MPRI 2.4, Functional programming and type systems
Metatheory of System F

Didier Rémy

Plan of the course

Metatheory of System F

ADTs, Recursive types, Existential types, GATDs

Going higher order with F^ω!

Logical relations

Side effects, References, Value restriction

Type reconstruction

Overloading

Side effects, References, Value restriction

Contents

Referential transparency

What is it?

An expression is *referentially transparent* or *pure* if it can be replaced with its corresponding value without changing the program behavior. Applying a pure funtion to the same arguments returns the same result.

Why is it useful?

Allows to reason about programs as a rewrite system, which may help

- prove the correction,
- perform code optimization.
- typically, it allows for: memoization, common expression elimination, lazy evaluation, . . .
- with code parallelization, optimistic evaluation, transactions, . . .

Referential transparency *counter examples*

Examples of impure constructs

- Exceptions, References, reading/printing functions.
- Interaction with the file system.
- Date and random primitives, etc.

Termination?

According to the definition, the status of termination is unclear. (As they never return, they cannot actually be replaced by the result of their evaluation—except in Haskell that uses an explicit bottom value $\perp$.) Non-termination is usually considered impure: it breaks equational reasoning and most program transformations, as other impure constructs.

In practice, high-complexity is not so different from non-termimation. . .

Effects

Any source of impurity is usually called an *effect*.

Referential transparency *Summary*

Effects are unavoidable

Any programming language must have some impure aspects to communicate with the operating system.
Side effects may sometimes be encapsulated, *e.g.* a module with side effetcs may sometimes have a pure interface.

Mitigation of effects

So the questions are more whether:

- a large core of the language is pure/effect free (*e.g.* Haskell, Coq, Core System F) or effectful (most other languages); and/or
- side effects can be tracked, *e.g.* by the type system.
(Haskell, Koka, Rust, Mezzo, or algebraic effects)

The semantics of effects

Programs with effects cannot be described as a pure rewrite system.

- The semantics must be changed.
- Some of the properties will be lost

We shall see:

- Exceptions, which require a small change to the semantics
- References, which:
  - require a major change to the semantics
  - do not fit well with polymorphism—which needs to be restricted in the presence of effects.
- Values, or a larger class of *non-expansive expressions*, whose evaluation is effect free play a key role in the presence of effects.

In the presence of effects, deterministic, call-by-value semantics is always a huge source of simplification when not a requirement.

Contents

Exceptions Semantics

Exceptions are a mechanism for changing the normal order of evaluation usually, but not necessarily, in case something abnormal occurred.

When an exception is raised, the evaluation does not continue as usual: Shortcutting normal evaluation rules, the exception is propagated up into the evaluation context until some handler is found at which the evaluation resumes with the exceptional value received; if no handler is found, the exception had reached the toplevel and the result of the evaluation is the exception instead of a value.

We extend the language with a constructor form to raise an exception and a destructor form to catch an exception; we also extend the evaluation contexts:

$$\begin{array}{rcl} M & ::= & \ldots \mid \textit{raise } M \mid \textit{try } M \textit{ with } M \\ E & ::= & \ldots \mid \textit{raise } [] \mid \textit{try } [] \textit{ with } M \end{array}$$

Exceptions Semantics

We do not treat *raise* V as a value, since it stops the normal order of evaluation. Instead, reduction rules propagate and handle exceptions:

Raise

$$F[\textit{raise}\, V] \longrightarrow \textit{raise}\, V$$

Handle-Val

$$\textit{try}\, V\ \textit{with}\ M \longrightarrow V$$

Handle-Raise

$$\textit{try raise}\, V\ \textit{with}\ M \longrightarrow M\ V$$

Rule Raise uses an evaluation context F which stands for *any* E *other than try* $[]$ *with* M, so that it propagates an exception up the evaluation contexts, but not through a handler.

The case of the handler is treated by two specific rules:

- Rule Handle-Raise passes an exceptional value to its handler;
- Rule Handle-Val removes the handler around a value.

Exceptions Example

For example, assuming that K is $\lambda x.\,\lambda y.\,y$ and $M \longrightarrow V$, we have the following reduction:

$$\begin{array}{rlr}
 & \textit{try}\ K\ (\textit{raise}\ M)\ \textit{with}\ \lambda x.\,x & \text{by } \textsc{Context} \\
\longrightarrow & \textit{try}\ K\ (\textit{raise}\ V)\ \textit{with}\ \lambda x.\,x & \text{by } \textsc{Raise} \\
\longrightarrow & \textit{try}\ \textit{raise}\ V\ \textit{with}\ \lambda x.\,x & \text{by } \textsc{Handle-Raise} \\
\longrightarrow & (\lambda x.\,x)\ V & \text{by } \beta_v \\
\longrightarrow & V &
\end{array}$$

In particular, we do not have the following step,

$$\begin{array}{rlr}
 & \textit{try}\ K\ (\textit{raise}\ V)\ \textit{with}\ \lambda x.\,x & \text{by } \beta_v \\
\not\longrightarrow & \textit{try}\ \lambda y.\,y\ \textit{with}\ \lambda x.\,x \longrightarrow \lambda y.\,y &
\end{array}$$

since *raise* V is *not* a value, so the first β-reduction step is not allowed.

Exceptions — Typing rules

We assume given a *fixed type τ_{exn}* for exceptional values.

$$\textsc{Raise}\quad \frac{\Gamma \vdash M : \tau_{exn}}{\Gamma \vdash \textit{raise}\; M : \tau} \qquad \textsc{Try}\quad \frac{\Gamma \vdash M_1 : \tau \qquad \Gamma \vdash M_2 : \tau_{exn} \to \tau}{\Gamma \vdash \textit{try}\; M_1 \;\textit{with}\; M_2 : \tau}$$

There are some subtleties:

- Raise turns an expression of type τ_{exn} into an exception.
- Consistently, the handler has type $\tau_{exn} \to \tau$, since it receives the exception value of type *exn* as argument;
- An exceptional value of type *exn* may be raised in M_1 and used in M_2 without any visible flow at the type level.
  Hence, *raise* $\cdot$ and *try* $\cdot$ *with* $\cdot$ must agree on the type *exn*.
- Both premises of Rule Try must return values of the same type τ.
- *raise* M can have any type, as the current computation is aborted.

Exceptions — The type of exception

What should we choose for τ_{exn}? Well, any type:

- Choosing *unit*, exceptions will not carry any information.
- Choosing *int*, exceptions can report some error code.
- Choosing *string*, exceptions can report error messages.
- Using a sum type or better a variant type (tagged sum), with one case to describe each exceptional situation.

  This is the approach followed by ML, which declares a new extensible type *exn* for exceptions: this is a sum type, except that all cases are not declared in advance, but only as needed.
  (Extensible datatypes are available in OCaml since version 4.02.)

In all cases, the type of exception must be fixed in the whole program.

This is because *raise* · and *try* · *with* · must agree beforehand on the type of exceptions as this type is not passed around by the typing rules.

Encoding of multiple exceptions

Introduce a data type:

$$\text{type } \mathit{exn} = \Sigma(E_i : \tau_i \to \mathit{exn})^{i \in I}$$

Use syntactic sugar:

$$\mathit{raise}\ E_i\ v \triangleq \mathit{raise}\ (E_i\ v)$$

$$\begin{aligned} &\mathit{try}\ M\ \mathit{with}\ (E_j\ x \Rightarrow M_k)^{j \in J} \\ &\qquad \triangleq \quad \mathit{try}\ M\ \mathit{with} \\ &\qquad\qquad (\lambda z.\, \mathit{match}\ z\ \mathit{with}\ (E_j\ x \Rightarrow M_k)^{j \in J} \mid z \Rightarrow \mathit{raise}\ z) \end{aligned}$$

Exceptions — Type soundness

How do we state type soundness, since exceptions may be uncaught?

By saying that this is the only "exception" to progress:

Theorem (Progress)

A well-typed, irreducible term is either a value or an uncaught exception. if $\varnothing \vdash M : \tau$ and $M \not\longrightarrow$, then M is either V or raise V for some value V.

Exceptions Structured exceptions

What is the type *exn* for exceptions? Well, it could be any type:

- If we take the unit type, we only know that an except has been raised but cannot pass any other information.
- Hence, we could take the type of integers (*e.g.* passing error codes, much as commands do in Unix)
- Use some richer data type with one constructor per kind of error.
- Use a variant type (tagged sum)

The handler may analyze the argument of the exception.

Exceptions On uncaught exceptions

An uncaught exception is often a programming error. It may be surprising that they are not detected by the type system.

Exceptions may be detected using more expressive type systems. Unfortunately, the existing solutions are often complicated for some limited benefit, and are still not often used in practice.

The complication comes from the treatment of functions, which have some *latent effect* of possibly raising or catching an exception when applied. To be precise, the analysis must therefore enrich types of functions with latent effects, which is quite invasive and obfuscating.

Uncaught exceptions must be declared in the language Java. (Java also has untraced exceptions.)

See Leroy and Pessaux [2000] for a solution in ML.

Exceptions — Small semantic variation

Once raised, exceptions are propagated step-by-step by Rule RAISE until they reach a handler or the toplevel.

We can also describe their semantics by replacing propagation of exceptions by deep handling of exceptions inside terms.

Replace the three previous reduction rules by:

HANDLE-VAL'
$$\textit{try}\, V \;\textit{with}\; M \longrightarrow V$$

HANDLE-RAISE'
$$\textit{try}\, \bar{F}[\textit{raise}\, V]\; \textit{with}\; M \longrightarrow M\ V$$

where $\bar{F}$ is a sequence of F contexts, *i.e.* handler-free evaluation context of arbitrary depth.

This semantics is perhaps more intuitive, closer to what a compiler does, but the two presentations are equivalent.

In this case, uncaught exceptions are of the form $\bar{F}[\textit{raise}\, V]$.

Exceptions Interesting syntactic variation

Benton and Kennedy [2001] have argued for merging let and try constructs into a unique form *let* $x = M_1$ *with* M_2 *in* M_3.

The expression M_1 is evaluated first and

- if it returns a value it is substituted for x in M_3, as if we had evaluated *let* $x = M_1$ *in* M_3;
- otherwise, *i.e.*, if it raises an exception *raise* V, then the exception is handled by M_2, as if we had evaluated *try* M_1 *with* M_2.

This combined form captures a common programming pattern:

```
let rec read_config_in_path filename (dir :: dirs) →
  let fd = open_in (Filename.concat dir filename)
  with Sys_error _ → read_config filename dirs in
  read_config_from_fd fd
```

Workarounds are inelegant and inefficient. This form is also better suited for program transformations (see Benton and Kennedy [2001]).

Exceptions — Interesting syntactic variation

Encoding the new form *let* $x = M_1$ *with* M_2 *in* M_3 with "let" and "try" is not easy:

In particular, it is not equivalent to: *try let* $x = M_1$ *in* M_3 *with* M_2.

The continuation M_3 could raise an exception that would then be handled by M_2, which is not intended.

There are several encodings:

- Use a sum type to know whether M_1 raised an exception:
  case (*try Val* M_1 *with* $\lambda y.$ *Exc* y) *of* (*Val* : $\lambda x. M_3$ $[]$ *Exc* : M_2)
- Freeze the continuation M_3 while handling the exception:
  (*try let* $x = M_1$ *in* $\lambda(). M_3$ *with* $\lambda y. \lambda(). M_2\ y$) ()

Unfortunately, they are both hardly readable—and inefficient.

Exceptions — Interesting syntactic variation

A similar construct has been added in OCaml version 4.02, allowing exceptions combined with pattern matching.

The previous example can now be written in OCaml as:

```
let rec read_config_in_path filename path =
  match path with  [] → [] | dir :: dirs →
     match open_in (Filename.concat dir filename) with
     | fd → read_config_from_fd fd
     | exception Sys_error _ → read_config_in_path filename dirs
```

Exceptions Termination

Do all well-typed programs terminate in the presence of exceptions?

No, because exceptions hide the type of values that they communicate to the handler, which can be used to emulate recursive types.

Encode values of type τ_0 as lazy values of type $\textit{unit} \rightarrow \tau_0$, say τ
Let encode be `fun x () -> x` and decode be `fun x -> x ()`.
Let dummy be some value of type τ_0.
Let type *exn* be $\tau \rightarrow \tau$, say σ.

Define the two coercion functions between types σ and τ:

$$\begin{aligned} \textit{fold} : \sigma \rightarrow \tau &\triangleq \lambda f{:}\sigma.\,\lambda().\ \textit{let}\ \_ = \textit{raise}\ f\ \textit{in}\ \mathrm{dummy} \\ \textit{unfold} : \tau \rightarrow \sigma &\triangleq \lambda f{:}\tau.\ \textit{try}\ \textit{let}\ \_ = f\,()\ \textit{in}\ \lambda x{:}\tau.\,x\ \textit{with}\ \lambda y{:}\tau \rightarrow \tau.\,y \end{aligned}$$

We may then define $\omega \triangleq \lambda x.\,(\textit{unfold}\ x)\ x$ so that $\omega\ (\textit{fold}\ \omega)$ loops.

Or a call-by-value fixpoint of type $(\sigma \rightarrow \sigma) \rightarrow \sigma$ that allows recursive definition of functions of type $\tau \rightarrow \tau$ (encoding type $\tau_0 \rightarrow \tau_0$).

Exercise

Program factorial with the previous encoding without using recursion (nor recursive types, nor references)

Exercise Semantics of *let* $\cdot = \cdot$ *with* $\cdot$ *in*

Describes the dynamic semantics of the *let* $x = M_1$ *with* M_2 *in* M_3.

Solution

We need a new evaluation context:

$$E ::= \ldots \mid \textit{let}\ x = E\ \textit{with}\ M_2\ \textit{in}\ M_3$$

and the following reduction rules:

Raise
$$F[\textit{raise}\ V] \longrightarrow \textit{raise}\ V$$

Handle-Val
$$\textit{let}\ x = V\ \textit{with}\ M_2\ \textit{in}\ M_3 \longrightarrow [x \mapsto V]M_3$$

Handle-Raise
$$\textit{let}\ x = \textit{raise}\ V\ \textit{with}\ M_2\ \textit{in}\ M_3 \longrightarrow M_2\ V$$

Exercise — Try finalize

A finalizer is some code that should always be run, whether the evaluation ends normally or an exception is being raised.

Write a function try_finalize that takes four arguments f, x, g, and y and returns the application $f\ x$ with finalizing code $g\ y$. *i.e.* $g\ y$ should be called before returning the result of the application of f to x whether it executed normally or raised an exception.

(You may try first without using binding mixed with exceptions, then using it, and compare.)

Exercise (Solution to) Try finalize

Without *let* · = · *with* · *in* :

```
let finalize f x g y =
  let result = try f x with exn → g y; raise exn in g y; result
```

An alternative version that does not duplicate the finalizing code and could be inlined, but allocates an intermediate result, is:

```
type 'a result = Val of 'a | Exc of exn
let finalize f x g y =
  let result = try Val (f x) with exn → Exc exn in
  g y; match result with Val x → x | Exc exn → raise exn
```

More concisely:

```
let finalize f x g y =
  match f x with
  | result → g y; result
  | exception exn → g y; raise exn
```

Generalizing exceptions — Effect handlers

Exceptions allow to abort the current computation to the dynamically enclosing handler.

Effect handlers are a variant of control operators.

As exceptions, they allow to abort the current computation to the dynamically enclosing handler, but offer the handler the possibility to resume the computation where it was aborted.

They are (much) more expressive.

They also allow to model a global state, where a toplevel heap handler is setup so that allocation, read, and write can be implemented by passing control to the handler together with the current continuation, *i.e.* evaluation context, which may change the heap and then resume or throw away the continuation.

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:

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

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 ?) would reduce to:

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

(which yields 3).

How shall we solve this problem?

References

$(\textit{ref}\ 3)$ 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 $M \,/\, \mu$ 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:

$$\begin{aligned} (\lambda x{:}\tau.\, M)\, V \,/\, \mu &\longrightarrow [x \mapsto V]M \,/\, \mu \\ E[M] \,/\, \mu &\longrightarrow E[M'] \,/\, \mu' \qquad \text{if } M \,/\, \mu \longrightarrow M' \,/\, \mu' \end{aligned}$$

Three new reduction rules are added:

$$\begin{aligned} \textit{ref}\, V \,/\, \mu &\longrightarrow \ell \,/\, \mu[\ell \mapsto V] \qquad \text{if } \ell \notin \mathrm{dom}(\mu) \\ \ell := V \,/\, \mu &\longrightarrow () \,/\, \mu[\ell \mapsto V] \\ !\, \ell \,/\, \mu &\longrightarrow \mu(\ell) \,/\, \mu \end{aligned}$$

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 $M \mathrel{/} \mu$ well-typed?

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."

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

Comments:

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

References

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

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

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

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 $M \,/\, \mu \longrightarrow M' \,/\, \mu'$ and $\vdash M \,/\, \mu : \tau$, then $\vdash M' \,/\, \mu' : \tau$.

Theorem (Progress)

If $M \,/\, \mu$ 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 $M \mathrel{/} \mu \longrightarrow M' \mathrel{/} \mu'$ *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:

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

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, 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:

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

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.

Exercise

Exercise (Recommended)

Prove subject reduction and progress for simply-typed λ-calculus equipped with unit, pairs, sums, recursive functions, exceptions, and references!

Monads

Haskell adopts a different route and chooses to distinguish effectful computations [Peyton Jones and Wadler, 1993; Peyton Jones, 2009].

$$\begin{aligned}
\text{return} &: \alpha \to \mathit{IO}\,\alpha \\
\text{bind} &: \mathit{IO}\,\alpha \to (\alpha \to \mathit{IO}\,\beta) \to \mathit{IO}\,\beta \\
\text{main} &: \mathit{IO}\,() \\
\text{newIORef} &: \alpha \to \mathit{IO}\,(\mathit{IORef}\,\alpha) \\
\text{readIORef} &: \mathit{IORef}\,\alpha \to \mathit{IO}\,\alpha \\
\text{writeIORef} &: \mathit{IORef}\,\alpha \to \alpha \to \mathit{IO}\,()
\end{aligned}$$

Haskell offers many monads other than IO. In particular, the ST monad offers references whose lifetime is statically controlled.

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 *ref* τ_1 and *ref* τ_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* $\forall\alpha.\,$*ref* τ (where α appears in τ) as it could then be specialized to both types *ref* $([\alpha \mapsto \tau_1]\tau)$ and *ref* $([\alpha \mapsto \tau_2]\tau)$.

By contrast, *a location* ℓ *can have type 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 *ref* $(\forall\alpha.\,\tau)$.

A counter example

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

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

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 $\forall\alpha.\,\textit{ref}\,(\alpha \to \alpha)$;
- writes the location at type *ref* (*bool* → *bool*) and reads it back at type *ref* (*int* → *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 $\alpha \to \alpha$).

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

While we have

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

We have

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

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 $\textsc{Tabs}$ 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.

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

Value forms:

$$u ::= x \mid V \mid \Lambda\alpha.\, u \mid u\ \tau$$

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) *let rec* $m : \Sigma = \mu$ *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

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

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:

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

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

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 *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 $\vec{\alpha}, \Sigma, \Gamma'$)

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

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:

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

Indeed, $\textit{ref}\,(\lambda z{:}\alpha.\, z)$ 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:

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

This expression becomes ill-typed:

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

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

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

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

Value restriction

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.

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

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

For example, the following expression is non-exapnsive:

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

Value restriction

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

For instance, $\textit{let}\ x : \forall\alpha.\, \textit{list}\ \alpha = \Lambda\alpha.\,(M_1\ M_2)\ \textit{in}\ M$ 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

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

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

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 $(\lambda x{:}\alpha.\, x)$ *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.)

▷ Nick Benton and Andrew Kennedy. Exceptional syntax journal of functional programming. *J. Funct. Program.*, 11(4):395–410, 2001.

▷ 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.

▷ Xavier Leroy and François Pessaux. Type-based analysis of uncaught exceptions. *ACM Trans. Program. Lang. Syst.*, 22(2):340–377, 2000. ISSN 0164-0925. doi: http://doi.acm.org/10.1145/349214.349230.

Bibliography II

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

▷ 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.

Bibliography III

- Jean-Pierre Talpin and Pierre Jouvelot. The type and effect discipline. *Information and Computation*, 11(2):245–296, 1994.
- 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.