Metalanguages and Applications

Eugenio Moggi*
DISI, Univ. of Genova
v. Dodecaneso 35, 16146 Genova, Italy
moggi@disi.unige.it

Introduction

The aim of these notes is to describe the monadic and incremental approaches to the denotational semantics of programming languages. This is done via the use of suitable typed metalanguages, which capture the relevant structure of semantic categories. The monadic and incremental approaches are formulated in the setting of a type-theoretic framework for the following reasons:

- a type theory with **dependent types** allows a precise, concise and general description of the two approaches, based on *signatures* as abstract representations for languages;
- there are various implementations (e.g. LEGO and CoQ) which provide computer assistance for several type-theories, and without computer assistance it seems unlikely that any of the two approaches can go beyond toy languages.

On the other hand, the monadic and incremental approaches can be described already with a naive set-theoretic semantics. Therefore, knowledge of Domain Theory and Category Theory becomes essential only in Section 5.

The presentation adopted differs from advanced textbooks on denotational semantics in the following aspects:

- it makes significant use of type theory as a tool for describing languages and calculi, while this is usually done via a set of formation or inference rules;
- it incorporates ideas from Axiomatic and Synthetic Domain Theory into metalanguages, while most metalanguages for denotational semantics are variants of LCF;
- it stresses the use of metalanguages to give semantics via translation (using the monadic and incremental approaches), but avoids a detailed analysis of the categories used in denotational semantics.

The remaining sections are organized as follows:

- Section 1 introduces few toy languages used in examples.
- Section 2 gives a brief overview of different approaches to programming language semantics, in order to place the use of metalanguages into context.
- Section 3 introduces a logical framework suitable for our applications, and explains how it may be used.

*This work was supported by ESPRIT BRA 6811 (Categorical Logic In Computer Science II) and EC SCIENCE twinning ERBSC1*CT920795 (Progr. Lang. Semantics and Program Logics)

- Section 4 explains possible uses of typed metalanguages for giving semantics to programming languages. Then, it introduces computational types and describes the monadic and incremental approaches.
- Section 5 addresses the issue of recursive definitions in the context of metalanguages. This is done by incorporating ideas from axiomatic and synthetic domain theory. Finally, it revises the notion of computational types (and its applications) in this richer setting.

Acknowledgements. These notes are partly based on joint work with P. Cenciarelli, some unpublished work by A. Simpson, and the MSc thesis of N. Signa. I would like to thank several people for discussions on Axiomatic and Synthetic Domain Theory: B. Reus, M. Fiore, P. Freyd, M. Hyland, A. Pitts, P. Rosolini, A. Simpson and T. Streicher. This notes were completed during my stay at the Newton Institute, and I would like to thanks the institute and the organizers of the program on "Semantics of Computation" for inviting me and providing such a pleasant working environment. I have used Paul Taylor's package for diagrams.

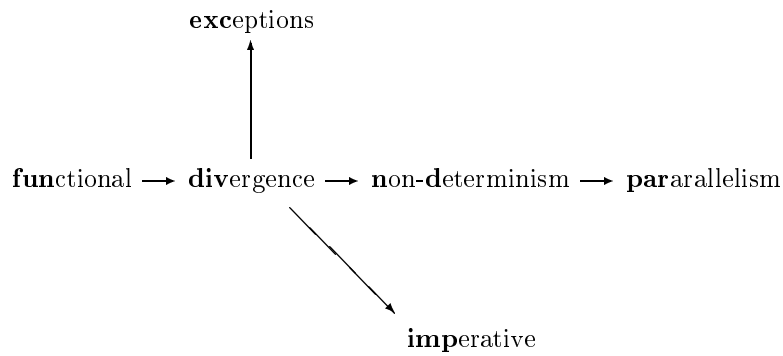
1 Toy programming languages

In this section we introduce few toy programming languages. They will be used as running examples to illustrate the use of metalanguages in describing the denotational semantics of programming languages.

These toy languages include a simple functional language and simple extensions of it obtained by adding the following features: divergence, mutable store, exception handling, nondeterminism, parallelism.

We present the functional language first, by giving its syntax, its typing rules, and two operational semantics (one call-by-value and one call-by-name). Then we introduce its extensions, by saying how their presentations differ from that of the simple functional language.

Here programming languages are given via an operational semantics, since this requires very little mathematical sophistication and it is enough to to give meaning to *observations*. However, in Section 2 we will describe alternative approaches, the trade-offs involved, and criteria to compare different semantics for the same language.



1.1 A simple functional language

A simple way of describing the syntax of a programming language is first to define *pseudo-expressions* by a BNF, and then define *well-formed expressions* by a set of typing rules. Often, the operational semantics can be defined for pseudo-expressions, while the denotational semantics is given by induction on the derivation that an expression is well-formed.

1.1.1 Syntax

The syntax for types and expressions of the language is given by the following BNF:

types	$\tau \in T ::=$	$unit \mid bool \mid nat \mid \tau_1 \Rightarrow \tau_2$
identifiers	$x \in Id ::=$	an infinite set
expressions	$e \in Exp ::=$	$x \mid$
		$* \mid$
		$tt \mid ff \mid if(e, e_1, e_2) \mid$
		$0 \mid s(e) \mid It(e_0, (\lambda x : \tau . e_s), e) \mid$
		$(\lambda x : \tau_1 . e_2) \mid ap(e, e_1)$

1.1.2 Typing rules

We give a set of rules to derive judgements of the form $\Gamma \vdash e : Exp[\tau]$, i.e. "expression e has type τ in typing context Γ". A **typing context** Γ is a sequence $x_1 : Id[\tau_1], \ldots, x_n : Id[\tau_n]$ s.t. the x_i are distinct.

Notation 1.1 We write:

- DV(Γ) for the set of the x_i, i.e. "the set of declared variables in Γ";
- $\Gamma(x_i)$ for $Id[\tau_i]$, i.e. "the type of x_i in Γ";
- $\Gamma; x : Id[\tau]$ for "the extension of Γ with $x : Id[\tau]$", i.e. $\Gamma_1, x : Id[\tau], \Gamma_2$ when $\Gamma \equiv \Gamma_1, x : Id[\tau'], \Gamma_2$ and $\Gamma, x : Id[\tau]$ otherwise.

$$var\frac{}{\Gamma \vdash x : Exp[\tau]}\ \Gamma(x) = Id[\tau] \qquad *\frac{}{\Gamma \vdash * : Exp[unit]}$$

$$tt\frac{}{\Gamma \vdash tt : Exp[bool]} \qquad ff\frac{}{\Gamma \vdash ff : Exp[bool]}$$

$$0\frac{}{\Gamma \vdash 0 : Exp[nat]} \qquad s\frac{\Gamma \vdash e : Exp[nat]}{\Gamma \vdash s(e) : Exp[nat]}$$

$$if\frac{\begin{array}{c}\Gamma \vdash e : Exp[bool]\\ \Gamma \vdash e_1, e_2 : Exp[\tau]\end{array}}{\Gamma \vdash if(e, e_1, e_2) : Exp[\tau]} \qquad It\frac{\begin{array}{c}\Gamma \vdash e : Exp[nat]\\ \Gamma \vdash e_0 : Exp[\tau]\\ \Gamma; x : Id[\tau] \vdash e_s : Exp[\tau]\end{array}}{\Gamma \vdash It(e_0, (\lambda x : \tau.e_s), e) : Exp[\tau]}$$

$$ap\frac{\begin{array}{c}\Gamma \vdash e : Exp[\tau_1 \Rightarrow \tau_2]\\ \Gamma \vdash e_1 : Exp[\tau_1]\end{array}}{\Gamma \vdash ap(e, e_1) : Exp[\tau_2]} \qquad ab\frac{\Gamma; x : Id[\tau_1] \vdash e_2 : Exp[\tau_2]}{\Gamma \vdash \lambda x : \tau_1.e_2 : Exp[\tau_1 \Rightarrow \tau_2]}$$

Remark 1.2 There are some basic properties of the typing rules one would like: it should be decidable whether a judgement $\Gamma \vdash e : Exp[\tau]$ is derivable, and its derivation should be unique, so that the interpretation of a judgement can be given unambiguously by induction on its derivation.

1.1.3 CBV operational semantics

The call-by-value (CBV) operational semantics is given by an evaluation relation $\Downarrow \subset Exp \times Val$, where $Val \subset Exp$ is the set of values:

$$v \in Val ::= \quad x \mid * \mid tt \mid ff \mid 0 \mid s(v) \mid (\lambda x : \tau_1.e_2)$$

The evaluation relation is defined by giving a set of rules for deriving judgements of the form $e \Downarrow v$, i.e. "expression e evaluates to value v".

$$\text{val}\frac{}{v \Downarrow v}$$

$$s\frac{e \Downarrow v}{s(e) \Downarrow s(v)}$$

$$if\frac{\begin{array}{c}e \Downarrow tt\\ e_1 \Downarrow v\end{array}}{if(e, e_1, e_2) \Downarrow v} \qquad if\frac{\begin{array}{c}e \Downarrow ff\\ e_2 \Downarrow v\end{array}}{if(e, e_1, e_2) \Downarrow v}$$

$$It\frac{\begin{array}{c}e \Downarrow 0\\ e_0 \Downarrow v\end{array}}{It(e_0, (\lambda x : \tau.e_s), e) \Downarrow v} \qquad It\frac{\begin{array}{c}e \Downarrow s(v_n)\\ It(e_0, (\lambda x : \tau.e_s), v_n) \Downarrow v_s\\ e_s[x := v_s] \Downarrow v\end{array}}{It(e_0, (\lambda x : \tau.e_s), e) \Downarrow v}$$

$$ap\frac{\begin{array}{c}e \Downarrow (\lambda x : \tau_1.e_2)\\ e_1 \Downarrow v_1\\ e_2[x := v_1] \Downarrow v\end{array}}{ap(e, e_1) \Downarrow v}$$

where $e'[x := v]$ is the substitution of identifier x with value v in expression e' (with a suitable renaming of the bound variables in e' to avoid clashes with the free variables in v).

Remark 1.3 Although the operational semantics is given independently from the typing rules, one expects some properties relating the two, e.g. **subject reduction**: $\emptyset \vdash e : Exp[\tau]$ and $e \Downarrow v$ implies $\emptyset \vdash v : Exp[\tau]$. This operational semantics is deterministic, i.e. there is at most one v s.t. $e \Downarrow v$, but this is not a property one may expect in general.

1.1.4 CBN operational semantics

The call-by-name (CBN) operational semantics is similar to CBV. The only differences are the definition of value (identifiers range over expressions) and the rule for evaluating $ap(e, e_1)$ (e_1 is not evaluated before substitution).

$$v \in Val ::= \quad * \mid tt \mid ff \mid 0 \mid s(v) \mid (\lambda x : \tau_1.e_2)$$

The Set of rules defining the CBN evaluation relation are those for CBV except

$$ap\frac{\begin{array}{c} e \Downarrow (\lambda x : \tau_1.e_2) \\ e_2[x := e_1] \Downarrow v \end{array}}{ap(e, e_1) \Downarrow v}$$

where $e'[x := e]$ is the substitution of identifier x with expression e in e'.

1.2 Extensions to the functional language

For each extension we give the corresponding additions to the syntax and typing rules of the functional language, and the modifications to the CBV operational semantics (the modifications to CBN operational semantics are left as an exercise).

1.2.1 Extension with divergence

- Syntax

types	$\tau \in T ::= \ldots$
identifiers	$x \in Id ::= \ldots$
expressions	$e \in Exp ::= \ldots \mid \bot$

- Typing rules

$$\bot\frac{}{\Gamma \vdash \bot : Exp[\tau]}$$

- CBV operational semantics

  $\Downarrow \subset Exp \times Val$, there are no changes to Val and the evaluation rules.

1.2.2 Extension with mutable store

- Syntax

types	$\tau \in T ::= \ldots$
identifiers	$x \in Id ::= \ldots$
locations	$l \in Loc ::=$ a set
expressions	$e \in Exp ::= \ldots \mid l \mid l := e$

- Typing rules

$$get\frac{}{\Gamma \vdash l : Exp[nat]}\ l \in Loc \qquad set\frac{\Gamma \vdash e : Exp[nat]}{\Gamma \vdash (l := e) : Exp[unit]}\ l \in Loc$$

- CBV operational semantics

$\Downarrow \subset (Exp \times St) \times (Val \times St)$, there are no changes to Val, and St is the set of stores $s \in St \stackrel{\Delta}{=} Loc \Rightarrow Val$

the evaluation rules for the functional language are changed, e.g.

$$\text{val}\frac{}{v, s \Downarrow v, s} \qquad \text{ap}\frac{\begin{array}{c} e, s_0 \Downarrow (\lambda x.e_2), s_1 \\ e_1, s_1 \Downarrow v_1, s_2 \\ e_2[x := v_1], s_2 \Downarrow v, s_3 \end{array}}{ap(e, e_1), s_0 \Downarrow v, s_3}$$

the following evaluation rules are added

$$\text{get}\frac{}{l, s \Downarrow v, s}\ v = s(l) \qquad \text{set}\frac{e, s_0 \Downarrow v, s_1}{l := e, s_0 \Downarrow *, s_1[l \mapsto v]}$$

where $s[l \mapsto v]$ is the function which maps l to v and is like s elsewhere.

1.2.3 Extension with exception handling

- Syntax

types	$\tau \in T ::= \ldots$
identifiers	$x \in Id ::= \ldots$
exception names	$n \in Exn ::=$ a set
expressions	$e \in Exp ::= \ldots \mid raise(n) \mid handle(n, e_1, e_2)$

- Typing rules

$$raise\frac{}{\Gamma \vdash raise(n) : Exp[\tau]}\ n \in Exn$$

$$handle\frac{\Gamma \vdash e_1, e_2 : Exp[\tau]}{\Gamma \vdash handle(n, e_1, e_2) : Exp[\tau]}\ n \in Exn$$

- CBV operational semantics

  $\Downarrow \subset Exp \times (Val + Exn)$ there are no changes to Val

  the evaluation rules for the functional language are changed, e.g.

$$\text{val}\frac{}{v \Downarrow v}$$

$$ap\frac{\begin{array}{c} e \Downarrow (\lambda x.e_2) \\ e_1 \Downarrow v_1 \\ e_2[x := v_1] \Downarrow r \end{array}}{ap(e, e_1) \Downarrow r} \qquad ap\frac{\begin{array}{c} e \Downarrow (\lambda x.e_2) \\ e_1 \Downarrow n \end{array}}{ap(e, e_1) \Downarrow n} \qquad ap\frac{e \Downarrow n}{ap(e, e_1) \Downarrow n}$$

the following evaluation rules are added

$$raise\frac{}{raise(n) \Downarrow n}$$

$$handle\frac{\begin{array}{c} e_2 \Downarrow n \\ e_1 \Downarrow r \end{array}}{handle(n, e_1, e_2) \Downarrow r} \qquad handle\frac{e_2 \Downarrow r}{handle(n, e_1, e_2) \Downarrow r}\ r \not\equiv n$$

1.2.4 Extension with nondeterministic choice

- Syntax

types	$\tau \in T ::= \ldots$
identifiers	$x \in Id ::= \ldots$
expressions	$e \in Exp ::= \ldots \mid or(e_1, e_2)$

- Typing rules

$$or\frac{\Gamma \vdash e_1, e_2 : Exp[\tau]}{\Gamma \vdash or(e_1, e_2) : Exp[\tau]}$$

- CBV operational semantics

  $\Downarrow \subset Exp \times Val$, there are no changes to Val and the evaluation rules

  the following evaluation rule is added

$$or\frac{e_i \Downarrow v}{or(e_1, e_2) \Downarrow v}$$

1.2.5 Extension with parallelism

- Syntax

types	$\tau \in T ::= \ldots$
identifiers	$x \in Id ::= \ldots$
expressions	$e \in Exp ::= \ldots \mid por(e_1, e_2) \mid pap(e_1, e_2)$

- Typing rules

$$por\frac{\Gamma \vdash e_1, e_2 : Exp[\tau]}{\Gamma \vdash por(e_1, e_2) : Exp[\tau]} \qquad pap\frac{\Gamma \vdash e : Exp[\tau_1 \Rightarrow \tau_2] \quad \Gamma \vdash e_1 : Exp[\tau_1]}{\Gamma \vdash pap(e, e_1) : Exp[\tau_2]}$$

- small step CBV operational semantics

  $\Rightarrow \subset Exp \times Exp$, there are no changes to Val

  the evaluation rules for the functional language are changed, e.g.

$$if\frac{e \Rightarrow e'}{if(e, e_1, e_2) \Rightarrow if(e', e_1, e_2)}$$

$$if\frac{}{if(tt, e_1, e_2) \Rightarrow e_1} \qquad if\frac{}{if(ff, e_1, e_2) \Rightarrow e_2}$$

$$s\frac{e \Rightarrow e'}{s(e) \Rightarrow s(e')} \qquad \ldots$$

$$ap\frac{e \Rightarrow e'}{ap(e, e_1) \Rightarrow ap(e', e_1)} \qquad ap\frac{e_1 \Rightarrow e_1'}{ap(v, e_1) \Rightarrow ap(v, e_1')}$$

$$ap\frac{}{ap((\lambda x.e_2), v_1) \Rightarrow e_2[x := v_1]}$$

  the following evaluation rules are added

$$pap\frac{e \Rightarrow e'}{pap(e, e_1) \Rightarrow pap(e', e_1)} \qquad pap\frac{e_1 \Rightarrow e_1'}{pap(e, e_1) \Rightarrow pap(e, e_1')}$$

$$pap\frac{}{pap((\lambda x.e_2), v_1) \Rightarrow e_2[x := v_1]}$$

$$por\frac{e_1 \Rightarrow e_1'}{por(e_1, e_2) \Rightarrow por(e_1', e_2)} \qquad por\frac{e_2 \Rightarrow e_2'}{por(e_1, e_2) \Rightarrow por(e_1, e_2')}$$

$$por\frac{}{por(v, e_2) \Rightarrow v} \qquad por\frac{}{por(e_1, v) \Rightarrow v}$$

2 Equivalence of programs

So far we have considered only operational semantics. In this section we review the denotational and algebraic (axiomatic) approaches, and explain how one can compare different semantics. The denotational approach assigns meaning to well-formed expressions (by induction on the derivation that they are well-formed), while the algebraic approach gives a set of rules for deriving when two well-formed expressions are equivalent. The denotational approach is particularly useful to validate reasoning principles, including equational rules, which can be used to prove properties of programs formally (without direct reference to the denotational model).

2.1 Observational equivalence

Operational semantics (even for the same programming language) can be substantially different, however one can compared the induced *observational equivalences*, which are equivalence relations (indeed congruences) on well-formed expressions. In fact, to associate an observational equivalence/preorder to an operational semantics one has to fix a notion of observation on *programs*, e.g. "program p has value tt", a notion of program context $C[_]$, and an (operational) interpretation of observations. We exemplify these notions for some of the operational semantics introduced so far:

- p is a program ($p \in Prg$ for short) $\stackrel{\Delta}{\Longleftrightarrow}$ p is a well-formed expression of type *bool* without free variables, i.e. $\vdash p : Exp[bool]$ is derivable.

- $C[_]$ is a program context for expressions of type τ in typing context Γ $\stackrel{\Delta}{\Longleftrightarrow}$ $C[_]$ is an *expression with one hole* s.t. $C[e] \in Prg$, whenever $\Gamma \vdash e : Exp[\tau]$ is derivable.

- given an operational semantics of the form $\Downarrow \subseteq Exp \times Val$ we say that

  "p has value tt" $\stackrel{\Delta}{\Longleftrightarrow}$ ($p \in Prg$ and) $p \Downarrow tt$

- given two well-formed expressions $\Gamma \vdash e_i : Exp[\tau]$ we say that

  $\Gamma \vdash e_1 \leq^{op} e_2 : Exp[\tau]$ (observational preorder) $\stackrel{\Delta}{\Longleftrightarrow}$
  $C[e_1] \Downarrow tt$ implies $C[e_2] \Downarrow tt$, whenever $C[_]$ is a program context for expressions of type τ in typing context Γ

  $\Gamma \vdash e_1 =^{op} e_2 : Exp[\tau]$ (observational equivalence) $\stackrel{\Delta}{\Longleftrightarrow}$
  $\Gamma \vdash e_1 \leq^{op} e_2 : Exp[\tau]$ and $\Gamma \vdash e_2 \leq^{op} e_1 : Exp[\tau]$.

Definition 2.1 *Given two programming languages PL_i and corresponding (operational) semantics $\Downarrow_i$ s.t. the well-formed expressions of PL_1 are included in those of PL_2 (i.e. $\Gamma \vdash_1 e : Exp[\tau]$ implies $\Gamma \vdash_2 e : Exp[\tau]$), we say that*

- *$\Downarrow_2$ is* **computationally adequate** *w.r.t. $\Downarrow_1$ $\stackrel{\Delta}{\Longleftrightarrow}$ the meaning of observations agrees, i.e. $p \Downarrow_1 tt$ iff $p \Downarrow_2 tt$ for any program p of PL_1.*

- *$\Downarrow_2$ is equationally* **sound** *w.r.t. $\Downarrow_1$ $\stackrel{\Delta}{\Longleftrightarrow}$ $\Gamma \vdash_2 e_1 =^{op} e_2 : Exp[\tau]$ implies $\Gamma \vdash_1 e_1 =^{op} e_2 : Exp[\tau]$, whenever e_i are well-formed expressions of PL_1 (i.e. $\Gamma \vdash_1 e_i : Exp[\tau]$)*

- *$\Downarrow_2$ is* **abstraction preserving** *w.r.t. $\Downarrow_1$ $\stackrel{\Delta}{\Longleftrightarrow}$ (it is equationally sound and) $\Gamma \vdash_1 e_1 =^{op} e_2 : Exp[\tau]$ implies $\Gamma \vdash_2 e_1 =^{op} e_2 : Exp[\tau]$.*

Under reasonable assumptions about programs and program contexts (which we do not spell out), the following result hold.

Proposition 2.2 *If $\Downarrow_2$ is computationally adequate (w.r.t. $\Downarrow_1$), then it is also sound.*

Let us consider the following operational semantics: $\Downarrow_{f_v}$ (CBV for the functional language), $\Downarrow_{f_n}$ (CBN for the functional language), $\Downarrow_v$ (CBV for the functional language with divergence), $\Downarrow_n$ (CBN for the functional language with divergence).

Proposition 2.3 *The following relations holds:*

- *$p \Downarrow_{f_v} tt$ iff $p \Downarrow_{f_n} tt$, so the two semantics induce the same observational equivalence $=^{op}$;*
- *$\Downarrow_v$ and $\Downarrow_n$ are computationally adequate w.r.t. $\Downarrow_{f_v}$;*
- *$\Downarrow_v$ and $\Downarrow_n$ are not computationally adequate w.r.t. each other, e.g. $p \Downarrow_n tt$ but $p \not\Downarrow_v tt$ when $p = (\lambda x : \tau.tt)\bot$.*

If one compares the observational equivalences $=^{op}$, $=^{op}_v$ and $=^{op}_n$ on the well-formed expressions of the functional language, then $=^{op}_v$ and $=^{op}_n$ are incomparable, and both are properly included in $=^{op}$. In fact:

- $x, y : Id[unit] \vdash x = y : Exp[unit]$ holds for $=^{op}$ and $=^{op}_v$, but not for $=^{op}_n$
- $y : Id[unit], f : Id[unit \Rightarrow unit] \vdash ap(\lambda x : unit.y, ap(f, *)) = y : Exp[unit]$ holds for $=^{op}$ and $=^{op}_n$, but not for $=^{op}_v$,

Remark 2.4 Instead of observing "p has value tt" (where $\vdash p : Exp[bool]$), we could have observed "p has a value" (where $\vdash p : Exp[\tau]$ for some type τ). The observational equivalences considered above are affected by this change as follows:

- $=^{op}$ collapses to the equivalence which identifies well-formed expressions of the same type;
- $=^{op}_v$ is unchanged;
- $=^{op}_n$ discriminates more, e.g. $f : Id[\tau_1 \Rightarrow \tau_2] \vdash \lambda x : \tau_1.ap(f, x) = f : Exp[\tau_1 \Rightarrow \tau_2]$ is no longer true.

2.2 Denotational semantics

The general pattern of a set-theoretic semantics for the functional language (and its extensions) is:

- types, like τ, syntactic categories, like $Exp[\tau]$, and typing contexts, like Γ, are interpreted as sets
- well-formed expressions, like $\Gamma \vdash e : Exp[\tau]$, are interpreted as functions from the interpretation of Γ to that of $Exp[\tau]$.

The same pattern applies, mutatis mutandis, to the interpretation in any category (e.g. the category of cpos), just replace sets with objects and functions with morphisms.

In fact, we will view the interpretation of $\Gamma \vdash e : Exp[\tau]$ as a family $\langle a_\rho | \rho \in [\![\Gamma]\!] \rangle$ assigning to each *environment* ρ the interpretation of e in it. This is closer to the usual Tarski's semantics for predicate calculus, but it does not generalize as easily to other categories. The interpretation is defined by induction on the syntax, in the case of a well-formed expression this means by induction on the derivation of $\Gamma \vdash e : Exp[\tau]$.

As a sample we give the *standard* set-theoretic interpretation of the functional language and two interpretations of the extension with divergence, which *correspond* to CBV and CBN (in a sense to be made precise later).

2.2.1 Standard semantics of the functional language

- The interpretation of types, syntactic categories and typing contexts is:

$$
\begin{array}{rl}
[\![unit]\!] = & \{*\} \\
[\![bool]\!] = & \{tt, ff\} \\
[\![nat]\!] = & \mathsf{N} \text{ the set of natural numbers} \\
[\![\tau_1 \Rightarrow \tau_2]\!] = & [\![\tau_1]\!] \to [\![\tau_2]\!] \text{ the set of total functions} \\
[\![Id[\tau]]\!] = & [\![\tau]\!] \\
[\![Exp[\tau]]\!] = & [\![\tau]\!] \\
[\![\Gamma]\!] = & [\![Id[\tau_1]]\!] \times \ldots \times [\![Id[\tau_n]]\!] \text{ where } \Gamma \text{ is } x_1 : Id[\tau_1], \ldots, x_n : Id[\tau_n]
\end{array}
$$

- The interpretation of well-formed expressions (of functional type) is:

$$
\begin{array}{rl}
[\![\Gamma \vdash x_i : Exp[\tau_i]]\!]_\rho = & \pi_i^n(\rho) : A_i \\
\hline
[\![\Gamma, x : Id[\tau_1] \vdash e_2 : Exp[\tau_2]]\!]_{\rho,a} = & b_a : A_2 \\
\hline
[\![\Gamma \vdash \lambda x : \tau_1.e_2 : Exp[\tau_1 \Rightarrow \tau_2]]\!]_\rho = & \lambda a \in A_1.b_a \\
\hline
[\![\Gamma \vdash e : Exp[\tau_1 \Rightarrow \tau_2]]\!]_\rho = & f : A_1 \to A_2 \\
[\![\Gamma \vdash e_1 : Exp[\tau_1]]\!]_\rho = & a : A_1 \\
\hline
[\![\Gamma \vdash ap(e, e_1) : Exp[\tau_2]]\!]_\rho = & f(a)
\end{array}
$$

where $A = [\![\tau]\!]$ and $A_i = [\![\tau_i]\!]$.

2.2.2 CBV semantics of the functional language with divergence

- CBV semantics differs from standard semantics in the interpretation of $\tau_1 \Rightarrow \tau_2$ and $Exp[\tau]$:

$$
\begin{array}{rl}
[\![\tau_1 \Rightarrow \tau_2]\!] = & [\![\tau_1]\!] \to [\![Exp[\tau_2]]\!] \\
[\![Exp[\tau]]\!] = & [\![\tau]\!] + \{\bot\}
\end{array}
$$

- while the interpretation of well-formed expressions has to be changed consistently (and extended in the obvious way):

$$
\begin{array}{rl}
[\![\Gamma \vdash x_i : Exp[\tau_i]]\!]_\rho = & in_1(\pi_i^n(\rho)) : A_i + \{\bot\} \\
[\![\Gamma \vdash \bot : Exp[\tau]]\!]_\rho = & in_2(\bot) : A + \{\bot\} \\
[\![\Gamma, x : Id[\tau_1] \vdash e_2 : Exp[\tau_2]]\!]_{\rho,a} = & b_a : A_2 + \{\bot\} \\
\hline
[\![\Gamma \vdash \lambda x : \tau_1.e_2 : Exp[\tau_1 \Rightarrow \tau_2]]\!]_\rho = & in_1(\lambda a \in A_1.b_a) \\
[\![\Gamma \vdash e : Exp[\tau_1 \Rightarrow \tau_2]]\!]_\rho = & f : (A_1 \to (A_2 + \{\bot\})) + \{\bot\} \\
[\![\Gamma \vdash e_1 : Exp[\tau_1]]\!]_\rho = & a : A_1 + \{\bot\} \\
\hline
[\![\Gamma \vdash ap(e, e_1) : Exp[\tau_2]]\!]_\rho = & \begin{cases} ga_1 & \text{if } f = in_1(g),\ a = in_1(a_1) \\ in_2(\bot) & \text{otherwise} \end{cases}
\end{array}
$$

where $A = [\![\tau]\!]$ and $A_i = [\![\tau_i]\!]$.

2.2.3 CBN semantics of the functional language with divergence

- The CBN semantics differs from CBV semantics in the interpretation of $\tau_1 \Rightarrow \tau_2$ and $Id[\tau]$:

$$
\begin{array}{rl}
[\![\tau_1 \Rightarrow \tau_2]\!] = & [\![Exp[\tau_1]]\!] \to [\![Exp[\tau_2]]\!] \\
[\![Id[\tau]]\!] = & [\![\tau]\!] + \{\bot\}
\end{array}
$$

- while the interpretation of well-formed expressions has to be changed consistently:

$$
\begin{array}{rl}
[\![\Gamma \vdash x_i : Exp[\tau_i]]\!]_\rho = & \pi_i^n(\rho) : A_i + \{\bot\} \\
[\![\Gamma \vdash \bot : Exp[\tau]]\!]_\rho = & in_2(\bot) : A + \{\bot\} \\
\hline
\dfrac{[\![\Gamma, x : Id[\tau_1] \vdash e_2 : Exp[\tau_2]]\!]_{\rho,a} =}{[\![\Gamma \vdash \lambda x : \tau_1.e_2 : Exp[\tau_1 \Rightarrow \tau_2]]\!]_\rho =} & \begin{array}{l} b_a : A_2 + \{\bot\} \\ in_1(\lambda a \in A_1.b_a) \end{array} \\
\hline
\dfrac{\begin{array}{r} [\![\Gamma \vdash e : Exp[\tau_1 \Rightarrow \tau_2]]\!]_\rho = \\ [\![\Gamma \vdash e_1 : Exp[\tau_1]]\!]_\rho = \end{array}}{[\![\Gamma \vdash ap(e, e_1) : Exp[\tau_2]]\!]_\rho =} & \begin{array}{l} f : ((A_1 + \{\bot\}) \to (A_2 + \{\bot\})) + \{\bot\} \\ a : A_1 + \{\bot\} \\ \hline \begin{cases} g(a) & \text{if } f = in_1(g) \\ in_2(\bot) & \text{otherwise} \end{cases} \end{array}
\end{array}
$$

where $A = [\![\tau]\!]$ and $A_i = [\![\tau_i]\!]$.

2.2.4 Denotational versus observational equivalence

An interpretation $[\![_]\!]$ of well-formed expressions induces an equivalence relation indeed a congruence (under reasonable assumptions about $[\![_]\!]$):

- $[\![\Gamma \vdash e_1 = e_2 : Exp[\tau]]\!]$ (denotational equivalence) $\stackrel{\Delta}{\Longleftrightarrow}$ the interpretations $[\![\Gamma \vdash e_i : Exp[\tau]]\!]$ are equal.

However, one can also associate an observational equivalence/preorder to a denotational semantics. What is needed is to fix the interpretation of observations (and proceed like in Section 2.1), e.g.:

- "p has value tt" $\stackrel{\Delta}{\Longleftrightarrow}$ $[\![\vdash p = tt : Exp[bool]]\!]$.

Remark 2.5 At this point we can relate the three denotational semantics considered in this section with the operational semantics considered in Section 2.1:

- $[\![\vdash p = tt : Exp[bool]]\!]_{fun}$ iff $p \Downarrow_{fun} tt$
- $[\![\vdash p = tt : Exp[bool]]\!]_v$ iff $p \Downarrow_v tt$
- $[\![\vdash p = tt : Exp[bool]]\!]_n$ iff $p \Downarrow_n tt$

i.e. corresponding semantics agree on the interpretation of observations.

In general, denotational and observational equivalence for a given denotational semantics do not coincide, but under reasonable assumptions about $[\![_]\!]$, the following result hold.

Proposition 2.6 *Denotational equivalence implies observational equivalence.*

It is usually easier to establish denotational equivalence than observational equivalence, since the latter involves a universal quantification over program contexts. Therefore, the result above gives a simpler way to prove observational equivalence (on the other hand, it is easy to prove that two expressions are not observationally equivalent, just find a discriminating program context!).

Definition 2.7 *Given a denotational semantics $[\![_]\!]$ for a programming language PL (and an interpretation of observations), we say that*

- $[\![_]\!]$ *is* **fully abstract** $\stackrel{\Delta}{\Longleftrightarrow}$ *denotational and observational equivalence coincide.*

Remark 2.8 A broader and deeper discussion on "good fit criteria" between operational and denotational semantics can be found in [MC88].

2.3 Equational calculi

A simple way to prove observational equivalence is to identify inference rules which are *admissible* w.r.t. denotational equivalence, and use them to derive formally $\Gamma \vdash e_1 = e_2 : Exp[\tau]$. It is usually easy to check that an inference rule is admissible w.r.t. denotational equivalence (just check that the conclusion of the rule is true in the model, whenever the premisses are!). When one cannot rely on a denotational model, one can establish that provable equivalence implies observational equivalence, by analogy with Section 2.2.4:

- first express observations in the equational calculus, e.g.

  "p has value tt" $\stackrel{\Delta}{\Longleftrightarrow}$ "$\vdash p = tt : Exp[bool]$" is derivable

- then prove that: $\vdash p = tt : Exp[bool]$ is derivable iff $p \Downarrow tt$.

Under reasonable assumptions about program contexts this entails that provable equivalence implies observational equivalence. Note that this is weaker than "the inference rules are admissible w.r.t. observational equivalence".

For each of the denotational semantics considered in Section 2.2 we give a set of admissible rules for deriving denotational equivalence (we skip the congruence rules, which are always admissible):

- standard equational calculus

$$\beta' \frac{}{\Gamma \vdash ap(\lambda x : \tau_1.e_2, x) = e_2 : Exp[\tau_2]}$$

$$\eta' \frac{}{\Gamma \vdash \lambda x : \tau_1.ap(f, x) = f : Exp[\tau_1 \Rightarrow \tau_2]}$$

$$sub \frac{\begin{array}{c}\Gamma \vdash e_i : Exp[\tau_i] \quad (i = 1, \ldots, n) \\ x_1 : Id[\tau_1], \ldots, x_n : Id[\tau_n] \vdash e = e' : Exp[\tau]\end{array}}{\Gamma \vdash e[\overline{x} := \overline{e}] = e'[\overline{x} := \overline{e}] : Exp[\tau]}$$

- CBV equational calculus, like the standard calculus but (sub) is replaced by

$$sub_v \frac{\begin{array}{c}\Gamma \vdash e_i : Exp[\tau_i] \quad (i = 1, \ldots, n) \\ x_1 : Id[\tau_1], \ldots, x_n : Id[\tau_n] \vdash e = e' : Exp[\tau]\end{array}}{\Gamma \vdash e[\overline{x} := \overline{e}] = e'[\overline{x} := \overline{e}] : Exp[\tau]} \; e_i \text{ values}$$

  where values are either variables or lambda-abstractions

- CBN equational calculus, like the standard calculus but without (η').

3 Abstract syntax and encoding in LF

In this section we introduce a *logical framework* with a cumulative hierarchy of predicative universes (in this way we don't need to distinguish between contexts Γ and signatures Σ). Our main motivation for introducing a logical framework is to have precise and concise descriptions of (the well-formed expressions of) languages and translations: languages are described by signatures and translations by signature realizations.

3.1 The logical framework LF

The logical framework is give by a set of inference rules for deriving judgements of the following forms:

- $\Gamma \vdash$, i.e. Γ is a context
- $\Gamma \vdash A : Type_i$, i.e. A is a type (in the i-th universe) in context Γ
- $\Gamma \vdash M : A$, i.e. M is a term of type A in context Γ

where M and A range over pseudo-terms described by the following BNF:

identifiers	$x \in Id ::=$ an infinite set
pseudo-terms	$A, M \in Exp ::= \ x \mid Type_i \mid \Pi x : A_1.A_2 \mid \lambda x : A.M \mid M_1 M_2$

$$\text{empty}\frac{}{\emptyset \vdash}$$

$$\text{ext}\frac{\Gamma \vdash A : Type_i}{\Gamma, x : A \vdash}\ x \notin \mathrm{DV}(\Gamma)$$

$$\text{type-}\in\frac{\Gamma \vdash}{\Gamma \vdash Type_i : Type_{i+1}}\ i \geq 0$$

$$\text{type-}\subset\frac{\Gamma \vdash A : Type_i}{\Gamma \vdash A : Type_j}\ i < j$$

$$\text{var}\frac{\Gamma \vdash}{\Gamma \vdash x : A}\ A = \Gamma(x)$$

$$\Pi\frac{\Gamma \vdash A_1 : Type_i \qquad \Gamma, x : A_1 \vdash A_2 : Type_i}{\Gamma \vdash (\Pi x : A_1.A_2) : Type_i}$$

$$\lambda\frac{\Gamma \vdash A_1 : Type_i \qquad \Gamma, x : A_1 \vdash M_2 : A_2}{\Gamma \vdash (\lambda x : A_1.M_2) : (\Pi x : A_1.A_2)}$$

$$\text{app}\frac{\Gamma \vdash M : (\Pi x : A_1.A_2) \qquad \Gamma \vdash M_1 : A_1}{\Gamma \vdash M M_1 : A_2[x := M_1]}$$

$$\text{conv}\frac{\Gamma \vdash M : A_1 \qquad \Gamma \vdash A_2 : Type_i}{\Gamma \vdash M : A_2}\ A_1 =_{\beta\eta} A_2$$

where $=_{\beta\eta}$ is $\beta\eta$-conversion on pseudo-terms, i.e. the congruence induced by α-conversion, $(\lambda x : A.M_2)M_1 = M_2[x := M_1]$ and $(\lambda x : A.Mx) = M$ provided $x \notin \mathrm{FV}(M)$.

Remark 3.1 The meta-theory of LF is rather delicate because of the (conv) rule and the failure of the Church-Rosser property for $\beta\eta$-reduction on pseudo-terms, anyway one can prove the following properties (see [HHP87, Geu92, Luo94]):

- it is decidable whether $\Gamma \vdash M : A$ is derivable

- if $\Gamma \vdash M_i : A$ (for $i = 1, 2$) is derivable, then $M_1 =_{\beta\eta} M_2$ is decidable.

It is convenient to separate the initial part of a context, which is intended to consist of constants, from the remaining part, consisting of variables. Therefore, we introduce the following derived notation:

- a LF-signature Σ is a well-formed context $\Sigma \vdash$
- a relativized judgement $\Gamma \vdash_\Sigma J$ stands for $\Sigma, \Gamma \vdash J$.

3.2 Set-theoretic semantics

The set-theoretic interpretation of LF has the following pattern (to model the cumulative hierarchy of universes we need a sequence of inaccessible cardinals α_i, so that $Type_i$ is interpreted by the set V_{α_i} of the von Neumann's hierarchy):

- the interpretation $[\![\Gamma \vdash]\!]$ of a context is a set I
- the interpretation $[\![\Gamma \vdash A : Type_j]\!]$ of a type is a family of sets $\langle X_i | i \in I \rangle$ s.t. $X_i \in V_{\alpha_j}$
- the interpretation $[\![\Gamma \vdash M : A]\!]$ of a term is a family of elements $\langle x_i | i \in I \rangle$ s.t. $x_i \in X_i$.

Remark 3.2 Because of the (conv) rule one may have different derivations of the same judgement, therefore the interpretation of a judgement cannot be defined by induction on the derivation. In any case, for defining the interpretation it is better to work with an equivalent semantic system (see [GW94]), in which one has also judgements of the form $\Gamma \vdash M_1 = M_2 : A$. One can proceed in two ways, either define the interpretation of a derivation and prove that derivations of the same judgement are interpreted in the same way (this is called a coherence result), or give a partially defined interpretation of pseudo-judgements (by induction on their *size*) and prove that whenever a pseudo-judgement is derivable its interpretation is defined (and satisfies certain properties).

We Follow the second approach and define a partial function $[\![\Gamma \vdash M : A]\!]$ by induction on $s(\Gamma) + s(M)$, where $s(_)$ gives the number of *symbols* in $_$.

Notation 3.3 Given a set X and a family of sets $\langle Y_x | x \in X \rangle$ we write $\Sigma x \in X.Y_x$ for the set $\{(x, y) | x \in X, y \in Y_x\}$ and $\Pi x \in X.Y_x$ for the set of all functions with domain X s.t. $\forall x \in X. f(x) \in Y_x$. Moreover, we identify a function $\lambda x \in X.y_x$ with its graph, i.e. the set $\{(x, y_x) | x \in X\}$, and write $X \times Y$ and $X \to Y$ instead of $\Sigma x \in X.Y_x$ and $\Pi x \in X.Y_x$, when Y_x is constantly Y.

empty	$[\![\emptyset \vdash]\!] =$	$1 = \{*\}$
ext	$[\![\Gamma \vdash A]\!] =$	$\langle X_i \mid i \in I \rangle$
	$[\![\Gamma, x : A \vdash]\!] =$	$\Sigma i \in I.X_i$
type-$\in$	$[\![\Gamma \vdash]\!] =$	I
	$[\![\Gamma \vdash Type_j]\!] =$	$\langle V_{\alpha_j} \mid i \in I \rangle$
var	$[\![\Gamma \vdash]\!] =$	I
	$[\![\Gamma \vdash x]\!] =$	$\langle \pi_x^\Gamma(i) \mid i \in I \rangle$
Π	$[\![\Gamma \vdash A_1]\!] =$	$\langle X_i \mid i \in I \rangle$
	$[\![\Gamma, x : A_1 \vdash A_2]\!] =$	$\langle Y_{(i,x)} \mid i \in I, x \in X_i \rangle$
	$[\![\Gamma \vdash (\Pi x : A_1.A_2)]\!] =$	$\langle \Pi x \in X_i.Y_{(i,x)} \mid i \in I \rangle$
λ	$[\![\Gamma \vdash A_1]\!] =$	$\langle X_i \mid i \in I \rangle$
	$[\![\Gamma, x : A_1 \vdash M_2]\!] =$	$\langle y_{(i,x)} \mid i \in I, x \in X_i \rangle$
	$[\![\Gamma \vdash (\lambda x : A_1.M_2)]\!] =$	$\langle \lambda x \in X_i.y_{(i,x)} \mid i \in I \rangle$
app	$[\![\Gamma \vdash M]\!] =$	$\langle f_i \mid i \in I \rangle$
	$[\![\Gamma \vdash M_1]\!] =$	$\langle x_i \mid i \in I \rangle$
	$[\![\Gamma \vdash M M_1]\!] \simeq$	$\langle f_i(x_i) \mid i \in I \rangle$

Note that it is only in the last case that the interpretation can be undefined even when the premisses are satisfied. At this point one can prove (by induction on the derivation of a judgement in the semantic system), that:

- if $\Gamma \vdash$ is derivable, then $[\![\Gamma \vdash]\!] = I$ for some set I
- if $\Gamma \vdash M : A$ is derivable, then
  $[\![\Gamma \vdash M]\!] = \langle x_i \mid i \in I \rangle$, $[\![\Gamma \vdash A]\!] = \langle X_i \mid i \in I \rangle$ and $\forall i \in I.x_i \in X_i$
  for some set I and I-indexed families x and X

  therefore we can define $[\![\Gamma \vdash M : A]\!]$ as $[\![\Gamma \vdash M]\!]$
- if $\Gamma \vdash M = N : A$ is derivable, then
  $[\![\Gamma \vdash M]\!] = \langle x_i \mid i \in I \rangle = [\![\Gamma \vdash N]\!]$, $[\![\Gamma \vdash A]\!] = \langle X_i \mid i \in I \rangle$ and $\forall i \in I.x_i \in X_i$
  for some set I and I-indexed families x and X.

Given a model for a LF-signature, i.e. $\mathrm{M} \in [\![\Sigma]\!]$, one can define the interpretation $[\![\Gamma \vdash_\Sigma J]\!]^{\mathrm{M}}$ of relativized judgements in M in the obvious way, e.g. $[\![\Gamma \vdash_\Sigma]\!]^{\mathrm{M}} \triangleq \{i \mid \mathrm{M} * i \in I\}$, where $I = [\![\Sigma, \Gamma \vdash]\!]$ and $*$ is concatenation.

3.3 Encodings

One can give a compact and uniform description of the typing rules for the functional language (and the other languages introduced in Section 1) in terms of a LF-signature.

Notation 3.4 We use the convention of writing: $Type$ for some unspecified $Type_i$ with i big enough, $A_1 \to A_2$ for $\Pi x : A_1.A_2$ with $x \notin \mathrm{FV}(A_2)$, $A_1, \ldots, A_n \to A$ for $A_1 \to \ldots A_n \to A$.

- LF-signature Σ_{fun} for the functional language

types	$T : Type$
	$unit, bool, nat : T$
	$\Rightarrow : T, T \to T$
ident	$Id : T \to Type$
expr	$Exp : T \to Type$
	$var : \Pi X : T.Id(X) \to Exp(X)$
unit	$* : Exp(unit)$
bool	$tt, ff : Exp(bool)$
	$if : \Pi X : T.Exp(bool), Exp(X), Exp(X) \to Exp(X)$
nat	$0 : Exp(nat)$
	$s : Exp(nat) \to Exp(nat)$
	$It : \Pi X : T.Exp(X), (Id(X) \to Exp(X)), Exp(nat) \to Exp(X)$
$\Rightarrow$	$ab : \Pi X_1, X_2 : T.(Id(X_1) \to Exp(X_2)) \to Exp(X_1 \Rightarrow X_2)$
	$ap : \Pi X_1, X_2 : T.Exp(X_1 \Rightarrow X_2), Exp(X_1) \to Exp(X_2)$

The correspondence between the syntax of the functional language and the LF-signature Σ_{fun} is expressed by the following adequacy result (more examples of encodings and similar adequacy results can be found in [HHP87]).

Proposition 3.5 (Syntactic adequacy) *There is a translation $_^*$ of types $\tau \in T$ and expressions $e \in Exp$ of the functional language into pseudo-terms of LF s.t.:*

- $\vdash_{\Sigma_{fun}} \tau^* : T$, *whenever* $\tau \in T$
- $\Gamma^* \vdash_{\Sigma_{fun}} e^* : Exp(\tau^*)$, *whenever* $\Gamma \vdash_{fun} e : Exp[\tau]$.

Moreover, the translation induces the following bijections

- *types* $\tau \in T \iff$ *pseudo-terms* M *(up to* $\beta\eta$*-conversion) s.t.* $\vdash_{\Sigma_{fun}} M : T$
- *expression* $e \in Exp$ *(up to* α*-conversion) s.t.* $\Gamma \vdash_{fun} e : Exp[\tau] \iff$ *pseudo-terms* M *(up to* $\beta\eta$*-conversion) s.t.* $\Gamma^* \vdash_{\Sigma_{fun}} M : Exp(\tau^*)$.

LF-signatures for the other languages of Section 1 can be obtain by a simple extension of Σ_{fun} (and similar syntactic adequacy results can be proved):

- LF-signature extension Σ_{div}

  $\bot : \Pi X : T.Exp(X)$

- LF-signature extension Σ_{imp}

locations	$Loc : Type$
	$get : Loc \to Exp(nat)$
	$set : Loc, Exp(nat) \to Exp(unit)$

- LF-signature extension Σ_{exc}

exception names	$Exn : Type$
	$raise : \Pi X : T.Exn \to Exp(X)$
	$handle : \Pi X : T.Exn, Exp(X), Exp(X) \to Exp(X)$

- LF-signature extension Σ_{nd}

  $or : \Pi X : T.Exp(X), Exp(X) \to Exp(X)$

- LF-signature extension Σ_{par}

  $por : \Pi X : T.Exp(X), Exp(X) \to Exp(X)$
  $pap : \Pi X_1, X_2 : T.Exp(X_1 \Rightarrow X_2), Exp(X_1) \to Exp(X_2)$

3.4 Semantics via translation

A model M for the LF-signature Σ_{fun} induces an interpretation of the functional language via the **encoding** $_^*$ as follows:

- the interpretation $[\![\tau]\!]^{\mathrm{M}}$ of a type τ is (the only element in the family) $[\![\vdash_{\Sigma_{fun}} \tau^* : T]\!]^{\mathrm{M}}$
- the interpretation $[\![\Gamma \vdash e : Exp[\tau]]\!]^{\mathrm{M}}_{\rho}$ of a well-formed expression e in an environment ρ is $[\![\Gamma^* \vdash_{\Sigma_{fun}} e^* : Exp[\tau^*]]\!]^{\mathrm{M}}_{\rho}$.

Moreover, the standard semantics for the functional language can be recovered by a suitable choice of M.

In general, given a translation $_^* : L' \to L$ from the language/formalism L' to L, we can turn an interpretation $[\![_]\!]$ of L into an interpretation $[\![_]\!]'$ of L' by defining $[\![_]\!]'$ to be the interpretation of the translation of $_$, i.e. $[\![_^*]\!]$. We call this way of defining an interpretation **semantics via translation**.

When one considers only languages induced by a LF-signature (this is not a strong restriction because of syntactic adequacy results), translations can be described in a compact way as signature *realizations*:

- given a LF-signature Σ, let $L(\Sigma)$ be the set of all derivable judgements of the form $\Gamma \vdash_{\Sigma} J$
- given two LF-signatures Σ and Σ', a **realization** $I : \Sigma' \to \Sigma$ of Σ' in Σ is a sequence of substitutions, one for each constant declared in Σ'. The precise definition is given by induction on the length of Σ':
  - $\emptyset : \emptyset \to \Sigma$
  - $(I, x := M) : (\Sigma', x : A) \to \Sigma$ iff $I : \Sigma' \to \Sigma$ and $\vdash_{\Sigma} M : A[I]$ is derivable, where $A[I]$ is the substitution instance of A obtained by applying in parallel all substitutions in I
- given a realization $I : \Sigma' \to \Sigma$ between LF-signatures, the induced translation (also denoted by I) from $L(\Sigma')$ to $L(\Sigma)$ maps $\Gamma \vdash_{\Sigma'} J$ to $\Gamma[I] \vdash_{\Sigma} J[I]$.

In the sequel, we will describe languages by LF-signatures and translations by realizations. However, for readability we will often use some derived notation or suppress some type information, which can be recovered from the context.

4 Metalanguages for denotational semantics

In this section we specialize the technique of giving semantics via translation to the case of programming languages. The general idea is to define the denotational semantics of a programming language PL by translating it into a typed metalanguage ML. The idea is as old as denotational semantics (see [Sco93]), so the main issue is whether it can be made into a viable technique capable of dealing with complex programming languages.

Before being more specific about metalanguages, let us discuss what are the main advantages in using them to give semantics via translation:

- to reuse the same ML for translating several programming languages.

$$\begin{tikzcd} PL_1 \arrow[dr, "transl"] & & \\ \dots & ML \arrow[r, "interp"] & \mathcal{C} \\ PL_n \arrow[ur, "transl"'] & & \end{tikzcd}$$

  Here we are implicitly assuming that defining a translation from PL to ML is simpler than directly defining an interpretation of PL.

  In this case it is worth putting some effort in the study of ML. In fact, once certain properties of ML have been established (e.g. reasoning principles or computational adequacy), it is usually *easy* to transfer them to PL via the translation.

- to choose ML according to certain criteria, which are usually not met by programming languages, for instance:
  - a metalanguage which is built around few *orthogonal* concepts will be simpler to study, on the contrary programming languages often introduce *syntactic sugar* for the benefit of programmers;
  - ML may be equipped with a logic so that it can be used for formalizing reasoning principles or for translating specification languages;
  - ML may be chosen as the *internal language* for a class of categories (e.g. cartesian closed or order-enriched categories) or for a specific semantic category (e.g. the category of sets or cpos).

- to use ML for hiding details of semantic categories (see [Gor79]).

  For instance, when ML is the internal language for a class of categories, it has one intended interpretation in each of them, therefore a translation into ML will induce a variety of interpretations

$$\begin{tikzcd} & & \mathcal{C}_1 \\ PL \arrow[r, "transl"] & ML \arrow[ur, "interp"] \arrow[dr, "interp"'] & \dots \\ & & \mathcal{C}_n \end{tikzcd}$$

  even when ML has only one intended interpretation, it may be difficult to work with the semantic category directly.

A good starting point for a metalanguage is to build it on top of a fairly standard typed λ-calculus, more controversial issues are:

- whether the metalanguage should be equipped with some logic (ranging from equational logic to higher order predicate logic).

  We believe that it should, since this is the simplest way to abstract from the semantics of the metalanguage, and still have something usable to establish properties of programs.

- whether the metalanguage should be itself a programming language (i.e. to have an operational semantics).

  In fact, this may force to choose among a variety of operational semantics (CBV or CBN), to place restriction on the types (dependent types would be problematic), and to use *non-standard* equational axiomatizations (see Section 2.3).

- whether one gains something by giving semantics to a (complex) programming language PL via translation into a metalanguage ML insteads of giving the semantics of PL directly. In particular, it would be unsatisfactory if giving either the translation of PL into ML or the semantics of ML are as difficult as giving the semantics of PL directly.

  We will discuss how the monadic approach can help in structuring the translation from PL to ML by the introduction of **auxiliary notation** (see [Mos90a, Mog91])

  $$PL \xrightarrow{transl} ML(\Sigma) \xrightarrow{transl} ML$$

  and in incrementally defining the semantics of auxiliary notation (see [CM93])

  $$PL \xrightarrow{transl} ML(\Sigma_n) \xrightarrow{transl} \ldots \xrightarrow{transl} ML(\Sigma_0) \xrightarrow{transl} ML$$

Remark 4.1 The metalanguages we consider can be described in terms of LF-signatures. This may cause problems in defining their interpretation in semantic categories other than sets, since some of them (e.g. the category of cpos) are not suitable for interpreting LF. One way around this is to define the interpretation of these metalanguages directly, without going via LF. An alternative way, is to follow the approach of *Synthetic Domain Theory* (SDT) and view these semantic categories as fully embedded in a constructive set-theoretic universe, like a *topos* or *quasitopos*, in which one can interpret LF (and much more). The latter approach has considerable advantages, provided one can *avoid* (e.g. by axiomatizing the relation between the semantics category and the constructive set-theoretic universe) the extra mathematical sophistication involved in SDT models.

4.1 Typed lambda-calculi

The metalanguages we consider are extensions of the simply typed $\lambda\beta\eta$-calculus (with type variables). Formally they will be described by LF-signatures, but we will introduce and use more standard syntax and notational conventions as shorthand for the formal (and almost unreadable) notation.

4.1.1 Equational logic and extensionality

An integral part of any typed lambda-calculi is the set of equational rules, which often give a complete characterization of types (like the universal properties used in Category Theory). Therefore, we consider LF-signatures which incorporate equational logic.

- LF-signature Σ_{eq} for equational logic

propositions	$Prop : Type$
proofs	$pr : Prop \to Type$
we write ϕ for $pr(\phi)$, identifying a prop with the type of its proofs	
logical constants	
equality	$eq : \Pi X : Type.X, X \to Prop$
we write $M_1 =_\tau M_2$ or $M_1 = M_2$ for $eq(\tau, M_1, M_2)$	
axioms	
reflexivity	$\Pi X : Type.\Pi x : X.x = x$
substitutivity	$\Pi X : Type, P : X \to Prop, x, y : X.x = y, P(x) \to P(y)$

We have not introduce explicit constants for reflexivity of equality and substitutivity of equals (in propositions), since they are never needed for defining signature extensions.

One can show that the following rules are derivable (in equational logic), where A is derivable (in equational logic) formally means that there exists a term M s.t. $\vdash_{\Sigma_{eq}} M : A$:

- symmetry, i.e. $x = y \to y = x$
- transitivity, i.e. $x = y, y = z \to x = x$
- congruence for application, i.e. $x =_{\tau_1} y, f =_{(\tau_1 \to \tau_2)} g \to fx =_{\tau_2} gy$

  this cannot be extended to arbitrary Π-types, because fx and gy may have different types, nevertheless the following rule is derivable

  $f =_{(\Pi x:\tau.Xx)} g \to fx =_{Xx} gx$

Remark 4.2 Instead of equational logic one can start with the more powerful Higher Order Logic, taking as sole logical constant universal quantification $\forall : \Pi X : Type.(X \to Prop) \to Prop$. Then one defines implication $\phi_1 \supset \phi_2$ as $\forall p : \phi_1.\phi_2$ and equality $x =_\tau y$ as Leibniz' equality $\forall P : \tau \to Prop.P(x) \supset P(y)$.

Although in LF one can identify functional types with Π-types, extensionality for functions is not derivable (from Σ_{eq}). Therefore, it has to be included explicitly as additional rule.

- LF-signature extension Σ_{ext} for extensionality rules

Π-ext	$\Pi X : Type, F : X \to Type, f, g : (\Pi x : X.Fx).$ $(\Pi x : X.fx = gx) \to f = g$
$Prop$-ext	$\Pi \phi_1, \phi_2 : Prop.(\phi_1 \to \phi_2) \to (\phi_2 \to \phi_1) \to \phi_1 = \phi_2$
pr-irrel	$\Pi \phi : Prop, x, y : \phi.x = y$

  Π-extensionality generalizes extensionality for functions, $Prop$-extensionality says that logical equivalence implies equality, and proof-irrelevance says that all proofs of a given proposition are equal.

In this extension one can easily derive (using only Π-ext) extensionality for functions, i.e. $(\Pi x : \tau_1.fx =_{\tau_2} gx) \to f =_{(\tau_1 \to \tau_2)} g$.

The standard set-theoretic interpretation of Σ_{eq} (and Σ_{ext}) is as follows:

$$\begin{array}{rl} [\![Prop]\!] = & 2 \triangleq \{\emptyset, \{\emptyset\}\} \text{ the set of truth values} \\ [\![pr]\!](\phi) = & \phi \\ [\![eq]\!](\tau, x, y) = & \begin{cases} 1 & \text{if } x = y \\ 0 & \text{otherwise} \end{cases} \end{array}$$

Remark 4.3 Other interesting models of Σ_{eq}, including intensional ones (which fail to satisfy Σ_{ext}), can be obtained by interpreting LF in realizability toposes.

4.1.2 Products, sums and natural numbers

We describe products, sums and the type of natural numbers as extensions of the LF-signature Σ_{eq} for equational logic.

- LF-signature extension $\Sigma_\times$ for product types

  types
  unit $\quad 1 : Type$
  product $\quad \times : Type, Type \to Type$
  we write $\tau_1 \times \tau_2$ for $\times(\tau_1, \tau_2)$

  operations
  $\quad *: 1$
  pairing $\quad pair : \Pi X_1, X_2 : Type. X_1, X_2 \to X_1 \times X_2$
  we write $\langle M_1, M_2 \rangle$ for $pair(\tau_1, \tau_2, M_1, M_2)$
  projections $\quad \pi_i : \Pi X_1, X_2 : Type. (X_1 \times X_2) \to X_i$
  we write $\pi_i(M)$ for $\pi_i(\tau_1, \tau_2, M)$

  axioms
  $$\Pi x : 1. x = *$$
  $$\Pi X_1, X_2 : Type, x_1 : X_1, x_2 : X_2. \pi_i(\langle x_1, x_2 \rangle) = x_i$$
  $$\Pi X_1, X_2 : Type, x : X_1 \times X_2. \langle \pi_1(x), \pi_2(x) \rangle = x$$

- LF-signature extension Σ_+ for sum types

  types
  empty $\quad 0 : Type$
  sum $\quad + : Type, Type \to Type$
  we write $\tau_1 + \tau_2$ for $+(\tau_1, \tau_2)$

  operations
  $\quad 0 : \Pi X : Type. 0 \to X$
  inclusions $\quad in_i : \Pi X_1, X_2 : Type. X_i \to (X_1 + X_2)$
  we write $in_i(M)$ for $in_i(\tau_1, \tau_2, M)$
  case $\quad case : \Pi X_1, X_2, X : Type.$
  $$(X_1 \to X), (X_2 \to X) \to (X_1 + X_2) \to X$$
  we write $case(M_1, M_2, M)$ for $case(\tau_1, \tau_2, \tau, M_1, M_2, M)$

  axioms
  $$\Pi X : Type, x : 0, y : X. 0(X, x) = y$$
  $$\Pi X_1, X_2 : Type, f_1 : (X_1 \to X), f_2 : (X_2 \to X), x : X_i.$$
  $$case(f_1, f_2, in_i(x)) = f_i(x)$$
  $$\Pi X_1, X_2, X : Type, f : (X_1 \times X_2) \to X.$$
  $$case(f \circ in_1, f \circ in_2) = f$$

  we may write (case M of $x_1.M_1 | x_2.M_2$) for $case(\lambda x_1 : \tau_1. M_1, \lambda x_2 : \tau_2. M_2, M)$

- LF-signature extension Σ_N for natural numbers

  types
  NNO $\quad N : Type$

  operations
  zero $\quad 0 : N$
  successor $\quad s : N \to N$
  iteration $\quad I : \Pi X : Type. X, (X \to X) \to N \to X$
  we write $M^n(N)$ for $I(\tau, N, M, n)$

  axioms
  $$\Pi X : Type, x : X, f : (X \to X). f^0(x) = x$$
  $$\Pi X : Type, x : X, f : (X \to X), n : N. f^{s(n)}(x) = f(f^n(x))$$
  $$\Pi X : Type, x : X, f : (X \to X), h : (N \to X), n : N.$$
  $$h(0) = x, (\Pi m : N. h(sm) = f(hm)) \to h(n) = f^n(x)$$

Remark 4.4 there are stronger axiomatizations of natural numbers (and other types) as *inductive types* (see [CP88]), which introduce also additional conversion rules on pseudo-terms:

types	
NNO	$N : Type$
operations	
zero	$0 : N$
successor	$s : N \to N$
induction	$R : \Pi P : N \to Type.P(0), (\Pi n : N.P(n) \to P(sn)) \to \Pi n : N.P(n)$
axioms	
	$\Pi P : N \to Type, x : P(0), f : (\Pi n : N.P(n) \to P(sn)).$ $R(P, x, f, 0) = x$
	$\Pi P : N \to Type, x : P(0), f : (\Pi n : N.P(n) \to P(sn)), n : N.$ $R(P, x, f, s(n)) = f(n, R(P, x, f, n))$

Iteration $I(\tau, a, f)$ can be defined by induction as $R(\lambda n : N.\tau, a, \lambda n : N.f)$. The first two axioms for I follow easily from the two axioms for R, while the third axiom for I is proved by induction. Namely, given proofs $b : (h(0) = a)$ and $c : (\Pi n : N.h(sn) = f(hn))$, one can construct a proof $g : (\Pi n : N.P(n) \to P(sn))$, where $P(n)$ stands for $h(n) = f^n(a)$. Therefore, $R(P, b, g, n) : P(n)$ proves the conclusion of the third axiom for I.

The stronger axiomatization of natural numbers becomes equivalent to the weaker one only in an *extensional* version of LF, where convertibility and equality coincide. However, in this extensional version of LF it is no longer decidable whether a judgement is derivable.

4.2 Computational types and structuring

A typical problem of denotational and operational semantics is the following: when a programming language is extended, its semantics may need to be extensively redefined. For instance, in Section 1 we kept redefining the operational semantics of the functional part, every time we considered a different extension. The problem remains even when the semantics is given via translation in a typed lambda-calculus (like the one introduce so far): one would keep redefining the translation of the functional part. In [Mos90b] this problem is identified very clearly, and it is stressed how the use of **auxiliary notation** may help in making semantic definitions more reusable.

[Mog91] identifies *monads* as an important structuring device for denotational semantics (but not for operational semantics!). The basic idea is that there is a unary type constructor T, called a **notion of computation**, and terms of type $T\tau$, called a **computational type**, should be thought as programs which computes values of type τ. The interpretation of T is not fixed, it varies according to the *computational features* of the programming language under consideration. Nevertheless, one can identifies some operations (for specifying the order of evaluation) and basic properties of them, which should be common to all notions of computation. This suggests to translate a programming language PL into a metalanguage $ML_T(\Sigma)$ with computational types, where the signature Σ give additional operations (and their basic properties). In summary, the **monadic approach** to denotational semantics consists of three steps, i.e. given a programming language PL:

- identify a suitable metalanguage $ML_T(\Sigma)$, this hides the interpretation of T and Σ like an interface hides the implementation of an abstract datatype,
- define a translation of PL into $ML_T(\Sigma)$,
- construct a model of $ML_T(\Sigma)$, e.g. via translation into a metalanguage ML without computational types.

By a suitable choice of Σ, one can find a simple translation from PL to $ML_T(\Sigma)$ (in comparison to a direct translation into ML), which usually does not have to be redefined when PL is extended, and at the same time one can keep the translation of $ML_T(\Sigma)$ into ML fairly manageable.

- LF-signature extension Σ_T for computational types

  types
  T $\quad T : Type \to Type$

  operations
  val $\quad val^T : \Pi X : Type.X \to TX$
  let $\quad let^T : \Pi X_1, X_2 : Type.(X_1 \to TX_2), TX_1 \to TX_2$
  we write $val^T(M)$ for $val^T(\tau, M)$ and similarly for let^T
  we drop the superscript T, when it is clear from the context

  axioms
  $$\Pi X_1, X_2 : Type, x : X_1, f : (X_1 \to TX_2).let(f, val(x)) = f(x)$$
  $$\Pi X : Type, c : TX.let(val, c) = c$$
  $$\Pi X_1, X_2, X_3 : Type, c : TX_1, f : (X_1 \to TX_2), g : (X_2 \to TX_3).$$
  $$let(g, let(f, c)) = let(let(g) \circ f, c)$$

Notation 4.5 We introduce some derived notation for computational type:

- $[e]_T$ stands for $val^T(e)$
- $\mathrm{let}_T\ x{\Leftarrow}e_1\ \mathrm{in}\ e_2$ stands for $let^T(\lambda x : \tau_1.e_2, e_1)$
- $Tf : T\tau_1 \to T\tau_2$, where $f : \tau_1 \to \tau_2$, stands for $\lambda c : T\tau_1.\mathrm{let}\ x{\Leftarrow}c\ \mathrm{in}\ [f(x)]$
- $\mu : T^2\tau \to T\tau$ stands for $\lambda c : T^2\tau.\mathrm{let}\ x{\Leftarrow}c\ \mathrm{in}\ x$
- let $\overline{x}{\Leftarrow}\overline{e}\ \mathrm{in}\ e$ stands for let $x_1{\Leftarrow}e_1$ in $(\ldots(\mathrm{let}\ x_n{\Leftarrow}e_n\ \mathrm{in}\ e)\ldots)$
- $\langle \overline{x} \Leftarrow \overline{e}, e\rangle$ stands for let $\overline{x}, x{\Leftarrow}\overline{e}, e\ \mathrm{in}\ [\langle \overline{x}, x\rangle]$
- let $\langle \overline{x}\rangle{\Leftarrow}c\ \mathrm{in}\ e(x_1, \ldots, x_n)$ stands for let $x{\Leftarrow}c\ \mathrm{in}\ e(\pi_1(x), \ldots, \pi_n(x))$.

Intuitively, the program $[e]$ simply returns the value e, while (let $x{\Leftarrow}e_1$ in e_2) first evaluates e_1 and binds the result to x, then evaluates e_2. With the above notation the third axiom becomes

$$\mathrm{let}\ x_2{\Leftarrow}(\mathrm{let}\ x_1{\Leftarrow}e_1\ \mathrm{in}\ e_2)\ \mathrm{in}\ e_3 = \mathrm{let}\ x_1{\Leftarrow}e_1\ \mathrm{in}\ (\mathrm{let}\ x_2{\Leftarrow}e_2\ \mathrm{in}\ e_3)$$

which says that only the order of evaluation matters.

Unlike the previous types, computational types are not uniquely determined (up to isomorphism) by their equational axioms.

Example 4.6 We give a list of possible interpretations of computational types in **Set**, and indicate which *computational feature* they try to capture. It is left as an exercise to find a suitable interpretation of *val* and *let* and verify the equational axioms.

- Given a single sorted algebraic theory Th (i.e. a signature and a set of equational axioms), let $TX = |T_{Th}(X)|$, i.e. the carrier of the free Th-algebra $T_{Th}(X)$ over X. By a suitable choice of Th one can obtain interesting notions of computation, e.g.:
  - $TX = X$, which captures terminating functional programs
  - $TX = X + \{\perp\}$, which captures functional programs which may diverge, where $\perp$ represents programs which diverge

- $TX = \mathcal{P}_{fin}(X)$ the set of finite subsets of X, which captures nondeterministic programs
- $TX = (X + E)$, which captures programs with exceptions, where E is the set of exceptions
- $TX = \mu X'.\mathcal{P}_{fin}(X + X')$ the set of finite trees with leaves labeled in X, which captures parallel programs

- $TX = (X \times S)^S$, which captures imperative programs, where S is the set of states, while $TX = X^S$ captures programs that can only look at the state
- $TX = R^{(R^X)}$, which captures programs with a continuation, where R is the set of final results
- $TX = X \times N$, which captures programs with timers, where N is the set of natural numbers

One could consider variations and combinations of the examples above, e.g.

- variations for nondeterministic programs are: $TX = \mathcal{P}(X)$ the set of subsets of X, and $TX = \mathcal{P}_\omega(X)$ the set of countable subsets of X
- two possible combinations for imperative programs with exceptions are: $TX = ((X + E) \times S)^S$ and $TX = ((X \times S) + E)^S$
- $TX = \mu X'.\mathcal{P}_{fin}(X + (A \times X'))$, which captures parallel programs which interact, where A is the set of actions (in fact TX is the set of finite synchronization trees up to strong bisimulation)
- $TX = \mu X'.\mathcal{P}_{fin}((X + X') \times S)^S$, which captures parallel imperative programs (in fact TX is closely related to resumptions).

One could give further examples (in the category of cpos) based on the denotational semantics for various programming languages (see [Sch86, GS89, Mos89]).

To exemplify the use of computational types, for each of the programming languages introduced in Section 1 we define a translation into a metalanguage $ML_T(\Sigma)$ with computational types, for a suitable choice of Σ, and indicate a possible interpretation for computational types and Σ. More formally, we give a realization of the LF-signature Σ_{PL} for a programming language PL (as given in Section 3.3) in the LF-signature $\Sigma_{ML} + \Sigma_T + \Sigma$ for the metalanguage, where Σ_{ML} could be taken as $\Sigma_{eq} + \Sigma_\times + \Sigma_+ + \Sigma_N$. Incidentally, the axioms for the metalanguage could be safely ignored when defining the realization.

4.2.1 CBV translation of the functional language with divergence

We define a translation of $\Sigma_{fun} + \Sigma_{div}$ into $ML_T(\Sigma)$. For this translation the LF-signature extension Σ is

operations

divergence $\quad \bot : \Pi X : Type.T(X)$

$$
\begin{array}{rl}
T^* := & Type \\
\textbf{types} & \\
unit^* := & 1 \\
bool^* := & 1+1 \\
nat^* := & N \\
\Rightarrow^*(X_1,X_2) := & X_1 \to T(X_2) \\
Id^*(X) := & X \\
Exp^*(X) := & T(X) \\
\textbf{expressions} & \\
var^*(X,x) := & [x] \\
^ := & [*] \\
tt^* := & [in_1(*)] \\
ff^* := & [in_2(*)] \\
if^*(X,c,c_1,c_2) := & \text{let } x{\Leftarrow}c \text{ in } (\text{case } x \text{ of } _:1.c_1|_:1.c_2) \\
0^* := & [0] \\
s^*(c) := & \text{let } n{\Leftarrow}c \text{ in } [s(n)] \\
It^*(X,c_0,f,c) := & \text{let } n{\Leftarrow}c \text{ in } g^n(c_0) \text{ , where } g \triangleq \lambda c:TX.\text{let } c{\Leftarrow}x \text{ in } f(x) \\
ab^*(X_1,X_2,f) := & [f] \\
ap^*(X_1,X_2,c,c_1) := & \text{let } f{\Leftarrow}c \text{ in } (\text{let } x{\Leftarrow}c_1 \text{ in } f(x)) \\
\bot^* := & \bot
\end{array}
$$

One gets the CBV denotational semantics by taking $TX = X + \{\bot\}$, and the standard semantics (for the fragment Σ_{fun} without divergence) by taking $TX = X$.

4.2.2 CBN translation of the functional language with divergence

We define a translation of $\Sigma_{fun} + \Sigma_{div}$ into $ML_T(\Sigma)$. Also for this translation the LF-signature extension Σ is

operations
divergence $\bot : \Pi X : Type.T(X)$

We report only those clauses, where the CBN translation differs from the CBV translation

$$
\begin{array}{rl}
\textbf{types} & \\
\Rightarrow^*(X_1,X_2) := & T(X_1) \to T(X_2) \\
Id^*(X) := & T(X) \\
Exp^*(X) := & T(X) \text{ , this is like CBV} \\
\textbf{expressions} & \\
var^*(X,c) := & c \\
ap^*(X_1,X_2,c,c_1) := & \text{let } f{\Leftarrow}c \text{ in } f(c_1)
\end{array}
$$

One gets the CBN denotational semantics by taking $TX = X + \{\bot\}$, and again the standard semantics (for the fragment without divergence) by taking $TX = X$.

4.2.3 CBV translation of $\Sigma_{fun} + \Sigma_{imp}$

For this translation the LF-signature extension Σ is

types
locations $L : Type$

---

operations
lookup $lkp : L \to TN$
update $upd : L, N \to T1$

We report only those clauses which extend the CBV translation

types

$Loc^* := L$

expressions

$get^*(l) := lkp(l)$

$set^*(l,c) :=$ let $x \Leftarrow c$ in $upd(l,x)$

One gets a CBV denotational semantics for the imperative language by taking $L = Loc$, $TX = (X \times \mathsf{S})^{\mathsf{S}}$ with $\mathsf{S} = Loc \to N$, $lkp(l) = \lambda s : \mathsf{S}.\langle s(l), s\rangle$, and $upd(l,n) = \lambda s : \mathsf{S}.\langle *, s[l \mapsto n]\rangle$. More precisely, one could prove this semantics to be computationally adequate w.r.t. the CBV operational semantics, when observing "p has value tt".

4.2.4 CBV translation of $\Sigma_{fun} + \Sigma_{exc}$

For this translation the LF-signature extension Σ is

types

exceptions $E : Type$

---

operations

test $eq : E, E \to 1 + 1$

raise $raise : \Pi X : Type.E \to TX$

handle $handle : \Pi X : Type.(E \to TX), TX \to TX$

We report only those clauses which extend the CBV translation

types

$Exn^* := E$

expressions

$raise^* := raise$

$handle^*(X, n, c_1, c_2) := handle(X, (\lambda x : E.\text{case } eq(x,n) \text{ of } c_1 | raise(x)), c_2)$

One gets a CBV denotational semantics for the language with exceptions by taking $E = Exn$, $TX = (X+E)$, $eq(n_1, n_2)$ is the equality on E, $raise(n) = in_2(n)$, $handle(f, in_1(x)) = x$ and $handle(f, in_2(n)) = f(n)$.

4.2.5 CBV translation of $\Sigma_{fun} + \Sigma_{nd}$

For this translation the LF-signature extension Σ is

operations

choice $or : \Pi X : Type.TX, TX \to TX$

We report only those clauses which extend the CBV translation

expressions

$or^* := or$

One gets a CBV denotational semantics for the non-deterministic language by taking $TX = \mathcal{P}_{fin}(X)$ and $or(x_1, x_2) = x_1 \cup x_2$. More precisely, one could prove this semantics to be computationally adequate w.r.t. the CBV operational semantics, when observing "p **may** have value tt".

4.2.6 CBV translation of $\Sigma_{fun} + \Sigma_{par}$

For this translation the LF-signature extension Σ is

operations

one-step $\delta : \Pi X : Type.TX \to TX$

or-parallelism $por : \Pi X : Type.TX, TX \to TX$

and-parallelism $pand : \Pi X_1, X_2 : Type.TX_1, TX_2 \to T(X_1 \times X_2)$

we write $\delta(M)$ for $\delta(\tau, M)$ and similarly for the others

Unlike the previous examples of CBV translations, here we need both to extend and redefine the CBV translation of expressions. For brevity we consider only the redefinition of constructors associated to boolean and functional types

expressions

$$
\begin{aligned}
var^*(X,x) &:= [x] \\
tt^* &:= [in_1(*)] \\
ff^* &:= [in_2(*)] \\
if^*(X,c,c_1,c_2) &:= \text{let } x \Leftarrow c \text{ in } \delta(\text{case } x \text{ of } _ : 1.c_1 | _ : 1.c_2) \\
ab^*(X_1,X_2,f) &:= [f] \\
ap^*(X_1,X_2,c,c_1) &:= \text{let } f \Leftarrow c \text{ in } (\text{let } x \Leftarrow c_1 \text{ in } \delta(fx)) \\
pap^*(X_1,X_2,c,c_1) &:= \text{let } \langle f,x\rangle \Leftarrow pand(c,c_1) \text{ in } \delta(fx) \\
por^* &:= por
\end{aligned}
$$

One gets a *trace semantics* of the parallel language by taking $TX = \mathcal{P}_{fne}(N \times X)$ the set of finite nonempty subsets of $N \times X$, where $\langle n, x\rangle$ represents a sequence of n steps with final result x. The operations *pand* and *por* can be defined as the additive functions s.t.

$$pand(\langle n_1,x_1\rangle,\langle n_2,x_2\rangle) = \{\langle n_1+n_2,\langle x_1,x_2\rangle\rangle\}$$

$$por(\langle n_1,x_1\rangle,\langle n_2,x_2\rangle) = \{\langle n_1+m,x_1\rangle | 0 \le m \le n_2\} \cup \{\langle n_2+m,x_2\rangle | 0 \le m \le n_1\}$$

these correspond to an interleaved implementation of *pand* and *por*, a concurrent (lock-step) implementation would be better represented as follows

$$pand(\langle n_1,x_1\rangle,\langle n_2,x_2\rangle) = \{\langle max(n_1,n_2),\langle x_1,x_2\rangle\rangle\}$$

$$por(\langle n_1,x_1\rangle,\langle n_2,x_2\rangle) = \{\langle n_1,x_1\rangle | n_1 \le n_2\} \cup \{\langle n_2,x_2\rangle | n_2 \le n_1\}$$

Remark 4.7 The translation for the parallel language could be considered a refinement of that for the functional language, obtained by inserting δ in few places. One can consider many variations of the above translation, which differ only in the use of δ, each of them corresponds to a different choice of *granularity* for the steps of the operational semantics.

4.3 Translations and incremental approach

The monadic approach to denotational semantics outlined in Section 4.2 has a caveat. When the programming language PL is complex, the signature Σ identified by the monadic approach can to get fairly large, and the translation of $ML_T(\Sigma)$ into ML may become quite complicated. There is no magic, but one can alleviate the problem by adopting an **incremental approach** in defining the translation of $ML_T(\Sigma)$ into ML.

The basic idea is to adapt to this setting the techniques and modularization facilities advocated for formal software development, in particular the desired translation of $ML_T(\Sigma)$ into ML corresponds to the implementation of an abstract datatype (in some given language). In an incremental approach, the desired implementation is obtained by a sequence of steps, where each step constructs an implementation for a more complex datatype from an implementation for a simpler datatype. To make the approach viable, we need a collection of self-contained parameterized *polymorphic* modules with the following features:

- they should be polymorphic, i.e. for any signature Σ (or at least for a wide range of signatures) the module should take an implementation of Σ and construct an implementation of $\Sigma + \Sigma_{new}$, where Σ_{new} is fixed
- they could be parametric, i.e. the construction and the signature Σ_{new} may depend on parameters of some fixed signature Σ_{par}.

The polymorphic requirement can be easily satisfied, when one can implement Σ_{new} without changing the implementation of Σ (this is often the case in software development). However,

the constructions we are interested in are not *persistent*, since they involve a reimplementation of computational types, and consequently of Σ. The translations we need to consider are of the form

$$I : ML_T(\Sigma_{par} + \Sigma + \Sigma_{new}) \to ML_T(\Sigma_{par} + \Sigma)$$

where Σ_{new} are the new symbols defined by I, Σ are the old symbols *redefined* by I and Σ_{par} are the parameters of the construction (which are unaffected by I). In general I can be decomposed in

- a translation $I_{new} : ML_T(\Sigma_{par} + \Sigma_{new}) \to ML_T(\Sigma_{par})$ defining the new symbols (in Σ_{new}) and redefining computational types,
- translations $I_{op} : ML_T(\Sigma_{op}) \to ML_T(\Sigma_{par} + \Sigma_{op})$ redefining an old symbol *op* in *isolation* (consistently with the redefinition of computational types), for each possible type of symbol one may have in Σ.

Remark 4.8 An obvious question is: why do we not apply the incremental approach directly to programming languages? One reason is that we take programming languages as given, so we have no scope to make them suitable to an incremental approach. A deeper reason is that programming languages cannot separate computational types from other type constructors, so one could not delimit the part of the language which need to be redefined as sharply as in metalanguages.

We exemplify the ideas above with a variety of translations for adding one computational feature at a time and do the necessary redefinitions. For each translation we give

- LF-signature extensions Σ_{par} and Σ_{new}
- an LF-signature realization

  $$\Sigma_{ML} + \Sigma_{par} + \Sigma_T + \Sigma_{op} + \Sigma_{new} \to \Sigma_{ML} + \Sigma_{par} + \Sigma_T + \Sigma_{op}$$

  where Σ_{op} is the following LF-signature extension

  $A, B : Type$
  old $op : \Pi X : Type.A, (B \to TX) \to TX$
  we write $op(a, f)$ for $op(\tau, a, f)$

  The realization leaves unchanged the symbols in $\Sigma_{ML} + \Sigma_{par}$ and the types A and B in Σ_{op}, so they are not explicitly redefined. For simplicity the axiom part of signatures is ignored.

4.3.1 Translation I_{se} for adding side-effects

- LF-signature Σ_{par} for parameter symbols

  states $S : Type$

- LF-signature Σ_{new} for new symbols

  lookup $lkp : TS$
  update $upd : S \to T1$

- LF-signature realization

redefinition of computational types

$$\begin{aligned} T^*X &:= S \to T(X \times S) \\ val^*(X,x) &:= \lambda s : S.[\langle x, s\rangle] \\ let^*(X,Y,f,c) &:= \lambda s : S.\text{let } \langle x, s'\rangle \Leftarrow c(s) \text{ in } f(x,s') \end{aligned}$$

definition of new symbols

$$\begin{aligned} lkp^* &:= \lambda s : S.[\langle s, s\rangle] \\ upd^*(s) &:= \lambda s' : S.[\langle *, s\rangle] \end{aligned}$$

redefinition of old operation

$$op^*(X,a,f) := \lambda s : S.op(X \times S, a, \lambda b : B.f(b,s))$$

Remark 4.9 The operations lkd and upd do not fit (by a suitable instantiation of A and B) the format for a redefinable operation, as specified in Σ_{op}. However, they can be massaged to fit into the required format. In fact, given an operation of the form $op' : A \to TB$ (like lkd and upd) one can define an operation $op : \Pi X : Type.A, (B \to TX) \to TX$ of the right format by taking $op(X,a,f) = \text{let } b \Leftarrow op'(a) \text{ in } f(b)$. Moreover, op' can be recovered from op as follows $op'(a) = op(B, a, val(B))$.

4.3.2 Translation I_{ex} for adding exceptions

- LF-signature Σ_{par} for parameter symbols

  exceptions $\quad E : Type$

- LF-signature Σ_{new} for new symbols

  raise $\quad raise : \Pi X : Type.E \to TX$

  handle $\quad handle : \Pi X : Type.(E \to TX), TX \to TX$

- LF-signature realization

  redefinition of computational types

$$\begin{aligned} T^*X &:= T(X+E) \\ val^*(X,x) &:= [in_1(x)] \\ let^*(X,Y,f,c) &:= \text{let } u \Leftarrow c \text{ in } (\text{case } u \text{ of } x : X.f(x) | n : E.raise^*(Y,n)) \end{aligned}$$

  definition of new symbols

$$\begin{aligned} raise^*(X,n) &:= [in_2(n)] \\ handle^*(X,f,c) &:= \text{let } u \Leftarrow c \text{ in } (\text{case } u \text{ of } x : X.val^*(X,x) | n : E.f(n)) \end{aligned}$$

  redefinition of old operation

$$op^*(X) := op(X+E)$$

Remark 4.10 In this translation we have improperly used the symbols to be realized on the rhs of the realization. This could always be replaced with a proper realization, since we have been careful enough to avoid circular definitions.

In this translation the redefinition of op is particularly simple, and one can show that the same redefinition works for a more general type of operations, given by the following LF-signature extension

$$\begin{aligned} & F : Type \to Type \\ \text{old} \quad & op : \Pi X : Type.F(TX) \end{aligned}$$

4.3.3 Translation I_{co} for adding complexity

- LF-signature Σ_{par} for parameter symbols

  monoid $\quad M : Type$

  $\quad 1 : M$

  $\quad * : M, M \to M$

  we write $m * n$ for $*(m,n)$

- LF-signature Σ_{new} for new symbols

  cost $\quad \delta : M \to T1$

- LF-signature realization

  redefinition of computational types

$$\begin{aligned} T^*X &:= T(X \times M) \\ val^*(X,x) &:= [\langle x, 1\rangle] \\ let^*(X,Y,f,c) &:= \text{let } \langle x,m\rangle \Leftarrow c \text{ in } (\text{let } \langle y,n\rangle \Leftarrow f(x) \text{ in } [\langle y, m*n\rangle]) \end{aligned}$$

  definition of new symbols

$$\delta^*(m) := [\langle *, m\rangle]$$

  redefinition of old operation

$$op^*(X) := op(X \times M)$$

Remark 4.11 We should have added to Σ_{par} axioms saying that $(M, 1, *)$ is a monoid. In fact, without them one cannot proved that the redefinition of computational types satisfies the axioms in Σ_T.

4.3.4 Translation I_{con} for adding continuations

- LF-signature Σ_{par} for parameter symbols

  results $\quad R : Type$

- LF-signature Σ_{new} for new symbols

  abort $\quad abort : \Pi X : Type.R \to TX$

  call-cc $\quad call_{cc} : \Pi X, Y : Type.((X \to TY) \to TX) \to TX$

- LF-signature realization

  redefinition of computational types

$$\begin{aligned} T^*X &:= (X \to TR) \to TR \\ val^*(X,x) &:= \lambda k.k(x) \\ let^*(X,Y,f,c) &:= \lambda k.c(\lambda x : X.f(x)k) \end{aligned}$$

  definition of new symbols

$$\begin{aligned} abort^*(X,r) &:= \lambda k.[r] \\ call_{cc}{}^*(X,Y,f) &:= \lambda k.f(\lambda x : X.\lambda k'.abort^*(X,kx))k \end{aligned}$$

  redefinition of old operation

$$op^*(X,a,f) := \lambda k.op(R,a,\lambda b : B.f(b)k)$$

Remark 4.12 The operation $call_{cc}$ does not fit the format for a redefinable operation, as specified in Σ_{op} (nor the more general one). This translation is quite different from the others, since computational types are used very little on the lhs of the realization.

4.3.5 Other Translations

One can consider also a translation I_{res} for adding resumptions, i.e. $T^*X := \mu X'.T(X + X')$. However, in the category of sets the type expression on the rhs does not have a semantics, unless one makes strong restrictions about T. A proper treatment of resumptions can be done only after extending the metalanguages $ML_T(\Sigma)$ with suitable machinery for dealing with recursive definitions. We have not provided any translation for adding non-determinism. In fact, it seems that this (and some other notions of computation) should be used as starting point for the incremental approach.

4.3.6 Incremental approach at work

Now that we have introduced several translations for adding one computational feature at a time, one can compose them to obtain more complex translations and richer metalanguages $ML_T(\Sigma)$, as advocated by the incremental approach. We consider the realization of computational types given by some of the composite translations, and indicate the kind of programming languages for which they could be used:

- $I_{ex}(I_{se}T)X = S \to T((X+E)\times S)$

  this is suitable for imperative language with exceptions, like Standard ML

- $I_{se}(I_{ex}T)X = S \to T((X\times S)+E)$

  this is suitable for languages with recovery blocks, where an error is handled by executing some alternative piece of code starting from a checkpoint state

- $I_{se}(I_{con}T)X = (X \to S \to TR) \to S \to TR$

  this is suitable for imperative languages with goto

- $I_{con}(I_{se}T)X = (X \to S \to T(R\times S)) \to S \to T(R\times S)$

  as above but with R replaced by $R\times S$

- $I_{res}(I_{se}T)X = \mu X'.S \to T((X+X')\times S)$

  this is suitable for parallel imperative languages, when T is suitable for nondeterministic languages (e.g when T is the finite powerset)

- $I_{se}(I_{res}T)X = S \to \mu X'.T((X\times S)+X')$

  this is suitable for transaction based languages, where a change of state can happen only after the interaction with other processes has been successfully completed.

One can pursue further the analogies with formal software development. For instance, an important issue that we have ignored so far is: what properties of the symbols defined by a realization can be proved, knowing that the symbols used on the rhs of the realization have certain properties? Of course, one wants to know that the redefinition of computational types preserves at least the axioms given in Σ_T. More interesting properties of translations one can investigate are:

- Which equations for the new operations are validated by the translation?

  For instance, I_{se} validates the following equations

  $upd(s); lkp = upd(s); [s]$
  $upd(s); upd(s') = upd(s')$
  $\text{let } s \Leftarrow lkp \text{ in } upd(s) = [*]$
  $lkp; c = c$

  while I_{ex} validates

  $\text{let } x \Leftarrow raise(n) \text{ in } f(x) = raise(n)$
  $handle(f, [x]) = [x]$
  $handle(f, raise(n)) = f(n)$
  $handle(raise, c) = c$
  $handle(f_2, handle(f_1, c)) = handle(\lambda n : E.handle(f_2, f_1(n)), c)$

- Which equations for the old operation are preserved by the translation?

  For instance, one can show that all translations preserve algebraic equations such as commutativity, associativity and idempotency for a binary polymorphic operation $op : \Pi X : Type.TX, TX \to TX$.

5 Metalanguages and recursive definitions

The typed metalanguages considered so far do not allow for recursive definitions of programs or types. The extensions we consider are inspired by Synthetic Domain Theory (SDT) and Axiomatic Domain Theory (ADT). From SDT we take the idea that predomains should be part of a set-theoretic universe with an expressive logic. From ADT we take equational reasoning principles, which are valid in many categories used in Denotational Semantics, such as the category of cpos. We have not taken the more traditional approach of LCF (see [Sco93]), in which predomains come equipped with a partial order, because in some semantic categories (e.g. complete extensional PERs and effective morphisms) the order structure is not the most important one, while in others (e.g. in dI-domains and stable functions) some of the LCF axioms fail. The axiomatization is structured as follows:

- the axioms clarifying the relation between predomains and the predicative universes $Type_i$;
- axioms for lifting as the classifier of partial *computable* functions, at this point we introduce the derived notion of partial map, domain and strict map;
- axioms asserting that the category of predomains and partial maps is algebraically compact (see [Fre92]), from which one derives the existence of a fix-type and a unique uniform fix-point combinator. At this point, we mention an equivalent axiomatization, based on the fix-type and existence of special invariant objects.
- revised axioms for computational types, which take into account recursive definitions.

5.1 The category of cpos

The category **Cpo** of cpos is the *intended* model of ADT, and we use it to exemplify the ADT part of the axiomatization. In this section we recall the basic definitions, and the key properties of **Cpo** and related categories.

- A **cpo** (also called **predomain**) is a poset $\underline{X} = (X, \leq_X)$ (the subscript is omitted when clear from the context) s.t. every ω-chain $\langle x_i | i \in \omega \rangle$ has lub (least upper bound) $\sqcup_i x_i$, where

  an ω-chain is a sequence $\langle x_i \in X | i \in \omega \rangle$ s.t. $\forall i. x_i \leq x_{i+1}$, and

  the lub of a sequence/set $\langle x_i \in X | i \in I \rangle$ is the unique $x \in X$ s.t. $\forall y \in X. (\forall i \in I. x_i \leq y) \leftrightarrow x \leq y$.

- A **cppo** (also called **domain**) is a cpo $\underline{X}$ with least element $\perp_X$, where

  the least element of a cpo/poset $\underline{X}$ is the unique $x \in X$ s.t. $\forall y \in X. x \leq y$.

- A **continuous** function $f : \underline{X} \to \underline{Y}$ between cpos is a function $f : X \to Y$ which is monotonic, i.e. $x_1 \leq x_2 \supset f(x_1) \leq f(x_2)$, and preserves lubs of ω-chains, i.e. $f(\sqcup_i x_i) = \sqcup_i f(x_i)$.

- An **open** subset X' of a cpo $\underline{X}$ is an upward closed subset of X, i.e. $x_1 \in X'$ and $x_1 \leq_X x_2$ imply $x_2 \in X'$, s.t. $(\sqcup_i x_i) \in X'$ implies $\exists i. x_i \in X'$ for any ω-chain $\langle x_i | i \in \omega \rangle$. We write $\underline{X}$' for the cpo $(X', \leq_X \cap (X' \times X'))$, i.e. X' with the induced order.

- A **partial** continuous function $f : \underline{X} \rightharpoonup \underline{Y}$ between cpos is a partial function $f : X \rightharpoonup Y$ s.t. its domain X' is an open subset of $\underline{X}$ and f is continuous on $\underline{X}$', i.e. $f : \underline{X}' \to \underline{Y}$.

- A **strict** function $f : \underline{X} \circ\!\!\to \underline{Y}$ between cppos is a continuos function $f : \underline{X} \to \underline{Y}$ which preserves the least element, i.e. $f(\perp) = \perp$.

There are four categories of domains and predomains one could consider:

- **Cpo** is the category of predomains and continuous functions,
- $\mathbf{Cpo}_\perp$ is the category of predomains and partial continuous functions,
- **Cppo** is the category of domains and continuous functions,
- $\mathbf{Cpo}^\perp$ is the category of domains and strict functions.

Remark 5.1 **Cpo** is a biCCC (i.e. a cartesian closed category with finite coproducts) with NNO and the category **Set** of sets is a full sub-biCCC with NNO of **Cpo**, therefore it is the appropriate replacement for **Set**. **Cppo** is a full sub-CCC of **Cpo** and every endomorphism has a fix point, therefore it is the right setting for a fix-point combinator. $\mathbf{Cpo}_\perp$ is algebraically compact, therefore it is the right setting for solving domain equations. Moreover, the inclusion of **Cpo** in $\mathbf{Cpo}_\perp$ is bijective on objects and reflects isomorphisms, therefore a solution in $\mathbf{Cpo}_\perp$ to a domain equation is also a solution in **Cpo**. In fact, $\mathbf{Cpo}^\perp$ is isomorphic to $\mathbf{Cpo}_\perp$, but in other models of ADT the situation can be different.

5.2 The category of predomains

The main slogan of SDT is "predomains are sets". In LF this can be formalized by adding a new universe $Pdom$ included in $Type_0$ and closed under Π-types over $Type_0$, i.e.

$$\Pi\frac{\Gamma \vdash A_1 : Type_0 \qquad \Gamma, x : A_1 \vdash A_2 : Pdom}{\Gamma \vdash (\Pi x : A_1.A_2) : Pdom}$$

One could impose additional properties on $Pdom$:

- $Pdom$ is a full reflective subcategory of $Type_0$, this ensures that many universal constructions in $Pdom$ are like in $Type_0$, and so "predomain constructions are set constructions";
- $Pdom$ is an impredicative universe, i.e.

  $$\Pi\frac{\Gamma \vdash A_1 : Type_i \qquad \Gamma, x : A_1 \vdash A_2 : Pdom}{\Gamma \vdash (\Pi x : A_1.A_2) : Pdom}$$

  this is satisfiable in realizability models, and it is particularly useful when modeling programming languages with polymorphic types.

It is consistent to assume that $Prop \subset Pdom$, while $Prop \in Pdom$ is inconsistent with $Pdom$ being an impredicative universe (and with other axioms for predomains introduced in the sequel). The rest of the axiomatization of predomains is rather independent from the above assumptions. This reflects the ADT approach, which tries to identify only the essential structure and properties to give meaning to recursive definitions.

5.3 The classifier for partial computable functions

The axiomatization of predomains takes has sole additional structure a monad (L, η, let^L) on the category of predomains. Other structure introduced by the axiomatization consists of universal constructions, and therefore unique up to isomorphism. Intuitively, $L\tau$ is the type of partial (deterministic) computations, which may either diverge or produce a value in τ. The intended interpretation in **Cpo** is as follows:

- $L\underline{X}$ is lifting $\underline{X}_\perp$ of $\underline{X}$, i.e. the domain $(\{\perp\} + X, \leq)$ s.t. $\perp < in(x)$ and $in(x) \leq in(x') \leftrightarrow x \leq x'$, i.e. L adds an element $\perp$ below $\underline{X}$;

- $\eta : \underline{X} \to L\underline{X}$ is given by $\eta(x) = in(x)$;
- $let^L : (L\underline{Y})^{\underline{X}} \times L\underline{X} \to L\underline{Y}$ is given by $let^L(f, \bot) = \bot$ and $let^L(f, in(x)) = f(x)$.

Moreover, (L, η) **classifies** partial continuous maps, i.e. there is a one-one correspondence between partial continuous maps $f : \underline{X} \rightharpoonup \underline{Y}$ and continuous maps $g : \underline{X} \to L\underline{Y}$ given by

$$\begin{tikzcd}
\underline{X} \arrow[r, "g"] & L\underline{Y} \\
\underline{X}' \arrow[u, hook] \arrow[r, "f"'] \arrow[ur, phantom, "\urcorner"] & \underline{Y} \arrow[u, "\eta"']
\end{tikzcd}$$

- LF-signature extension Σ_L for lifting

types	
lifting	$L : Pdom \to Pdom$
operations	
val	$\eta : \Pi X : Pdom.X \to LX$
diverge	$\bot : \Pi X : Pdom.LX$
let	$let^L : \Pi X_1, X_2 : Pdom.(X_1 \to LX_2), LX_1 \to LX_2$
same conventions as for computational types in Section 4.2	
axioms	
$L.0$	$\Pi X_1, X_2 : Pdom, x : X_1, f : (X_1 \to LX_2).let(f, \eta(x)) = f(x)$
$L.1$	$\Pi X_1, X_2 : Pdom, f : (X_1 \to LX_2).let(f, \bot) = \bot$
$L.2$	$\Pi X_1, X_2 : Pdom, f, g : (LX_1 \to X_2).$ $f(\bot) = g(\bot), (\Pi x : X_1.f(\eta(x)) = g(\eta(x))) \to f = g$
$L.3$	$\Pi X : Pdom, x, y : X.\eta(x) = \eta(y) \to x = y$
$L.4$	$\Pi X : Pdom, f, g : (LX \to X).$ $(\Pi x : X_1.f(\eta(x)) = x), (\Pi x : X_1.g(\eta(x)) = x) \to f = g$
derived properties	
	$\Pi X : Pdom, c : LX.let(\eta, c) = c$
	$\Pi X_1, X_2, X_3 : Pdom, c : LX_1, f : (X_1 \to LX_2), g : (X_2 \to LX_3).$ $let(g, let(f, c)) = let(let(g) \circ f, c)$

Remark 5.2 We have not axiomatize that (L, η) is a partial map classifier, because it is too clumsy. We will indicate explicitly, when this extra axiom would have been useful.

Some comments about the axioms. $(L.0)$ is the only axiom for computational we need to assume, the other are derivable. $(L.1)$ says that $\bot$ represents the diverging program. $(L.2)$ says that $[\eta, \bot] : X + 1 \to LX$ is epic in $Pdom$, but not in $Type$. $(L.3)$ says that $\eta : X \to LX$ is monic in $Pdom$ as well as $Type$, this is true for any partial map classifier. $(L.4)$ says that there is at most one left inverse to $\eta : X \to LX$, this axiom is convenient but not essential.

In **Cpo** there are only three partial map classifiers: lifting $\underline{X}_\bot$, topping $\underline{X}^\top$ and $\underline{X} + 1$. Only the first two satisfy $(L.4)$. In a preorder (viewed as a category) the only monad, which satisfies (trivially) all the axioms, is the one mapping every object to the terminal one.

Various domain-theoretic notions can be defined solely in terms of L. When L is the lifting monad in **Cpo** these notions agree with those introduced already for cpos, and defined in terms of the partial order. Moreover, most of their properties, valid in **Cpo**, can be derived formally from the axioms for L.

- $(X, \alpha_X : LX \to X)$ (i.e. is a **domain**) iff

$$\begin{tikzcd} X \arrow[r, "\eta_X"] \arrow[dr, no head] & LX \arrow[d, "\alpha_X"] \\ & X \end{tikzcd}$$

  We often say that X is a domain, since by $(L.4)$ the left inverse is unique, and write Dom for the collection of domains.

- $f : X \rightharpoonup Y$ (i.e. is **partial**) iff $f : X \to LY$, therefore predomains and partial maps are the Kleisli category for the monad L.

- Given X and Y domains, $f : X \circ\!\!\to Y$ (i.e. is **strict**) iff $f(\alpha_X(\bot)) = \alpha_Y(\bot)$.

The following assertions are derivable:

- $\bot$ is the unique element satisfaying $(L.1)$, i.e.
  $\Pi X : Pdom, c : LX.(\Pi f : X \to LX.let(f, c) = c) \to c = \bot$

- If X and Y are domains and $f : X \circ\!\!\to Y$, then

$$\begin{tikzcd} L^2X \arrow[r, "L\alpha_X"] \arrow[d, "\mu_X"'] & LX \arrow[d, "\alpha_X"] \\ LX \arrow[r, "\alpha_X"'] & X \end{tikzcd} \qquad \begin{tikzcd} LX \arrow[r, "Lf"] \arrow[d, "\alpha_X"'] & LY \arrow[d, "\alpha_Y"] \\ X \arrow[r, "f"'] & Y \end{tikzcd}$$

  therefore domains and strict maps are the category of Eilenberg-Moore algebras for the monad L.

- If $X \lhd Y$ via (m, e), i.e. $m; e = \mathrm{id}_X$, and Y is a domain, then X is a domain and e is strict.

- 1, $Y_1 \times Y_2$ and $X_1 \to Y_2$ are domains, provided Y_1 and Y_2 are domains.

5.4 Algebraic compactness

The main contribution of Domain Theory to Denotational Semantics are general techniques (applicable to a variety of categories) for giving semantics to recursive definitions. Recursive definitions comes in two forms: $x = f(x)$ where f is a function on a domain, or $X \cong FX$ where F is a type constructor. The more traditional approach to recursive definitions works in the context of **Cpo**-categories (see [SP82, Ten91]). More recently Freyd has identified *algebraic compactness* as the general abstract property to give semantics to recursive definitions (see [Fre90, Fre92]).

Definition 5.3 (Freyd) *Given a category $\mathcal{C}$ we say that:*

- *$\sigma_F : F(\mu F) \to \mu F$ is a* **free** *algebra for the functor $F : \mathcal{C} \to \mathcal{C}$, when σ_F is an initial F-algebra and σ_F^{-1} is a final F-coalgebra;*

- *$\mathcal{C}$ is* **algebraically complete** *iff* every *$F : \mathcal{C} \to \mathcal{C}$ has an initial algebra*

- $\mathcal{C}$ *is* **algebraically cocomplete** *iff* every $F : \mathcal{C} \to \mathcal{C}$ *has a final coalgebra*
- $\mathcal{C}$ *is* **algebraically compact** *iff* every $F : \mathcal{C} \to \mathcal{C}$ *has a free algebra.*

Remark 5.4 The quantification over *every* endofunctor should be understood 2-categorically. Algebraic compactness implies that $0 \cong 1$, therefore it is equationally inconsistent with $\mathcal{C}$ being cartesian closed. We will consider only covariant domain equations. However, one can solve also domain equations of mixed variance, since $\mathcal{C}^{op} \times \mathcal{C}$ is algebraically compact when $\mathcal{C}$ is.

Neither **Set** nor **Cpo** are algebraically complete (or cocomplete), even w.r.t. *strong* functors, e.g. take $FX = (X \to 2) \to 2$. The category of PER (Partial Equivalence Relations over Kleene's partial applicative structure) is algebraically complete and cocomplete w.r.t. *realizable* functors.

T *h*

e category $\mathcal{C}_\perp$ of predomains and partial maps is algebraically compact w.r.t. **Cpo**-functors, where $F : \mathcal{C}_\perp \to \mathcal{C}_\perp$ is a **Cpo**-functor iff its action on morphisms is given by continuous maps $F : (LY)^X \to L(FY)^{FX}$.

Proof This follows from a general result on **Cpo**-categories (see [SP82]). More explicitly, the free F-algebra μF is given by the set (with the pointwise order)

$$\{x \in \Pi_n L(F^n 0) | \forall n : N. x_n = let(F^n !. x_{n+1}) \wedge \exists n : N, x' : F^n 0. x_n = \eta(x')\}$$

where ! is the unique map in $F0 \to L0$, i.e. $!(x') = \perp$. ∎

The following axioms say that for each endofunctor $F : \mathcal{C}_L \to \mathcal{C}_L$, F-algebra $\alpha : F\tau \rightharpoonup \tau$ and F-coalgebra $\beta : \tau \rightharpoonup F\tau$ exist unique $\alpha^\dagger$ and $\beta_\dagger$ s.t.

$$\begin{tikzcd} F(\mu F) \arrow[r, "F(\alpha^\dagger)"] \arrow[d, "\sigma_F"'] & F\tau \arrow[d, "\alpha"] \\ \mu F \arrow[r, "\alpha^\dagger"'] & \tau \end{tikzcd} \quad \text{and} \quad \begin{tikzcd} F\tau \arrow[r, "F(\beta_\dagger)"] & F(\mu F) \\ \tau \arrow[u, "\beta"] \arrow[r, "\beta_\dagger"'] & \mu F \arrow[u, "\sigma_F^{-1}"'] \end{tikzcd}$$

The precise axiomatization is quite complicated in comparison to the above description. First, one has to represent by a suitable type $Endo_L$ the collection of endofunctors on the Kleisli category $\mathcal{C}_L$, to do this one uses the cumulative hierarchy of universes. Then, one has to introduce some common conventions for functors. Finally, one can introduce the following LF-signature.

- LF-signature extension $\Sigma_{\mu\nu}$ for free algebras

types

free $\mu : Endo_L \to Pdom$

---

operations

$$\sigma : \Pi F : Endo_L . F(\mu F) \to \mu F$$
$$\sigma^{-1} : \Pi F : Endo_L . \mu F \to F(\mu F)$$
$$I : \Pi F : Endo_L, X : Pdom.(FX \to LX), (\mu F) \to LX$$
$$J : \Pi F : Endo_L, X : Pdom.(X \to L(FX)), X \to L(\mu F)$$

we write $\alpha^\dagger$ for $I(F, \tau, \alpha)$ and $\alpha_\dagger$ for $J(F, \tau, \alpha)$

---

axioms

iso $\Pi F : Endo_L, x : F(\mu F).\sigma_F^{-1}(\sigma_F x) = x$

$\Pi F : Endo_L, x : \mu F.\sigma_F(\sigma_F^{-1} x) = x$

initial $\Pi F : Endo_L, X : Pdom, \alpha : (FX \to LX), x : F(\mu F).$
$$\alpha^\dagger(\sigma_F x) = let^L(\alpha, F(\alpha^\dagger)x)$$
$\Pi F : Endo_L, X : Pdom, \alpha : (FX \to LX), f : (\mu F \to LX).$
$$(\Pi x : F(\mu F).let^L(\alpha, F(f)x)) \to f = \alpha^\dagger$$

final $\Pi F : Endo_L, X : Pdom, \beta : (X \to L(FX)), x : F(X).$
$$let^L(L\sigma_F^{-1}, \beta_\dagger x) = let^L(F(\beta_\dagger), \beta x)$$
$\Pi F : Endo_L, X : Pdom, \beta : (X \to L(FX)), f : (X \to L(\mu F)).$
$$(\Pi x : X.let^L(F\sigma_F^{-1}, fx) = let^L(Ff, \beta x)) \to f = \beta_\dagger$$

Remark 5.5 To be faithful to the definition of algebraic compactness, we should have required σ_F to be an isomorphism in $\mathcal{C}_L$ instead of $\mathcal{C}$. In fact, when L is a partial map classifier (we have not axiomatized this) one can show that: the inclusion of $\mathcal{C}$ in $\mathcal{C}_L$ reflects isomorphisms, and that an initial F-algebra in $\mathcal{C}_L$ is initial also in $\mathcal{C}$, when F cuts down to $\mathcal{C}$.

There are several functors on $\mathcal{C}$, which can be extended to $\mathcal{C}_L$

- binary products, although $\tau_1 \times \tau_2$ is not the product of τ_1 and τ_2 in $\mathcal{C}_L$
- binary sums, and $\tau_1 + \tau_2$ is also the coproduct of τ_1 and τ_2 in $\mathcal{C}_L$

a restricted form of exponentiation, which suffices for most applications to Denotational Semantics, is also available

Proposition 5.6 *If $T : \mathcal{C}_L \to \mathcal{C}_L$ factors through $\mathcal{C}^L$, i.e. FX is a domain and Ff is (the image of) a strict map, then T cuts down to $\mathcal{C}$ and*

- *the functor $_ \to T_ : \mathcal{C}^{op} \times \mathcal{C} \to \mathcal{C}$ extend to a functor from $\mathcal{C}_L^{op} \times \mathcal{C}_L$ to $\mathcal{C}_L$ which factors through $\mathcal{C}^L$;*
- *the free T-algebra $T(\mu T) \to \mu T$ in $\mathcal{C}_L$ is also a free T-algebra in $\mathcal{C}$.*

Remark 5.7 One could have assumed algebraic compactness for $\mathcal{C}^L$, i.e. the category of domains and strict maps, rather than for $\mathcal{C}_L$. In our opinion, it is preferable to use $\mathcal{C}_L$. In fact, in $\mathcal{C}^L$ one can define by recursion only domains (not predomains), and some additional type constructors, i.e. $\otimes$ (smash product) and $\circ\!\!\to$ (strict function spaces), for dealing with CBV programming languages.

It is very cumbersome to work directly with functors. However, one can define once for all functors *corresponding* to (some) type constructors, and then *canonically* associate functors to type expressions, by exploiting the usual closure properties for functors. In fact, most metalanguages for denotational semantics avoid the problem by working with a restricted form of type expressions, which are guaranteed to have a *corresponding* functor.

At this point we can introduce some constructions with universal properties, whose existence follows from algebraic compactness: the fix-type (introduce by [CP92]) and a uniform fix-point combinator (see [Sim92]).

5.4.1 The fix-type

The monad L extends to an endofunctor L' on the category $\mathcal{C}_L$ of predomains and partial maps, namely $L'f = Lf; \mu_Y; \eta_Y$ whenever $f : X \to LY$. Let $\sigma_L : L(\Omega) \to \Omega$ be the free L'-algebra in $\mathcal{C}_L$, then one can prove that it is also the free L-algebra in $\mathcal{C}$. In fact, [CP92] introduces the equivalent (but apparently weaker) notion of fix-type, which is enough for defining a uniform fix-point combinator and prove the *consistent algebraic compactness* of $\mathcal{C}_L$ and $\mathcal{C}^L$ (see [Sim92]).

- LF-signature extension Σ_Ω for the fix-type

types	
fix-type	$\Omega : Pdom$
operations	
	$\sigma : L\Omega \to \Omega$
	$I_L : \Pi X : Pdom.(LX \to X), \Omega \to X$
	$\omega : \Omega$
we write $\alpha^\dagger$ for $I_L(X, \alpha)$, $\underline{0}$ for $\sigma_L(\bot)$ and $\underline{s}$ for $\eta; \sigma_L$	
axioms	
$\Omega.1$	$\Pi X : Pdom, \alpha : (LX \to X), c : L\Omega.\alpha^\dagger(\sigma c) = \alpha(L(\alpha^\dagger)c)$
	$\Pi X : Pdom, \alpha : (LX \to X), f : (\Omega \to X).$
	$(\Pi c : L\Omega.f(\sigma c) = \alpha(L(f)c)) \to f = \alpha^\dagger$
$\Omega.2$	$\underline{s}(\omega) = \omega$
	$\Pi x : L\Omega.\underline{s}(x) = x \to x = \omega$
additional axioms	
$\Omega.3$	$\Pi X : Pdom, f, g : (\Omega \to X).(\Pi n : N.f(\underline{s}^n(\underline{0})) = g(\underline{s}^n(\underline{0}))) \to f = g$

Remark 5.8 ($\Omega.1$) says that σ_L is the initial L-algebra. ($\Omega.2$) that there exists a unique fix-point for $\underline{s}$, or equivalently a unique L-coalgebra morphism from $\eta : 1 \to L1$ to σ_L^{-1}. These axioms are enough to derive **consistent algebraic compactness** of $\mathcal{C}_L$, i.e. any initial algebra (when it exists) is also a free algebra. Therefore, in the presence of a fix-type algebraic completeness of $\mathcal{C}_L$ implies algebraic compactness.

The axiom ($\Omega.3$) involves a NNO N in $Type$, and it is not derivable from algebraic compactness. It amounts to computational induction for the fix-point combinator (defined in terms of the fix-type), which is a useful proof principle.

In **Cpo** (when L is lifting) the fix-type can be described as follows:

- Ω is the domain $\{0 < 1 < \ldots < n < n+1 < \ldots \omega\}$, i.e. the set of ordinals $\leq \omega$ with the natural order.

  There is a one-one correspondence between continuous maps $f : \Omega \to \underline{X}$ and ω-chains $\langle x_i | i \in \omega \rangle$ given by $f(n) = x_n$ for $n \in N$ and $f(\omega) = \sqcup_i x_i$.

- $\sigma : L\Omega \to \Omega$ is the map s.t. $\sigma(\bot) = 0$, $\sigma(in(n)) = n+1$ and $\sigma(in(\omega)) = \omega$.

  Moreover one has that $\underline{0} = 0$, $\underline{s}(n) = n+1$ and $\underline{s}(\omega) = \omega$. Therefore, ω is the unique fix-point of $\underline{s}$, and the unique $e : N \to \Omega$ s.t. $e(n) = \underline{s}^n(\underline{0})$ is $e(n) = n$.

- Given $\alpha : L\underline{X} \to \underline{X}$, the unique $\alpha^\dagger : \Omega \to \underline{X}$ s.t.

$$\begin{tikzcd} L\Omega \arrow[r, "\sigma"] \arrow[d, "L(\alpha^\dagger)"'] & \Omega \arrow[d, "\alpha^\dagger"] \\ L\underline{X} \arrow[r, "\alpha"'] & \underline{X} \end{tikzcd}$$

  is $\alpha^\dagger n = \underline{s}_\alpha^n(\alpha(\bot))$ and $\alpha^\dagger \omega = \sqcup_n \alpha^\dagger n$, where $\underline{s}_\alpha(x) = \alpha(\eta x)$.

- It is easy to see (because of continuity) that $e : N \to \Omega$ is an epic in **Cpo**, so the additional axiom (Ω.3) is valid.

When $L\underline{X}$ is $\underline{X}^{\top}$ or $\underline{X}+1$ the initial L-algebra exists, but $\underline{s}$ has no fix-points.

5.4.2 The fix-point combinator

Following [CP92], we use the fix-type to define a *canonical* fix-point for a map $f : X \to X$ over a domain X, namely $\mathrm{fix}(f) = f^*(\omega)$, where $f^* : \Omega \to X$ is the unique map s.t.

$$\begin{tikzcd}
L\Omega \arrow[r, "\sigma_L"] \arrow[d, "L(f^*)"'] & \Omega \arrow[d, "f^*"] & \text{i.e.} & \underline{s}^n(\underline{0}) \arrow[d, mapsto, "f^*"] \\
LX \arrow[r, "Lf"'] & LX \arrow[r, "\alpha_X"'] & X & f^n(\alpha_X(\bot))
\end{tikzcd}$$

- LF-signature extension Σ_{fix} for the fix-point combinator

operations

$\mathrm{fix} : \Pi X : Dom.(X \to X) \to X$

we write $\mathrm{fix}f$ for $\mathrm{fix}(\tau, f)$

---

axioms

fix-point $\quad \Pi X : Dom, f : (X \to X).\mathrm{fix}f = f(\mathrm{fix}f)$

uniform $\quad \Pi X_1, X_2 : Dom, f_1 : (X_1 \to X_1), f_2 : (X_2 \to X_2), h : (X_1 \circ\!\!\to X_2).$
$f_2 \circ (h = h \circ f_1 \to \mathrm{fix}f_2 = h(\mathrm{fix}f_1)$

---

additional axioms

comp-ind $\quad \Pi X_1 : Dom, X_2 : Pdom, f : (X_1 \to X_1), g_1, g_2 : (X_1 \to X_2).$
$(\Pi n : N.\phi(f^n(\bot))) \to \phi(\mathrm{fix}f)$
where $\phi(M)$ stands for $g_1(M) = g_2(M)$ and $\bot$ for $\alpha_\tau(\bot)$

Bekic $\quad \Pi X_1, X_2 : Dom, f : (X_1 \times X_2 \to X_1 \times X_2).$
$(\mathrm{fix}f) = \langle \mathrm{fix}(F_1), F_2(\mathrm{fix}(F_1))\rangle$
where $F_2(x_1) = \mathrm{fix}(\lambda x_2.\pi_2(f(\langle x_1, x_2\rangle)))$
and $F_1(x_1) = \lambda x_1.\pi_1(f(\langle x_1, F_2(x_1)\rangle))$

Remark 5.9 From the axioms for the fix-type one can prove that there exists a unique fix satisfying (fix-point) and (uniform).

(Bekic) involves products in $Pdom$ and follows from algebraic compactness. Basically, it says that to solve a (finite) system of recursive equations, it does not matter in which order one constructs a solution, the result is anyway the same.

(comp-ind) follows from (Ω.3) and closure of $Pdom$ under equalizers computed in $Type$ (which is true when $Pdom$ is a full reflective sub-category of $Type$). However a weaker form of induction, called Scott's induction

$$\phi(\bot), (\Pi x : \phi(x) \to \phi(fx)) \to \phi(\mathrm{fix}f)$$

follows from the axioms for the fix-type (and the assumption about equalizers).

In **Cpo** the fix-point combinator obtained from the fix-type Ω coincides with the least prefix-point combinator.

Definition 5.10 (Least prefix-point point) *Given a domain $\underline{X}$ and a map $f : \underline{X} \to \underline{X}$, the least x s.t. $f(x) \leq x$ exists, and it is given by* $\mathrm{fix}(f) \triangleq \sqcup_i f^i(\bot)$.

5.4.3 Special invariant objects

Instead of assuming algebraic compactness (of $\mathcal{C}_L$) and then deriving a fix-type and fix-point combinator, we can proceed in the other way. More precisely, first assume a fix-type, which is enough to prove that there exists a unique uniform fix-point combinator, then assume the existence of *special invariant objects.*

- LF-signature extension Σ_{inv} for special invariant objects

types	
invariant	$\mu : Endo_L \to Pdom$
operations	
	$\sigma : \Pi F : Endo_L.F(\mu F) \to \mu F$
	$\sigma^{-1} : \Pi F : Endo_L.\mu F \to F(\mu F)$
axioms	
iso	$\Pi F : Endo_L, x : \mu F.\sigma(\sigma^{-1}x) = x$
	$\Pi F : Endo_L, x : F(\mu F).\sigma^{-1}(\sigma x) = x$
special	$\Pi F : Endo_L.\mathrm{fix}(\lambda f : \mu F \to L(\mu F).(L\sigma) \circ (Ff) \circ \sigma^{-1}) = \eta$

Remark 5.11 The definition of special invariant object is taken from [Sim92] (which differs from Freyd's definition). In the presence of a fix-type, one can prove that for any endofunctor F over $\mathcal{C}_L$ an isomorphism $\sigma : FX \to X$ is a free F-algebra (a global property) iff it is a special invariant object (a local property). This correspondence generalizes the equivalence, established in [SP82], between O-limits and ω-colimits for an ω-chain of embeddings.

5.5 Computational types revised

Once the monad L has been introduced, it is natural to revise the axioms for computational types given in Section 4.2. More specifically, T should act on predomains rather then arbitrary types, moreover one wants the possibility of defining programs by recursion and to use T in recursive domain equations.

- LF-signature extension Σ_T for revised computational types

types	
T	$T : Pdom \to Pdom$
operations	
val	$val^T : \Pi X : Pdom.X \to TX$
let	$let^T : \Pi X_1, X_2 : Pdom.(X_1 \to TX_2), TX_1 \to TX_2$
dom	$\alpha^T : \Pi X : Pdom.L(TX) \to TX$
same conventions as in Section 4.2	
axioms	
	$\Pi X_1, X_2 : Pdom, x : X_1, f : (X_1 \to TX_2).let(f, val(x)) = f(x)$
	$\Pi X : Pdom, c : TX.let(val, c) = c$
	$\Pi X_1, X_2, X_3 : Pdom, c : TX_1, f : (X_1 \to TX_2), g : (X_2 \to TX_3).$ $let(g, let(f, c)) = let(let(g) \circ f, c)$
$T.1$	$\Pi X : Pdom, c : TX.\alpha(\eta(c)) = c$
$T.2$	$\Pi X_1, X_2 : Pdom, f : (X_1 \to TX_2).let(f, \bot) = \bot$
	where $\bot$ stands for $\alpha_\tau(\bot)$

The first three axioms are the one already introduced in Section 4.2. $(T.1)$ says that TX is a domain, therefore it has a fix-point combinator (because of the fix-type), which we can be use to define programs by recursion. $(T.2)$ says that $let(f) : TX \to TY$ is strict, this has two important consequences.

Proposition 5.12 *Let* in_X *be* $LX \xrightarrow{L(val)} L(TX) \xrightarrow{\alpha} TX$*, then*

- $in : LX \to TX$ *is the unique strict monad morphism from* L *to* T*, i.e.* $in([x]_L) = [x]_T$ *and* $in(\mathrm{let}_L\, x \Leftarrow c \,\mathrm{in}\, f(x)) = \mathrm{let}_T\, x \Leftarrow in(c) \,\mathrm{in}\, in(fx)$
- *the functor* $T : \mathcal{C} \to \mathcal{C}$ *extends to an endofunctor* T' *on* $\mathcal{C}_L$ *which factors through* $\mathcal{C}^L$*, namely* $T'f$ *is (the image of)*

$$TX \xrightarrow{Tf} T(LY) \xrightarrow{T(in)} T^2Y \xrightarrow{\mu} TY$$

The second consequence of $(T.2)$ says not only that we can use T in recursive domain equations, but we can also exponentiate it (see Proposition 5.6).

5.6 Structuring and incremental approach revised

At this point we can replace the metalanguages $ML_T(\Sigma)$ introduced in Section 4.2 with more expressive ones, which distinguish between predomains $Pdom$ and type $Type$, and moreover incorporate the monad L, algebraic compactness of $\mathcal{C}_L$, the fix-type and fix-point combinator, and the revised LF-signature for computational types. Therefore, one can give semantics to more realistic programming languages via translation into these metalanguages. On the other hand, the more complex LF-signature for computational types requires only a minor overhead to the incremental approach, since we have to check two additional axioms (the structure α^T is unique). One can easily check that all translations given in Section 4.3 preserve these two axioms.

As a sample application, we define a translation I_{res} for adding resumptions, which uses recursive types, and show a better way to give semantics to the parallel language by translating it into a metalanguage with a fix-point combinator.

5.6.1 Translation for adding resumptions

In this section we define a translation parameterized w.r.t. an endofunctor H on $\mathcal{C}$ (not necessarily on $\mathcal{C}_L$). To deal with the simple parallel language introduced in Section 1 the parameter H should be instantiated to the identity functor. However, different instances of H may be needed for more complex parallel languages. We proceed as in Section 4.3, by given LF-signature extensions Σ_{par} and Σ_{new}, and an LF-signature realization

$$\Sigma_{ML} + \Sigma_{par} + \Sigma_T + \Sigma_{op} + \Sigma_{new} \to \Sigma_{ML} + \Sigma_{par} + \Sigma_T + \Sigma_{op}$$

but now we should use the variant of LF incorporating predomains, Σ_{ML} should be $\Sigma_{eq} + \Sigma_\times + \Sigma_+ + \Sigma_N + \Sigma_L + \Sigma_{\mu\nu}$ and Σ_T is the LF-signature of Section 5.5.

- LF-signature Σ_{par} for parameter symbols

  resumptions $\quad H : Endo$
  where $Endo$ is the type of endofunctors on $\mathcal{C}$

- LF-signature Σ_{new} for new symbols

  step $\quad S : \Pi X : Pdom.H(TX) \to TX$
  case $\quad C : \Pi X, Y : Pdom.(X \to TY), (H(TX) \to TY), TX \to TY$

- LF-signature realization

redefinition of computational types

$$
\begin{aligned}
T^*X := &\quad T(\mu(G_X)) \\
&\quad \text{where } G_X : Endo_L \text{ is } G_X(X') = X + H(TX') \\
val^*(X,x) := &\quad [in_1(x)] \\
let^*(X,Y,f) := &\quad \text{fix}(\ \lambda f' : T^*X \to T^*Y. \\
&\qquad\qquad C^*(X,Y,f,\lambda x' : H(T^*X).S^*(Y,H(f')x'))\ )
\end{aligned}
$$

definition of new symbols

$$
\begin{aligned}
S^*(X,x') := &\quad [in_2(x')] \\
C^*(X,Y,f,g,c) := &\quad \text{let } u \Leftarrow c \text{ in (case } u \text{ of } x : X.val^*(x) | x' : H(TX).g(x'))
\end{aligned}
$$

redefinition of old operation

$$
op^*(X) := \quad op(X + H(T^*X))
$$

In the definition above we have been somewhat imprecise, in order to keep the translation concise and readable:

- the exact definition of the endofunctor G_τ, where $\tau : Pdom$, is

$$
G_\tau = \mathcal{C}_L \xrightarrow{\ T'\ } \mathcal{C} \xrightarrow{\ \tau + H\ } \mathcal{C} \hookrightarrow \mathcal{C}_L
$$

  where T' was defined in Proposition 5.12

- we have not explicitly written the isomorphisms $\sigma_{G_\tau} : G_\tau(\mu G_\tau) \to \mu G_\tau$ nor their inverses.

Remark 5.13 The translation I_{ex} for adding exceptions can be viewed as an instance of I_{res} obtained by instantiating H to a constant functor.

$C(f,g,c)$ performs a case analysis on the *first step* of c: when the result is in X it applies f, when the result is in $H(T^*X)$ it applies g. When $H(X') = E$ the operation $handle(X,f,c)$ defined by I_{ex} can be expressed in terms of C as $C(X,X,val^*,f,c)$.

The definition of $let^*(f)$ uses the fact that $T^*X \to T^*Y$ is a domain (as T^*Y is), and so it has a fix-point combinator.

Like for the other translations, one can investigate the equations for the new operations validated by the translation. For convenience, we divide the equations in two groups.

- Rewriting rules for C

  $C(f,g,[x]) = f(x)$
  $C(f,g,S(c)) = g(c)$
  $C(f_2,g_2,C(f_1,g_1,c)) = C(C(f_2,g_2) \circ f_1, C(f_2,g_2) \circ g_1, c)$

  these follow from the equational axioms for computational types

- special invariant property of T

  $\Pi X : Pdom.\text{fix}(\lambda h : TX \to TX.C(val, S \circ (Hh))) = (\lambda c : TX.c)$

  This follows from the special invariant property for $\mu(G_\tau)$. Using this property in combination with computational induction, one can prove that an equation $\phi(c) \equiv (g_1(c) = g_2(c))$ is true for any $c : TX$ by proving (e.g. by induction on N) that $\Pi n : N.\phi(c^n)$, where $c^n \triangleq \Phi^n(\lambda c : TX.\bot)$ and $\Phi(h) \triangleq C(val, S \circ (Hh))$.

  A consequence of this property (and the rewriting rules for C) is existence and uniqueness of $h : TX \to TY$ s.t. $h = C(f, g \circ (Hh))$, where $f : X \to TY$ and $g : H(TY) \to TY$ are fixed.

  In particular, $let(f)$ is the unique h s.t. $h = C(f, S \circ (Hh))$, and $\lambda c : TX.c$ is the unique h s.t. $h = C(val, S \circ (Hh))$.

Besides studying which equations for the old operation are preserved by the translation, one can also study whether equations for the old operation implies new equations involving the old operation and the new operations. For instance one can easily prove that:

- If before translation *op* **distributes** over *let*, i.e.

$$\Pi X, Y : Pdom, a : A, h : (B \to TX), f : (X \to TY).$$
$$let(f, op(a, h)) = op(a, \lambda b : B.let(f, hb))$$

  then after the I_{res} translation *op* distributes over C, i.e.

$$\Pi X, Y : Pdom, a : A, h : (B \to TX), f : (X \to TY), g : (H(TX) \to TY).$$
$$C(f, g, op(a, h)) = op(a, \lambda b : B.C(f, g, hb))$$

5.6.2 Revised CBV translation of $\Sigma_{fun} + \Sigma_{par}$

In Section 4.2 we gave a translation of the parallel language into a fairly ad hoc metalanguage $ML_T(\Sigma)$, where Σ was the LF-signature:

one-step	$\delta : \Pi X : Pdom.TX \to TX$
or-parallelism	$por : \Pi X : Pdom.TX, TX \to TX$
and-parallelism	$pand : \Pi X_1, X_2 : Pdom.TX_1, TX_2 \to T(X_1 \times X_2)$

This was not very satisfactory, because most of the operations in Σ are not obtained via the incremental approach. We now show that such operations are definable (by making an essential use of the fix-point combinator) from non-deterministic choice and the operations introduced by the translation I_{res} for adding resumptions when $HX' = X'$:

choice	$or : \Pi X : Pdom.TX, TX \to TX$
step	$S : \Pi X : Pdom.TX \to TX$
case	$C : \Pi X, Y : Pdom.(X \to TY), (TX \to TY), TX \to TY$

The translation makes essential use of the fix-point combinator:

$$\begin{aligned}
\delta(X) &\triangleq S(X) \\
por(X) &\triangleq \text{fix}(\ \lambda h : TX, TX \to TX.\lambda c_1, c_2 : TX. \\
&\quad or\ \ (C(val, \lambda c_1'.S(h(c_1', c_2)), c_1), \\
&\qquad C(val, \lambda c_2'.S(h(c_1, c_2')), c_2))\ ) \\
pand(X_1, X_2) &\triangleq \text{fix}(\ \lambda h : TX_1, TX_2 \to T(X_1 \times X_2).\lambda c_1 : TX_1, c_2 : TX_2. \\
&\quad or\ \ (C(\lambda x_1.S(\text{let } x_2 \Leftarrow c_2 \text{ in } [\langle x_1, x_2 \rangle]), \lambda c_1'.S(h(c_1', c_2)), c_1), \\
&\qquad C(\lambda x_2.S(\text{let } x_1 \Leftarrow c_1 \text{ in } [\langle x_1, x_2 \rangle]), \lambda c_2'.S(h(c_1, c_2')), c_2))\ )
\end{aligned}$$

Remark 5.14 The computations $por(c_1, c_2)$ and $pand(c_1, c_2)$ are quite similar. In both cases the computations c_1 and c_2 are interleaved. However, $por(c_1, c_2)$ may terminate as soon as one of the c_i has terminated, while $pand(c_1, c_2)$ terminates only when both c_i have terminated.

One can derive various *expected* properties of *por* and *pand* from the properties of S and C validated by the translation for adding resumptions and the following properties of *or*:

- algebraic properties

  $or(c_1, or(c_2, c_3)) = or(or(c_1, c_2), c_3)$
  $or(c_1, c_2) = or(c_2, c_1)$
  $or(c, c) = c$

- distributivity

  $C(f, g, or(c_1, c_2)) = or(C(f, g, c_1), C(f, g, c_2))$

We exemplify a fairly general technique to prove algebraic properties of these and other derived operations, by proving associativity of *por*:

- first prove, by induction on $m+n+p$, that

$$\Pi X : Pdom.c_1, c_2, c_3 : TX.por(c_1^m, por(c_2^n, c_3^p)) = por(por(c_1^m, c_2^n), c_3^p)$$

- then, by repeatedly applying computational induction, conclude that

$$\Pi X : Pdom.c_1, c_2, c_3 : TX.por(c_1, por(c_2, c_3)) = por(por(c_1, c_2), c_3)$$

Remark 5.15 The operations S and C, in combination with the fix-point operator, can be used to define many other forms of parallel composition or other kinds of combinators. Moreover, by instantiating H with different functors one can deal with other (parallel) languages, e.g.:

- $HX' = L \times X'$ is suitable for processes which can only synchronize with signals in L (like pure CCS);
- $HX' = X' + L \times (V \to X') + L \times V \times X'$ is suitable for processes which can communicate values in V using channels in L;
- $HX' = I \to (O \times X')$ is suitable for I/O-automata, with input alphabet I and output alphabet O.

References

[CM93] P. Cenciarelli and E. Moggi. A syntactic approach to modularity in denotational semantics. In *Proceedings of the Conference on Category Theory and Computer Science, Amsterdam, Sept. 1993*, 1993. CWI Tech. Report.

[CP88] T. Coquand and C. Paulin. Inductively defined types. In *Conference on Computer Logic*, volume 417 of *LNCS*. Springer Verlag, 1988.

[CP92] R.L. Crole and A.M. Pitts. New foundations for fixpoint computations: Fix hyperdoctrines and the fix logic. *Information and Computation*, 98, 1992.

[Fre90] P. Freyd. Recursive types reduced to inductive types. In J. Mitchell, editor, *Proc. 5th Symposium in Logic in Computer Science*, Philadelphia, 1990. I.E.E.E. Computer Society.

[Fre92] P. Freyd. Algebraically complete categories. In A. Carboni, M.C. Pedicchio, and G. Rosolini, editors, *Category Theory '90*, volume 1144 of *Lecture Notes in Mathematics*, Como, 1992. Springer-Verlag.

[Geu92] H. Geuvers. The church-rosser property for $\beta\eta$-reduction in typed lambda calculi. In A. Scedrov, editor, *Proc. 7th Symposium in Logic in Computer Science*, Santa Cruz, 1992. I.E.E.E. Computer Society.

[Gor79] M.J.C. Gordon. *The Denotational Description of Programming Languages.* Springer-Verlag, 1979.

[GS89] C. Gunter and D.S. Scott. Semantic domains. Technical Report MS-CIS-89-16, Dept. of Comp. and Inf. Science, Univ. of Pennsylvania, 1989. to appear in North Holland Handbook of Theoretical Computer Science.

[GW94] H. Geuvers and B. Werner. On the church-rosser property for expressive type systems and its consequences for their metatheory. In S. Abramsky, editor, *Proc. 9th Symposium in Logic in Computer Science*, Paris, 1994. I.E.E.E. Computer Society.

[HHP87] R. Harper, F. Honsell, and G. Plotkin. A framework for defining logics. In R. Constable, editor, *Proc. 2th Symposium in Logic in Computer Science*, Ithaca, NY, 1987. I.E.E.E. Computer Society.

[Luo94] Z. Luo. *Computation and Reasoning: A Type Theory for Computer Science*. International Series of Monographs on Computer Science. Oxford University Press, 1994.

[MC88] A.R. Meyer and S.S. Cosmodakis. Semantic paradigms: Notes for an invited lecture. In *3rd LICS Conf.* IEEE, 1988.

[Mog91] E. Moggi. Notions of computation and monads. *Information and Computation*, 93(1), 1991.

[Mos89] P. Mosses. Denotational semantics. Technical Report DAIMI-PB-276, CS Dept., Aarhus University, 1989. to appear in North Holland Handbook of Theoretical Computer Science.

[Mos90a] P. Mosses. *Action Semantics*. Cambridge Tracts in Theoretical Computer Science. Cambridge University Press, 1990.

[Mos90b] P. Mosses. Denotational semantics. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*. North Holland, 1990.

[Sch86] D.A. Schmidt. *Denotational Semantics: a Methodology for Language Development*. Allyn & Bacon, 1986.

[Sco93] D.S. Scott. A type-theoretic alternative to CUCH, ISWIM, OWHY. *Theoretical Computer Science*, 121, 1993.

[Sim92] A.K. Simpson. Recursive types in kleisli categories. available via FTP from theory.doc.ic.ac.uk, 1992.

[SP82] M. Smyth and G. Plotkin. The category-theoretic solution of recursive domain equations. *SIAM Journal of Computing*, 11, 1982.

[Ten91] R.D. Tennent. *Semantics of Programming Languages*. Prentice Hall, 1991.

Contents

1 Toy programming languages **3**
- 1.1 A simple functional language . . . 3
  - 1.1.1 Syntax . . . 3
  - 1.1.2 Typing rules . . . 3
  - 1.1.3 CBV operational semantics . . . 4
  - 1.1.4 CBN operational semantics . . . 5
- 1.2 Extensions to the functional language . . . 5
  - 1.2.1 Extension with divergence . . . 5
  - 1.2.2 Extension with mutable store . . . 5
  - 1.2.3 Extension with exception handling . . . 6
  - 1.2.4 Extension with nondeterministic choice . . . 6
  - 1.2.5 Extension with parallelism . . . 7

2 Equivalence of programs **8**
- 2.1 Observational equivalence . . . 8
- 2.2 Denotational semantics . . . 9
  - 2.2.1 Standard semantics of the functional language . . . 10
  - 2.2.2 CBV semantics of the functional language with divergence . . . 10
  - 2.2.3 CBN semantics of the functional language with divergence . . . 10
  - 2.2.4 Denotational versus observational equivalence . . . 11
- 2.3 Equational calculi . . . 12

3 Abstract syntax and encoding in LF **13**
- 3.1 The logical framework LF . . . 13
- 3.2 Set-theoretic semantics . . . 14
- 3.3 Encodings . . . 15
- 3.4 Semantics via translation . . . 17

4 Metalanguages for denotational semantics **18**
- 4.1 Typed lambda-calculi . . . 19
  - 4.1.1 Equational logic and extensionality . . . 19
  - 4.1.2 Products, sums and natural numbers . . . 21
- 4.2 Computational types and structuring . . . 22
  - 4.2.1 CBV translation of the functional language with divergence . . . 24
  - 4.2.2 CBN translation of the functional language with divergence . . . 25
  - 4.2.3 CBV translation of $\Sigma_{fun} + \Sigma_{imp}$ . . . 25
  - 4.2.4 CBV translation of $\Sigma_{fun} + \Sigma_{exc}$ . . . 26
  - 4.2.5 CBV translation of $\Sigma_{fun} + \Sigma_{nd}$ . . . 26
  - 4.2.6 CBV translation of $\Sigma_{fun} + \Sigma_{par}$ . . . 26
- 4.3 Translations and incremental approach . . . 27
  - 4.3.1 Translation I_{se} for adding side-effects . . . 28
  - 4.3.2 Translation I_{ex} for adding exceptions . . . 29
  - 4.3.3 Translation I_{co} for adding complexity . . . 29
  - 4.3.4 Translation I_{con} for adding continuations . . . 30
  - 4.3.5 Other Translations . . . 30
  - 4.3.6 Incremental approach at work . . . 31

5 Metalanguages and recursive definitions **32**
5.1 The category of cpos 32
5.2 The category of predomains 33
5.3 The classifier for partial computable functions 33
5.4 Algebraic compactness 35
5.4.1 The fix-type 38
5.4.2 The fix-point combinator 39
5.4.3 Special invariant objects 40
5.5 Computational types revised 40
5.6 Structuring and incremental approach revised 41
5.6.1 Translation for adding resumptions 41
5.6.2 Revised CBV translation of $\Sigma_{fun} + \Sigma_{par}$ 43