Monadic Encapsulation of Effects: a Revised Approach (Extended Version)

E.Moggi and F.Palumbo*
DISI, Univ. of Genova, v. Dodecaneso 35, 16146 Genova, Italy

August 20, 1999

Abstract

Launchbury and Peyton-Jones came up with an ingenious idea for embedding regions of imperative programming in a pure functional language like Haskell. The key idea was based on a simple modification of Hindley-Milner's type system. Our first contribution is to propose a more natural encapsulation construct exploiting higher-order kinds, which achieves the same encapsulation effect, but avoids the *bogus* type parameter of the original proposal. The second contribution is a stronger type safety result, namely encapsulation of strict state in higher-order lambda-calculus. We formalise the intended implementation as a very simple big-step operational semantics on untyped terms, which captures interesting implementation details not captured by the reduction semantics proposed previously.

Introduction

Launchbury and Peyton-Jones (see [LP95]) came up with an ingenious idea for encapsulating *regions* of imperative programming in a pure functional language. More specifically, they introduced a simple modification of Hindley-Milner's type system, and proved (using logical relations) that if a program is well-typed (in a restricted system, where all locations contain expressions of a fixed *base* type), then different state threads don't interfere. Subsequently Launchbury and Sabry (see [LS97]) gave a formal account of type safety (for the whole system), by *proving* subject reduction for a reduction semantics. Unfortunately, there is a bug in the proof of type safety in [LS97], which "can be traced to the complicated semantics of lazy state threads" (see [SS99]). However, Sabry and Semmelroth are able to adapt the formal developments in [LS97] to prove type safety for monadic encapsulation of strict state.

Our goals are very similar to those stated in [SS99], while we differ substantially in the methodology and strength of the results. We formalise the intended implementation as a big-step operational semantics (which will be referred to as the dynamic semantics), then we prove soundness of β-equivalence and type safety. The dynamic semantics is substantially simpler than the reduction semantics in [SS99], and we argue that it formalises certain implementation details more accurately, such as deallocation of local state and *leakage* of locations referring to a deallocated state. On the other hand, reduction semantics provides a more direct support for sound equational reasoning.

Methodology and techniques. We follow a standard approach for proving soundness of equational reasoning and type safety. The techniques used are fairly elementary and well-established:

- the dynamic behaviour of programs is specified *operationally* with an SOS;

*Research partially supported by MURST progetto cofinanziato "Tecniche formali per la specifica, l'analisi e la trasformazione di sistemi software", ESPRIT WG APPSEM.

- soundness of β-equivalence w.r.t. observational congruence is proved by a non-trivial adaptation of the *syntactic techniques* used in [Plo75];
- type systems are presented a la Church (see [Bar91, Car97]), which make explicit information not used at run-time, but easy to remove by *erasure*;
- type safety is established for an instrumented SOS, which handles also type and region information, and performs additional run-time checks.

These techniques are quite robust w.r.t. language extensions such as recursive definitions of terms and types, therefore we ignore such desirable (but technically easy) extensions.

Summary. The paper is structured as follows. Section 1 gives a big-step operational semantics (dynamic semantics) for an untyped λ-calculus with a run-construct, describing the *intended implementation*, including what constitutes a run-time error. We prove that β-conversion is sound for establishing *observational congruence* of program fragments. We have refrained from using a reduction semantics along the lines of [WF94, LS97, SS99], because it fails to capture certain low-level implementations details (see Remark 1.1 and 1.2).

Section 2 introduces a type system a la Church, basically higher-order λ-calculus with constants. The expressiveness of the type systems allows us

- to adopt a more natural encapsulation construct, which avoids the bogus type parameter introduced by [LP95, LS97, SS99] for monadic types and operations. Our construct relies on higher-order kinds, but it can be easily added to Haskell and also to SML (since only first-order kinds are needed);
- to establish a stronger type safety result, since more untyped terms are typable (but one must restrict to Haskell, to get a type inference algorithm).

Section 3 introduces an *instrumented semantics* for the pseudo-expressions of the type system a la Church. The instrumented semantics makes explicit the two-dimensional structure of the address space, typical of region-based memory management (see [TT97]), and enables a more accurate description of *improper program behaviour*. We prove type safety for the instrumented semantics, by exploiting region information in a crucial way. Then we relate the instrumented and dynamic semantics independently from well-typedness assumptions (in general the instrumented semantics does not agree with the dynamic semantics, e.g. the former does not permit to access the state of a thread with a location generated by another thread, while the latter semantics does). Finally, we derive type safety for the dynamic semantics, namely the *erasure* of a well-typed term cannot cause a run-time error.

Section 4 draws some conclusions and discusses related and future work.

Notation. We summarise some conventions for λ-calculi used throughout:

- an overline, e.g. $\overline{e}$, indicates a sequence (of terms), and $|\overline{e}|$ denotes its length
- We write $e\ \overline{e}/\lambda\overline{x}.e$ for iterated application/abstraction (similarly for other binary constructs and binders, e.g. $\overline{\tau} \to \tau$ and $\forall\overline{X}\colon K.\tau$)
- terms are treated up to α-conversion, and $e[\overline{x}\colon= \overline{e}]$ stands for parallel substitution with renaming of bound variables
- Γ stands for a typing context, i.e. a sequence of declarations $x\colon\tau$ (and $X\colon K$) without repetitions of variables; we write $\overline{x}\colon\tau$ for declaring several variables of the same type.

Acknowledgements. We would like to thanks the referees for their comments and criticisms, J.Launchbury and A.Sabry for enlightening discussions.

1 Dynamic semantics and β-conversion

We extend the pure untyped λ-calculus with a run-construct $\mathsf{run}\ \overline{x}.e$. Intuitively speaking, when an interpreter for the λ-calculus meets $\mathsf{run}\ \overline{x}.e$, it calls a *monadic interpreter*, which interprets e in an environment where the variables $\overline{x}$ are bound to *internal implementations*, and then evaluates the term returned by the monadic interpreter. One can envisage different monadic interpreters, to which the same *abstract code* $\overline{x}.e$ could be passed.

To define the dynamic semantics for such a language, we introduce auxiliary semantic domains and extend the syntax for terms with additional constants.

- names $m, n \in \mathsf{N}$, e.g. natural numbers, for naming *locations*
- term operators $o \in \mathsf{Op} \stackrel{\Delta}{=} \{ret, do, new, get, set\}$ with the following arities

$$\#ret = 1 \quad \#do = 2 \quad \#new = 1 \quad \#get = 1 \quad \#set = 2$$

- constants $c \in \mathsf{Const}: := o \mid \ell_m$
- terms $e \in \mathsf{E}: := c \mid x \mid \lambda x.e \mid e_1\ e_2 \mid \mathsf{run}\ \overline{x}.e$ where $|\overline{x}| = |\mathsf{Op}|$;
  we write E_0 for the set of closed terms
- values $v \in \mathsf{Val}: := \lambda x.e \mid \ell_m \mid o\ \overline{e}$ where $|\overline{e}| \leq \#o$
- stores $\mu \in \mathsf{S} \stackrel{\Delta}{=} \mathsf{N} \stackrel{fin}{\to} \mathsf{E}_0$, i.e. partial maps from N to E_0 with finite domain;
  we write Loc_μ for the set $\{\ell_m | m \in dom(\mu)\}$ of locations in μ
- descriptions $d \in \mathsf{D}: := e \mid (\mu, e) \mid err$.

The dynamic semantics is given in terms of two mutually recursive interpreters, which evaluate **closed terms** and may also raise run-time errors. We have two evaluation judgements:

- $e \Longrightarrow v \mid err$ says that evaluation of $e \in \mathsf{E}_0$ by the pure interpreter returns $v \in \mathsf{Val}_0$ (or raises an error);
- $\mu, e \Longrightarrow \mu', e' \mid err$ says that evaluation of $e \in \mathsf{E}_0$ in local store μ by the monadic interpreter returns $e' \in \mathsf{E}_0$ and changes the store to μ' (or raises an error).

Figure 1 gives the evaluation rules for the dynamic semantics.

Remark 1.1 On pure λ-terms pure evaluation coincides with CBN (Call-by-Name) evaluation. Pure evaluation treats locations ℓ_m as values (like *nil* for the empty list) and term operators o as term-constructors (like *cons*). Monadic evaluation proceeds very much like evaluation for an imperative language, but sequencing and termination of monadic evaluation are made explicit through *do* and *ret*. Moreover, monadic evaluation calls pure evaluation, whenever it needs the value of a term. The dynamic semantics is non-deterministic, since we don't fix a deterministic strategy for choosing an $m \notin dom(\mu)$. Finally, evaluation is rather permissive:

- locations referring to a deallocated state can be returned as values, e.g. $\mathsf{run}\ \overline{x}.x_{new}\ 0 \Longrightarrow \ell_m$ where x_{new} is the variable in $\overline{x}$ which get bound to *new* and m can be any name;
- *new* can be implemented by a local *name server*, which does not require communication with other threads, e.g. $\mathsf{run}\ \overline{x}.x_{new}\ \ell_m \Longrightarrow \ell_n$ where n can be any name (including m);
- there is no check on whether a location generated by a thread is used to access the state of another, e.g. $\mathsf{run}\ \overline{y}.y_{do}\ (y_{new}\ 1)\ (\lambda\_.y_{get}\ (\mathsf{run}\ \overline{x}.x_{new}\ 0))$ may evaluate to 1 or err. 1 is returned when the name servers for the two threads choose the same name, while the run-time error occurs if they choose different names.

Pure Evaluation

$$v \Longrightarrow v \qquad \frac{e_1 \Longrightarrow \lambda x.e \quad e[x{:}=e_2] \Longrightarrow v}{e_1\ e_2 \Longrightarrow v}$$

$$\frac{e_1 \Longrightarrow o\ \overline{e}}{e_1\ e_2 \Longrightarrow o\ \overline{e}\ e_2}\ |\overline{e}| < \#o \qquad \frac{\emptyset, e[\overline{x}{:}=\mathsf{Op}] \Longrightarrow \mu, e' \quad e' \Longrightarrow v}{\mathsf{run}\ \overline{x}.e \Longrightarrow v}$$

Monadic Evaluation

$$\frac{e \Longrightarrow ret\ e'}{\mu, e \Longrightarrow \mu, e'} \qquad \frac{e \Longrightarrow do\ e_0\ e_1 \quad \mu_0, e_0 \Longrightarrow \mu_1, e_0' \quad \mu_1, e_1\ e_0' \Longrightarrow \mu_2, e'}{\mu_0, e \Longrightarrow \mu_2, e'}$$

$$\frac{e \Longrightarrow new\ e_0}{\mu, e \Longrightarrow \mu\{m = e_0\}, \ell_m}\ m \notin dom(\mu)$$

$$\frac{e \Longrightarrow get\ e_0 \quad e_0 \Longrightarrow \ell_m}{\mu, e \Longrightarrow \mu, e'}\ e' = \mu(m) \qquad \frac{e \Longrightarrow set\ e_0\ e_1 \quad e_0 \Longrightarrow \ell_m}{\mu, e \Longrightarrow \mu\{m = e_1\}, \ell_m}\ m \in dom(\mu)$$

Pure and Monadic Run-Time Errors

Besides the obvious rules for error propagation, there are the following rules for error generation

$$\frac{e_1 \Longrightarrow v}{e_1\ e_2 \Longrightarrow err}\ v \not\equiv \lambda x.e \text{ or } v \not\equiv o\ \overline{e} \text{ with } |\overline{e}| < \#o \qquad \frac{e \Longrightarrow v}{\mu, e \Longrightarrow err}\ v \not\equiv o\ \overline{e} \text{ with } |\overline{e}| = \#o$$

$$\frac{e \Longrightarrow get\ e_0 \quad e_0 \Longrightarrow v}{\mu, e \Longrightarrow err}\ v \notin \mathsf{Loc}_\mu \qquad \frac{e \Longrightarrow set\ e_0\ e_1 \quad e_0 \Longrightarrow v}{\mu, e \Longrightarrow err}\ v \notin \mathsf{Loc}_\mu$$

Figure 1: Evaluation Rules for Dynamic Semantics

Remark 1.2 [LP95, LS97, SS99] adopt a run-construct $\mathsf{run}\ e$, instead of our $\mathsf{run}\ \overline{x}.e$. However, it is straightforward to accommodate $\mathsf{run}\ e$ in our dynamic semantics by adding the evaluation rule $\dfrac{\emptyset, e \Longrightarrow \mu, e' \quad e' \Longrightarrow v}{\mathsf{run}\ e \Longrightarrow v}$.

In the untyped language the two run-constructs are interdefinable: $\mathsf{run}\ \overline{x}.e \equiv \mathsf{run}\ (e[\overline{x}{:}=\mathsf{Op}])$ and $\mathsf{run}\ e \equiv \mathsf{run}\ \overline{x}.e$ provided $\overline{x}$ do not occur free in e. However, they are no longer interdefinable in the typed language.

There are important differences between our dynamic semantics and the reduction semantics of [LS97, SS99]. The latter introduce an auxiliary construct $\mathsf{sto}(\mu, e)$ (and $\mathsf{sto}(\overline{\mu}, e)$ in [SS99]), which roughly speaking corresponds to the configuration (μ, e) for our monadic interpreter. However, in $\mathsf{sto}(\mu, e)$ the locations in μ are considered bound variables (while for us they are constants), therefore one has that

- reduction $e \longrightarrow e'$ has to be defined on open terms (and thus it is convenient to identify terms modulo α-conversion), while our dynamic semantics is given on closed terms;
- the reduction $\mathsf{sto}(\mu, ret\ e) \longrightarrow e$ makes no sense when $\mathsf{sto}(\mu, e)$ is a binder (unless no locations in μ occur in e), so one has to postpone deallocation of the local store (in a lazy state semantics there are other reasons why one may have to postpone deallocation);
- the reduction $\mathsf{sto}(\mu, do\ (new\ e_0)\ e_1) \longrightarrow \mathsf{sto}(\mu\{m = e_0\}, e_1\ \ell_m)$ is correct only if ($m \notin dom(\mu)$ and) ℓ_m is not free in e and μ, so the name server has to look at the whole term.

One can device a reduction semantics providing a faithful account of store deallocation, but other implementation details (e.g. name generation) seem beyond the descriptive abilities of reduction semantics. On the other hand, reduction semantics is *directly related* to sound equational reasoning.

We briefly discuss how to extend the dynamic semantics with a fix-point operation fix and a test

$$\frac{\emptyset, e[\overline{x}, x := \mathsf{Op}, eq] \Longrightarrow \mu, e' \quad e' \Longrightarrow v}{\mathsf{run}\ \overline{x}, x.e \Longrightarrow v} \qquad \frac{e_1 \Longrightarrow fix \quad e_2\ (fix\ e_2) \Longrightarrow v}{e_1\ e_2 \Longrightarrow v} \qquad \frac{e \Longrightarrow eq}{e\ e_0 \Longrightarrow eq\ e_0}$$

$$\frac{e \Longrightarrow eq\ e_0 \quad e_0 \Longrightarrow \ell_m \quad e_1 \Longrightarrow \ell_m}{e\ e_1 \Longrightarrow \lambda x, y.x} \qquad \frac{e \Longrightarrow eq\ e_0 \quad e_0 \Longrightarrow \ell_m \quad e_1 \Longrightarrow \ell_n}{e\ e_1 \Longrightarrow \lambda x, y.y}\ m \neq n \qquad \frac{e \Longrightarrow eq\ e_0 \quad e_i \Longrightarrow v}{e\ e_1 \Longrightarrow err}\ v \notin \mathsf{Loc}$$

Figure 2: Additional Evaluation Rules in the presence of fix and eq

for equality of locations eq. However, we do not consider these extensions in the subsequent formal developments, but they are interesting (though unproblematic) for the following reasons:

- fix is *independent* from the run-construct (like many other extensions one could envisage);
- eq is a peculiar operation, in fact its type involves the type constructor for locations (so it should be introduced by the run-construct), but its result type does not involve the type constructor for computational types (so it should be evaluated by the pure interpreter).

The changes to the syntactic categories are as follows

- constants $c \in \mathsf{Const} ::= fix \mid eq \mid o \mid \ell_m$, i.e. we have added two new constants; we write Loc for the set $\{\ell_m | m \in \mathsf{N}\}$ of locations
- untyped terms $e \in \mathsf{E} ::= c \mid x \mid \lambda x.e \mid e_1\ e_2 \mid \mathsf{run}\ \overline{x}, x.e$ where $|\overline{x}| = |\mathsf{Op}|$, thus we modify the monadic interpreter so that it binds the new variable x to eq
- values $v \in \mathsf{Val} ::= \lambda x.e \mid fix \mid eq \mid eq\ e \mid \ell_m \mid o\ \overline{e}$ where $|\overline{e}| \leq \#o$.

Figure 2 gives the additional evaluation rules, and that for the modified run.

The dynamic semantics induces an observational congruence $\approx$ on open terms. We have decided to observe as much as allowed by the dynamic semantics, i.e. both successful termination and run-time errors. On the other hand, the given dynamic semantics does not allow to observe whether the evaluation of a term e *must always* terminate. Therefore, our observational congruence suffers of the same weaknesses as may testing (see [dH84]).

Definition 1.3 *Given $e \in \mathsf{E}_0$, the set $O(e) \subseteq \{ok, err\}$ of possible observations is s.t. $err \in O(e) \stackrel{\Delta}{\Longleftrightarrow} e \Longrightarrow err$ and $ok \in O(e) \stackrel{\Delta}{\Longleftrightarrow} \exists v \in \mathsf{Val}.e \Longrightarrow v$.*

Two terms $e_1, e_2 \in \mathsf{E}$ are **observationally equivalent** *($e_1 \approx e_2$ for short) $\stackrel{\Delta}{\Longleftrightarrow} O(C[e_1]) = O(C[e_2])$ for every closing context $C[\_]$.*

The following result says that β-conversion is a safe transformation regardless of well-formedness of terms. Of course, if one would consider only well-formed terms and contexts, then the resulting observational congruence would identify more (well-formed) terms, and more transformations could be proved safe.

Theorem 1.4 (soundness of β) *If $e_1 \longrightarrow_\beta e_2$, then $e_1 \approx e_2$.*

Proof. The proof adapts the technique used by Plotkin (see [Plo75]) for proving that $\lambda_N \vdash e_1 = e_2$ implies $e_1 \approx_N e_2$. There are some additional complications due to the possibility of run-time errors, the non-determinism of evaluation, and the fact that β-reduction does not subsume evaluation. Technical details of the proof are given in Appendix A. ∎

2 Type system a la Church

We formalise the type system as an higher-order λ-calculus a la Church (see [Bar91, Geu93]). For convenience, we distinguish between constants (declared in signatures) and variables (declared in contexts). The type system uses the following syntactic categories:

- constructor constants $C \in \mathsf{CONST}$ and constructor variables $X \in \mathsf{VAR}$, (term) constants $c \in \mathsf{Const}$ and (term) variables $x \in \mathsf{Var}$;

  these sets are assumed to be infinite and mutually disjoint
- kinds $K \in \mathsf{K}::= * \mid K_1 \to K_2$; $*$ is the kind of all types
- constructors $u, \tau \in \mathsf{U}::= C \mid X \mid \tau_1 \to \tau_2 \mid \forall X{:}K.\tau \mid \Lambda X{:}K.u \mid u_1[u_2]$;

  we write τ for a constructor that is expected to have kind $*$
- terms $e \in \mathsf{E}^*::=$ $c \mid x \mid \lambda x{:}\tau.e \mid e_1\ e_2 \mid \Lambda X{:}K.e \mid e[u] \mid$ $\mathsf{run}\ X_M, X_R, x_{ret}, x_{do}, x_{new}, x_{get}, x_{set}.e\ \mathsf{with}\ \tau$
- signatures $\Sigma \in \mathsf{Sig}::= \emptyset \mid \Sigma, C{:}K \mid \Sigma, c{:}\tau$
- contexts $\Delta, \Gamma \in \mathsf{Ctx}::= \emptyset \mid \Gamma, X{:}K \mid \Gamma, x{:}\tau$

Notation 2.1 There are several notions of reduction one may consider:

- $(\Lambda X{:}K.u')[u] \longrightarrow^u_\beta u'[X{:=}u]$ and $(\Lambda X{:}K.u[X]) \longrightarrow^u_\eta u$ when $X \notin \mathrm{FV}(u)$
- $(\Lambda X{:}K.e)[u] \longrightarrow^\forall_\beta e[X{:=}u]$ and $(\Lambda X{:}K.e[X]) \longrightarrow^\forall_\eta e$ when $X \notin \mathrm{FV}(e)$
- $(\lambda x{:}\tau.e')\ e \longrightarrow^e_\beta e'[x{:=}e]$ and $(\lambda x{:}\tau.e\ x) \longrightarrow^e_\eta e$ when $x \notin \mathrm{FV}(e)$

The only notion of reduction needed for defining the type system is $\longrightarrow^u_{\beta\eta}$, i.e. the union of $\longrightarrow^u_\beta$ and $\longrightarrow^u_\eta$. With some abuse of notation we denote with $\longrightarrow^u_{\beta\eta}$ also the reduction induced by the notion of reduction $\longrightarrow^u_{\beta\eta}$ (on constructors), i.e. the *compatible closure* of $\longrightarrow^u_{\beta\eta}$ (on any syntactic category). Moreover, we denote with $=^u_{\beta\eta}$ the reflexive, symmetric and transitive closure of the reduction $\longrightarrow^u_{\beta\eta}$ (and similarly for other notions of reduction).

Figure 3 gives the rules of the type system for deriving judgements of the form

- $\Sigma \vdash$, i.e. Σ is a well-formed signature
- $\Sigma; \Gamma \vdash$, i.e. Γ is a well-formed context
- $\Sigma; \Gamma \vdash u{:}K$, i.e. u is a well-formed constructor of kind K
- $\Sigma; \Gamma \vdash e{:}\tau$, i.e. e is a well-formed term of type τ.

2.1 Intermezzo: types for encapsulation

This section, which is irrelevant for the following developments, compares our proposal for encapsulation of effects with the original one and existential types.

Instead of the run-with construct we could have introduced a constant *run* of a suitable type, and define ($\mathsf{run}\ \overline{X}, \overline{x}.e\ \mathsf{with}\ \tau$) as derived notation. For the instrumented semantics it is more convenient to take run-with as primitive. In this section we describe the alternative presentation in terms of *run*, since it is easier to relate to *runST* of [LP95] and existential types (see [MP88]).

For conciseness, we use the derived notation in Figure 4 defined by induction on the structure of a context Δ or a sequence ρ, where sequences are given by the BNF $\rho, \theta \in \mathsf{Seq}::= \emptyset \mid \rho,\ u \mid \rho,\ e$.

Signatures and Contexts : $\Sigma \vdash$ and $\Sigma;\Gamma \vdash$

$$\emptyset\text{-}\Sigma\ \frac{}{\emptyset \vdash} \qquad C\text{-}\Sigma\ \frac{\Sigma \vdash}{\Sigma, C\colon K \vdash}\ C \text{ fresh in } \Sigma \qquad c\text{-}\Sigma\ \frac{\Sigma;\emptyset \vdash \tau\colon *}{\Sigma, c\colon \tau \vdash}\ c \text{ fresh in } \Sigma$$

$$\emptyset\text{-}\Gamma\ \frac{\Sigma \vdash}{\Sigma;\emptyset \vdash} \qquad X\text{-}\Gamma\ \frac{\Sigma;\Gamma \vdash}{\Sigma;\Gamma, X\colon K \vdash}\ X \text{ fresh in } \Gamma \qquad x\text{-}\Gamma\ \frac{\Sigma;\Gamma \vdash \tau\colon *}{\Sigma;\Gamma, x\colon \tau \vdash}\ x \text{ fresh in } \Gamma$$

Constructors : $\Sigma;\Gamma \vdash u\colon K$

$$C\ \frac{\Sigma;\Gamma \vdash}{\Sigma;\Gamma \vdash C\colon K}\ C\colon K \in \Sigma \qquad X\ \frac{\Sigma;\Gamma \vdash}{\Sigma;\Gamma \vdash X\colon K}\ X\colon K \in \Gamma$$

$$\Lambda\ \frac{\Sigma;\Gamma, X\colon K_1 \vdash u\colon K_2}{\Sigma;\Gamma \vdash \Lambda X\colon K_1.u\colon (K_1 \to K_2)} \qquad \mathit{app}\ \frac{\Sigma;\Gamma \vdash u_1\colon K_1 \to K_2 \quad \Sigma;\Gamma \vdash u_2\colon K_1}{\Sigma;\Gamma \vdash u_1[u_2]\colon K_2}$$

$$\forall\ \frac{\Sigma;\Gamma, X\colon K \vdash \tau\colon *}{\Sigma;\Gamma \vdash (\forall X\colon K.\tau)\colon *} \qquad \to\ \frac{\Sigma;\Gamma \vdash \tau_1\colon * \quad \Sigma;\Gamma \vdash \tau_2\colon *}{\Sigma;\Gamma \vdash (\tau_1 \to \tau_2)\colon *}$$

Term : $\Sigma;\Gamma \vdash e\colon \tau$

$$c\ \frac{\Sigma;\Gamma \vdash}{\Sigma;\Gamma \vdash c\colon \tau}\ c\colon \tau \in \Sigma \quad x\ \frac{\Sigma;\Gamma \vdash}{\Sigma;\Gamma \vdash x\colon \tau}\ x\colon \tau \in \Gamma \quad \text{conv}\ \frac{\begin{array}{c}\Sigma;\Gamma \vdash e\colon \tau_1\\ \Sigma;\Gamma \vdash \tau_2\colon *\end{array}}{\Sigma;\Gamma \vdash e\colon \tau_2}\ \tau_1 =^{u}_{\beta\eta} \tau_2$$

$$\to\text{I}\ \frac{\Sigma;\Gamma, x\colon \tau_1 \vdash e\colon \tau_2}{\Sigma;\Gamma \vdash \lambda x\colon \tau_1.e\colon (\tau_1 \to \tau_2)} \qquad \to\text{E}\ \frac{\Sigma;\Gamma \vdash e_1\colon (\tau_1 \to \tau_2) \quad \Sigma;\Gamma \vdash e_2\colon \tau_1}{\Sigma;\Gamma \vdash e_1\ e_2\colon \tau_2}$$

$$\forall\text{I}\ \frac{\Sigma;\Gamma, X\colon K \vdash e\colon \tau}{\Sigma;\Gamma \vdash \Lambda X\colon K.e\colon (\forall X\colon K.\tau)} \qquad \forall\text{E}\ \frac{\Sigma;\Gamma \vdash e\colon (\forall X\colon K.\tau) \quad \Sigma;\Gamma \vdash u\colon K}{\Sigma;\Gamma \vdash e[u]\colon \tau[X\colon= u]}$$

$$\text{run}\ \frac{\Sigma;\Gamma \vdash \tau\colon * \quad \Sigma;\Gamma, \Gamma_M \vdash e\colon X_M[\tau]}{\Sigma;\Gamma \vdash (\mathsf{run}\ \overline{X}, \overline{x}.e\ \mathsf{with}\ \tau)\colon \tau}\ \overline{X}, \overline{x} \equiv |\Gamma_M|$$

where $\Gamma_M \equiv X_M, X_R\colon * \to *$,
$x_{ret}\colon \forall X\colon *.X \to X_M[X]$,
$x_{do}\colon \forall X, Y\colon *.X_M[X], (X \to X_M[Y]) \to X_M[Y]$,
$x_{new}\colon \forall X\colon *.X \to X_M[X_R[X]]$,
$x_{get}\colon \forall X\colon *.X_R[X] \to X_M[X]$,
$x_{set}\colon \forall X\colon *.X_R[X], X, \to X_M[X_R[X]]$
and $|\Gamma_M| \equiv X_M, X_R, x_{ret}, x_{do}, x_{new}, x_{get}, x_{set}$

Figure 3: Formation rules for type system

	Δ	$\emptyset$	$X\colon K',\ \Delta'$	$x\colon \tau',\ \Delta'$
	ρ	$\emptyset$	$u',\ \rho'$	$e',\ \rho'$
derived notation for				
kinds	$\Delta \to K$	K	$K' \to \Delta' \to K$	$\Delta' \to K$
constructors	$\Lambda\Delta.u$	u	$\Lambda X\colon K'.\Lambda\Delta'.u$	$\Lambda\Delta'.u$
	$u(\rho)$	u	$u[u'](\rho')$	$u(\rho')$
	$\forall\Delta.\tau$	τ	$\forall X\colon K'.\forall\Delta'.\tau$	$\tau' \to \forall\Delta'.\tau$
terms	$\Lambda\Delta.e$	e	$\Lambda X\colon K'.\Lambda\Delta'.e$	$\lambda x\colon \tau'.\Lambda\Delta'.e$
	$e(\rho)$	e	$e[u'](\rho')$	$e\ e'(\rho')$
signatures/contexts	$\Pi\Gamma.\Delta$	$\emptyset$	$X\colon \Gamma \to K',\ \Pi\Gamma.\Delta'[X\colon= X(\lvert\Gamma\rvert)]$	$x\colon \forall\Gamma.\tau',\ \Pi\Gamma.\Delta'$
sequences	$\lvert\Delta\rvert$	$\emptyset$	$X,\ \lvert\Delta'\rvert$	$x,\ \lvert\Delta'\rvert$
	$\lambda\Gamma.\rho$	$\emptyset$	$\Lambda\Gamma.u',\ \lambda\Gamma.\rho'$	$\Lambda\Gamma.e',\ \lambda\Gamma.\rho'$
	$\rho(\theta)$	$\emptyset$	$u'(\theta),\ \rho'(\theta)$	$e'(\theta),\ \rho'(\theta)$

The two top lines recall the three cases of the inductive definitions of Δ and ρ, while the others introduce notation defined by induction on the structure of Δ or ρ.
For instance, line 3 defines $\Delta \to K$ (first entry), namely the last three entries give the three cases of the inductive definition of $\Delta \to K$, i.e.
$\emptyset \to K \stackrel{\Delta}{\equiv} K$, $(X\colon K',\ \Delta') \to K \stackrel{\Delta}{\equiv} K' \to \Delta' \to K$ and $(x\colon \tau',\ \Delta') \to K \stackrel{\Delta}{\equiv} \Delta' \to K$.

Figure 4: Derived notation

Run-with as a constant. We can replace the run-with construct with a constant *run* of type $\forall X\colon\ast.(\forall\Gamma_M.X_M[X]) \to X$, where

$$\begin{aligned}\Gamma_M \equiv\ & X_M, X_R\colon \ast \to \ast\ ,\\ & x_{ret}\colon \forall X\colon\ast.X \to X_M[X]\ ,\\ & x_{do}\colon \forall X, Y\colon\ast.X_M[X], (X \to X_M[Y]) \to X_M[Y]\ ,\\ & x_{new}\colon \forall X\colon\ast.X \to X_M[X_R[X]]\ ,\\ & x_{get}\colon \forall X\colon\ast.X_R[X] \to X_M[X]\ ,\\ & x_{set}\colon \forall X\colon\ast.X_R[X], X \to X_M[X_R[X]]\end{aligned}$$

More precisely, one can define run-with in terms of *run* and conversely:

$$\begin{aligned}(\mathsf{run}\ X_M, X_R, x_{ret}, x_{do}, x_{new}, x_{get}, x_{set}.e\ \mathsf{with}\ \tau) &\stackrel{\Delta}{\equiv} run\ [\tau]\ (\Lambda\Gamma_M.e)\\ run &\stackrel{\Delta}{\equiv} (\Lambda X\colon\ast.\ \lambda x\colon(\forall\Gamma_M.X_M[X]).\ \mathsf{run}\ \lvert\Gamma_M\rvert.x\ (\lvert\Gamma_M\rvert)\ \mathsf{with}\ X)\end{aligned}$$

A more appealing way of writing the type for *run* is $\forall X\colon\ast.(M[X]) \to X$, where the type constructor M is defined as $M \equiv \ \ \Lambda X\colon\ast.\forall\Gamma_M.X_M[X] \ \ \colon\ast \to \ast$, intuitively $M[X]$ is the type of *monadic code* (in higher-order abstract syntax).

One can almost define an *initial algebra* for the *specification* Γ_M. In second-order λ-calculus one can *represent* initial algebras for algebraic specifications. For instance, given the specification $\Gamma_N \equiv X_N\colon\ast,\ x_{zero}\colon X_N,\ x_{succ}\colon X_N \to X_N$ of the natural numbers, one can define the type $N \equiv \forall\Gamma_N.X_N$ (of Church's numerals), which has the structure of a *weakly* initial algebra (see [RP93]). The specification Γ_M is not algebraic, but one can mimic the definition of the initial algebra given in [RP93], except for the operation *new* (and *set*), since the type $\forall X\colon\ast.X \to M[R[X]]$ of *new* has a nesting of M and R.

$$\begin{array}{rcl} M \equiv & \Lambda X\colon\ast.\forall\Gamma_M.X_M[X] & \colon\ast \to \ast\\ R \equiv & \Lambda X\colon\ast.\forall\Gamma_M.X_R[X] & \colon\ast \to \ast\\ ret \equiv & \Lambda X\colon\ast.\lambda x\colon X.\Lambda\Gamma_M.x_{ret}[X]\ x & \colon\forall X\colon\ast.X \to M[X]\\ new \equiv & ???? & \forall X\colon\ast.X \to M[R[X]]\\ get \equiv & \Lambda X\colon\ast.\lambda x\colon R[X].(\Lambda\Gamma_M.x_{get}[X]\ (x(\lvert\Gamma_M\rvert)) & \colon\forall X\colon\ast.R[X] \to M[X]\end{array}$$

Comparison with existential types. With the notation of Figure 4 one can define the *existential type* $\exists\Delta$ as $\forall X{:}*.(\forall\Delta.X) \to X$. The type $\forall X{:}*.(\forall\Gamma_M.X_M[X]) \to X$ of *run* is very similar to the existential type $\exists\Gamma_M \equiv \forall X{:}*.(\forall\Gamma_M.X) \to X$. This suggest a common generalisation, namely $\forall X{:}*.(\forall\Delta.\tau) \to X$, where $\Sigma; \Gamma, X{:}*, \Delta \vdash \tau{:}*$. In the type of *run* one takes $\tau \equiv X_M[X]$, while in $\exists\Delta$ one takes $\tau \equiv X$. This generalisation does not seem to have a *logical* reading, but it subsumes also the type of $runST$.

Comparison with $runST$. It is easy to recast the original proposal of [LP95] in higher-order λ-calculus, and thus compare its expressiveness with that of our *run*. The type system of [LP95] introduces several constants. If we write Σ_M for Γ_M viewed as a signature (we write M in place of X_M, etc.), then the constants of [LP95] are those declared in

$$\Sigma'_M,\ runST{:}\forall X{:}*.(\forall\alpha{:}*.M(\alpha, X)) \to X \quad \text{where } \Sigma'_M \stackrel{\Delta}{\equiv} \Pi\alpha{:}*.\Sigma_M$$

(every constant in Σ'_M takes an extra type parameter w.r.t. the corresponding constant in Σ_M). In higher-order λ-calculus one can define our *run* in terms of the constants in [LP95]

$$run \stackrel{\Delta}{\equiv} \Lambda X{:}*.\ \lambda x{:}(\forall\Gamma_M.X_M[X]).\ runST[X]\ (\Lambda\alpha{:}*.x\ (|\Sigma'_M|\ (\alpha)))$$

In other words, $run[X]\ x$ first specialises the monadic code x with the constants in Σ'_M applied to a generic type parameter α, i.e.

$$\Sigma'_M;\ X{:}*,\ x{:}\forall\Gamma_M.X_M[X],\ \alpha{:}* \vdash x\ (|\Sigma'_M|\ (\alpha)){:}\, M(\alpha, X)\ ,$$

then applies $runST$ to the abstraction of the specialised code w.r.t. α.

Remark 2.2 We conjecture that $runST$ cannot be defined in terms of *run* (and the other constants in Σ'_M). Even if the original proposal seems to be more expressive than ours, we expect it to enjoy similar safety properties (in the framework of higher-order λ-calculus). We advocate *run* in place of $runST$ because it avoids the bogus type parameter α, and thus it complies with standard monadic programming style. Moreover, we have given an intuitive reading for the type of *run* in terms of the definable type constructor for *monadic code*.

3 Instrumented semantics

The instrumented semantics is a refinement of the dynamic semantics of Section 1 which uses terms of the type system a la Church described in Section 2 instead of untyped λ-terms. The instrumented semantics serves two technical purposes: to give a more accurate description of *improper program behaviour*, to prove type safety for the dynamic semantics. In order to transfer the type safety result from the instrumented to the dynamic semantics, one has to establish a *compatibility* result linking dynamic and instrumented semantics.

To define the instrumented semantics, we introduce auxiliary semantic domains and syntactic categories. Most of them are analogues of those introduced for the dynamic semantics, and are indicated by a superscript *.

- names $m, n, r \in \mathsf{N}$, e.g. natural numbers, for naming *regions* and locations inside a region
- type operators $O \in \mathsf{OP} \stackrel{\Delta}{\equiv} \{M, R\}$
- term operators $o \in \mathsf{Op} \stackrel{\Delta}{\equiv} \{ret, do, new, get, set\}$ with type- and term-arities

term operator	o	ret	do	new	get	set
type-arity	$\Box o$	1	2	1	1	1
term-arity	$\#o$	1	2	1	1	2

- type- and term-constants, kinds, constructors and terms

$$\begin{array}{rcl} C & ::= & C_m \mid O_r \\ c \in \mathsf{Const}^\star & ::= & o_r \mid \ell_{r,m} \\ K & ::= & * \mid K_1 \to K_2 \\ u & ::= & C \mid X \mid u_1 \to u_2 \mid \forall X\colon K.u \mid \Lambda X\colon K.u \mid u_1[u_2] \\ e \in \mathsf{E}^\star & ::= & c \mid x \mid \lambda x\colon u.e \mid e_1\ e_2 \mid \Lambda X\colon K.e \mid e[u] \mid \\ & & \mathsf{run}\ \overline{X}, \overline{x}.e\ \mathsf{with}\ u \text{ where } |\overline{X}| = |\mathsf{OP}| \text{ and } |\overline{x}| = |\mathsf{Op}| \end{array}$$

  we write OP_r for $\{O_r | O \in \mathsf{OP}\}$ and Op_r for $\{o_r | o \in \mathsf{Op}\}$

- values $\begin{array}{rcl} v \in \mathsf{Val}^\star & ::= & \lambda x\colon u.e \mid \ell_{r,m} \mid o_r[\overline{u}] \text{ where } |\overline{u}| < \Box o \\ & \mid & o_r[\overline{u}]\ \overline{e} \text{ where } |\overline{u}| = \Box o \text{ and } |\overline{e}| \leq \# o \end{array}$

- stores $\mu \in \mathsf{S}^\star \stackrel{\Delta}{=} \mathsf{N} \stackrel{fin}{\to} \mathsf{E}^\star_0$;

  we write $\mathsf{Loc}_{r,\mu}$ for the set of location $\{\ell_{r,m} | m \in dom(\mu)\}$

- dynamic signatures $\Sigma \in \mathsf{Sig}\colon := \emptyset \mid \Sigma, C\colon K \mid \Sigma, c\colon u$
  and static signatures $\Delta\colon := \emptyset \mid \Delta, C_m\colon K$

- descriptions $d \in \mathsf{D}^\star\colon := (\Delta|\Sigma; e) \mid (\Sigma; \mu, e) \mid err$.

Remark 3.1 Notice that the values in $\mathsf{Val}^\star$ do not include polymorphic terms $\Lambda X\colon K.e$, therefore the instrumented semantics has to account for evaluation under Λ. More specifically, to evaluate $\Lambda X\colon K.e$ we replace X with a fresh C_m, and evaluate $e[X\colon = C_m]$. In certain cases we will blur the difference between constants and variables, in particular we will write $e[C_m\colon = u]$ and $\Lambda C_m\colon K.e$ for substitution and binding of a constant.

One can obtain a term in E from a term in $\mathsf{E}^\star$ by *erasing* types and regions.

Definition 3.2 *Given $e \in \mathsf{E}^\star$ its* **erasure** *$|e| \in \mathsf{E}$ is defined by induction as*

$$|o_r| = o \quad |\ell_{r,m}| = \ell_m \quad |x| = x \quad |\lambda x\colon u.e| = \lambda x.|e| \quad |e_1\ e_2| = |e_1|\ |e_2|$$
$$|\Lambda X\colon K.e| = |e[u]| = |e| \quad |\mathsf{run}\ \overline{X}, \overline{x}.e\ \mathsf{with}\ u| = \mathsf{run}\ \overline{x}.|e|$$

Erasure is extended to stores $\mu \in \mathsf{S}^\star$ and descriptions $d \in \mathsf{D}^\star$ as follows

$$|\mu|(m) = |\mu(m)| \qquad |(\Delta|\Sigma; e)| = |e| \qquad |(\Sigma; \mu, e)| = (|\mu|, |e|)$$

Proposition 3.3 *Erasure satisfies the following properties:*

- $|e[X\colon = u]| = |e|$ *and* $|e[x\colon = e']| = |e|[x\colon = |e'|]$
- $v \in \mathsf{Val}^\star$ *implies* $|v| \in \mathsf{Val}$
- $e \longrightarrow^{u\forall}_{\beta\eta} e'$ *implies* $|e| \equiv |e'|$.

The instrumented semantics (like the dynamic one) is given in terms of two mutually recursive interpreters, which evaluate **closed terms** and may also raise run-time errors:

- $\Delta|\Sigma; e \Longrightarrow \Delta'|\Sigma'; v \mid err$ says that evaluation of $e \in \mathsf{E}^\star_0$ by the pure interpreter returns $v \in \mathsf{Val}^\star_0$ and extends the dynamic signature from Σ to Σ' (or raises an error); moreover the polymorphism of e *is made explicit* by extending the static signature Δ with Δ';

- $\Delta|\Sigma; \mu, e \stackrel{r}{\Longrightarrow} \Sigma'; \mu', e' \mid err$ says that evaluation of $e \in \mathsf{E}^\star_0$ in local store μ by the monadic interpreter for region r returns $e' \in \mathsf{E}^\star_0$, changes the store to μ' and extends the dynamic signature from Σ to Σ' (or raises an error).

Figure 5 gives the evaluation rules for the instrumented semantics. Each of these rules is either the counterpart of an evaluation rule for the dynamic semantics (as given in Figure 1) or is for evaluating terms of the form $e[u]$ and $\Lambda X{:}\,K.e$. The rule for evaluating $\Lambda X{:}\,K.e$ is the one forcing the introduction of static signatures Δ and Δ'.

Proposition 3.4

- $\Delta|\Sigma; e \Longrightarrow \Delta'|\Sigma'; v$ *implies* $\Sigma \subseteq \Sigma'$ *and* $v \in \mathsf{Val}_0^\star$
- $\Delta|\Sigma; \mu, e \stackrel{r}{\Longrightarrow} \Sigma'; \mu', e'$ *implies* $\Sigma \subseteq \Sigma'$, *dom*$(\mu) \subseteq$ *dom*(μ') *and* $e' \in \mathsf{E}_0^\star$

Proof. By induction on the derivation of an evaluation judgement. ∎

3.1 Type safety and compatibility

We show that well-formed programs cannot go wrong, which amounts to prove subject reduction for the instrumented semantics.

Notation 3.5 We introduce auxiliary definitions useful in stating type safety.

- $\Delta, \Sigma \models J \stackrel{\Delta}{\Longleftrightarrow} \Delta, \Sigma \vdash J$ and for all constants C and c declared in Σ
  - $C \equiv M_r$ implies $C{:}\, * \to *$ in Σ
  - $C \equiv R_r$ implies $C{:}\, * \to *$ in Σ
  - $c \equiv ret_r$ implies $c{:}\,\forall X{:}\,*.X \to M_r[X]$ in Σ
  - $c \equiv do_r$ implies $c{:}\,\forall X, Y{:}\,*.M_r[X], (X \to M_r[Y]) \to M_r[Y]$ in Σ
  - $c \equiv new_r$ implies $c{:}\,\forall X{:}\,*.X \to M_r[R_r[X]]$ in Σ
  - $c \equiv get_r$ implies $c{:}\,\forall X{:}\,*.R_r[X] \to M_r[X]$ in Σ
  - $c \equiv set_r$ implies $c{:}\,\forall X{:}\,*.R_r[X], X \to M_r[R_r[X]]$ in Σ
  - $c \equiv \ell_{r,m}$ implies $c{:}\,R_r[\tau]$ in Σ for some τ
- Reg_Σ is the set of regions in Σ, i.e.

  $r \in \mathsf{Reg}_\Sigma \stackrel{\Delta}{\Longleftrightarrow}$ at least one O_r is declared in Σ.
- $\mathsf{Loc}_{r,\Sigma}$ is the set of locations of region r in Σ, i.e.

  $\ell_{r,m} \in \mathsf{Loc}_{r,\Sigma} \stackrel{\Delta}{\Longleftrightarrow} \ell_{r,m}$ is declared in Σ.
- $\Sigma \hookrightarrow \Sigma' \stackrel{\Delta}{\Longleftrightarrow} \Sigma \subseteq \Sigma'$ and $\forall n \in \mathsf{Reg}_\Sigma.\mathsf{Loc}_{n,\Sigma} = \mathsf{Loc}_{n,\Sigma'}$, i.e.

  Σ' extends Σ but the set of locations in pre-existing regions does not change.
- $\Sigma \stackrel{r}{\hookrightarrow} \Sigma' \stackrel{\Delta}{\Longleftrightarrow} \Sigma \subseteq \Sigma'$ and $\forall n \in \mathsf{Reg}_\Sigma - \{r\}.\mathsf{Loc}_{n,\Sigma} = \mathsf{Loc}_{n,\Sigma'}$, i.e.

  Σ' extends Σ but the set of locations in pre-existing regions, except region r, does not change.
- $\Delta, \Sigma \models_r \mu \stackrel{\Delta}{\Longleftrightarrow} \Delta, \Sigma \models$ and $\mathsf{Loc}_{r,\mu} = \mathsf{Loc}_{r,\Sigma}$ and for all names $m \in \mathsf{N}$

  $\ell_{r,m}{:}\,R_r[\tau]$ in Σ and $e \equiv \mu(m)$ imply $\Delta, \Sigma \vdash e{:}\,\tau$

To establish type safety one has to prove a much stronger result, which involves in an essential way regions, namely pure evaluation does not add new locations to pre-existing regions, while monadic evaluation can add new locations to the *active region* r, but not to other pre-existing regions. However, both evaluations may add new locations to newly generated regions.

Pure Evaluation

$$\Delta|\Sigma;v \Longrightarrow \emptyset|\Sigma;v \qquad \frac{\Delta|\Sigma_0;e_1 \Longrightarrow \emptyset|\Sigma_1;\lambda x\colon u.e \quad \Delta|\Sigma_1;e[x\colon= e_2] \Longrightarrow \Delta'|\Sigma_2;v}{\Delta|\Sigma_0;e_1\ e_2 \Longrightarrow \Delta'|\Sigma_2;v}$$

$$\frac{\Delta|\Sigma;e \Longrightarrow \emptyset|\Sigma';o_r[\overline{u}]}{\Delta|\Sigma;e[u] \Longrightarrow \emptyset|\Sigma';o_r[\overline{u}\ u]}\ |\overline{u}| < \Box o \qquad \frac{\Delta|\Sigma;e_1 \Longrightarrow \emptyset|\Sigma';o_r[\overline{u}]\ \overline{e}}{\Delta|\Sigma;e_1\ e_2 \Longrightarrow \emptyset|\Sigma';o_r[\overline{u}]\ \overline{e}\ e_2}\ |\overline{u}| = \Box o \wedge |\overline{e}| < \#o$$

$$\frac{\Delta|\Sigma;e \Longrightarrow C_m\colon K,\Delta'|\Sigma';v}{\Delta|\Sigma;e[u] \Longrightarrow \Delta'|(\Sigma';v)[C_m\colon= u]} \qquad \frac{\Delta,C_m\colon K|\Sigma;e[X\colon= C_m] \Longrightarrow \Delta'|\Sigma';v}{\Delta|\Sigma;\Lambda X\colon K.e \Longrightarrow C_m\colon K,\Delta'|\Sigma';v}\ m \text{ fresh}$$

$$\frac{\Delta|\Sigma_0+\Sigma_r^{op};\emptyset,e[\overline{X},\overline{x}\colon= \mathsf{OP}_r,\mathsf{Op}_r] \overset{r}{\Longrightarrow} \Sigma_1;\mu,e' \quad \Delta|\Sigma_1;e' \Longrightarrow \Delta'|\Sigma_2;v}{\Delta|\Sigma_0;\mathsf{run}\ \overline{X},\overline{x}.e\ \mathsf{with}\ u \Longrightarrow \Delta'|\Sigma_2;v}\ r \text{ fresh}$$

where $\Sigma_r^{op} \stackrel{\Delta}{\equiv} M_r,\ R_r\colon * \to *\ ,$
$\quad ret_r\colon \forall X\colon *.X \to M_r[X]\ ,$
$\quad do_r\colon \forall X,Y\colon *.M_r[X],(X \to M_r[Y]) \to M_r[Y]\ ,$
$\quad new_r\colon \forall X\colon *.X \to M_r[R_r[X]]\ ,$
$\quad get_r\colon \forall X\colon *.R_r[X] \to M_r[X]\ ,$
$\quad set_r\colon \forall X\colon *.R_r[X],X \to M_r[R_r[X]]$

Monadic Evaluation

$$\frac{\Delta|\Sigma;e \Longrightarrow \emptyset|\Sigma';ret_r[u]\ e'}{\Delta|\Sigma;\mu,e \overset{r}{\Longrightarrow} \Sigma';\mu,e'} \qquad \frac{\begin{array}{l}\Delta|\Sigma_0;e \Longrightarrow \emptyset|\Sigma_1;do_r[u_0\ u_1]\ e_0\ e_1\\ \Delta|\Sigma_1;\mu_0,e_0 \overset{r}{\Longrightarrow} \Sigma_2;\mu_1,e_0'\\ \Delta|\Sigma_2;\mu_1,e_1\ e_0' \overset{r}{\Longrightarrow} \Sigma_3;\mu_2,e'\end{array}}{\Delta|\Sigma_0;\mu_0,e \overset{r}{\Longrightarrow} \Sigma_3;\mu_2,e'}$$

$$\frac{\Delta|\Sigma;e \Longrightarrow \emptyset|\Sigma';new_r[u]\ e_0}{\Delta|\Sigma;\mu,e \overset{r}{\Longrightarrow} \Sigma'+\Sigma_{r,u}^m;\mu\{m=e_0\},\ell_{r,m}}\ m \notin dom(\mu)$$

where $\Sigma_{r,u}^m$ is the dynamic signature $\quad \ell_{r,m}\colon R_r u$

$$\frac{\Delta|\Sigma_0;e \Longrightarrow \emptyset|\Sigma_1;get_r[u]\ e_0 \quad \Delta|\Sigma_1;e_0 \Longrightarrow \emptyset|\Sigma_2;\ell_{r,m}}{\Delta|\Sigma_0;\mu,e \overset{r}{\Longrightarrow} \Sigma_2;\mu,e'}\ e' = \mu(m)$$

$$\frac{\Delta|\Sigma_0;e \Longrightarrow \emptyset|\Sigma_1;set_r[u]\ e_0\ e_1 \quad \Delta|\Sigma_1;e_0 \Longrightarrow \emptyset|\Sigma_2;\ell_{r,m}}{\Delta|\Sigma_0;\mu,e \overset{r}{\Longrightarrow} \Sigma_2;\mu\{m=e_1\},\ell_{r,m}}\ m \in dom(\mu)$$

Pure and Monadic Run-Time Errors

Besides the obvious rules for error propagation, there are the following rules for error generation

$$\frac{\Delta|\Sigma;e_1 \Longrightarrow \Delta'|\Sigma';v}{\Delta|\Sigma;e_1\ e_2 \Longrightarrow err}\ \Delta' \not\equiv \emptyset \text{ or } v \not\equiv \lambda x\colon u.e \text{ or } v \not\equiv o_r[\overline{u}]\ \overline{e} \text{ with } |\overline{u}| = \Box o \wedge |\overline{e}| < \#o$$

$$\frac{\Delta|\Sigma;e \Longrightarrow \emptyset|\Sigma';v}{\Delta|\Sigma;e[u] \Longrightarrow err}\ v \not\equiv o_r[\overline{u}] \text{ with } |\overline{u}| < \Box o$$

$$\frac{\Delta|\Sigma;e \Longrightarrow \Delta'|\Sigma';v}{\Delta|\Sigma;\mu,e \overset{r}{\Longrightarrow} err}\ \Delta' \not\equiv \emptyset \text{ or } v \not\equiv o_r[\overline{u}]\ \overline{e} \text{ with } |\overline{u}| = \Box o \wedge |\overline{e}| = \#o$$

$$\frac{\Delta|\Sigma;e \Longrightarrow \emptyset|\Sigma';get_r[u]\ e_0 \quad \Delta|\Sigma';e_0 \Longrightarrow \Delta'|\Sigma'';v}{\Delta|\Sigma;\mu,e \overset{r}{\Longrightarrow} err}\ \Delta' \not\equiv \emptyset \text{ or } v \notin \mathsf{Loc}_{r,\mu}$$

$$\frac{\Delta|\Sigma;e \Longrightarrow \emptyset|\Sigma';set_r[u]\ e_0\ e_1 \quad \Delta|\Sigma';e_0 \Longrightarrow \Delta'|\Sigma'';v}{\Delta|\Sigma;\mu,e \overset{r}{\Longrightarrow} err}\ \Delta' \not\equiv \emptyset \text{ or } v \notin \mathsf{Loc}_{r,\mu}$$

Figure 5: Evaluation Rules for Instrumented Semantics

Theorem 3.6 (type safety for instrumented sem.)

1. *If* $\Delta|\Sigma; e \Longrightarrow d$ *and* $\Delta, \Sigma \models e\colon\tau$, *then exist* Δ', Σ', v *and* τ' *s.t.* $d \equiv (\Delta'|\Sigma'; v)$, $\tau =^{u}_{\beta\eta} \forall\Delta'.\tau'$, $\Sigma \hookrightarrow \Sigma'$ *and* $\Delta, \Delta', \Sigma' \models v\colon\tau'$

   The type $\forall\Delta'.\tau'$ *is defined by induction on* Δ' *(see Figure 4), and by blurring the distinction between constants and variables (see Remark 3.1)*

2. *If* $\Delta|\Sigma; \mu, e \stackrel{r}{\Longrightarrow} d$, $\Delta, \Sigma \models_r \mu$ *and* $\Delta, \Sigma \models e\colon M_r[\tau]$, *then exist* Σ', μ' *and* e' *s.t.* $d \equiv (\Sigma'; \mu', e')$, $\Sigma \stackrel{r}{\hookrightarrow} \Sigma'$, $\Delta, \Sigma' \models_r \mu'$ *and* $\Delta, \Sigma' \models e'\colon\tau$

Proof. By induction on the derivation of an evaluation judgement, and by applying the generation lemma to the typing assumption. The details for some cases are given in Appendix C. ∎

The following theorem says that the instrumented semantics is *compatible* with the dynamic semantics modulo erasure, more precisely: if e may terminate *properly* so does $|e|$, if $|e|$ may raise an error so does e, if $|e|$ may terminate *properly* then e may do the same or raise an error. When e may raise an error, we cannot say anything about the behaviour of $|e|$, it may even fail to terminate.

Theorem 3.7 (compatibility) *For every* Δ, Σ, e, μ, r *and* d' *the following implications hold*

- $|e| \Longrightarrow d'$ *implies exists* d *s.t.* $\Delta|\Sigma; e \Longrightarrow d$ *and* $(d' \equiv |d|$ *or* $d \equiv err)$
- $|\mu|, |e| \Longrightarrow d'$ *implies exists* d *s.t.* $\Delta|\Sigma; \mu, e \stackrel{r}{\Longrightarrow} d$ *and* $(d' \equiv |d|$ *or* $d \equiv err)$

Proof. The implications are proved by lexicographic induction on the derivation of an evaluation judgement $|e| \Longrightarrow d'$ / $|\mu|, |e| \Longrightarrow d'$ for the dynamic semantics, and the *size* of e / (μ, e). The details for some cases are given in in Appendix C. ∎

Corollary 3.8 (type safety for dynamic sem.) *If* $\Delta, \Sigma \models e\colon\tau$ *and* $|e| \Longrightarrow d'$, *then* $d' \not\equiv err$.

Proof. By compatibility we know that exists d s.t. $\Delta|\Sigma; e \Longrightarrow d$ and $(d' \equiv |d|$ or $d \equiv err)$. By type safety for the instrumented semantics we know that $d \not\equiv err$, therefore $d' \equiv |d| \not\equiv err$. ∎

4 Conclusions and related work

In this section we discuss what we have done, also in the light of related work, and point out possible future developments.

- **Lazy versus Strict state semantics**. We have focused on strict state because we are mainly interested in identifying generic mechanisms for encapsulating a programming style into another (and mainstream imperative languages handle the state in a strict way). However, one may ask whether our approach can be adapted to prove type safety for lazy state.

  Some preliminary attempts in collaboration with A.Sabry seem to indicate that this can be done. but the dynamic semantics for lazy state is substantially more complex, and it does not allow the simple deallocation strategy typical of region-based memory management.

- **CBV, CBN and lazy semantics**. We have adopted a CBN semantics for the pure interpreter (following [LP95, LS97]), while [SS99] adopts a CBV semantics. Technically speaking, it does not make much of a difference if the pure interpreter adopts CBN or CBV.

  However, CBN is very inefficient, so it would be interesting to adopt a lazy semantics for the pure interpreter, and then prove that it would induce the same observational congruence $\approx$ of the pure CBN interpreter.

- **Effect Masking and Monadic Encapsulation**. [SS99] shows that $runST$ implements a cheap form of effect masking (see [LG88, TJ92]), thus extending the relation between effects and monads established in [Wad98]. More precisely they give a translation from a type system with effects and regions (EML) to one with $runST$ (MML).

  It seems plausible that the translation given in [SS99] can be adapted so that the target language (MML) uses our run-with construct instead of $runST$. Indeed, replacing $runST$ by run-with might help to establish a reverse translation from MML to EML.

- **Relations to region-based memory management**. While the languages we consider do not have syntactic categories of effects and regions, the instrumented semantics exhibits certain similarities with region-based memory management (see [TT97]), namely the two-dimensional structure of the address space, and the store deallocation strategy.

In the paper we have also pointed out what we believe to be intrinsic limitations of reduction semantics (as advocated by [WF94]), which prevent the formalisation of certain implementation details. On the other hand, substantially more work is needed to establish soundness of equational reasoning w.r.t. our dynamic semantics (even for something as unsurprising as β-equivalence).

References

[Bar91] H. P. Barendregt. Lambda calculi with types. In S. Abramsky, D. M. Gabbay, and T. S. E. Maibaum, editors, *Handbook of Logic in Computer Science*. Oxford University Press, 1991.

[Car97] L. Cardelli. Type systems. In Jr Allen B. Tucker, editor, *The Computer Science and Engineering Handbook*. CRC Press, 1997.

[dH84] R. de Nicola and M. C. B. Hennessy. Testing equivalences for processes (concurrent programming). *Theoretical Computer Science*, 34(1-2):83–133, November 1984.

[Geu93] H. Geuvers. *Logics and Type Systems*. PhD thesis, Computer Science Institute, Katholieke Univ. Nijmegen, 1993.

[LG88] J. M. Lucassen and D. K. Gifford. Polymorphic effect systems. In ACM, editor, *POPL '88. Proceedings of the conference on Principles of programming languages, January 13–15, 1988, San Diego, CA*, pages 47–57. ACM Press, 1988.

[LP95] J. Launchbury and S.L. Peyton Jones. State in Haskell. *Lisp and Symbolic Computation*, 8(4), December 1995.

[LS97] J. Launchbury and A. Sabry. Monadic state: Axiomatization and type safety. In *Proc. of the 1997 Int. Conf. on Functional Programming*, Amsterdam, The Netherlands, 9–11 June 1997.

[MP88] J. C. Mitchell and G. D. Plotkin. Abstract types have existential types. *ACM Trans. on Programming Languages and Systems*, 10(3):470–502, 1988.

[Plo75] G. Plotkin. Call-by-name, call-by-value and the λ-calculus. *Theoretical Computer Science*, 1(2), 1975.

[RP93] J. C. Reynolds and G. D. Plotkin. On functors expressible in the polymorphic lambda calculus. *Information and Computation*, 105:1–29, 1993.

[SS99] M. Semmelroth and A. Sabry. Monadic encapsulation in ML. In *Proc. of the 1999 Int. Conf. on Functional Programming*, Paris, France, 27–29 September 1999.

[TJ92] J.-P. Talpin and P. Jouvelot. Polymorphic type, region and effect inference. *Journal of Functional Programming*, 2(3):245–271, 1992.

[TT97] M. Tofte and J.P. Talpin. Region-based memory management. *Information and Computation*, 132:109–176, 1997.

[Wad98] P. Wadler. The marriage of effects and monads. In *International Conference on Functional Programming*, pages 63–74, Baltimore, September 1998. ACM.

[WF94] A. K. Wright and M. Felleisen. A syntactic approach to type soundness. *Information and Computation*, 115:38–94, 1994.

A Proof of soundness of β-conversion

Lemma A.1 *If $e \longrightarrow_\beta e'$ and $e \in \mathsf{Val}$, then $e' \in \mathsf{Val}$ and has the same top-level structure of e.*

Definition A.2 (1-step parallel reduction) *1-step parallel reduction $e >_p e'$ is the relation on E induced by the following inference rules*

$$e >_p e \qquad \frac{e >_p e'}{\lambda x.e >_p \lambda x.e'} \qquad \frac{e >_p e'}{\mathsf{run}\ \overline{x}.e >_p \mathsf{run}\ \overline{x}.e'} \qquad \frac{\begin{array}{c} e_1 >_p e_1' \\ e_2 >_p e_2' \end{array}}{e_1\ e_2 >_p e_1'\ e_2'} \qquad \frac{\begin{array}{c} e >_p e' \\ e_2 >_p e_2' \end{array}}{(\lambda x.e)\ e_2 >_p e'[x{:=}e_2']}$$

1-step parallel reduction is extended in the obvious way to stores $\mu \in \mathsf{S}$ and descriptions $d \in \mathsf{D}$, in particular $\mu >_p \mu' \stackrel{\Delta}{\Longleftrightarrow} dom(\mu) = dom(\mu')$ and $\forall m \in dom(\mu).\mu(m) >_p \mu'(m)$.

The following are well-known facts about 1-step parallel reduction.

Lemma A.3 substitutivity $\dfrac{e_1 >_p e_1' \quad e_2 >_p e_2'}{e_1[x{:=}e_2] >_p e_1'[x{:=}e_2']}$ *is admissible. $e \longrightarrow_\beta^* e'$ iff $e >_p^* e'$.*

Proposition A.4 *If $d_1 >_p d_1'$ and $d_1 \Longrightarrow d_2$, then exists d_2' s.t. $d_2 >_p d_2'$ and $d_1' \Longrightarrow d_2'$.*

Proof. By induction on the derivation of $d_1 \Longrightarrow d_2$ and case analysis on the derivation of $d_1 >_p d_1'$. We consider the inductive steps covering the case of d_1 being an application. If $d_1 \equiv d_1'$, then the proof is immediate.

If $\dfrac{e_1 \Longrightarrow \lambda x.e \quad e[x{:=}e_2] \Longrightarrow d_2}{d_1 \equiv e_1\ e_2 \Longrightarrow d_2}$ and $\dfrac{e_1 >_p e_1' \quad e_2 >_p e_2'}{d_1 \equiv e_1\ e_2 >_p e_1'\ e_2' \equiv d_1'}$, then

- exists e' s.t. $e >_p e'$ and $e_1' \Longrightarrow \lambda x.e'$, by IH for $e_1 \Longrightarrow \lambda x.e$
- $e[x{:=}e_2] >_p e'[x{:=}e_2']$, by substitutivity for $>_p$
- exists d_2' s.t. $d_2 >_p d_2'$ and $d_1' \Longrightarrow d_2'$, by IH for $e[x{:=}e_2] \Longrightarrow d_2$ (and definition of evaluation).

If $\dfrac{\lambda x.e \Longrightarrow \lambda x.e \quad e[x{:=}e_2] \Longrightarrow d_2}{d_1 \equiv (\lambda x.e)\ e_2 \Longrightarrow d_2}$ and $\dfrac{e >_p e' \quad e_2 >_p e_2'}{d_1 \equiv (\lambda x.e)\ e_2 >_p e'[x{:=}e_2'] \equiv d_1'}$, then

- $e[x{:=}e_2] >_p e'[x{:=}e_2'] \equiv d_1'$, by substitutivity for $>_p$
- exists d_2' s.t. $d_2 >_p d_2'$ and $d_1' \Longrightarrow d_2'$, by IH for $e[x{:=}e_2] \Longrightarrow d_2$.

∎

Definition A.5 (CBN and standard reduction) *CBN reduction $e >_n e'$ is the (functional) relation induced by the following inference rules*

$$(\lambda x.e_1)\ e_2 >_n e_1[x:=e_2] \qquad \frac{e_1 >_n e_1'}{e_1\ e_2 >_n e_1'\ e_2}$$

Standard reduction $e >_s e'$ is the relation induced by the following inference rules

$$x >_s x \quad c >_s c \quad \frac{e >_s e'}{\lambda x.e >_s \lambda x.e'} \quad \frac{e >_s e'}{\mathsf{run}\ \overline{x}.e >_s \mathsf{run}\ \overline{x}.e'} \quad \frac{\begin{array}{c} e >_n e' \\ e' >_s e'' \end{array}}{e >_s e''} \quad \frac{\begin{array}{c} e_1 >_s e_1' \\ e_2 >_s e_2' \end{array}}{e_1\ e_2 >_s e_1'\ e_2'}$$

Standard reduction is extended in the obvious way to stores $\mu \in \mathsf{S}$ and descriptions $d \in \mathsf{D}$, in particular $\mu >_s \mu' \stackrel{\Delta}{\iff} dom(\mu) = dom(\mu')$ and $\forall m \in dom(\mu).\mu(m) >_s \mu'(m)$.

Lemma A.6 (standardisation) *$e \longrightarrow_\beta^* e'$ iff $e >_s e'$.*

Proposition A.7 *If $e >_n e'$ and $e' \Longrightarrow d$, then $e \Longrightarrow d$.*

Proof. By induction on the derivation of $e >_n e'$. ∎

Proposition A.8 *If $d_1 >_s d_1'$ and $d_1' \Longrightarrow d_2'$, then exists d_2 s.t. $d_2 >_s d_2'$ and $d_1 \Longrightarrow d_2$.*

Proof. By lexicographic induction on the derivation of $d_1' \Longrightarrow d_2'$ and on the derivation of $d_1 >_s d_1'$. We consider the inductive steps covering the case of d_1 being an application.

If $\dfrac{e_1' \Longrightarrow \lambda x.e' \quad e'[x:=e_2'] \Longrightarrow d_2'}{d_1' \equiv e_1'\ e_2' \Longrightarrow d_2'}$ and $\dfrac{e_1 >_s e_1' \quad e_2 >_s e_2'}{d_1 \equiv e_1\ e_2 >_s e_1'\ e_2' \equiv d_1'}$, then

- exists e s.t. $e >_s e'$ and $e_1 \Longrightarrow \lambda x.e$, by IH for $e_1' \Longrightarrow \lambda x.e'$ and $e_1 >_s e_1'$ (and Lemma A.1)
- $e[x:=e_2] >_s e'[x:=e_2']$, by substitutivity for $>_s/\longrightarrow_\beta^*$
- exists d_2 s.t. $d_2 >_s d_2'$ and $d_1 \Longrightarrow d_2$, by IH for $e'[x:=e_2'] \Longrightarrow d_2'$ and $e[x:=e_2] >_s e'[x:=e_2']$ (and definition of evaluation).

If $\dfrac{e_1' \Longrightarrow \lambda x.e' \quad e'[x:=e_2'] \Longrightarrow d_2'}{d_1' \equiv e_1'\ e_2' \Longrightarrow d_2'}$ and $\dfrac{d_1 >_n d \quad d >_s d_1'}{d_1 >_s d_1'}$, then

- exists d_2 s.t. $d_2 >_s d_2'$ and $d_1 \Longrightarrow d_2$, by IH for $d_1' \Longrightarrow d_2'$ and $d >_s d_1'$ (and Lemma A.7).

∎

B Basic properties of the type system

This sections states some properties of the type system.

Proposition B.1 $<.1\ \dfrac{\Sigma, \Sigma' \vdash}{\Sigma \vdash} \qquad <.2\ \dfrac{\Sigma, \Sigma'; \Gamma \vdash J}{\Sigma \vdash} \qquad <.3\ \dfrac{\Sigma; \Gamma, \Gamma' \vdash J}{\Sigma; \Gamma \vdash}$

Proposition B.2 (Thinning)

$$T.\Sigma\ \frac{\Sigma; \Gamma \vdash J \quad \Sigma' \vdash}{\Sigma'; \Gamma \vdash J}\ \Sigma \subseteq \Sigma' \qquad T.\Gamma\ \frac{\Sigma; \Gamma, \Delta \vdash J \quad \Sigma; \Gamma' \vdash}{\Sigma; \Gamma', \Delta \vdash J}\ \Gamma \subseteq \Gamma' \land dom(\Gamma') \cap dom(\Delta) = \emptyset$$

Proposition B.3 (Substitution)

$$S.X\ \frac{\Sigma;\Gamma_1, X\colon K, \Gamma_2 \vdash J \quad \Sigma;\Gamma_1 \vdash u\colon K}{\Sigma;\Gamma_1, \Gamma_2[X\colon= u] \vdash J[X\colon= u]} \qquad S.x\ \frac{\Sigma;\Gamma_1, x\colon u, \Gamma_2 \vdash J \quad \Sigma;\Gamma_1 \vdash e\colon u}{\Sigma;\Gamma_1, \Gamma_2 \vdash J[x\colon= e]}$$

Proposition B.4 (Proper typing) $\dfrac{\Sigma;\Gamma \vdash e\colon \tau}{\Sigma;\Gamma \vdash \tau\colon *}$

Proposition B.5 (Generation Lemma for Terms) *The following implications hold*

1. $\Sigma;\Gamma \vdash c\colon \tau \Rightarrow \exists \tau'.[\Sigma;\Gamma \vdash \wedge c\colon \tau' \in \Sigma \wedge \tau =^u_{\beta\eta} \tau']$
2. $\Sigma;\Gamma \vdash x\colon \tau \Rightarrow \exists \tau'.[\Sigma;\Gamma \vdash \wedge x\colon \tau' \in \Gamma \wedge \tau =^u_{\beta\eta} \tau']$
3. $\Sigma;\Gamma \vdash (\lambda x\colon \tau_1.e)\colon \tau \Rightarrow \exists \tau_2.[\Sigma;\Gamma, x\colon \tau_1 \vdash e\colon \tau_2 \wedge \tau =^u_{\beta\eta} (\tau_1 \to \tau_2)]$
4. $\Sigma;\Gamma \vdash (e_1 e_2)\colon \tau \Rightarrow \exists \tau_1, \tau_2.[\Sigma;\Gamma \vdash e_1\colon (\tau_1 \to \tau_2) \wedge \Sigma;\Gamma \vdash e_2\colon \tau_1 \wedge \tau =^u_{\beta\eta} \tau_2]$
5. $\Sigma;\Gamma \vdash (\Lambda X\colon K.e)\colon \tau \Rightarrow \exists \tau'.[\Sigma;\Gamma, X\colon K \vdash e\colon \tau' \wedge \tau =^u_{\beta\eta} (\forall X\colon K.\tau')]$
6. $\Sigma;\Gamma \vdash (e[u])\colon \tau \Rightarrow \exists X, K, \tau'.[\Sigma;\Gamma \vdash e\colon (\forall X\colon K.\tau') \wedge \Sigma;\Gamma \vdash u\colon K \wedge \tau =^u_{\beta\eta} \tau'[X\colon= u]\ ]$
7. $\Sigma;\Gamma \vdash (\textsf{run}\ \overline{X}, \overline{x}.e\ \textsf{with}\ \tau)\colon \tau' \Rightarrow [\Sigma;\Gamma \vdash \tau\colon * \wedge \Sigma;\Gamma, \Gamma_M \vdash e\colon X_M[\tau] \wedge \tau =^u_{\beta\eta} \tau']$
   where $\overline{X}, \overline{x} \equiv |\Gamma_M|$ *and* Γ_M *is given in Figure 3.*

C Proofs of type safety and compatibility

Theorem C.1 (type safety for instrumented sem.)

1. *If* $\Delta|\Sigma; e \Longrightarrow d$ *and* $\Delta, \Sigma \models e\colon \tau$*, then exist* Δ', Σ', v *and* τ' *s.t.*
   $d \equiv (\Delta'|\Sigma'; v)$ *and* $\tau =^u_{\beta\eta} \forall \Delta'.\tau'$ *and* $\Sigma \hookrightarrow \Sigma'$ *and* $\Delta, \Delta', \Sigma' \models v\colon \tau'$
   The type $\forall\Delta'.\tau'$ *is defined by induction on the structure of* Δ' *(see Figure 4), and by blurring the distinction between constants and variables (as indicated in Remark 3.1)*
2. *If* $\Delta|\Sigma; \mu, e \stackrel{r}{\Longrightarrow} d$ *and* $\Delta, \Sigma \models_r \mu$ *and* $\Delta, \Sigma \models e\colon M_r[\tau]$ *then exist* Σ', μ' *and* e' *s.t.*
   $d \equiv (\Sigma'; \mu', e')$ *and* $\Sigma \stackrel{r}{\hookrightarrow} \Sigma'$ *and* $\Delta, \Sigma' \models_r \mu'$ *and* $\Delta, \Sigma' \models e'\colon \tau$

Proof. By induction on the derivation of an evaluation judgement, and by applying the generation lemma to the typing assumption. We consider in detail few cases.

- $\dfrac{(1)\ \Delta|\Sigma; e \Longrightarrow \emptyset|\Sigma'; o_r\ [\overline{u}]}{\Delta|\Sigma; e\ [u] \Longrightarrow \emptyset|\Sigma'; o_r\ [\overline{u}\ u]}\ |\overline{u}| < \Box o$

  from the assumption $\Delta, \Sigma \models e[u]\colon \tau$ and the generation lemma exist X, K, τ_1 such that:
  (2) $\Delta, \Sigma \models e\colon (\forall X\colon K.\tau_1)$ (3) $\Delta, \Sigma \models u\colon K$ (4) $\tau =^u_{\beta\eta} \tau_1[X\colon= u]$
  from (2) and the IH for (1) exists τ_2 such that
  (5) $\Delta, \Sigma' \models o_r[\overline{u}]\colon \tau_2$ (6) $(\forall X\colon K.\tau_1) =^u_{\beta\eta} \tau_2$ (7) $\Sigma \hookrightarrow \Sigma'$
  from (5) and (6) by the formation rule (*conv*) one has
  (8) $\Delta, \Sigma' \models o_r[\overline{u}]\colon (\forall X\colon K.\tau_1)$
  from (3) by thinning one has
  (9) $\Delta, \Sigma' \models u\colon K$
  from (8) and (9) by the formation rule ($\forall$E) follows
  (10) $\Delta, \Sigma' \models o_r[\overline{u}\ u]\colon \tau_1[X\colon= u]$
  from (4), (10) and (7), taking $\tau' \equiv \tau_1[X\colon= u]$, follows the thesis.

- $$\frac{(1)\ \Delta|\Sigma; e \Longrightarrow \emptyset|\Sigma'; v}{\Delta|\Sigma; e[u] \Longrightarrow err}\ v \not\equiv o_r[\overline{u}] \text{ with } |\overline{u}| < \Box o$$

  from the assumption $\Delta, \Sigma \models e[u]\colon \tau$ and the generation lemma exist X, K, τ_1 such that:

  (2) $\Delta, \Sigma \models e\colon (\forall X\colon K.\tau_1)$ (3) $\Delta, \Sigma \models u\colon K$ (4) $\tau =^u_{\beta\eta} \tau_1[X\colon= u]$

  from (2) and the IH for (1) exists τ_2 such that

  (5) $\Delta, \Sigma' \models v\colon \tau_2$ (6) $(\forall X\colon K.\tau_1) =^u_{\beta\eta} \tau_2$ (7) $\Sigma \hookrightarrow \Sigma'$

  from (5) and (6) by the formation rule (*conv*) one has

  (8) $\Delta, \Sigma' \models v\colon (\forall X\colon K.\tau_1)$

  however (8) is in contradiction with the assumption $v \not\equiv o_r[\overline{u}]$ with $|\overline{u}| < \Box o$.

  In fact the remaining possibilities for v, i.e.

  $\lambda x\colon u.e \mid \ell_{r,m} \mid o_r[\overline{u}]\ \overline{e}$ where $|\overline{u}| = \Box o$ and $|\overline{e}| \leq \# o$

  cannot have a polymorphic type, only a functional type or one of the form $R_r[\tau]$ or $M_r[\tau]$.

  Therefore the thesis follows, because this case of the proof by induction cannot occur.

- $$\frac{(1)\ \Delta, C_m\colon K|\Sigma; e[X\colon= C_m] \Longrightarrow \Delta'|\Sigma'; v}{\Delta|\Sigma; \Lambda X\colon K.e \Longrightarrow C_m\colon K, \Delta'|\Sigma'; v}\ (2)\ m \text{ fresh}$$

  from the assumption $\Delta, \Sigma \models \Lambda X\colon K.e\colon \tau$ and the generation lemma exists τ_1 such that:

  (3) $\Delta, \Sigma; X\colon K \models e\colon \tau_1$ (4) $\tau =^u_{\beta\eta} (\forall X\colon K.\tau_1)$

  from (3) and (2) by the substitution lemma

  (5) $\Delta, C_m\colon K, \Sigma \models e[X\colon= C_m]\colon \tau_1[X\colon= C_m]$

  from (5) and the IH for (1) exists τ_2 such that

  (6) $\tau_1[X\colon= C_m] =^u_{\beta\eta} (\forall \Delta'.\tau_2)$ (7) $\Delta, C_m\colon K, \Delta', \Sigma' \models v\colon \tau_2$ (8) $\Sigma \hookrightarrow \Sigma'$

  from (4) and (6) by properties of $=^u_{\beta\eta}$

  (9) $\tau =^u_{\beta\eta} (\forall C_m\colon K, \Delta'.\tau_2)$

  from (9), (7) and (8), taking $\tau' \equiv \tau_2$, follows the thesis.

- $$\frac{(1)\ \Delta|\Sigma; e \Longrightarrow \emptyset|\Sigma'; ret_r[\tau']\ e'}{\Delta|\Sigma; \mu, e \overset{r}{\Longrightarrow} \Sigma'; \mu, e'}$$

  from the assumption $\Delta, \Sigma \models e\colon M_r[\tau]$ and the IH for (1) exists τ_1 such that:

  (2) $M_r[\tau] =^u_{\beta\eta} \tau_1$ (3) $\Delta, \Sigma' \models ret_r[\tau']\ e'\colon \tau_1$ (4) $\Sigma \hookrightarrow \Sigma'$

  from (2) and (3) by the generation lemma (and properties of $=^u_{\beta\eta}$

  (5) $\Delta; \Sigma' \models e'\colon \tau'$ (6) $\tau' =^u_{\beta\eta} \tau$

  from (3), (4) and the assumption $\Delta, \Sigma \models_r \mu$ follows that

  (7) $\Delta, \Sigma' \models_r \mu$

  from (5) and (6) by the formation rule (*conv*) one has

  (8) $\Delta, \Sigma' \models e'\colon \tau$

  from (4), since $\Sigma \hookrightarrow \Sigma'$ implies $\Sigma \overset{r}{\hookrightarrow} \Sigma'$, follows that

  (9) $\Sigma \overset{r}{\hookrightarrow} \Sigma'$

  from (9), (7) and (8) follows the thesis.

∎

Theorem C.2 (compatibility) *For every Δ, Σ, e, μ, r and d' the following implications hold*

- $|e| \Longrightarrow d'$ *implies exists* d *s.t.* $\Delta|\Sigma; e \Longrightarrow d$ *and* ($d' \equiv |d|$ *or* $d \equiv err$)
- $|\mu|, |e| \Longrightarrow d'$ *implies exists* d *s.t.* $\Delta|\Sigma; \mu, e \overset{r}{\Longrightarrow} d$ *and* ($d' \equiv |d|$ *or* $d \equiv err$)

Proof. The implications are proved by lexicographic induction on the derivation of an evaluation judgement $|e| \Longrightarrow d'$ $\;|\mu|, |e| \Longrightarrow d'$ for the dynamic semantics, and the *size* of e $\;(\mu, e)$. We consider in detail one case, to illustrate why we need the lexicographic induction.

Suppose that $|e| \equiv e'_1\ e'_2$ and $\dfrac{e'_1 \Longrightarrow \lambda x.e' \quad e'[x{:=}\,e'_2] \Longrightarrow d'}{e'_1\ e'_2 \Longrightarrow d'}$ there are 3 possibilities for e

$e_1\ e_2$ therefore $|e_1| \equiv e'_1$and $|e_2| \equiv e'_2$. In this case we apply the IH to Δ, Σ, e_1 and $(\lambda x.e')$. In fact, the derivation of the dynamic evaluation judgement $e'_1 \Longrightarrow \lambda x.e'$ is shorter.

Therefore we have a d s.t. $\Delta|\Sigma; e_1 \Longrightarrow d$ and ($d' \equiv |d|$ or $d \equiv err$).

If $d \equiv (\emptyset|\Sigma_1; \lambda x{:}\,u.e)$ and thus $|e| = e'$, then we proceed (and reach the desired conclusion) by applying the IH to Δ, Σ_1, $e[x{:=}\,e_2]$ and d'. In fact, $|e[x{:=}\,e_2]| \equiv e'[x{:=}\,e'_2]$ and the derivation of the dynamic evaluation judgement $e'[x{:=}\,e'_2] \Longrightarrow d'$ is shorter.

In all the other cases, namely $d \equiv err$ or $d \equiv (\Delta'|\Sigma_1; \lambda x{:}\,u.e)$ with $\Delta' \not\equiv \emptyset$, we get $\Delta|\Sigma; e_1\ e_2 \Longrightarrow err$.

$\Lambda X{:}\,K.e$ therefore $|e| \equiv e'_1\ e'_2$. In this case we apply the IH to $\Delta, C_m{:}\,K$ (with m fresh), Σ, $e[X{:=}\,C_m]$ and d'. In fact, we have reduced the size of the term (from $\Lambda X{:}\,K.e$ to $e[X{:=}\,C_m]$), while the dynamic evaluation judgement is unchanged (since $|\Lambda X{:}\,K.e| \equiv |e[X{:=}\,C_m]|$).

Therefore we have a d s.t. $\Delta, C_m{:}\,K|\Sigma; e[X{:=}\,C_m] \Longrightarrow d$ and ($d' \equiv |d|$ or $d \equiv err$).

If $d \equiv (\Delta'|\Sigma'; v)$ and $|d| = d'$, then we derive $\Delta|\Sigma; \Lambda X{:}\,K.e \Longrightarrow C_m{:}\,K, \Delta'|\Sigma'; v$, and obviously $|C_m{:}\,K, \Delta'|\Sigma'; v| \equiv d'$. Otherwise we get $\Delta|\Sigma; \Lambda X{:}\,K.e \Longrightarrow err$.

$e[u]$ therefore $|e| \equiv e'_1\ e'_2$. In this case we apply the IH to Δ, Σ, e and d'. In fact, we have reduced the size of the term (from $e[u]$ to e), while the dynamic evaluation judgement is unchanged (since $|e[u]| \equiv |e|$).

Therefore we have a d s.t. $\Delta|\Sigma; e \Longrightarrow d$ and ($d' \equiv |d|$ or $d \equiv err$).

If $d \equiv (C_m{:}\,K, \Delta'|\Sigma'; v)$ and $|d| = d'$, then we derive $\Delta|\Sigma; e[u] \Longrightarrow \Delta'|(\Sigma'; v)[C_m{:=}\,u]$, and obviously $|\Delta'|(\Sigma'; v)[C_m{:=}\,u]| \equiv d'$. Otherwise we get $\Delta|\Sigma; e[u] \Longrightarrow err$.

∎

Contents

1 Dynamic semantics and β-conversion **3**

2 Type system a la Church **6**

2.1 Intermezzo: types for encapsulation 6

3 Instrumented semantics **9**

3.1 Type safety and compatibility 11

4 Conclusions and related work **13**

A Proof of soundness of β-conversion **15**

B Basic properties of the type system **16**

C Proofs of type safety and compatibility **17**