tau$, *then*
$$\Gamma \vdash \texttt{if}_\tau\, e\, \texttt{then}\, e_1\, \texttt{else}\, e_2\, \texttt{fi} \lesssim^{uci} \texttt{if}_\tau\, e\, \texttt{then}\, e_1\, \texttt{else}\, e_2'\, \texttt{fi} : \tau.$$
6. *If* $\Gamma \vdash e_1 \lesssim^{uci} e_1' : \tau_1$, *then* $\Gamma \vdash \texttt{<}e_1\texttt{,}e_2\texttt{>} \lesssim^{uci} \texttt{<}e_1'\texttt{,}e_2\texttt{>} : \tau_1 \times \tau_2$.
7. *If* $\Gamma \vdash e_2 \lesssim^{uci} e_2' : \tau_2$, *then* $\Gamma \vdash \texttt{<}e_1\texttt{,}e_2\texttt{>} \lesssim^{uci} \texttt{<}e_1\texttt{,}e_2'\texttt{>} : \tau_1 \times \tau_2$.
8. *If* $\Gamma \vdash e \lesssim^{uci} e' : \tau_1 \times \tau_2$, *then* $\Gamma \vdash \texttt{proj}_i(e) \lesssim^{uci} \texttt{proj}_i(e') : \tau_i$ *(*$i = 1, 2$*).*
9. *If* $\Gamma \vdash e_1 \lesssim^{uci} e_1' : \tau_2 \rightharpoonup \tau$, *then* $\Gamma \vdash e_1(e_2) \lesssim^{uci} e_1'(e_2) : \tau$.
10. *If* $\Gamma \vdash e_2 \lesssim^{uci} e_2' : \tau_2$, *then* $\Gamma \vdash e_1(e_2) \lesssim^{uci} e_1(e_2') : \tau$.
11. *If* $\Gamma[x{:}\tau_1] \vdash e \lesssim^{uci} e' : \tau_2$, *then* $\Gamma \vdash \texttt{fun}\, f(x{:}\tau_1)\, \texttt{is}\, e \lesssim^{uci} \texttt{fun}\, f(x{:}\tau_1)\, \texttt{is}\, e'$.

Proof: Compatibility of the uci ordering with most of the eliminatory forms is obvious: if $\mathcal{E}$ is an evaluation context, then so are $\mathcal{E}[\bullet\ op\ e]$, $\mathcal{E}[v\ op\ \bullet]$, $\mathcal{E}[\bullet = e]$, $\mathcal{E}[v = \bullet]$, $\mathcal{E}[\texttt{proj}_i(\,\bullet\,)]$, $\mathcal{E}[\bullet(e)]$, and $\mathcal{E}[v(\,\bullet\,)]$. In the case of the second argument of an arithmetic operation, if $\mathcal{E}[e_1\ op\ e_2] \Downarrow a$, then there exists a value v_1 such that $\mathcal{E}[e_1\ op\ e_2] \mapsto^* \mathcal{E}[v_1\ op\ e_2]$.

For the conditional expression, ...[1]

For ordered pairs, we consider the left- and right-hand positions separately. It suffices to consider closed terms. Suppose $e_1 \lesssim^{uci} e_1' : \tau_1$ and $\mathcal{E}'[e_1] \Downarrow a$ for some $a : o$, where $\mathcal{E}' = \mathcal{E}[<\bullet, e_2>]$. Since $\mathcal{E}'$ is an evaluation context, it follows that $\mathcal{E}'[e_1'] \Downarrow a$. Now suppose $e_2 \lesssim^{uci} e_2' : \tau_2$ and $\mathcal{E}'[e_2] \Downarrow a$ for some $a : o$, where $\mathcal{E}' = \mathcal{E}[<e_1, \bullet>]$ and e_1 is a value. The result then follows directly from the assumptions. If e_1 is a not a value and $\mathcal{E}[<e_1,e_2>] \Downarrow a$, then there must exist v_1 such that $\mathcal{E}[<e_1,e_2>] \mapsto^* \mathcal{E}[<v_1,e_2>] \Downarrow a$. Taking $\mathcal{E}' = \mathcal{E}[<v_1, \bullet>]$, we have that $\mathcal{E}'[e_2] \Downarrow a$, and hence $\mathcal{E}'[e_2'] \Downarrow a$, from which the result follows.

For functions, we must significantly strengthen the inductive hypothesis to account for the substitution of arguments for parameters at function applications. Let $u = \texttt{fun}\, f(x{:}\tau_1)\, \texttt{is}\, e$ and $u' = \texttt{fun}\, f(x{:}\tau_1)\, \texttt{is}\, e'$. We prove for every expression e_0 such that $z{:}\tau \vdash e : o$ that $\{u/z\}e_0 \lesssim^{kl} \{u'/z\}e_0 : o$, given that $[f{:}\tau_1 \rightharpoonup \tau_2, x{:}\tau_1] \vdash e \lesssim^{uci} e' : \tau_2$. The result follows by taking $e_0 = \mathcal{E}[z]$ so that $\mathcal{E}[e] \lesssim^{kl} \mathcal{E}[e'] : o$, and hence $e \lesssim^{uci} e' : \tau_1 \rightharpoonup \tau_2$.

The proof proceeds by induction on the length of the evaluation sequence $\{u/z\}e_0 \Downarrow a$. If the first step is independent of the value substituted for z, then the result follows immediately from the inductive hypothesis. In particular, this covers the case $e_0 = \mathcal{E}[v(z)]$ (for some value $v = \texttt{fun}\, f(x{:}\tau)\, \texttt{is}\, e$). For then $\{u/z\}e_0 \mapsto \{u/z\}\mathcal{E}[\{v, z/f, x\}e] \Downarrow a$, hence by induction $\{u'/z\}\mathcal{E}[\{v, z/f, x\}e] \Downarrow a$ and therefore by assumption $\{u'/z\}\mathcal{E}[\{v, z/f, x\}e'] \Downarrow a$ from which it follows that $\{u'/z\}\mathcal{E}[v(z)] \Downarrow a$.

We are left with the case $e_0 = \mathcal{E}[z(v)]$. Obviously,

$$\{u/z\}e_0 \mapsto (\{u/z\}\mathcal{E})[\{u, \{u/z\}v/f, x\}e] \Downarrow a.$$

That is, $\{u/z\}\mathcal{E}[\{z, v/f, x\}e] \Downarrow a$ by a shorter reduction sequence, hence by induction we have that $\{u'/z\}\mathcal{E}[\{z, v/f, x\}e] \Downarrow a$. By the assumption governing e and e', it follows that $\{u'/z\}\mathcal{E}[\{z, v/f, x\}e'] \Downarrow a$, and hence $\{u'/z\}\mathcal{E}[z(v)] \Downarrow a$. ∎

Lemma 28.13
The UCI ordering is a consistent pre-congruence.

Proof: Consistency is proved by considering the empty evaluation context. Congruence follows from the preceding lemma by induction on the structure of contexts. ∎

Corollary 28.14
The UCI ordering is contained in the contextual ordering: if $\Gamma \vdash e_1 \lesssim^{uci} e_2 : \tau$, *then* $\Gamma \vdash e_1 \lesssim^{ctx} e_2 : \tau$.

Proof: The contextual ordering is the largest consistent congruence. ∎

[1] Complete this argument.

Theorem 28.15 (Equivalence of UCI and Contextual Orderings)
$\Gamma \vdash e_1 \lesssim^{uci} e_2 : \tau$ *iff* $\Gamma \vdash e_1 \lesssim^{ctx} e_2 : \tau$.

With this in hand we may establish some useful properties of contextual equivalence.

Lemma 28.16
If $\Gamma \vdash e_1 \lesssim^{kl} e_2 : \tau$, *then* $\Gamma \vdash e_1 \lesssim^{uci} e_2 : \tau$.

Proof: Suppose that $\mathcal{E} : \tau \leadsto \tau'$ is an evaluation context and $\gamma : \Gamma$ is a value substitution. We are to show that $\mathcal{E}[\hat{\gamma}(e_1)] \lesssim^{kl} \mathcal{E}[\hat{\gamma}(e_2)]$. By definition of the open extension of the Kleene ordering we have $\hat{\gamma}(e_1) \lesssim^{kl} \hat{\gamma}(e_2)$. It follows that $\mathcal{E}[\hat{\gamma}(e_1)] \lesssim^{kl} \mathcal{E}[\hat{\gamma}(e_2)]$, as required. ∎

Corollary 28.17
If $\Gamma \vdash e_1 \lesssim^{kl} e_2 : \tau$, *then* $\Gamma \vdash e_1 \lesssim^{ctx} e_2 : \tau$.

Lemma 28.18 (Stability Under Substitution)
If $\Gamma \vdash e_1 \lesssim^{ctx} e_2 : \tau$ *and* $\gamma : \Gamma$ *is a value substitution, then* $\hat{\gamma}(e_1) \lesssim^{ctx} \hat{\gamma}(e_2) : \tau$.

Proof: This is easily seen to be true of the uci ordering, which coincides with the contextual ordering. ∎

28.2.4 Applicative Ordering

Another characterization of the contextual ordering limits the observations even further. Let us first define the set of *atomic tests* as follows:

$$\mathcal{T} \quad ::= \quad \bullet \mid \bullet \; op \; e \mid v \; op \bullet \mid \bullet = e \mid v = \bullet \mid \texttt{if}_\tau \; \bullet \; \texttt{then} \, e_1 \, \texttt{else} \, e_2 \, \texttt{fi} \mid \bullet(e)$$

An *applicative evaluation context* is a composition of atomic tests yielding a value of observable type, *i.e.*, a context of the form $\mathcal{T}_1[\cdots \mathcal{T}_n[\bullet]]$ of type $\tau \leadsto o$. Let $\mathcal{A}$ range over the set of applicative evaluation contexts.

Definition 28.19 (Applicative Ordering)
The applicative ordering $e_1 \lesssim^{app} e_2 : \tau$ *between closed terms* e_1 *and* e_2 *of type* τ *holds iff for every applicative evaluation context* $\mathcal{A} : \tau \leadsto o$, $\mathcal{A}[e_1] \lesssim^{kl} \mathcal{A}[e_2]$.

Lemma 28.20
If $\Gamma \vdash e_1 \lesssim^{uci} e_2 : \tau$, *then* $\Gamma \vdash e_1 \lesssim^{app} e_2 : \tau$.

Proof: Every applicative evaluation context is an evaluation context. ∎

An atomic test $\mathcal{T} : \tau \leadsto \tau'$ is *non-trivial* iff $\mathcal{T} = \bullet$ implies $\tau = \tau' = o$. That is, a non-trivial context may only be empty at observable type.

Lemma 28.21
If $\mathcal{E}[\mathcal{T}[v_1]] \lesssim^{kl} \mathcal{E}[\mathcal{T}[v_2]] : o$ for every $\mathcal{E} : \tau' \rightsquigarrow o$ and every non-trivial atomic test $\mathcal{T} : \tau \rightsquigarrow \tau'$, then for every $x{:}\tau \vdash \mathcal{E} : \tau' \rightsquigarrow o$ and every $x{:}\tau \vdash e : \tau'$, $\{v_1/x\}\mathcal{E}[e] \lesssim^{kl} \{v_2/x\}\mathcal{E}[e]$.

Proof: By induction on the length of the evaluation sequence $\{v_1/x\}\mathcal{E}[e] \Downarrow v$. ■

Corollary 28.22
If $\mathcal{E}[\mathcal{T}[v_1]] \lesssim^{kl} \mathcal{E}[\mathcal{T}[v_2]] : o$ for every $\mathcal{E} : \tau' \rightsquigarrow o$ and every non-trivial atomic test $\mathcal{T} : \tau \rightsquigarrow \tau'$, then $v_1 \lesssim^{uci} v_2 : \tau$

Proof: Take $e = x$ in the preceding lemma. ■

Corollary 28.23
If $\Gamma \vdash e_1 \lesssim^{app} e_2 : \tau$, then $\Gamma \vdash e_1 \lesssim^{uci} e_2 : \tau$.

Proof: Suppose that $\mathcal{A}[e_1] \lesssim^{kl} \mathcal{A}[e_2] : o$ for every $\mathcal{A} : \tau \rightsquigarrow o$. We are to show that $\mathcal{E}[e_1] \lesssim^{kl} \mathcal{E}[e_2] : o$ for all $\mathcal{E} : \tau \rightsquigarrow o$. By repeated application of the preceding corollary, it is sufficient to show that $\mathcal{E}[\mathcal{A}_0[e_1]] \lesssim^{kl} \mathcal{E}[\mathcal{A}_0[e_2]] : o$ for some applicative context $\mathcal{A}_0 : \tau \rightsquigarrow o$ and all evaluation contexts $\mathcal{E} : o \rightsquigarrow o$.[2] But this follows from the fact that $\mathcal{A}_0[e_1] \lesssim^{kl} \mathcal{A}_0[e_2] : o$, which follows from the assumption that $e_1 \lesssim^{app} e_2 : \tau$. ■

Theorem 28.24 (Coincidence of UCI and Applicative Orderings)
The applicative and uci orderings coincide: $\Gamma \vdash e_1 \lesssim^{uci} e_2 : \tau$ iff $\Gamma \vdash e_1 \lesssim^{app} e_2 : \tau$.

Corollary 28.25
The applicative ordering is a pre-congruence.

Proof: By the theorem the applicative and uci orderings coincide with the contextual ordering, which is obviously a pre-congruence. ■

Lemma 28.26 (Extensionality of Functions)
$\Gamma \vdash \texttt{fun}\, f(x{:}\tau_1)\, \texttt{is}\, e \lesssim^{ctx} \texttt{fun}\, f(x{:}\tau_1)\, \texttt{is}\, e' : \tau_1 \rightharpoonup \tau_2$ iff $\Gamma \vdash (\texttt{fun}\, f(x{:}\tau_1)\, \texttt{is}\, e)(v) \lesssim^{ctx} (\texttt{fun}\, f(x{:}\tau_1)\, \texttt{is}\, e')(v) : \tau_1 \rightharpoonup \tau_2$ for all $\Gamma \vdash v : \tau_1$.

Proof: This is easily seen to hold for the applicative ordering, which coincides with the uci and contextual orderings. ■

[2] Clarify this argument.

28.2.5 Simulation Ordering

A third characterization of contextual equivalence interleaves evaluation with recursive descent through sub-expressions of values.

Definition 28.27 (Simulation)
A simulation *between closed terms is a type-indexed family of binary relations R such that if $e_1 \, R \, e_2 : \tau$, then*

1. *if $\tau = \texttt{Int}$ and $e_1 \Downarrow \overline{n}$, then $e_2 \Downarrow \overline{n}$.*
2. *if $\tau = \texttt{Bool}$ and $e_1 \Downarrow \texttt{true}$ ($e_1 \Downarrow \texttt{false}$), then $e_2 \Downarrow \texttt{true}$ ($e_2 \Downarrow \texttt{false}$).*
3. *if $\tau = \tau' \times \tau''$ and $e_1 \Downarrow \texttt{<}e_1',e_1''\texttt{>}$, then $e_2 \Downarrow \texttt{<}e_2',e_2''\texttt{>}$ and $e_1' \, R \, e_2' : \tau'$ and $e_1'' \, R \, e_2'' : \tau''$.*
4. *if $\tau = \tau' \rightharpoonup \tau''$ and $e_1 \Downarrow \texttt{fun}\, f(x{:}\tau') \,\texttt{is}\, e$, then $e_2 \Downarrow \texttt{fun}\, f(x{:}\tau') \,\texttt{is}\, e'$ and $(\texttt{fun}\, f(x{:}\tau') \,\texttt{is}\, e)(v) \, R \, (\texttt{fun}\, f(x{:}\tau') \,\texttt{is}\, e')(v) : \tau''$ for every closed value v of type τ.*

Definition 28.28 (Similarity Orderings)
For closed terms e_1 and e_2 of type τ, we define the similarity ordering, *$e_1 \lesssim^{sim} e_2 : \tau$, to be the largest simulation between closed terms.*[3]

Lemma 28.29
The similarity ordering is consistent: if $e_1 \lesssim^{sim} e_2 : o$, then $e_1 \lesssim^{kl} e_2 : o$.

Proof: Follows immediately from the definition, given that observations are at base type. ∎

Lemma 28.30
If $\Gamma \vdash e_1 \lesssim^{sim} e_2 : \tau$ and $\mathcal{A} : \tau \rightsquigarrow o$, then $\mathcal{A}[e_1] \lesssim^{sim} \mathcal{A}[e_2] : o$.

Proof: By induction on the structure of $\mathcal{A}$.[4] ∎

Corollary 28.31
If $\Gamma \vdash e_1 \lesssim^{sim} e_2 : \tau$, then $\Gamma \vdash e_1 \lesssim^{app} e_2 : \tau$.

Proof: Follows immediately from the preceding lemma and the consistency of the simulation ordering. ∎

Lemma 28.32
The applicative ordering is a simulation. Therefore, if $\Gamma \vdash e_1 \lesssim^{app} e_2 : \tau$, then $\Gamma \vdash e_1 \lesssim^{sim} e_2 : \tau$.

[3] In fact the least and greatest orderings coincide, as may be established by induction on types. The two fixed points separate only in the presence of a lazy recursive type such as streams.

[4] Check this proof.

Proof: [5] It suffices to consider closed terms. At base type b we observe that if $e_1 \lesssim^{app} e_2 : b$, then $e_1 \lesssim^{kl} e_2 : b$, as required for a simulation. At function type $\tau = \tau_1 \rightharpoonup \tau_2$, we observe that if $e_1 \lesssim^{app} e_2 : \tau$, and $e_1 \Downarrow v_1$, then $v_1 = \texttt{fun}\, f(x{:}\tau_1)\, \texttt{is}\, e_1'$ and $e_2 \Downarrow v_2$ with $v_2 = \texttt{fun}\, f(x{:}\tau_1)\, \texttt{is}\, e_2'$ because applicative evaluation contexts are "strict". But then $v_1(v) \lesssim^{app} v_2(v) : \tau_2$ for every $v : \tau_1$, since $e_1(v) \lesssim^{app} e_2(v) : \tau_2$, once again relying on "strictness" of applicative evaluation contexts. ■

Theorem 28.33
$\lesssim^{kl} \subsetneq \lesssim^{sim} = \lesssim^{app} = \lesssim^{uci} = \lesssim^{ctx}$

28.2.6 Logical Ordering

The final characterization of the contextual ordering that we will consider is based on the method of *logical relations.*

Definition 28.34
The logical *ordering $e_1 \lesssim^{log} e_2 : \tau$ between closed terms e_1 and e_2 of type τ is defined to hold iff whenever $e_1 \Downarrow v_1$, then $e_2 \Downarrow v_2$, and $v_1 \lessapprox^{log} v_2 : \tau$. The logical ordering $v_1 \lessapprox^{log} v_2 : \tau$ between closed values of type τ is defined by induction on the structure of τ as follows:*

$$
\begin{array}{rcl}
v_1 \lessapprox^{log} v_2 : \texttt{Int} & \textit{iff} & v_1 = v_2 = \overline{n} \\
v_1 \lessapprox^{log} v_2 : \texttt{Bool} & \textit{iff} & v_1 = v_2 = \texttt{true} \textit{ or } v_1 = v_2 = \texttt{false} \\
v_1 \lessapprox^{log} v_2 : \tau' \times \tau'' & \textit{iff} & \texttt{proj}_1(v_1) \lesssim^{log} \texttt{proj}_2(v_2) : \tau' \textit{ and } \texttt{proj}_2(v_1) \lesssim^{log} \texttt{proj}_2(v_2) : \tau'' \\
v_1 \lessapprox^{log} v_2 : \tau' \rightharpoonup \tau'' & \textit{iff} & \textit{if } v_1' \lessapprox^{log} v_2' : \tau',\ \textit{then } v_1(v_1') \lesssim^{log} v_2(v_2') : \tau''
\end{array}
$$

The logical ordering is extended to open expressions by defining $\Gamma \vdash e_1 \lesssim^{log} e_2 : \tau$ to hold iff $\hat{\gamma}_1(e_1) \lesssim^{log} \hat{\gamma}_2(e_2) : \tau$ whenever $\gamma_1 \lessapprox^{log} \gamma_2 : \Gamma$.

Notice that we do *not* define the logical ordering on open terms as the open extension of the logical ordering on closed terms!

Lemma 28.35
The logical ordering is closed under pre- and post-composition with the applicative ordering. That is, if $\Gamma \vdash e_1 \lesssim^{log} e_2 : \tau$, $\Gamma \vdash e_1' \lesssim^{app} e_1 : \tau$, and $\Gamma \vdash e_2 \lesssim^{app} e_2' : \tau$, then $\Gamma \vdash e_1' \lesssim^{log} e_2' : \tau$.

Proof: We first prove the result for closed terms by induction on the structure of types. The conditions on the simulation ordering determine applicative evaluation contexts that drive the expressions to type o, where the ordering coincides with the Kleene ordering. That is, if $e_1 \lesssim^{log} e_2 : \tau$, then there exists an

[5] Check this proof.

applicative evaluation context A_τ such that $\mathcal{A}_\tau[e_1] \lesssim^{kl} \mathcal{A}_\tau[e_2] : o$, from which the result follows. The extension to open terms is achieved by appeal to the result for closed terms.[6] ■

Lemma 28.36
The logical ordering is consistent and transitive.

Lemma 28.37
The logical ordering is admissible.[7]

Lemma 28.38
The logical ordering is preserved by each of the expression-forming constructors.

Proof: By case analysis on the possible constructors. For the case of recursive functions we rely on admissibility of the logical ordering. ■

Lemma 28.39
The logical ordering is a pre-congruence: if $\Gamma \vdash e_1 \lesssim^{log} e_2 : \tau$ *and* $\Gamma \vdash \mathcal{C} : \Gamma/\tau \rightsquigarrow \Gamma'/\tau'$*, then* $\Gamma' \vdash \mathcal{C}[e_1] \lesssim^{log} \mathcal{C}[e_2] : \tau'$.

Proof: We proceed by induction on the structure of contexts, appealing to the preceding lemma for each case of the induction. ■

Lemma 28.40
The logical ordering is reflexive: if $\Gamma \vdash e : \tau$*, then* $\Gamma \vdash e \lesssim^{log} e : \tau$.

Proof: By the preceding lemma, taking e to be a context with no holes. ■

It follows that the logical ordering contains the applicative ordering since the logical ordering is reflexive and closed under pre-composition with the applicative ordering. Since the logical ordering is a consistent pre-congruence, it is contained in the contextual ordering. Thus we have the following theorem.

Lemma 28.41
$\lesssim^{app} \subseteq \lesssim^{log} \subseteq \lesssim^{ctx}$

Since the contextual ordering contains the applicative ordering, we obtain the following result:

Theorem 28.42 (Relation Between Orderings)
$\lesssim^{kl} \subsetneq \lesssim^{app} = \lesssim^{uci} = \lesssim^{log} = \lesssim^{ctx}$

[6] Not very clear.
[7] Define admissibility.

28.3 Denotational Ordering

The denotational semantics given in Chapter 27 induces a consistent pre-congruence between terms defined as follows.

Definition 28.43 (Denotational Ordering)
We define the denotational ordering $\Gamma \vdash e_1 \lesssim^{den} e_2 : \tau$ *between open terms to hold iff* $||\Gamma \vdash e_1 : \tau||\, \rho \sqsubseteq ||\Gamma \vdash e_2 : \tau||\, \rho$ *for every* $\rho \in ||\Gamma||$.

Lemma 28.44
The denotational ordering is a consistent pre-congruence.

Proof: Consistency of the denotational ordering follows directly from adequacy: if $e_1 \lesssim^{den} e_2 : o$ and $e_1 \Downarrow v$, then $||e_1 : \tau||\emptyset = ||v : \tau||\emptyset = ||e_2 : \tau||\emptyset$, so by adequacy $e_2 \Downarrow v$, and conversely. The denotational ordering is a pre-congruence because it is defined by induction on the structure of expressions. ■

Corollary 28.45
The denotational ordering is contained in the contextual ordering.

The denotational ordering does not, however, *coincide* with the contextual ordering. Plotkin gave two closed terms that are distinguished in the denotational ordering, but that are contextually equivalent. In other words the denotational semantics is not fully abstract.

28.4 References

Most of the development is adapted directly from work of Mason, Talcott, and Smith, Pitts, and Winskel.

Part V

Background

Appendix A

Inductive Definitions

A.1 Introduction

Inductive definitions of sets and relations are a fundamental technique in the study of programming languages.

A.2 Inductive and Coinductive Definitions

We will consider a set-theoretic, rather than purely foundational, account of inductive and co-inductive definitions.

Fix a set $\mathcal{U}$ to serve as the *universe of discourse*. We will consider inductive and coinductive subsets of the universe $\mathcal{U}$. These will arise as fixed points of certain operators on the powerset, $\mathcal{P}(\mathcal{U})$, of $\mathcal{U}$.

A function $\Phi : \mathcal{P}(\mathcal{U}) \to \mathcal{P}(\mathcal{U})$ is *monotone* iff it preserves set inclusion: $\Phi(X) \subseteq \Phi(Y)$ whenever $X \subseteq Y$. For a monotone operator Φ on $\mathcal{P}(\mathcal{U})$, define the sets $\mu(\Phi) \subseteq \mathcal{U}$ and $\nu(\Phi) \subseteq \mathcal{U}$ by the following equations

$$\mu(\Phi) = \bigcap \{ X \subseteq \mathcal{U} \mid \Phi(X) \subseteq X \}$$

and

$$\nu(\Phi) = \bigcup \{ X \subseteq \mathcal{U} \mid X \subseteq \Phi(X) \}.$$

A subset $X \subseteq \mathcal{U}$ such that $\Phi(X) \subseteq X$ is said to be Φ-*closed*; a subset $X \subseteq \mathcal{U}$ such that $X \subseteq \Phi(X)$ is said to be Φ-*consistent*.

Lemma A.1
Let $\Phi : \mathcal{P}(\mathcal{U}) \to \mathcal{P}(\mathcal{U})$ be monotone. The set $\mu(\Phi)$ is the smallest Φ-closed subset of $\mathcal{U}$, and the set $\nu(\Phi)$ is the largest Φ-consistent subset of $\mathcal{U}$. That is, $\Phi(\mu(\Phi)) \subseteq \mu(\Phi)$ and $\mu(\Phi) \subseteq X$ whenever $\Phi(X) \subseteq X$, and, dually, $\nu(\Phi) \subseteq \Phi(\nu(\Phi))$ and $X \subseteq \nu(\Phi)$ whenever $X \subseteq \Phi(X)$.

Proof: It is obvious that $\mu(\Phi)$ is a subset of every Φ-closed set X since it is, by definition, the intersection of all such sets. Thus we need only show that $\mu(\Phi)$ is Φ-closed: $\Phi(\mu(\Phi)) \subseteq \mu(\Phi)$. It suffices to show that $\Phi(\mu(\Phi)) \subseteq X$ for every Φ-closed subset $X \subseteq \mathcal{U}$. Since $\mu(\Phi) \subseteq X$, we have by monotonicity of Φ that $\Phi(\mu(\Phi)) \subseteq \Phi(X)$. But $\Phi(X) \subseteq X$ since X is Φ-closed. Dually, it is obvious that $\nu(\Phi)$ contains every Φ-consistent set; we need only show that $\nu(\Phi)$ is itself Φ-consistent. It suffices to show that if $X \subset \Phi(X)$, then $X \subset \Phi(\nu(\Phi))$. Since $X \subseteq \nu(\Phi)$, it follows that $\Phi(X) \subseteq \Phi(\nu(\Phi))$ by monotonicity. But then $X \subseteq \Phi(\nu(\Phi))$, as required. ■

Theorem A.2 (Tarski)
The sets $\mu(\Phi)$ and $\nu(\Phi)$ are, respectively, the least and greatest fixed points of Φ: $\Phi(\mu(\Phi)) = \mu(\Phi)$ and $\Phi(\nu(\Phi)) = \nu(\Phi)$.

Proof: We have already shown that $\Phi(\mu(\Phi)) \subseteq \mu(\Phi)$. It remains only to show the opposite containment. By monotonicity we have that $\Phi(\Phi(\mu(\Phi))) \subseteq \Phi(\mu(\Phi))$, which means that $\Phi(\mu(\Phi))$ is Φ-closed. But then $\mu(\Phi) \subseteq \Phi(\mu(\Phi))$ by Lemma **??**. Dually, we have shown that $\nu(\Phi) \subseteq \Phi(\nu(\Phi))$. Therefore $\Phi(\nu(\Phi)) \subseteq \Phi(\Phi(\nu(\Phi))$, which means that $\Phi(\nu(\Phi))$ is Φ-consistent. By Lemma **??** it follows that $\Phi(\nu(\Phi)) \subseteq \nu(\Phi)$. ■

The sets $\mu(\Phi)$ and $\nu(\Phi)$ are called the sets *inductively-defined*, respectively *co-inductively-defined*, by the operator Φ. This terminology is justified by the following observation. Think of a property P of a set X as a subset of it: an element $x \in X$ has property P iff $x \in P$. With this in mind Lemma **??** tells us that every element $x \in \mu(\Phi)$ has property P iff for every $y \in \Phi(P)$ we have that $y \in P$. Dually, we may show that $x \in \nu(\Phi)$ by showing that $x \in P$ for some P such that for every $y \in P$, we have $y \in \Phi(P)$.

Some concrete examples will help guide your intuition. Let $0 = \emptyset$ and let $succ(x) = x \cup \{ x \}$. Fix a suitably large universe $\mathcal{U}$ (namely one containing 0 and closed under $succ$). Define the operator Φ as follows:

$$\Phi(X) = \{ 0 \} \cup \{ succ(x) \mid x \in X \}$$

In more concise notation we might write

$$\Phi(X) = \mathbf{0} \cup \mathbf{succ}(X)$$

where $\mathbf{0} = \{ 0 \}$ and $\mathbf{succ}(X) = \{ succ(x) \mid x \in X \}$. This is clearly a monotone operator on $\mathcal{P}(\mathcal{U})$. Let $N = \mu(\Phi)$, the smallest Φ-closed subset of $\mathcal{U}$. Expanding the definition of Φ-closure, this means that N is the smallest set X such that $0 \in X$ and if $x \in X$, then $succ(x) \in X$. Thus to prove that every element of N satisfies a property P, it suffices to show that $0 \in P$ and that $x \in P$ implies $succ(x) \in P$. This is the familiar notion of mathematical induction for the natural numbers.

For an example of a coinductively-defined set, let $\Phi(X) = N \times X$, the Cartesian product of N and X. This is clearly a monotone operator on $\mathcal{P}(\mathcal{U})$

for a suitable choice of $\mathcal{U}$.[1] Let $S = \nu(\Phi)$ be the largest Φ-consistent set. To show that $x \in S$, it suffices to show that $x \in P$ for some Φ-consistent property P. Unravelling the definition, we must show that $x \in P$ for some P such that if $y \in P$, then $\pi_1(y) \in N$ and $\pi_2(y) \in P$. For example, consider the set x such that $x = \langle 0, x \rangle$.[2] We claim that $x \in S$. To see this we take P to be the singleton set $\{ x \}$ (so that $x \in P$ is obvious). We must show that P is Φ-consistent. It suffices to show that $0 \in N$ (true by definition of N) and that $x \in P$ (true by definition of P), since $\pi_1(x) = 0$ and $\pi_2(x) = x$. Consequently, $P \subseteq S$, which is to say that $x \in S$.

A.3 Rule Sets

The example operators in the previous section were contrived to make a point. How do such operators arise in practice? One important source is through the notion of a *rule set*. A *rule* (over a universe $\mathcal{U}$) is a pair $X \to x$, where $X \subseteq \mathcal{U}$ and $x \in \mathcal{U}$. The set X is called the set of *premises* of the rule, and the element x is called its *consequence*. A rule with an empty set of premises is sometimes called an *axiom*. We often write $x_1 \ldots, x_n \to x$ for the rule $\{ x_1 \ldots, x_n \} \to x$. A *rule set* (over $\mathcal{U}$) is, obviously, a set of rules over $\mathcal{U}$. By a slight abuse of notation, we write $R \cup X$, where $X \subseteq \mathcal{U}$, for the rule set $R \cup \{ \to x \mid x \in X \}$.

Every rule set R over a universe $\mathcal{U}$ determines a monotone operator Φ_R on $\mathcal{P}(\mathcal{U})$ defined as follows:

$$\Phi_R(X) = \{ y \mid \exists Y \to y \in R \; Y \subseteq X \}$$

Intuitively, $\Phi_R(X)$ consists of all of the consequences of rules in R whose premises are in the set X. Thus Φ_R closes up a set X under one step of deduction using the rules in R. It should be evident that Φ_R is monotone.

The operators determining the sets N and S in the preceding section may be thought of as arising from the following rule sets:

$$R_N = \{ \to 0 \} \cup \{ x \to \mathit{succ}(x) \mid x \in \mathcal{U} \}$$

and

$$R_S = \{ x \to \langle n, x \rangle \mid n \in N \wedge x \in \mathcal{U} \}.$$

You may be accustomed to thinking of sets inductively defined by a set of rules as being generated from axioms by repeated applications of the inference rules. This characterization is not immediately apparent from the definition of $\mu(\Phi_R)$ given in the preceding section. Dually, the set co-inductively defined by a set of rules may be thought of as arising by starting with $\mathcal{U}$ and throwing out elements that cannot be "justified" as arising from an application of some rule. Once again, this characterization is not immediately obvious from the definition of $\nu(\Phi_R)$ given in the previous section.

[1] It may not be so obvious that there is a choice of $\mathcal{U}$; we blithely assume that there is.

[2] It's a big step to assume that such an x exists; non-well-founded sets are the key [?].

To state the relevant characterization we must introduce the *finite iterates* of an operator, defined as follows:

$$\begin{array}{rcl}\Phi^{(0)}(X) & = & \emptyset \\ \Phi^{(n+1)}(X) & = & \Phi(\Phi^{(n)}(X))\end{array}$$

A rule set R is *finitary* iff every rule in R has only finitely many premises: if $X \to x \in R$, then X is finite. Finitary rule sets induce monotone operators whose fixed points can be achieved by iteration.

Theorem A.3
Let R be a finitary rule set and let $\Phi = \Phi_R$ be the operator induced by R. Then

$$\mu(\Phi) = \bigcup_{n<\omega} \Phi^{(n)}(\emptyset)$$

and

$$\nu(\Phi) = \bigcap_{n<\omega} \Phi^{(n)}(\mathcal{U}).$$

Proof: Let $X_i = \Phi^{(i)}(\emptyset)$ and let $X = \bigcup_{i<\omega} X_i$. Then

$$\begin{array}{rcl}\Phi(X) & = & \{\, y \mid \exists Y \to y \in R \; Y \subseteq X \,\} \\ & = & \{\, y \mid \exists i < \omega \, \exists Y \to y \in R \; Y \subseteq X_i \,\} \\ & = & \bigcup_{i<\omega} \{\, y \mid \exists Y \to y \in R \; Y \subseteq X_i \,\} \\ & = & \bigcup_{i<\omega} \Phi(X_i) \\ & = & \bigcup_{i<\omega} X_{i+1} \\ & = & \bigcup_{i<\omega} X_i \\ & = & X\end{array}$$

We make use of the restriction to finitary rule sets in the second line of the proof; the rest follows by elementary set-theoretic manipulations. The characterization of $\nu(\Phi)$ is established by a dual argument. ■

A.4 Derivations

Let R be a finitary rule set defined over a universe $\mathcal{U}$. The *derivable* elements of $\mathcal{U}$ relative to R are precisely the elements of $\mu(\Phi_R)$. Thus $x \in \mathcal{U}$ is derivable in R iff there is a finite composition of rules of R yielding x. More generally, $x \in \mathcal{U}$ is derivable from premises $X \subseteq \mathcal{U}$ in R iff there is such a derivation, taking the elements of X as axioms.

The derivability relation may be defined more precisely as follows. For a finitary rule set R, the relation $X \Rightarrow_R x$, read "x is derivable from premises X relative to R", holds exactly when x is derivable in the rule set $R \cup X$, the extension of R with axioms X. The rule $X \to x$ is said to be *derivable* in R exactly when this is the case.

An equivalent characterization of derivability may be given as follows. The *composition* of the rules $X \to x$ and $Y \cup \{ x \} \to y$ is the rule $X \cup Y \to y$ — the "assumption" x, in the second rule is replaced by the assumptions X of the first required to derive x. It is easy to see that a rule $X \to x$ is derivable in R iff $X \to x$ is a composition of some set of rules of R.

The crucial property of a derivable rule is that it remains derivable in all extensions of the rule set R (and conversely).

Lemma A.4
A rule $X \to x$ is derivable in R iff for every extension $R' \supseteq R$, if $\Rightarrow_{R'} X$, then $\Rightarrow_{R'} x$.

Proof: Suppose that $X \to x$ is derivable in R — that is, $X \Rightarrow_R x$. If $R' \supseteq R$, then clearly $X \to_{R'} r$, since the derivability relation is clearly monotone in the rule set. Hence if $\Rightarrow_{R'} X$, then $\Rightarrow_{R'} x$. For the converse suppose that $\Rightarrow_{R'} X$ implies $\Rightarrow_{R'} x$ for every $R' \supseteq R$. We are to show $X \Rightarrow_R x$. Taking $R' = R \cup X$,, we observe that $\Rightarrow_{R'} X$, and hence $\Rightarrow_{R'} x$. But then $X \Rightarrow_R x$, by the definition of derivability. ■

Derivability should not be confused with the closely-related notion of *admissibility*. A rule $X \to x$ is said to be a *admissible*in R iff $\Rightarrow_R x$ whenever $\Rightarrow_R X$. This condition is strictly weaker than derivability of the rule since we make no claim that the derivation of x from X is stable under extensions to R. Thus every derivable rule is admissible, but an admissible rule need not also be derivable. Intuitively, the admissibility of a rule is sensitive to the specific rule set R; the proof that it is admissible typically relies on a complete analysis of R (*e.g.*, induction on the size of R) and hence is not stable under extensions to R.

A.5 References

The main reference is Aczel's chapter "Introduction to Inductive Definitions" in the *Handbook of Mathematical Logic* [**?**]. See also Aczel's monograph on non-well-founded sets [**?**].

Appendix B

Domains

B.1 Introduction

Ordered sets arise frequently in the study of programming languages. Of particular interest are ordered sets with additional structure corresponding to recursive definitions. Loosely speaking such sets are called *domains*, although the reader is warned that the exact definition of a domain varies depending on the context of use.

B.2 Domains

A *partially ordered set (poset)* is a pair $(D, \sqsubseteq)$ consisting of a set D together with a reflexive, transitive, and anti-symmetric binary relation $\sqsubseteq$ on D. That is, $\sqsubseteq \subseteq D \times D$ satisfies the following three conditions:

Reflexivity $d \sqsubseteq d$ for every $d \in D$.

Anti-symmetry If $d \sqsubseteq d'$ and $d' \sqsubseteq d$, then $d = d'$.

Transitivity If $d \sqsubseteq d'$ and $d' \sqsubseteq d''$, then $d \sqsubseteq d''$.

We generally omit explicit mention of the ordering relation whenever it is clear from context.

A subset $S \subseteq D$ of a poset is *directed* iff every pair of elements of S has an upper bound in S (equivalently, iff every finite subset of S has an upper bound in S). A poset D is *directed complete* iff every non-empty directed subset S of D has a *least upper bound*, or *supremum*, $\bigsqcup S \in D$. Directed complete posets are called *dcpo's*. A dcpo D is *pointed* if it has a least element, *i.e.*iff there exists $\perp \in D$ such that $\perp \sqsubseteq x$ for every $x \in D$. Pointed dcpo's are called *dcppo*'s.

Exercise B.1
A chain *in a poset D is an increasing sequence $x_0 \sqsubseteq x_1 \sqsubseteq \cdots \in D$. Show that every chain is a directed subset of D. Hence if D is a dcpo, then every chain in D has a least upper bound in D.*

Definition B.2
Let $f : D_1 \to D_2$ be a function between dcpo's D_1 and D_2.

1. *The function f is* order-preserving *(or* monotone*) iff $f(d) \sqsubseteq f(d')$ in D_2 whenever $d \sqsubseteq d'$ in D_1. If D_1 and D_2 are pointed, then f is* point-preserving *(or* strict*) iff $f(\bot) = \bot$.*

2. *An order-preserving function f is* continuous *iff it preserves suprema of non-empty directed subsets, i.e., $f(\bigsqcup S) = \bigsqcup\{\, f(x) \mid x \in S \,\}$ whenever S is a non-empty directed subset of D_1.*

Definition B.3

1. *The* lifting, *$D_\bot$, of a dcpo D has as elements $\{\, \lfloor d \rfloor \mid d \in D \,\} \cup \{\, \bot \,\}$, ordered by (1) $\bot \sqsubseteq \lfloor d \rfloor$ for every $d \in D$, and (2) $\lfloor d \rfloor \sqsubseteq \lfloor d' \rfloor$ iff $d \sqsubseteq d'$ in D.*

2. *The* Cartesian product, *$[D_1 \times D_2]$ of two dcpo's D_1 and D_2 has as elements the set $\{\, \langle d_1, d_2 \rangle \mid d_1 \in D_1,\ d_2 \in D_2 \,\}$ ordered component-wise. The* smash product, *$[D_1 \otimes D_2]$, of two dcppo's has as elements $\{\, \langle d_1, d_2 \rangle \mid \bot \neq d_1 \in D_1,\ \bot \neq d_2 \in D_2 \,\}_\bot$ ordered componentwise, with $\bot$ as least element.*

3. *The* function space, *$[D_1 \to D_2]$, between dcpo's D_1 and D_2 consists of all continuous functions $f : D_1 \to D_2$ ordered by the pointwise ordering (i.e., $f \sqsubseteq g$ in $[D_1 \to D_2]$ iff $f(d) \sqsubseteq g(d)$ in D_2 for every $d \in D_1$).*

Lemma B.4

1. *If D is a dcpo, then $D_\bot$ is a dcppo.*

2. *If D_1 and D_2 are dcpo's, then $[D_1 \times D_2]$ is a dcpo. If D_1 and D_2 are dcppo's, then $[D_1 \otimes D_2]$ is a dcppo.*

3. *If D_1 and D_2 are dcpo's, then $[D_1 \to D_2]$ is a dcpo. If D_2 is pointed, then so is $[D_1 \to D_2]$.*

Theorem B.5 (Kleene)
If $f : D \to D$ is a continuous function on a dcppo, then f has a least fixed point $\mathit{fix}_D(f)$ given by $\bigsqcup_{i \geq 0} f^{(i)}(\bot)$.

Proof: Observe that the sequence $f^{(i)}(\bot)$ is a chain, and hence has a least upper bound. To see that this provides a fixed point we argue as follows:

$$\begin{array}{rcl} f(\bigsqcup_{i \geq 0} f^{(i)}(\bot)) & = & \bigsqcup_{i \geq 0} f(f^{(i)}(\bot)) \\ & = & \bigsqcup_{i \geq 0} f^{(i+1)}(\bot) \\ & = & \bigsqcup_{i \geq 0} f^{(i)}(\bot) \end{array}$$

To see that this is the least fixed point, observe that if $f(d) = d$, then each iterate $f^{(i)}(\bot) \sqsubseteq d$, and hence $\bigsqcup_{i \geq 0} f^{(i)}(\bot) \sqsubseteq d$. ■

Of particular interest are those dcppo's of the form $D = [D_1 \to D_2]$, where D_2 is pointed (D_1 may or may not be). We apply Kleene's fixed point theorem to a continuous functional defined on D to obtain its least fixed point, which is used as the interpretation of a recursive function.

B.3 References

Winskel's text [**?**] is an excellent introduction to domain theory and denotational semantics.

Appendix C

Term Rewriting Systems

C.1 Introduction

C.2 Abstract Term Rewriting Systems

We begin by introducing some general terminology and deriving some basic results. A *notion of reduction* on a set X is a binary relation $\rightarrow \subseteq X \times X$. We write $\rightarrow^*$ for the reflexive and transitive closure of $\rightarrow$, $\rightarrow^+$ for its transitive closure, and $\leftrightarrow$ for its symmetric closure. It is easy to see that $\leftrightarrow^*$ is the least equivalence relation containing $\rightarrow$. We write $x \downarrow y$ iff there exists z such that $x \rightarrow^* z$ and $y \rightarrow^* z$. Similarly, we write $x \uparrow y$ iff there exists z such that $z \rightarrow^* x$ and $z \rightarrow^* y$.

If x is such that $x \rightarrow x'$ for no x', then x is said to be *irreducible*, or *in normal form*. An element x is *normalizable* iff there exists x' in normal form such that $x \rightarrow^* x'$; it is *strongly normalizable* iff there are no infinite reduction sequences starting from x. We say that $\rightarrow$ is normalizable (resp., strongly normalizable) iff every x is normalizable (resp., strongly normalizable). A predicate P on X is *$\rightarrow$-complete* iff $P(x)$ holds whenever $P(y)$ holds for every y such that $x \rightarrow^+ y$. If $\rightarrow$ is strongly normalizable and P is $\rightarrow$-complete, then $P(x)$ holds for every $x \in X$. This is called *transfinite induction on $\rightarrow$*.

We say that $\rightarrow$ *has the diamond property* iff whenever $x \rightarrow x_1$ and $x \rightarrow x_2$, then there exists y such that $x_1 \rightarrow y$ and $x_2 \rightarrow y$. We say that $\rightarrow$ is *locally confluent* iff whenever $x \rightarrow x_1$ and $x \rightarrow x_2$, then $x_1 \downarrow x_2$. It is *confluent* if $x_1 \downarrow x_2$ whenever $x_1 \uparrow x_2$, *i.e.*whenever $\rightarrow^*$ has the diamond property. Clearly if $\rightarrow$ has the diamond property, it is locally confluent, and if it is confluent, it is locally confluent. The converses do not, in general, hold. However, if $\rightarrow$ is strongly normalizable, then local confluence is equivalent to confluence.

Exercise C.1
Prove by transfinite induction on $\rightarrow$ that if a strongly normalizable relation is locally confluent, then it is confluent.

Confluence has a number of significant consequences. First, normal forms are unique: if $x \to x_1$ and $x \to x_2$ where x_1 and x_2 are normal forms, then $x_1 = x_2$. Second, the "Church-Rosser" property holds: $x \leftrightarrow^* y$ iff $x \downarrow y$. The "if" part is immediate since $\to^* \subseteq \leftrightarrow^*$. For the "only if", the interesting case is transitivity. We have by induction that $x_1 \downarrow x_2$ and $x_2 \downarrow x_3$; we are to show that $x_1 \downarrow x_3$. Expanding the definitions, this means that there exists y and z such that $x_1 \to^* y$, $x_2 \to^* y$, $x_2 \to^* z$, and $x_3 \to^* z$. By confluence at x_2, we have that $y \downarrow z$, and hence $x_1 \downarrow x_3$. Stated in other words, we use confluence to argue that $\downarrow$ is a transitive relation; since it is clearly reflexive and symmetric and contains $\to$, it follows that $\leftrightarrow^* \subseteq \downarrow$ since $\leftrightarrow^*$ is the smallest such relation.

Bibliography

[1] Peter Aczel. Introduction to inductive definitions. In John Barwise, editor, *The Handbook of Mathematical Logic*, pages 739–782. North-Holland, 1977.

[2] Peter Aczel. *Non-Well-Founded Sets.* Lecture Notes. CSLI, Stanford, CA, 1988.

[3] Andrew W. Appel. *Compiling with Continuations.* Cambridge University Press, 1992.

[4] Andrew W. Appel and David B. MacQueen. A Standard ML compiler. In *Functional Programming Languages and Computer Architecture*, pages 301–324. Springer-Verlag, 1987. Volume 274 of *Lecture Notes in Computer Science.*

[5] Andrew W. Appel and David B. MacQueen. Standard ML of New Jersey. In J. Maluszynski and M. Wirsing, editors, *Third Int'l Symp. on Prog. Lang. Implementation and Logic Programming*, pages 1–13, New York, August 1991. Springer-Verlag.

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

[7] Luca Cardelli. Phase distinctions in type theory. unpublished manuscript.

[8] Luca Cardelli. Typeful programming. Technical Report 45, DEC Systems Research Center, 1989.

[9] Luca Cardelli and Xavier Leroy. Abstract types and the dot notation. Technical Report 56, DEC Systems Research Center, Palo Alto, CA, March 1990.

[10] Luca Cardelli, Simone Martini, John C. Mitchell, and Andre Scedrov. An extension of System F with subtyping. Research Report 80, Digital Systems Research Center, Palo Alto, California, December 1991.

[11] Dominique Clément, Joëlle Despeyroux, Thierry Despeyroux, Laurent Hascoet, and Gilles Kahn. Natural semantics on the computer. Technical Report RR 416, INRIA, Sophia–Antipolis, France, June 1985.

[12] Dominique Clément, Joëlle Despeyroux, Thierry Despeyroux, and Gilles Kahn. A simple applicative language: Mini-ML. In *1986 ACM Conference on LISP and Functional Programming*, 1986.

[13] Robert L. Constable and Scott Fraser Smith. Partial objects in constructive type theory. In *Second Symposium on Logic in Computer Science*, pages 183–193, June 1987.

[14] Roy Crole. *Categories for Types.* Cambridge Mathematical Textbooks. Cambridge, 1993.

[15] Bruce Duba, Robert Harper, and David MacQueen. Typing first-class continuations in ML. In *Eighteenth ACM Symposium on Principles of Programming Languages*, January 1991.

[16] Matthias Felleisen and Robert Hieb. The revised report on the syntactic theories of sequential control and state. *Theoretical Computer Science*, 10(2):235–271, 1992.

[17] Andrzej Filinski. Declarative continuations: An investigation of duality in programming language semantics. In *Summer Conference on Category Theory and Computer Science*, volume 389 of *Lecture Notes in Computer Science*, Manchester, UK, 1989. Springer-Verlag.

[18] Andrzej Filinski. Linear continuations. In *Nineteenth ACM Symposium on Principles of Programming Languages*, pages 27–38, Albuquerque, NM, January 1992. ACM SIGPLAN/SIGACT.

[19] Michael J. Fischer. Lambda-calculus schemata. *LISP and Symbolic Computation*, 6(3/4):259–288, November 1993.

[20] Harvey Friedman. Equality between functionals. In Rohit Parikh, editor, *Logic Colloquium '75*, Studies in Logic and the Foundations of Mathematics, pages 22–37. North-Holland, 1975.

[21] Jean Gallier. On Girard's "Candidats de Reductibilité". In P. Odifreddi, editor, *Logic and Computation*, volume 31 of *The APIC Series*, pages 123–203. Academic Press, 1990.

[22] Jean-Yves Girard. Une extension de l'interpretation de Gödel à l'analyse, et son application à l'élimination des coupures dans l'analyse et la théorie des types. In J. E. Fenstad, editor, *Proceedings of the Second Scanindavian Logic Symposium*, Studies in Logic and the Foundations of Mathematics, pages 63–92. North-Holland, 1971.

[23] Jean-Yves Girard. *Interprétation Fonctionnelle et Élimination des Coupures dans l'Arithmétique d'Ordre Supérieure*. PhD thesis, Université Paris VII, 1972.

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

[25] Timothy Griffin. A formulae-as-types notion of control. In *Seventeenth ACM Symposium on Principles of Programming Languages*, San Francisco, CA, January 1990. ACM, ACM.

[26] Timothy G. Griffin. Logical interpretations and computational simulations. Tech. memo., AT&T Bell Laboratories, 1992. in preparation.

[27] Carl A. Gunter. *Semantics of Programming Languages*. Foundations of Computing. MIT Press, Cambridge, MA, 1992.

[28] Robert Harper. A simplified account of polymorphic references. Technical Report CMU–CS–93–169, School of Computer Science, Carnegie Mellon University, Pittsburgh, PA 15213, June 1993.

[29] Robert Harper, Bruce Duba, and David MacQueen. Typing first-class continuations in ML. *Journal of Functional Programming*, 3(4):465–484, October 1993. (See also [?].).

[30] Robert Harper and Mark Lillibridge. Polymorphic type assignment and CPS conversion. In Olivier Danvy and Carolyn Talcott, editors, *Proceedings of the ACM SIGPLAN Workshop on Continuations CW92*, pages 13–22, Stanford, CA 94305, June 1992. Department of Computer Science, Stanford University. Published as technical report STAN–CS–92–1426.

[31] Robert Harper and Mark Lillibridge. Polymorphic type assignment and CPS conversion. *LISP and Symbolic Computation*, 6(4):361–380, November 1993. (See also [?].).

[32] Robert Harper and Mark Lillibridge. A type-theoretic approach to higher-order modules with sharing. In *Twenty-First ACM Symposium on Principles of Programming Languages*, pages 123–137, Portland, OR, January 1994.

[33] Robert Harper and John C. Mitchell. On the type structure of Standard ML. *ACM Transactions on Programming Languages and Systems*, 15(2):211–252, April 1993. (See also [?].).

[34] Robert Harper, John C. Mitchell, and Eugenio Moggi. Higher-order modules and the phase distinction. In *Seventeenth ACM Symposium on Principles of Programming Languages*, San Francisco, CA, January 1990.

[35] Xavier Leroy. Manifest types, modules, and separate compilation. In *Proceedings of the Twenty-first Annual ACM Symposium on Principles of Programming Languages, Portland*. ACM, January 1994.

[36] David MacQueen. Using dependent types to express modular structure. In *Thirteenth ACM Symposium on Principles of Programming Languages*, 1986.

[37] Per Martin-Löf. Constructive mathematics and computer programming. In *Sixth International Congress for Logic, Methodology, and Philosophy of Science*, pages 153–175. North-Holland, 1982.

[38] Per Martin-Löf. *Intuitionistic Type Theory*, volume 1 of *Studies in Proof Theory*. Bibliopolis, Naples, 1984.

[39] Paul Mendler. *Recursive Definition in Type Theory*. PhD thesis, Cornell University, 1987.

[40] Albert R. Meyer and Mitchell Wand. Continuation semantics in typed lambda calculi (summary). In Rohit Parikh, editor, *Logics of Programs*, volume 193 of *Lecture Notes in Computer Science*, pages 219–224. Springer-Verlag, 1985.

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

[42] Robin Milner and Mads Tofte. *Commentary on Standard ML*. MIT Press, 1991.

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

[44] John Mitchell and Robert Harper. The essence of ML. In *Fifteenth ACM Symposium on Principles of Programming Languages*, San Diego, California, January 1988.

[45] John C. Mitchell. *Foundations for Programming Languages*. Foundations of Computing. MIT Press, 1996.

[46] John C. Mitchell and Gordon Plotkin. Abstract types have existential type. *ACM Transactions on Programming Languages and Systems*, 10(3):470–502, 1988.

[47] Eugenio Moggi. Computational lambda calculus and monads. In *Fourth Symposium on Logic in Computer Science*, Asilomar, California, June 1989.

[48] Chetan Murthy. *Extracting Constructive Content from Classical Proofs*. PhD thesis, Cornell University, Ithaca, NY, August 1990.

[49] Frank Pfenning. Partial polymorphic type inference and higher-order unification. In *Proceedings of the 1988 ACM Conference on Lisp and Functional Programming*, pages 153–163, Snowbird, Utah, July 1988. ACM Press.

[50] Frank Pfenning. On the undecidability of partial polymorphic type reconstruction. *Fundamenta Informaticae*, 199? To appear. Preliminary version available as Technical Report CMU–CS–92–105, School of Computer Science, Carnegie Mellon University, Pittsburgh, Pennsylvania, January 1992.

[51] Frank Pfenning and Peter Lee. LEAP: A language with eval and polymorphism. In *TAPSOFT '89, Proceedings of the International Joint Conference on Theory and Practice in Software Development, Barcelona, Spain*, pages 345–359. Springer-Verlag LNCS 352, March 1989.

[52] Gordon Plotkin. Call-by-name, call-by-value, and the lambda calculus. *Theoretical Computer Science*, 1:125–159, 1975.

[53] Gordon Plotkin. LCF considered as a programming language. *Theoretical Computer Science*, 5:223–257, 1977.

[54] Gordon Plotkin. Lambda-definability in the full type hierarchy. In J. P. Seldin and J. R. Hindley, editors, *To H. B. Curry: Essays in Combinatory Logic, Lambda Calculus and Formalism*, pages 363–373. Academic Press, 1980.

[55] Gordon Plotkin. A structural approach to operational semantics. Technical Report DAIMI–FN–19, Computer Science Department, Aarhus University, 1981.

[56] Gordon Plotkin. Notes for a post-graduate course in semantics. Available from the Computer Science Department, University of Edinburgh, 1983.

[57] John C. Reynolds. Definitional interpreters for higher-order programming languages. In *Conference Record of the 25th National ACM Conference*, pages 717–740, Boston, August 1972. ACM.

[58] John C. Reynolds. Towards a theory of type structure. In *Colloq. sur la Programmation*, volume 19 of *Lecture Notes in Computer Science*, pages 408–423. Springer-Verlag, 1974.

[59] John C. Reynolds. The discoveries of continuations. *LISP and Symbolic Computation*, 6(3/4):233–248, November 1993.

[60] Scott F. Smith. Partial computations in constructive type theory. (To appear, *Information and Computation*), 1991.

[61] Richard Statman. Logical relations and the typed λ-calculus. *Information and Control*, 65:85–97, 1985.

[62] Glynn Winskel. *The Formal Semantics of Programming Languages*. Foundations of Computing Series. MIT Press, 1993.

[63] Andrew K. Wright and Matthias Felleisen. A syntactic approach to type soundness. *Information and Computation*, 115(1):38–94, November 1994.