MPRI 2.4, Functional programming and type systems

Gabriel Scherer
reusing course material from Didier Rémy

References, Value restriction

Contents

References

In the ML vocabulary, a *reference cell*, also called *a reference*, is a dynamically allocated block of memory, which holds a value, and whose content can change over time.

A reference can be allocated and initialized (*ref*), written (*:=*), and read (*!*).

Expressions and evaluation contexts are extended:

$$\begin{array}{rcl} M & ::= & \ldots \mid \mathit{ref}\, M \mid M := M \mid !\, M \\ E & ::= & \ldots \mid \mathit{ref}\, [\,] \mid [\,] := M \mid V := [\,] \mid !\, [\,] \end{array}$$

References

A reference allocation is not a value. Otherwise, by β, the program:

$$(\lambda x{:}\tau.\,(x := 1; !\,x))\;(\mathit{ref}\,3)$$

(which intuitively should yield ?)

References

A reference allocation is not a value. Otherwise, by β, the program:

$$(\lambda x{:}\tau.\,(x := 1; !\,x))\;(\mathit{ref}\,3)$$

(which intuitively should yield 1) would reduce to:

References

A reference allocation is not a value. Otherwise, by β, the program:

$$(\lambda x{:}\tau.\,(x := 1; !\,x))\;(\mathit{ref}\,3)$$

(which intuitively should yield 1) would reduce to:

$$(\mathit{ref}\,3) := 1; !\,(\mathit{ref}\,3)$$

(which yields 3).

How shall we solve this problem?

References

$(\mathit{ref}\, 3)$ should first reduce to a value: the *address* of a fresh cell.

Not just the *content* of a cell matters, but also its address. Writing through one copy of the address should affect a future read via another copy.

References

We extend the simply-typed λ-calculus calculus with *memory locations*:

$$\begin{array}{rcl} V & ::= & \ldots \mid \ell \\ M & ::= & \ldots \mid \ell \end{array}$$

A memory location is just an atom (that is, a name). The value found at a location ℓ is obtained by indirection through a *memory* (or *store*).

A memory μ is a finite mapping of locations to *closed* values.

References

A *configuration* is a pair $M \mid \mu$ of a term and a store. The operational semantics (given next) reduces configurations instead of expressions.

The semantics maintains a no-dangling-pointers invariant: the locations that appear in M or in the image of μ are in the domain of μ.

- Initially, the store is empty, and the term contains no locations, because, by convention, memory locations cannot appear in source programs. So, the invariant holds.
- If we wish to start reduction with a non-empty store, we must check that the initial configuration satisfies the *no-dangling-pointers* invariant.

References

Because the semantics now reduces configurations, all existing reduction rules are augmented with a store, which they do not touch:

$$(\lambda x{:}\tau.\, M)\; V \,/\, \mu \longrightarrow [x \mapsto V]M \,/\, \mu$$

$$E[M] \,/\, \mu \longrightarrow E[M'] \,/\, \mu' \qquad \text{if } M \,/\, \mu \longrightarrow M' \,/\, \mu'$$

References

Because the semantics now reduces configurations, all existing reduction rules are augmented with a store, which they do not touch:

$$(\lambda x{:}\tau.\, M)\; V \,/\, \mu \longrightarrow [x \mapsto V]M \,/\, \mu$$

$$E[M] \,/\, \mu \longrightarrow E[M'] \,/\, \mu' \qquad \text{if } M \,/\, \mu \longrightarrow M' \,/\, \mu'$$

References

Because the semantics now reduces configurations, all existing reduction rules are augmented with a store, which they do not touch:

$$(\lambda x{:}\tau.\, M)\; V \,/\, \mu \longrightarrow [x \mapsto V]M \,/\, \mu$$

$$E[M] \,/\, \mu \longrightarrow E[M'] \,/\; \mu' \qquad \text{if } M \,/\, \mu \longrightarrow M' \,/\; \mu'$$

Three new reduction rules are added:

$$\mathit{ref}\, V \,/\, \mu \longrightarrow \ell \,/\, \mu[\ell \mapsto V] \qquad \text{if } \ell \notin \mathrm{dom}(\mu)$$

References

Because the semantics now reduces configurations, all existing reduction rules are augmented with a store, which they do not touch:

$$(\lambda x{:}\tau.\, M)\; V \,/\, \mu \longrightarrow [x \mapsto V]M \,/\, \mu$$

$$E[M] \,/\, \mu \longrightarrow E[M'] \,/\; \mu' \qquad \text{if } M \,/\, \mu \longrightarrow M' \,/\; \mu'$$

Three new reduction rules are added:

$$\mathit{ref}\, V \,/\, \mu \longrightarrow \ell \,/\, \mu[\ell \mapsto V] \qquad \text{if } \ell \notin \mathrm{dom}(\mu)$$

$$\ell := V \,/\, \mu \longrightarrow () \,/\, \mu[\ell \mapsto V]$$

References

Because the semantics now reduces configurations, all existing reduction rules are augmented with a store, which they do not touch:

$$(\lambda x{:}\tau.\, M)\; V \,/\, \mu \longrightarrow [x \mapsto V]M \,/\, \mu$$

$$E[M] \,/\, \mu \longrightarrow E[M'] \,/\; \mu' \qquad \text{if } M \,/\, \mu \longrightarrow M' \,/\; \mu'$$

Three new reduction rules are added:

$$\mathit{ref}\, V \,/\, \mu \longrightarrow \ell \,/\, \mu[\ell \mapsto V] \qquad \text{if } \ell \notin \mathrm{dom}(\mu)$$

$$\ell := V \,/\, \mu \longrightarrow () \,/\, \mu[\ell \mapsto V]$$

$$!\,\ell \,/\, \mu \longrightarrow \mu(\ell) \,/\, \mu$$

Notice: In the last two rules, the no-dangling-pointers invariant guarantees $\ell \in \mathrm{dom}(\mu)$.

References

The type system is modified as follows. Types are extended:

$$\tau ::= \ldots \mid \mathit{ref}\,\tau$$

Three new typing rules are introduced:

$$\textsc{Ref}\ \frac{\Gamma \vdash M : \tau}{\Gamma \vdash \mathit{ref}\,M : \mathit{ref}\,\tau} \qquad \textsc{Set}\ \frac{\Gamma \vdash M_1 : \mathit{ref}\,\tau \qquad \Gamma \vdash M_2 : \tau}{\Gamma \vdash M_1 := M_2 : \mathit{unit}} \qquad \textsc{Get}\ \frac{\Gamma \vdash M : \mathit{ref}\,\tau}{\Gamma \vdash\, !\,M : \tau}$$

Is that all we need?

References

The preceding setup is enough to typecheck *source terms*, but does not allow stating or proving type soundness.

Indeed, we have not yet answered these questions:

- What is the type of a memory location ℓ?
- When is a configuration $M \,/\, \mu$ well-typed?

References

When does a location ℓ have type *ref* τ?

?

References

When does a location ℓ have type $\mathit{ref}\,\tau$?

A possible answer is, *when it points to some value of type τ*.
Intuitively, this could be formalized by a typing rule of the form:

$$\frac{\mu, \varnothing \vdash \mu(\ell) : \tau}{\mu, \Gamma \vdash \ell : \mathit{ref}\,\tau}$$

Comments?

?

References

When does a location ℓ have type *ref* τ?

A possible answer is, *when it points to some value of type* τ.
Intuitively, this could be formalized by a typing rule of the form:

$$\frac{\mu, \varnothing \vdash \mu(\ell) : \tau}{\mu, \Gamma \vdash \ell : \mathit{ref}\,\tau}$$

Comments?

- Typing judgments would have the form $\mu, \Gamma \vdash M : \tau$.
  However, they would no longer be *inductively* defined (or else, every cyclic structure would be ill-typed). Instead, *co-induction* would be required.

References

When does a location ℓ have type *ref* τ?

A possible answer is, *when it points to some value of type* τ.
Intuitively, this could be formalized by a typing rule of the form:

$$\frac{\mu, \varnothing \vdash \mu(\ell) : \tau}{\mu, \Gamma \vdash \ell : \mathit{ref}\,\tau}$$

Comments?

- Typing judgments would have the form $\mu, \Gamma \vdash M : \tau$.
  However, they would no longer be *inductively* defined (or else, every cyclic structure would be ill-typed). Instead, *co-induction* would be required.
- Moreover, if the value $\mu(\ell)$ happens to admit two distinct types τ_1 and τ_2, then ℓ admits types *ref* τ_1 and *ref* τ_2. So, one can write at type τ_1 and read at type τ_2: this rule is *unsound!*

References

A simpler and sound approach is to fix the type of a memory location when it is first allocated. To do so, we use a *store typing* Σ, a finite mapping of locations to types.

So, when does a location ℓ have type *ref* τ? "When Σ says so."

$$\textsc{Loc}\quad \Sigma, \Gamma \vdash \ell : \mathit{ref}\, \Sigma(\ell)$$

Comments:

- Typing judgments now have the form $\Sigma, \Gamma \vdash M : \tau$.

References

How do we know that the store typing predicts appropriate types?

?

References

How do we know that the store typing predicts appropriate types?

This is required by the typing rules for stores and configurations:

STORE

$$\frac{}{\vdash \mu : \Sigma}$$

CONFIG

$$\frac{}{\vdash M \,/\, \mu : \tau}$$

References

How do we know that the store typing predicts appropriate types?

This is required by the typing rules for stores and configurations:

$$\textsc{Store}\ \frac{\forall \ell \in \mathrm{dom}(\mu), \qquad \vdash \mu(\ell) : \Sigma(\ell)}{\vdash \mu : \Sigma} \qquad \textsc{Config}\ \frac{}{\vdash M \,/\, \mu : \tau}$$

References

How do we know that the store typing predicts appropriate types?

This is required by the typing rules for stores and configurations:

$$\textsc{Store}\ \frac{\forall \ell \in \mathrm{dom}(\mu), \quad \Sigma, \varnothing \vdash \mu(\ell) : \Sigma(\ell)}{\vdash \mu : \Sigma} \qquad \textsc{Config}\ \frac{}{\vdash M \,/\, \mu : \tau}$$

References

How do we know that the store typing predicts appropriate types?

This is required by the typing rules for stores and configurations:

$$\textsc{Store}\quad \frac{\forall \ell \in \mathrm{dom}(\mu), \quad \Sigma, \varnothing \vdash \mu(\ell) : \Sigma(\ell)}{\vdash \mu : \Sigma} \qquad \textsc{Config}\quad \frac{\vdash \mu : \Sigma \qquad \Sigma, \varnothing \vdash M : \tau}{\vdash M \,/\, \mu : \tau}$$

References

How do we know that the store typing predicts appropriate types?

This is required by the typing rules for stores and configurations:

$$\textsc{Store}\quad \frac{\forall \ell \in \mathrm{dom}(\mu), \quad \Sigma, \varnothing \vdash \mu(\ell) : \Sigma(\ell)}{\vdash \mu : \Sigma} \qquad \textsc{Config}\quad \frac{\vdash \mu : \Sigma \qquad \Sigma, \varnothing \vdash M : \tau}{\vdash M \,/\, \mu : \tau}$$

Comments:

- This is an *inductive* definition. The store typing Σ serves both as an assumption (Loc) and a goal (Store). Cyclic stores are not a problem.
- The store typing is used only in the definition of a "well-typed configuration" and in the typechecking of locations. Thus, it is not needed for type-checking source programs, since the store is empty and the empty-store configuration is always well-typed.
- **. . . ?**

References

How do we know that the store typing predicts appropriate types?

This is required by the typing rules for stores and configurations:

$$\textsc{Store}\quad \frac{\forall \ell \in \mathrm{dom}(\mu), \quad \Sigma, \varnothing \vdash \mu(\ell) : \Sigma(\ell)}{\vdash \mu : \Sigma} \qquad \textsc{Config}\quad \frac{\vdash \mu : \Sigma \qquad \Sigma, \varnothing \vdash M : \tau}{\vdash M \,/\, \mu : \tau}$$

Comments:

- This is an *inductive* definition. The store typing Σ serves both as an assumption (Loc) and a goal (Store). Cyclic stores are not a problem.
- The store typing is used only in the definition of a "well-typed configuration" and in the typechecking of locations. Thus, it is not needed for type-checking source programs, since the store is empty and the empty-store configuration is always well-typed.
- Notice that Σ does not appear in the conclusion of Config.

Restating type soundness

The type soundness statements are slightly modified in the presence of the store, since we now reduce configurations:

Theorem (Subject reduction)

Reduction preserves types: if $M \mid \mu \longrightarrow M' \mid \mu'$ *and* $\vdash M \mid \mu : \tau$, *then* $\vdash M' \mid \mu' : \tau$.

Theorem (Progress)

If $M \mid \mu$ *is a well-typed, irreducible configuration, then* M *is a value.*

Restating subject reduction

Inlining CONFIG, subject reduction can also be restated as:

Theorem (Subject reduction, expanded)

If $M \mid \mu \longrightarrow M' \mid \mu'$ *and* $\vdash \mu : \Sigma$ *and* $\Sigma, \varnothing \vdash M : \tau$*, then there exists* Σ' *such that* $\vdash \mu' : \Sigma'$ *and* $\Sigma', \varnothing \vdash M' : \tau$*.*

This statement is correct, but *too weak*—its proof by induction will fail in one case. (Which one?)

Establishing subject reduction

Let us look at the case of reduction under a context.

The hypotheses are:

$$M \,/\, \mu \longrightarrow M' \,/\, \mu' \quad \text{and} \quad \vdash \mu : \Sigma \quad \text{and} \quad \Sigma, \varnothing \vdash E[M] : \tau$$

Establishing subject reduction

Let us look at the case of reduction under a context.

The hypotheses are:

$$M \,/\, \mu \longrightarrow M' \,/\, \mu' \quad \text{and} \quad \vdash \mu : \Sigma \quad \text{and} \quad \Sigma, \varnothing \vdash E[M] : \tau$$

Assuming compositionality, there exists τ' such that:

$$\Sigma, \varnothing \vdash M : \tau' \quad \text{and} \quad \forall M', \quad (\Sigma, \varnothing \vdash M' : \tau') \Rightarrow (\Sigma, \varnothing \vdash E[M'] : \tau)$$

Establishing subject reduction

Let us look at the case of reduction under a context.

The hypotheses are:

$$M \,/\, \mu \longrightarrow M' \,/\, \mu' \quad \text{and} \quad \vdash \mu : \Sigma \quad \text{and} \quad \Sigma, \varnothing \vdash E[M] : \tau$$

Assuming compositionality, there exists τ' such that:

$$\Sigma, \varnothing \vdash M : \tau' \quad \text{and} \quad \forall M', \quad (\Sigma, \varnothing \vdash M' : \tau') \Rightarrow (\Sigma, \varnothing \vdash E[M'] : \tau)$$

Then, by the induction hypothesis, there exists Σ' such that:

$$\vdash \mu' : \Sigma' \quad \text{and} \quad \Sigma', \varnothing \vdash M' : \tau'$$

Establishing subject reduction

Let us look at the case of reduction under a context.

The hypotheses are:

$$M \,/\, \mu \longrightarrow M' \,/\, \mu' \quad \text{and} \quad \vdash \mu : \Sigma \quad \text{and} \quad \Sigma, \varnothing \vdash E[M] : \tau$$

Assuming compositionality, there exists τ' such that:

$$\Sigma, \varnothing \vdash M : \tau' \quad \text{and} \quad \forall M', \quad (\Sigma, \varnothing \vdash M' : \tau') \Rightarrow (\Sigma, \varnothing \vdash E[M'] : \tau)$$

Then, by the induction hypothesis, there exists Σ' such that:

$$\vdash \mu' : \Sigma' \quad \text{and} \quad \Sigma', \varnothing \vdash M' : \tau'$$

Here, *we are stuck*. The context E is well-typed under Σ, but the term M' is well-typed under Σ', so we cannot combine them.

How can we fix this?

Establishing subject reduction

We are missing a key property: *the store typing grows with time*.
That is,

?

Establishing subject reduction

We are missing a key property: *the store typing grows with time*.
That is, although new memory locations can be allocated, *the type of an existing location does not change*.

This is formalized by strengthening the subject reduction statement:

Theorem (Subject reduction, strengthened)

If $M \mid \mu \longrightarrow M' \mid \mu'$ *and* $\vdash \mu : \Sigma$ *and* $\Sigma, \varnothing \vdash M : \tau$, *then there exists* Σ' *such that* $\vdash \mu' : \Sigma'$ *and* $\Sigma', \varnothing \vdash M' : \tau$ *and* $\Sigma \subseteq \Sigma'$.

At each reduction step, the new store typing Σ' extends the previous store typing Σ.

Establishing subject reduction

Growing the store typing preserves well-typedness:

Lemma (Stability under memory allocation)

If $\Sigma \subseteq \Sigma'$ *and* $\Sigma, \Gamma \vdash M : \tau$, *then* $\Sigma', \Gamma \vdash M : \tau$.

(This is a generalization of the weakening lemma.)

Establishing subject reduction

Stability under memory allocation allows establishing a strengthened version of compositionality:

Lemma (Compositionality)

Assume $\Sigma, \varnothing \vdash E[M] : \tau$. *Then, there exists* τ' *such that:*

- $\Sigma, \varnothing \vdash M : \tau'$,
- *for every* Σ' *such that* $\Sigma \subseteq \Sigma'$, *for every* M', $\Sigma', \varnothing \vdash M' : \tau'$ *implies* $\Sigma', \varnothing \vdash E[M'] : \tau$.

Establishing subject reduction

Let us now look again at the case of reduction under a context.

The hypotheses are:

$$\vdash \mu : \Sigma \quad \text{and} \quad \Sigma, \varnothing \vdash E[M] : \tau \quad \text{and} \quad M \,/\, \mu \longrightarrow M' \,/\, \mu'$$

Establishing subject reduction

Let us now look again at the case of reduction under a context.

The hypotheses are:

$$\vdash \mu : \Sigma \quad \text{and} \quad \Sigma, \varnothing \vdash E[M] : \tau \quad \text{and} \quad M \,/\, \mu \longrightarrow M' \,/\, \mu'$$

By compositionality, there exists τ' such that:

$\Sigma, \varnothing \vdash M : \tau'$

$\forall \Sigma', \forall M', \quad (\Sigma \subseteq \Sigma') \Rightarrow (\Sigma', \varnothing \vdash M' : \tau') \Rightarrow (\Sigma', \varnothing \vdash E[M'] : \tau')$

Establishing subject reduction

Let us now look again at the case of reduction under a context.

The hypotheses are:

$$\vdash \mu : \Sigma \quad \text{and} \quad \Sigma, \varnothing \vdash E[M] : \tau \quad \text{and} \quad M \,/\, \mu \longrightarrow M' \,/\, \mu'$$

By compositionality, there exists τ' such that:

$$\Sigma, \varnothing \vdash M : \tau'$$

$$\forall \Sigma', \forall M', \quad (\Sigma \subseteq \Sigma') \Rightarrow (\Sigma', \varnothing \vdash M' : \tau') \Rightarrow (\Sigma', \varnothing \vdash E[M'] : \tau')$$

By the induction hypothesis, there exists Σ' such that:

$$\vdash \mu' : \Sigma' \quad \text{and} \quad \Sigma', \varnothing \vdash M' : \tau' \quad \text{and} \quad \Sigma \subseteq \Sigma'$$

The goal immediately follows.

On memory deallocation

In ML, memory deallocation is implicit. It must be performed by the runtime system, possibly with the cooperation of the compiler.

The most common technique is *garbage collection*. A more ambitious technique, implemented in the ML Kit, is compile-time *region analysis* [Tofte et al., 2004].

References in ML are easy to type-check, thanks in large part to the *no-dangling-pointers* property of the semantics.

Making memory deallocation an explicit operation, while preserving type soundness, is possible, but difficult. This requires reasoning about *aliasing* and *ownership*. See Charguéraud and Pottier [2008] for citations.

See also the Mezzo language [Pottier and Protzenko, 2013] designed especially for the explicit control of resources.

A similar approach is taken in the language Rust.

Contents

Combining extensions

We have shown how to extend simply-typed λ-calculus, independently, with:

- polymorphism, and
- references.

Can these two extensions be combined?

Beware of polymorphic locations!

When adding references, we noted that type soundness relies on the fact that *every reference cell (or memory location) has a fixed type*.

Otherwise, if a location had two types $\mathit{ref}\,\tau_1$ and $\mathit{ref}\,\tau_2$, one could store a value of type τ_1 and read back a value of type τ_2.

Hence, it should also be *unsound if a location could have type* $\forall\alpha.\,\mathit{ref}\,\tau$ (where α appears in τ) as it could then be specialized to both types $\mathit{ref}\,([\alpha \mapsto \tau_1]\tau)$ and $\mathit{ref}\,([\alpha \mapsto \tau_2]\tau)$.

By contrast, *a location* ℓ *can have type* $\mathit{ref}\,(\forall\alpha.\,\tau)$: this says that ℓ stores values of polymorphic type $\forall\alpha.\,\tau$, but ℓ, as a value, is viewed with the monomorphic type $\mathit{ref}\,(\forall\alpha.\,\tau)$.

A counter example

Still, if naively extended with references, System F allows construction of polymorphic references, which breaks subject reduction:

A counter example

Still, if naively extended with references, System F allows construction of polymorphic references, which breaks subject reduction:

$$\begin{array}{l} \mathit{let}\ y : \forall\alpha.\ \mathit{ref}\ (\alpha \to \alpha) = \Lambda\alpha.\ \mathit{ref}\ (\alpha \to \alpha)\ (\lambda z{:}\alpha.\ z)\ \mathit{in} \\ \quad (y\ \mathit{bool}) := (\mathit{bool} \to \mathit{bool})\ not; \\ \quad !(\mathit{int} \to \mathit{int})\ (y\ \mathit{int})\ 1 \mid \varnothing \\ \xrightarrow{*}\ not\ 1 \mid \ell \mapsto not \end{array}$$

A counter example

Still, if naively extended with references, System F allows construction of polymorphic references, which breaks subject reduction:

$$\begin{array}{l} \mathit{let}\, y : \forall\alpha.\, \mathit{ref}\,(\alpha \to \alpha) = \Lambda\alpha.\, \mathit{ref}\,(\alpha \to \alpha)\ (\lambda z{:}\alpha.\, z)\ \mathit{in} \\ \quad (y\ \mathit{bool}) := (\mathit{bool} \to \mathit{bool})\ \mathit{not}; \\ \quad !(\mathit{int} \to \mathit{int})\ (y\ \mathit{int})\ 1 \,/\, \varnothing \\ \xrightarrow{*}\ \mathit{not}\ 1 \,/\, \ell \mapsto \mathit{not} \end{array}$$

What happens is that the evaluation of the reference:

- creates and returns a location ℓ bound to the identity function $\lambda z{:}\alpha.\, z$ of type $\alpha \to \alpha$,
- abstracts α in the result and binds it to y with the polymorphic type $\forall\alpha.\, \mathit{ref}\,(\alpha \to \alpha)$;
- writes the location at type $\mathit{ref}\,(\mathit{bool} \to \mathit{bool})$ and reads it back at type $\mathit{ref}\,(\mathit{int} \to \mathit{int})$.

Nailing the bug

In the counter-example, the first reduction step uses the following rule (where V is $\lambda x{:}\alpha.\, x$ and τ is $\alpha \to \alpha$).

$$\textsc{Context}\ \frac{\mathit{ref}\,\tau\ V \,/\, \varnothing \longrightarrow \ell \,/\, \ell \mapsto V}{\Lambda\alpha.\, \mathit{ref}\,\tau\ V \,/\, \varnothing \longrightarrow \Lambda\alpha.\, \ell \,/\, \ell \mapsto V}$$

?

Nailing the bug

In the counter-example, the first reduction step uses the following rule (where V is $\lambda x{:}\alpha.\, x$ and τ is $\alpha \to \alpha$).

$$\textsc{Context}\ \frac{\mathit{ref}\,\tau\ V \mid \varnothing \longrightarrow \ell \mid \ell \mapsto V}{\Lambda\alpha.\,\mathit{ref}\,\tau\ V \mid \varnothing \longrightarrow \Lambda\alpha.\,\ell \mid \ell \mapsto V}$$

While we have

$$\alpha \vdash \mathit{ref}\,\tau\ V \mid \varnothing : \mathit{ref}\,\tau \qquad \text{and} \qquad \alpha \vdash \ell \mid \ell \mapsto V : \mathit{ref}\,\tau$$

We have

$$\vdash \Lambda\alpha.\,\mathit{ref}\,\tau\ V \mid \varnothing : \forall\alpha.\,\mathit{ref}\,\tau \qquad \text{but not} \qquad \vdash \Lambda\alpha.\,\ell \mid \ell \mapsto V : \forall\alpha.\,\mathit{ref}\,\tau$$

Hence, the context case of subject reduction breaks.

Nailing the bug

The typing derivation of $\Lambda\alpha.\,\ell$ requires a store typing Σ of the form $\ell : \tau$ and a derivation of the form:

$$\textsc{Tabs}\ \frac{\Sigma, \alpha \vdash \ell : \mathit{ref}\,\tau}{\Sigma \vdash \Lambda\alpha.\,\ell : \forall\alpha.\,\mathit{ref}\,\tau}$$

However, the typing context Σ, α is ill-formed as α appears free in Σ.

Instead, a well-formed premise should bind α earlier as in $\alpha, \Sigma \vdash \ell : \mathit{ref}\,\tau$, but then, Rule Tabs cannot be applied.

By contrast, the expression $\mathit{ref}\,\tau\ V$ is pure, so Σ may be empty:

$$\textsc{Tabs}\ \frac{\alpha \vdash \mathit{ref}\,\tau\ V : \mathit{ref}\,\tau}{\varnothing \vdash \Lambda\alpha.\,\mathit{ref}\,\tau\ V : \forall\alpha.\,\mathit{ref}\,\tau}$$

The expression $\Lambda\alpha.\,\ell$ is correctly rejected as ill-typed, so $\Lambda\alpha.\,(\mathit{ref}\,\tau\ V)$ should also be rejected.

Fixing the bug

Mysterious slogan:

> *One must not abstract over a type variable that might, after evaluation of the term, enter the store typing.*

Indeed, this is what happens in our example. The type variable α which appears in the type $\alpha \to \alpha$ of V is abstracted in front of *ref* $(\alpha \to \alpha)\ V$.

When *ref* $(\alpha \to \alpha)\ V$ reduces, $\alpha \to \alpha$ becomes the type of the fresh location ℓ, which appears in the new store typing.

This is all well and good, but *how* do we enforce this slogan?

Fixing the bug

In the context of ML, a number of rather complex historic approaches have been followed: see Leroy [1992] for a survey.

Then came Wright [1995], who suggested an amazingly simple solution, known as the *value restriction:* only value forms can be abstracted over.

$$\textsc{TAbs}\ \frac{\Gamma, \alpha \vdash u : \tau}{\Gamma \vdash \Lambda\alpha.\, u : \forall\alpha.\tau}$$

$$\textsc{Value forms:}\quad u ::= x \mid V \mid \Lambda\alpha.\, u \mid u\ \tau$$

The problematic proof case *vanishes*, as we now never $\beta\delta$-reduce under type abstraction—only ι-reduction is allowed.

Subject reduction holds again.

A good intuition: internalizing configurations

A configuration $M \,/\, \mu$ is an expression M in a memory μ. The memory can be viewed as a recursive extensible record.

The configuration $M \,/\, \mu$ may be viewed as the recursive definition (of values) $\mathit{let\ rec}\ m : \Sigma = \mu\ \mathit{in}\ [\ell \mapsto m.\ell]M$ where Σ is a store typing for μ.

The store typing rules are coherent with this view.

Allocation of a reference is a reduction of the form

$$\begin{array}{rlll} & \mathit{let\ rec}\ m : \Sigma & = \mu & \mathit{in}\ E[\mathit{ref}\,\tau\ V] \\ \longrightarrow & \mathit{let\ rec}\ m : \Sigma, \ell : \tau & = \mu, \ell \mapsto V & \mathit{in}\ E[m.\ell] \end{array}$$

For this transformation to preserve well-typedness, it is clear that the evaluation context E *must not bind any free type variable of* τ.

Otherwise, we are violating the scoping rules.

Clarifying the typing rules

Let us review the typing rules for configurations:

?

Clarifying the typing rules

Let us review the typing rules for configurations:

CONFIG

$$\frac{\Sigma, \varnothing \vdash M : \tau \qquad \vdash \mu : \Sigma}{\vdash M \,/\, \mu : \tau}$$

STORE

$$\frac{\forall \ell \in \mathrm{dom}(\mu), \qquad \Sigma, \varnothing \vdash \mu(\ell) : \Sigma(\ell)}{\vdash \mu : \Sigma}$$

Clarifying the typing rules

Let us review the typing rules for configurations:

CONFIG

$$\frac{\vec{\alpha}, \Sigma, \varnothing \vdash M : \tau \qquad \vec{\alpha} \vdash \mu : \Sigma}{\vec{\alpha} \vdash M \,/\, \mu : \tau}$$

STORE

$$\frac{\forall \ell \in \mathrm{dom}(\mu), \quad \vec{\alpha}, \Sigma, \varnothing \vdash \mu(\ell) : \Sigma(\ell)}{\vec{\alpha} \vdash \mu : \Sigma}$$

Clarifying the typing rules

Let us review the typing rules for configurations:

$$\textsc{Config}\quad \frac{\vec{\alpha}, \Sigma, \varnothing \vdash M : \tau \qquad \vec{\alpha} \vdash \mu : \Sigma}{\vec{\alpha} \vdash M \,/\, \mu : \tau} \qquad \textsc{Store}\quad \frac{\forall \ell \in \mathrm{dom}(\mu), \quad \vec{\alpha}, \Sigma, \varnothing \vdash \mu(\ell) : \Sigma(\ell)}{\vec{\alpha} \vdash \mu : \Sigma}$$

Remarks:

- Closed configurations are typed in an environment just composed of type variables $\vec{\alpha}$.
- $\vec{\alpha}$ may appear in the store during reduction.
  Take for example, M equal to $\mathit{ref}\,(\alpha \to \alpha)\; V$ where V is $\lambda x{:}\alpha.\, x$.
- Thus $\vec{\alpha}$ will also appear in the store typing and should be placed in front of the store typing; no β in $\vec{\alpha}$ can be generalized.
- New type variables cannot be introduced during reduction.

Clarifying the typing rules

Judgments are now of the form $\vec{\alpha}, \Sigma, \Gamma \vdash M : \tau$ although we may see $\vec{\alpha}, \Sigma, \Gamma$ as a whole typing context Γ'.

For locations, we need a new context formation rule:

$$\frac{\vdash \Gamma \qquad \Gamma \vdash \tau \qquad \ell \notin \mathrm{dom}(\Gamma)}{\vdash \Gamma, \ell : \tau}\ \textsc{WfEnvLoc}$$

This allows locations to appear anywhere. However, in a derivation of a closed term, the typing context will always be of the form $\vec{\alpha}, \Sigma, \Gamma$ where:

- Σ only binds locations (to arbitrary types) and
- Γ does not bind locations.

Clarifying the typing rules

The typing rule for memory locations (where Γ is of the form $\vec{\alpha}, \Sigma, \Gamma'$)

$$\frac{}{\Gamma \vdash \ell : \mathit{ref}\, \Gamma(\ell)}\ \textsc{Loc}$$

In System F, typing rules for references need not be primitive.
We may instead treat them as constants of the following types:

$$\begin{array}{rcl} \mathit{ref} & : & \forall\alpha.\, \alpha \to \mathit{ref}\, \alpha \\ (!) & : & \forall\alpha.\, \mathit{ref}\, \alpha \to \alpha \\ (:=) & : & \forall\alpha.\, \mathit{ref}\, \alpha \to \alpha \to \mathit{unit} \end{array}$$

Which ones are constructors?

Clarifying the typing rules

The typing rule for memory locations (where Γ is of the form $\vec{\alpha}, \Sigma, \Gamma'$)

$$\frac{}{\Gamma \vdash \ell : \mathit{ref}\,\Gamma(\ell)}\ \textsc{Loc}$$

In System F, typing rules for references need not be primitive.
We may instead treat them as constants of the following types:

$$\begin{array}{rcl} \mathit{ref} & : & \forall\alpha.\,\alpha \to \mathit{ref}\,\alpha \\ (!) & : & \forall\alpha.\,\mathit{ref}\,\alpha \to \alpha \\ (:=) & : & \forall\alpha.\,\mathit{ref}\,\alpha \to \alpha \to \mathit{unit} \end{array}$$

There are all destructors (event *ref*) with the obvious arities.

The δ-rules are adapted to carry explicit type parameters:

$$\begin{array}{rcll} \mathit{ref}\,\tau\ V \,/\, \mu & \longrightarrow & \ell \,/\, \mu[\ell \mapsto V] & \text{if } \ell \notin \mathrm{dom}(\mu) \\ \ell := (\tau)\ V \,/\, \mu & \longrightarrow & () \,/\, \mu[\ell \mapsto V] & \\ !\tau\ \ell \,/\, \mu & \longrightarrow & \mu(\ell) \,/\, \mu & \end{array}$$

Stating type soundness

Lemma (Subject reduction for constants)

δ-rules preserve well-typedness of closed configurations.

Theorem (Subject reduction)

Reduction of closed configurations preserves well-typedness.

Lemma (Progress for constants)

A well-typed closed configuration M/μ where M is a full application of constants ref, (!), or (:=) to types and values can always be reduced.

Theorem (Progress)

A well-typed irreducible closed configuration M/μ is a value.

Consequences

The problematic program is now syntactically ill-formed:

$$\begin{array}{l} \mathit{let}\ y : \forall\alpha.\, \mathit{ref}\,(\alpha \to \alpha) = \Lambda\alpha.\, \mathit{ref}\,(\lambda z{:}\alpha.\, z)\ \mathit{in} \\ \quad (:=)\ (\mathit{bool} \to \mathit{bool})\ (y\ \mathit{bool})\ not; \\ \quad !\,(\mathit{int} \to \mathit{int})\ (y\ (\mathit{int}))\ 1 \end{array}$$

Indeed, $\mathit{ref}\,(\lambda z{:}\alpha.\, z)$ is not a value form, but the application of a unary destructor to a value, so it cannot be generalized.

Value restriction limitations

With the value restriction, some pure programs become ill-typed, even though they were well-typed in the absence of references.

Therefore, this style of introducing references in System F (or in ML) is *not a conservative extension.*

Assuming:

$$map : \forall\alpha.\,\forall\beta.\,(\alpha \to \beta) \to \mathit{list}\,\alpha \to \mathit{list}\,\beta \qquad id : \forall\alpha.\,\alpha \to \alpha$$

This expression becomes ill-typed:

$$\Lambda\alpha.\,map\;\alpha\;\alpha\;(id\;\alpha)$$

?

Value restriction limitations

With the value restriction, some pure programs become ill-typed, even though they were well-typed in the absence of references.

Therefore, this style of introducing references in System F (or in ML) is *not a conservative extension.*

Assuming:

$$map : \forall\alpha.\,\forall\beta.\,(\alpha \to \beta) \to \mathit{list}\,\alpha \to \mathit{list}\,\beta \qquad id : \forall\alpha.\,\alpha \to \alpha$$

This expression becomes ill-typed:

$$\Lambda\alpha.\,map\;\alpha\;\alpha\;(id\;\alpha)$$

A common work-around is to perform a manual η-expansion:

$$\Lambda\alpha.\,\lambda y{:}\mathit{list}\,\alpha.\,map\;\alpha\;(id\;\alpha)\;y$$

?

Value restriction limitations

With the value restriction, some pure programs become ill-typed, even though they were well-typed in the absence of references.

Therefore, this style of introducing references in System F (or in ML) is *not a conservative extension.*

Assuming:

$$map : \forall\alpha.\,\forall\beta.\,(\alpha \to \beta) \to \mathit{list}\,\alpha \to \mathit{list}\,\beta \qquad id : \forall\alpha.\,\alpha \to \alpha$$

This expression becomes ill-typed:

$$\Lambda\alpha.\,map\;\alpha\;\alpha\;(id\;\alpha)$$

A common work-around is to perform a manual η-expansion:

$$\Lambda\alpha.\,\lambda y{:}\mathit{list}\,\alpha.\,map\;\alpha\;(id\;\alpha)\;y$$

Of course, in the presence of side effects, η-expansion is *not* semantics-preserving, so this must not be done blindly.

Value restriction Extensions

Non-expansive expressions

The value restriction can be slightly relaxed by enlarging the class of value-forms to a syntactic category of so-called *non-expansive terms*—terms whose evaluation will definitely not allocate new reference cells. Non-expansive terms form a strict superset of value-forms.

$$\begin{array}{rcl} u & ::= & x \mid V \mid \Lambda\alpha.\, u \mid u\,\tau \\ & \mid & \mathit{let}\, x = u\, \mathit{in}\, u \mid (\lambda x{:}\tau.\, u)\, u \\ & \mid & C\, u_1 \ldots u_k \\ & \mid & d\, u_1 \ldots u_k \end{array} \qquad \text{where either} \left[\begin{array}{l} k < \mathit{arity}\,(d) \\ d \text{ is non-expansive.} \end{array}\right.$$

In particular, pattern matching is a non-expansive destructor! But *ref* $\cdot$ is an expansive one!.

For example, the following expression is non-exapnsive:

$$\Lambda\alpha.\, \mathit{let}\, x = (\mathit{match}\, y\, \mathit{with}\, (C_i\, \bar{x}_i \to u_i)^{\,i\in I})\, \mathit{in}\, u$$

Value restriction

Extensions

Positive occurrences: Garrigue [2004] relaxes the value restriction in a more subtle way, which is justified by a subtyping argument.

For instance, *let* $x : \forall\alpha.\, \mathit{list}\ \alpha = \Lambda\alpha.\, (M_1\ M_2)$ *in* M may be well-typed because because α appears only positively in the type of $M_1\ M_2$.

More generally, given a type context $T[\alpha]$ where α appears only positively

- $\forall\alpha.T[\alpha]$ can be instantiated to $T[\forall\alpha.\alpha]$, and
- $T[\forall\alpha.\alpha]$ is a subtype of $\forall\alpha.T[\alpha]$

Hence, a value of type $T[\alpha]$ can be given the monomorphic type $T[\forall\alpha.\alpha]$ by weakening before entering the store to please the value restriction, but retrieved at type $\forall\alpha.T[\alpha]$, a subtype of $T[\forall\alpha.\alpha]$.

OCaml implements this, but restricts it to *strictly positive* occurrences so as to keep the principal type property.

Value restriction Extensions

In fact, the two extensions can be combined: $\Lambda\alpha.\,M$ *need only be forbidden* when

> α *appears in the type of some exposed expansive subterm at some negative occurrence,*

where exposed subterms are those that do not appear under some λ-abstraction.

For instance, the expression

$$\begin{array}{l} \mathit{let}\ x : \forall\alpha.\ \mathit{int} \times (\mathit{list}\ \alpha) \times (\alpha \to \alpha) = \\ \quad \Lambda\alpha.\,(\mathit{ref}\ (1+2),\ (\lambda x{:}\alpha.\,x)\ \mathit{Nil},\ \lambda x{:}\alpha.\,x) \\ \mathit{in}\ M \end{array}$$

may be accepted because α appears only in the type of the non-expansive exposed expression $\lambda x{:}\alpha.\,x$ and only positively in the type of the expansive expression $(\lambda x{:}\alpha.\,x)\ \mathit{Nil}$.

Conclusions

Experience has shown that *the value restriction is tolerable.* Even though it is not conservative, the search for better solutions has been pretty much abandoned.

There is still on going research for tracing side effects more precisely, in particular to better circumvent their use.

Actually, there is a regained interest in tracing side effects, with the introduction of effect handlers.

Conclusions

In a type-and-effect system [Lucassen and Gifford, 1988; Talpin and Jouvelot, 1994], or in a type-and-capability system [Charguéraud and Pottier, 2008], the type system indicates which expressions may allocate new references, and at which type. This permits strong updates—updates that may also change the type of references.

There, the value restriction is no longer necessary.

However, if one extends a type-and-capability system with a mechanism for *hiding* state, the need for the value restriction re-appears.

Pottier and Protzenko [2012] (and [Protzenko, 2014]) designed a language, called Mezzo, where mutable state is tracked very precisely, using permissions, ownership, and afine types.

Bibliography I

(Most titles have a clickable mark "▷" that links to online versions.)

▷ Arthur Charguéraud and François Pottier. Functional translation of a calculus of capabilities. In *ACM International Conference on Functional Programming (ICFP)*, pages 213–224, September 2008.

▷ Jacques Garrigue. Relaxing the value restriction. In *Functional and Logic Programming*, volume 2998 of *Lecture Notes in Computer Science*, pages 196–213. Springer, April 2004.

▷ Xavier Leroy. *Typage polymorphe d'un langage algorithmique*. PhD thesis, Université Paris 7, June 1992.

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

Bibliography II

- Simon Peyton Jones. Tackling the awkward squad: monadic input/output, concurrency, exceptions, and foreign-language calls in Haskell. Online lecture notes, January 2009.
- Simon Peyton Jones and Philip Wadler. Imperative functional programming. In *ACM Symposium on Principles of Programming Languages (POPL)*, pages 71–84, January 1993.

François Pottier and Jonathan Protzenko. Programming with permissions in Mezzo. Submitted for publication, October 2012.

François Pottier and Jonathan Protzenko. Programming with permissions in Mezzo. In *Proceedings of the 2013 ACM SIGPLAN International Conference on Functional Programming (ICFP'13)*, pages 173–184, September 2013.

- Jonathan Protzenko. *Mezzo, a typed language for safe effectful concurrent programs*. PhD thesis, University Paris Diderot, 2014.
- Jean-Pierre Talpin and Pierre Jouvelot. The type and effect discipline. *Information and Computation*, 11(2):245–296, 1994.

Bibliography III

- Mads Tofte, Lars Birkedal, Martin Elsman, and Niels Hallenberg. A retrospective on region-based memory management. *Higher-Order and Symbolic Computation*, 17(3):245–265, September 2004.
- Andrew K. Wright. Simple imperative polymorphism. *Lisp and Symbolic Computation*, 8(4):343–356, December 1995.