MPRI, Typage

Didier Rémy
(With course material from François Pottier)

October 23, 2012

INRIA

Plan of the course

Introduction

Simply-typed λ-calculus

Introduction

Contents

Choosing the meta language of this course...

Choosing the meta language of this course...

English or French?

Choosing the meta language of this course...

English or French?

(Questions stay in French)

Online material

Visit the page

http://gallium.inria.fr/~remy/mpri/

All course material:

- Course notes (will be updated as we progress)
- Calendar of lessons and exams
- Information on the programming task
- All useful information and pointers

Online material

Written notes *v.s.* copies of the slides.

Both are available on line.

However, you should rather read the course notes that the slides.

- They contain more details.
- Proofs:
  - they are often more sketchy on the slides, as a proof explain on the board.
  - they are detailed in the course notes—as you should write them on paper (and at the exam).
- They contain more exercises and solutions to exercises in the course, I only point to a few)

Questions!

Questions are welcome!

Please, ask questions...

- during the lesson
- at the end of the lesson
- by email

Didier.Remy@inria.fr

Please, don't wait the end of the course to raise problems (if any).

You are there to learn and I am there to help you!

Programming task

Reminder:

- The task will be given by the end of November.
- The solution is due by the end of the course.
- It counts for 1/3 in the final grade.
- It is often fun!
- It focuses on one particular topic of the course and usually help understand it in detail.

Questions?

What is functional programming?

The term *"functional programming"* means various things:

- it views functions as ordinary data—which, in particular can be passed as argument to other functions and stored in data structures.
- it loosely or strongly discourages the use of modifiable data, in favor of effect-free transformations of data.

  (In contrast with mainstream object-oriented programming languages)
- encourages abstraction of repetitive patterns as functions that can be called multiple times so as to avoid code duplication.

What are functional programming languages?

They are usually:

- *typed* (Scheme and Erlang are exceptions),
  with close connections to logic.

  In this course, we focus on typed languages and types play a central role.

- given a precise *formal semantics* derived from that of the λ-calculus.

  Some are *strict* (ML) and some are *lazy* (Haskell) [Hughes, 1989].

  This difference has a large impact on the language design and on the programming style, but has usually little impact on typing.

- *sequential*: their model of evaluation is not concurrent, even if core languages may then be extended with primitives to support concurrency.

Contents

What are types?

- Types are:

  “a concise, formal description of the behavior of a program fragment.”

- For instance:

int	An integer
$int \rightarrow bool$	A function that maps an integer to a Boolean
$(int \rightarrow bool) \rightarrow (list\ int \rightarrow list\ int)$	A function that maps an integer predicate to an integer list transformer

- Types must be *sound*.
  That is, programs must behave as prescribed by their types.
- Hence, types must be *checked* and ill-typed programs must be rejected.

What are they useful for?

- Types serve as *machine-checked* documentation.
- Types provide a *safety* guarantee.

  > *"Well-typed expressions do not go wrong." [Milner, 1978]*

  (Advanced type systems can also guarantee various forms of security, resource usage, complexity, . . .)
- Types can be used to drive *compiler optimizations.*
- Types encourage *separate compilation*, *modularity*, and *abstraction*.

  > *"Type structure is a syntactic discipline for enforcing levels of abstraction." [Reynolds, 1983]*

  Type-checking is compositional. Types can be abstract.
  Even seemingly non-abstract types offer a degree of abstraction
  (e.g., a function type does not tell how a function is represented)

Type-preserving compilation

Types make sense in *low-level* programming languages as well—even *assembly languages* can be statically typed! [Morrisett et al., 1999]

In a *type-preserving* compiler, every intermediate language is typed, and every compilation phase maps typed programs to typed programs.

Preserving types provides insight into a transformation, helps *debug* it, and paves the way to a *semantics preservation* proof [Chlipala, 2007].

Interestingly enough, lower-level programming languages often require richer type systems than their high-level counterparts.

Typed or untyped?

Reynolds [1985] nicely sums up a long and rather acrimonious debate:

> *"One side claims that untyped languages preclude compile-time error checking and are succinct to the point of unintelligibility, while the other side claims that typed languages preclude a variety of powerful programming techniques and are verbose to the point of unintelligibility."*

The issues are *safety*, *expressiveness*, and *type inference*.

A sound type system with decidable type-checking (and possibly decidable type inference) must be *conservative*.

Typed, Sir! with better types.

In fact, Reynolds settles the debate:

> *"From the theorist's point of view, both sides are right, and their arguments are the motivation for seeking type systems that are more flexible and succinct than those of existing typed languages."*

Today, the question is more whether to stay with rather simple polymorphic types (*e.g.* as in ML or System F) or use more sophiscated types (*e.g.* dependent types, logical assertions, afine types, capabililties and ownership, *etc.*), or even towards full program proofs!

Explicit *v.s.* implicit types?

Annotating programs with types can lead to redundancy.
Types can even become extremely cumbersome when they have to be explicitly and repeatedly provided.

In some pathological cases, type information may be grow in square of the size of the underlying untyped expression.

This creates a need for a certain degree of *type reconstruction* (also called type inference), where the source program may contain some but not all type information.

In principle, types could be entirely left implicit, even if the language is typed. A well-typed program is then one that is the type erasure of a (well-typed) explicitly-typed program.

Full type reconstruction is undecidable for expressive type systems.

Some type annotations are required or type reconstruction is incomplete.

Outline of the course

This course is organized in 5 topics, spread over 7 lectures.

1. Simple types:
   - Type soundness
   - Unit, Pairs, Sums, Recursion
   - Normalization
   - Exceptions, References
2. Polymorphism
   - System F, ML
   - Type soundness
   - Polymorphism and references.
3. Type reconstruction
   - Simple types. ML.
   - System F
4. Extistential types
5. Overloading and type classes

Simply-typed lambda-calculus

Contents

Why λ-calculus?

In this course, the underlying programming language is the λ-calculus.

The λ-calculus supports *natural* encodings of many programming languages [Landin, 1965], and as such provides a suitable setting for studying type systems.

Following Church's thesis, any Turing-complete language can be used to encode any programming language. However, these encodings might not be natural or simple enough to help us in understanding their typing discipline.

Using λ-calculus, most of our results can also be applied to other languages (Java, assembly language, *etc.*).

Syntax

Types are:

$$\tau ::= \alpha \mid \tau \to \tau \mid \dots$$

Terms are:

$$M ::= x \mid \lambda x\!:\!\tau.\, M \mid M\ M \mid \dots$$

The dots are place holders for future extensions of the language.

Binders

$\lambda x{:}\tau.\, M$ *binds* variable x in M.

We write $\mathrm{ftv}(M)$ for the set of free variables of M:

$$\begin{aligned} \mathrm{ftv}(x) &\triangleq \{x\} \\ \mathrm{ftv}(\lambda x{:}\tau.\, M) &\triangleq \mathrm{ftv}(M) \setminus \{x\} \\ \mathrm{ftv}(M_1\ M_2) &\triangleq \mathrm{ftv}(M_1) \cup \mathrm{ftv}(M_2) \end{aligned}$$

We write $x \# M$ for $x \notin \mathrm{ftv}(M)$.

Terms are considered equal up to renaming of bound variables:

- $\lambda x_1{:}\tau_1.\, \lambda x_2{:}\tau_2.\, x_1\ x_2$ and $\lambda y{:}\tau_1.\, \lambda x{:}\tau_2.\, y\ x$ are really the same term!
- $\lambda x{:}\tau.\, \lambda x{:}\tau.\, M$ is equal to $\lambda y{:}\tau.\, \lambda x{:}\tau.\, M$ when $y \notin \mathrm{ftv}(M)$.

Substitution:

$[x \mapsto N]M$ is the capture avoiding substitution of N for x in M.

Concrete *v.s.* abstract syntax

For our metatheoretical study, we are interested in the abstract syntax of expressions rather than their concrete syntax. Hence, we like to think of expressions as their abstract syntax trees.

Still, we need to write expressions on paper, hence we need some concrete syntax for terms.

The compromise is to have some concrete syntax that is in one-to-one correspondence with the abstract syntax.

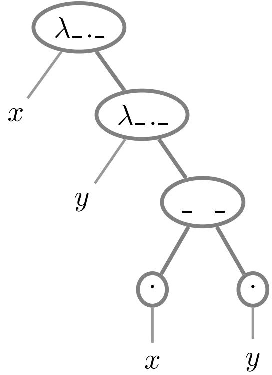
We may introduce syntactic sugar as short hand that should then be understood by its expansion into some primitive form.

Concrete *v.s.* abstract syntax

An expression in concrete notation

$$\lambda x.\, \lambda y.\, x\ y$$

must be understood as its abstract syntax tree:



Dynamic semantics

We use a *small-step operational* semantics.

We choose a *call-by-value* variant. When explaining *references*, exceptions, or other forms of side effects, this choice matters.

Otherwise, most of the type-theoretic machinery applies to call-by-name or call-by-need just as well.

Dynamic semantics

In the pure λ-calculus, the *values* are the functions:

$$V ::= \lambda x{:}\tau.\, M \mid \dots$$

The *reduction relation* $M_1 \longrightarrow M_2$ is inductively defined:

$$\beta_v \quad (\lambda x{:}\tau.\, M)\; V \longrightarrow [x \mapsto V]M \qquad\qquad \textsc{Context}\ \frac{M \longrightarrow M'}{E[M] \longrightarrow E[M']}$$

Evaluation contexts are defined as follows:

$$E ::= [\,]\; M \mid V\; [\,] \mid \dots$$

We only need evaluation contexts of depth one, using repeated applications of Rule Context.

An evaluation context of arbitrary depth can be defined as:

$$\bar{E} ::= [\,] \mid E[\bar{E}]$$

Static semantics

Technically, the type system is a 3-place predicate, whose instances are called *typing judgments*, written:

$$\Gamma \vdash M : \tau$$

where Γ is a typing context.

Typing context

A *typing context* (also called a *type environment*) Γ binds program variables to types.

We write $\varnothing$ for the empty context and $\Gamma, x : \tau$ for the extension of Γ with $x \mapsto \tau$.

To avoid confusion, we require $x \notin \mathrm{dom}(\Gamma)$ when we write $\Gamma, x : \tau$.

Bound variables in source programs can always be suitably renamed to avoid name clashes.

A typing context can then be thought of as a finite function from program variables to their types.

We write $\mathrm{dom}(\Gamma)$ for the set of variables bound by Γ and $x : \tau \in \Gamma$ to mean $x \in \mathrm{dom}(\Gamma)$ and $\tau = \Gamma(x)$.

Static semantics

Typing judgments are defined inductively by the following set of *inferences rules*:

$$\textsc{Var}\quad \Gamma \vdash x : \Gamma(x) \qquad\qquad \textsc{Abs}\quad \frac{\Gamma, x : \tau_1 \vdash M : \tau_2}{\Gamma \vdash \lambda x{:}\tau_1 .\, M : \tau_1 \rightarrow \tau_2}$$

$$\textsc{App}\quad \frac{\Gamma \vdash M_1 : \tau_1 \rightarrow \tau_2 \qquad \Gamma \vdash M_2 : \tau_1}{\Gamma \vdash M_1\, M_2 : \tau_2}$$

Notice that the specification is extremely simple.

In the simply-typed λ-calculus, the definition is *syntax-directed*. This is not true of all type systems.

Example

The following is a valid *typing derivation*:

$$
\dfrac{
  \dfrac{
    \dfrac{
      \dfrac{}{\Gamma \vdash f : \tau_1 \rightarrow \tau_2}\,\textsc{Var}
      \qquad
      \dfrac{}{\Gamma \vdash x : \tau_1}\,\textsc{Var}
    }{\Gamma \vdash f\ x : \tau_2}\,\textsc{App}
    \qquad
    \dfrac{
      \dfrac{}{\Gamma \vdash f : \tau_1 \rightarrow \tau_2}\,\textsc{Var}
      \qquad
      \dfrac{}{\Gamma \vdash y : \tau_1}\,\textsc{Var}
    }{\Gamma \vdash f\ y : \tau_2}\,\textsc{App}
  }{f : \tau_1 \rightarrow \tau_2, x : \tau_1, y : \tau_1 \vdash (f\ x, f\ y) : \tau_2 \times \tau_2}\,\textsc{Pair}
}{\varnothing \vdash \lambda f{:}\tau_1 \rightarrow \tau_2.\, \lambda x{:}\tau_1.\, \lambda y{:}\tau_1.\, (f\ x, f\ y) : (\tau_1 \rightarrow \tau_2) \rightarrow \tau_1 \rightarrow \tau_1 \rightarrow (\tau_2 \times \tau_2)}\,\textsc{Abs}
$$

Γ stands for $(f : \tau_1 \rightarrow \tau_2, x : \tau_1, y : \tau_1)$. Rule Pair is introduced later on.

This is in fact, the only typing derivation (in the empty environment).

This derivation is valid for any choice of τ_1 and τ_2.

Conversely, every derivation for this term must have this shape, for some τ_1 and τ_2.

Inversion of typing rules

The inversion Lemma states formally the previous informal reasoning.
It describes how the subterms of a well-typed term can be typed.

Lemma (Inversion of typing rules)

Assume $\Gamma \vdash M : \tau$.

- *If* M *is a variable* x*, then* $x \in \mathrm{dom}(\Gamma)$ *and* $\Gamma(x) = \tau$.
- *If* M *is* $M_1\ M_2$ *then* $\Gamma \vdash M_1 : \tau_2 \to \tau$ *and* $\Gamma \vdash M_2 : \tau_2$ *for some type* τ_0.
- *If* M *is* $\lambda x{:}\tau_2.\,M_1$*, then* τ *is of the form* $\tau_2 \to \tau_1$ *and* $\Gamma, x : \tau_2 \vdash M_1 : \tau_1$.

The inversion lemma is a basic property and used in many places when reasoning by induction on terms.

Although trivial in our simple setting, stating it explicitly avoids informal reasoning in proofs.

Uniqueness of typing derivations

Since typing rules are syntax-directed, the shape of the derivation tree is fully determined by the shape of the term.

In our simple setting, each term has actually a unique type.
Hence, typing derivations are unique, up to the typing context.
The proof, by induction on the structure of terms, is straightforward.

Explicitly-typed terms can thus be used to describe typing derivations (up to the typing context) in a precise and concise way, because terms of the language have a concrete syntax.

This enables reasoning by induction on terms instead of on typing derivations, which is often lighter.

Lacking this convenience, typing derivations must otherwise be described in the meta-language of mathematics.

Explicitly *v.s.* implicitly typed?

Our presentation of simply-typed λ-calculus is *explicitly typed* (we also say in *church-style*), as parameters of abstractions are annotated with their types.

Simply-typed λ-calculus can also be *implicitly typed* (we also say in *curry-style*) when parameters of abstractions are left unannotated, as in the pure λ-calculus.

Type erasure

We may translate explicitly-typed expressions into implicitly-typed ones by dropping type annotations. This is called *type erasure*.

We write $\lceil M \rceil$ for the type erasure of M, which is defined by structural induction on M:

$$
\begin{array}{rcl}
\lceil x \rceil & \triangleq & x \\
\lceil \lambda x{:}\tau.\, M \rceil & \triangleq & \lambda x.\, \lceil M \rceil \\
\lceil M_1\ M_2 \rceil & \triangleq & \lceil M_1 \rceil\, \lceil M_2 \rceil
\end{array}
$$

Type reconstruction

Conversely, can we convert implicitly-typed expressions back into explicitly-typed ones, that is, can we reconstruct the missing type information?

This is equivalent to finding a typing derivation for implicitly-typed terms. It is called *type reconstruction* (or *type inference*).
(See the course on type reconstruction.)

Untyped semantics

Observe that although the reduction carries types at runtime, types do not actually contribute to the reduction.

Intuitively, the semantics of terms is the same as that of their type erasure. We say that the semantics is *untyped* or *type-erasing*.

But how can we say that the semantics of typed and untyped terms coincide when these terms do not live in the same world?

?

Untyped semantics

Observe that although the reduction carries types at runtime, types do not actually contribute to the reduction.

Intuitively, the semantics of terms is the same as that of their type erasure. We say that the semantics is *untyped* or *type-erasing*.

But how can we say that the semantics of typed and untyped terms coincide when these terms do not live in the same world?

By showing that the reductions in the two languages can be put into close correspondence.

Untyped semantics

On the one hand, type erasure preserves reduction.

$$M_1 \xrightarrow{\beta} M_2$$

Lemma (Forward simulation)

If $M_1 \longrightarrow M_2$ *then* $\lceil M_1 \rceil \longrightarrow \lceil M_2 \rceil$.

Untyped semantics

On the one hand, type erasure preserves reduction.

$$\begin{tikzcd} M_1 \arrow[r, "\beta"] \arrow[d, dashed, "\lceil\rceil"'] & M_2 \arrow[d, dashed, "\lceil\rceil"] \\ a_1 \arrow[r, dashed, "\beta"'] & a_2 \end{tikzcd}$$

Lemma (Forward simulation)

If $M_1 \longrightarrow M_2$ *then* $\lceil M_1 \rceil \longrightarrow \lceil M_2 \rceil$.

Untyped semantics

On the one hand, type erasure preserves reduction.

$$\begin{tikzcd} M_1 \arrow[r, "\beta"] \arrow[d, dashed, "\lceil\,\rceil"'] & M_2 \arrow[d, dashed, "\lceil\,\rceil"] \\ a_1 \arrow[r, dashed, "\beta"'] & a_2 \end{tikzcd}$$

Lemma (Forward simulation)

If $M_1 \longrightarrow M_2$ *then* $\lceil M_1 \rceil \longrightarrow \lceil M_2 \rceil$.

Conversely, a reduction steps after type erasure could also have been performed on the term before type erasure.

$$\begin{tikzcd} M_1 \arrow[d, "\lceil\,\rceil"'] & \\ a_1 \arrow[r, "\beta"'] & a_2 \end{tikzcd}$$

Lemma (Backward simulation)

If $\lceil M \rceil \longrightarrow a$ *then there exists* M' *such that* $M \longrightarrow M'$ *and* $\lceil M' \rceil = a$.

Untyped semantics

On the one hand, type erasure preserves reduction.

$$\begin{tikzcd} M_1 \arrow[r, "\beta"] \arrow[d, dashed, "\lceil\rceil"'] & M_2 \arrow[d, dashed, "\lceil\rceil"] \\ a_1 \arrow[r, dashed, "\beta"'] & a_2 \end{tikzcd}$$

Lemma (Forward simulation)

If $M_1 \longrightarrow M_2$ *then* $\lceil M_1 \rceil \longrightarrow \lceil M_2 \rceil$.

Conversely, a reduction steps after type erasure could also have been performed on the term before type erasure.

$$\begin{tikzcd} M_1 \arrow[r, dashed, "\beta"] \arrow[d, "\lceil\rceil"'] & M_2 \arrow[d, dashed, "\lceil\rceil"] \\ a_1 \arrow[r, "\beta"'] & a_2 \end{tikzcd}$$

Lemma (Backward simulation)

If $\lceil M \rceil \longrightarrow a$ *then there exists* M' *such that* $M \longrightarrow M'$ *and* $\lceil M' \rceil = a$.

Untyped semantics

On the one hand, type erasure preserves reduction.

$$\begin{tikzcd} M_1 \arrow[r, "\beta"] \arrow[d, dashed, "\lceil\,\rceil"'] & M_2 \arrow[d, dashed, "\lceil\,\rceil"] \\ a_1 \arrow[r, dashed, "\beta"'] & a_2 \end{tikzcd}$$

Lemma (Forward simulation)

If $M_1 \longrightarrow M_2$ *then* $\lceil M_1 \rceil \longrightarrow \lceil M_2 \rceil$.

Conversely, a reduction steps after type erasure could also have been performed on the term before type erasure.

$$\begin{tikzcd} M_1 \arrow[r, dashed, "\beta"] \arrow[d, "\lceil\,\rceil"'] & M_2 \arrow[d, dashed, "\lceil\,\rceil"] \\ a_1 \arrow[r, "\beta"'] & a_2 \end{tikzcd}$$

Lemma (Backward simulation)

If $\lceil M \rceil \longrightarrow a$ *then there exists* M' *such that* $M \longrightarrow M'$ *and* $\lceil M' \rceil = a$.

What we have established is a *bisimulation* between explicitly-typed terms and implicitly-typed ones.

In general, there may be reduction steps on source terms that involved only types and have no counter-part (and disappear) on compiled terms.

Untyped semantics

It is an important property for a language to have an untyped semantics.

It then has an implicitly-typed presentation.

The metatheoretical study is often easier with explicitly-typed terms.

Properties of the implicitly-typed presentation can often be indirectly proved via an explicitly-typed presentation of the language.

This is the path we choose in this course.

Contents

Stating type soundness

What is a formal statement of Milner's slogan?

> *"Well-typed expressions do not go wrong"*

By definition, a closed term M is *well-typed* if it admits some type τ in the empty environment.

By definition, a closed, irreducible term is either a value or *stuck*.

A closed term must *converge* to a value, *diverge*, or *go wrong* by reducing to a stuck term.

Stating type soundness

Milner's slogan now has formal meaning:

Theorem (Type Soundness)

Well-typed expressions do not go wrong.

Proof.

By Subject Reduction and Progress. □

Establishing type soundness

We use the syntactic proof method of Wright and Felleisen [1994]. Type soundness follows from two properties:

Theorem (Subject reduction)

Reduction preserves types: if $M_1 \longrightarrow M_2$ then for any type τ such that $\varnothing \vdash M_1 : \tau$, we also have $\varnothing \vdash M_2 : \tau$.

Theorem (Progress)

A (closed) well-typed term is either reducible or a value:
if $\varnothing \vdash M : \tau$ then there exists M' such that $M \longrightarrow M'$, or M is a value.

Equivalently, we may say: *a closed, well-typed, irreducible term is a value.*

Establishing subject reduction

Subject reduction is proved by *induction* over the hypothesis $M_1 \longrightarrow M_2$. Thus, there is one case per reduction rule.

In the pure simply-typed λ-calculus, there are just two such rules: β-reduction and reduction under an evaluation context.

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

$$\text{Context} \quad \frac{M \longrightarrow M'}{E[M] \longrightarrow E[M']}$$

Establishing subject reduction — Case β

In the β-reduction case, the first hypothesis is

$$(\lambda x\!:\!\tau.\, M)\ V \longrightarrow [x \mapsto V]M \tag{1}$$

the second hypothesis is

$$\varnothing \vdash (\lambda x\!:\!\tau.\, M)\ V : \tau_0 \tag{2}$$

and the goal is

$$\varnothing \vdash [x \mapsto V]M : \tau_0 \tag{3}$$

How do we proceed?

Establishing subject reduction — Case β

Hyp: $(\lambda x\!:\!\tau.\, M)\ V \longrightarrow [x \mapsto V]M$ $\mathbf{(1)}$ and $\varnothing \vdash (\lambda x\!:\!\tau.\, M)\ V : \tau_0$ $\mathbf{(2)}$.

Goal: $\varnothing \vdash [x \mapsto V]M : \tau_0$ $\mathbf{(3)}$

We *decompose* the hypothesis (2).

By inversion of the typing rules, the derivation of (2) must be:

$$\dfrac{\dfrac{x : \tau \vdash M : \tau_0\ \mathbf{(4)}}{\varnothing \vdash (\lambda x\!:\!\tau.\, M) : \tau \to \tau_0}\ \textsc{Abs} \qquad \varnothing \vdash V : \tau\ \mathbf{(5)}}{\varnothing \vdash (\lambda x\!:\!\tau.\, M)\ V : \tau_0\ (2)}\ \textsc{App}$$

Where next?

Establishing subject reduction — Case β

Hyp: $(\lambda x\!:\!\tau.\, M)\ V \longrightarrow [x \mapsto V]M$ $\mathbf{(1)}$ and $\varnothing \vdash (\lambda x\!:\!\tau.\, M)\ V : \tau_0$ $\mathbf{(2)}$.

Goal: $\varnothing \vdash [x \mapsto V]M : \tau_0$ $\mathbf{(3)}$

We *decompose* the hypothesis (2).

By inversion of the typing rules, the derivation of (2) must be:

$$\textsc{App}\ \dfrac{\textsc{Abs}\ \dfrac{x : \tau \vdash M : \tau_0\ \mathbf{(4)}}{\varnothing \vdash (\lambda x\!:\!\tau.\, M) : \tau \to \tau_0} \qquad \varnothing \vdash V : \tau\ \mathbf{(5)}}{\varnothing \vdash (\lambda x\!:\!\tau.\, M)\ V : \tau_0\ (2)}$$

Where next?

We expect (3) to follow from (4) and (5)...

Establishing subject reduction (case β) Value substitution

Hence, to conclude, we only need the following lemma:

Lemma (Value substitution)

If $x : \tau \vdash M : \tau_0$ *and* $\varnothing \vdash V : \tau$*, then* $\varnothing \vdash [x \mapsto V]M : \tau_0$*.*

In plain words, replacing a formal parameter with a type-compatible actual argument preserves types.

How do we prove this lemma?

?

Establishing subject reduction (case β) Value substitution

Hence, to conclude, we only need the following lemma:

Lemma (Value substitution)

If $x : \tau \vdash M : \tau_0$ *and* $\varnothing \vdash V : \tau$, *then* $\varnothing \vdash [x \mapsto V]M : \tau_0$.

In plain words, replacing a formal parameter with a type-compatible actual argument preserves types.

How do we prove this lemma?

—By induction on the typing derivation for M...

However, one case does not go through. Which one?

Establishing subject reduction (case β) Value substitution

Hence, to conclude, we only need the following lemma:

Lemma (Value substitution)

If $x : \tau \vdash M : \tau_0$ *and* $\varnothing \vdash V : \tau$*, then* $\varnothing \vdash [x \mapsto V]M : \tau_0$*.*

In plain words, replacing a formal parameter with a type-compatible actual argument preserves types.

How do we prove this lemma?

—By induction on the typing derivation for M...

However, one case does not go through. Which one?

Establishing subject reduction (case β) Value substitution

The lemma must be suitably generalized so that *induction* can be applied in the case of abstraction:

Lemma (Value substitution, strengthened)

If $x : \tau, \Gamma \vdash M : \tau_0$ *and* $\varnothing \vdash V : \tau$, *then* $\Gamma \vdash [x \mapsto V]M : \tau_0$.

The proof is now straightforward—at variables, it uses another lemma:

Lemma (Weakening)

If $\varnothing \vdash V : \tau_1$ *then* $\Gamma \vdash V : \tau_1$.

This closes the case of the β-reduction rule.

Establishing subject reduction (case β) Weakening

The weakening lemma need only add one binding at a time, the general case follows as a corollary. *However, ...*

?

Establishing subject reduction (case β) Weakening

The weakening lemma need only add one binding at a time, the general case follows as a corollary. *However, it must be strengthened...*

Lemma (Weakening, strengthened)

If $\Gamma \vdash M : \tau$ *and* $y \notin \mathrm{dom}(\Gamma)$, *then* $\Gamma, y : \tau' \vdash M : \tau$.

The proof is by induction and cases on M applying the inversion lemma:

Case M is x: Then x must be bound to τ in Γ. Hence, it is also bound to τ in $(\Gamma, y : \tau')$. We conclude by rule Var.

Case M is $\lambda x{:}\tau_2.\, M_1$: *W.l.o.g*, we may choose $x \notin \mathrm{dom}(\Gamma)$ and $x \neq y$. We have $\Gamma, x : \tau_2 \vdash M_1 : \tau_1$ with $\tau_2 \to \tau_1$ equal to τ. By induction hypothesis, we have $\Gamma, x : \tau_2, y : \tau' \vdash M_1 : \tau_1$. Thanks to a *permutation* lemma, we have $\Gamma, y : \tau', x : \tau_2 \vdash M_1 : \tau_1$ and we conclude by Rule Abs.

Case M is $M_1\ M_2$: easy.

Establishing subject reduction (case β) Permutation

Lemma (Permutation lemma)

If $\Gamma \vdash M : \tau$ *and* Γ' *is a permutation of* Γ*, then* $\Gamma' \vdash M : \tau$*.*

?

Establishing subject reduction (case β) Permutation

Lemma (Permutation lemma)

If $\Gamma \vdash M : \tau$ *and* Γ' *is a permutation of* Γ*, then* $\Gamma' \vdash M : \tau$*.*

The result is obvious since a permutation of Γ does not change its interpretation as a finite function, which is all what is needed in the typing rules so far (this will no more be the case when we later extend Γ with type variable declarations).

Formally, the proof is by induction on M.

Establishing subject reduction — Case Context

In the context case, the first hypothesis is

$$M \longrightarrow M' \qquad (\mathbf{1})$$

where, by induction hypothesis, this reduction preserves types $(\mathbf{2})$.

The second hypothesis is

$$\varnothing \vdash E[M] : \tau \qquad (\mathbf{3})$$

where E is an *evaluation context* (reminder $E ::= [\,]\ M \mid V\ [\,] \mid \ldots$).

The goal is

$$\varnothing \vdash E[M'] : \tau \qquad (\mathbf{4})$$

How do we proceed?

Establishing subject reduction

Case Context

Type-checking is *compositional*: only the type of the sub-expression *in the hole* matters, not its exact form. The context case immediately follows from compositionality, which closes the proof of subject reduction.

Lemma (Compositionality)

If $\varnothing \vdash E[M] : \tau$*, then, there exists* τ' *such that:*

- $\varnothing \vdash M : \tau'$,
- *for every* M', $\quad \varnothing \vdash M' : \tau'$ *implies* $\varnothing \vdash E[M'] : \tau$.

The proof is straightforward, by cases over E.

Informally, τ' is the type of the hole in the pseudo judgment $\varnothing \vdash E[\tau'] : \tau$. Evaluation contexts do not bind variables, so the hole is typechecked in an empty environment as well.

Establishing progress

Progress ("A closed, well-typed, irreducible term M is a value") is proved by *structural induction* over the term M. Thus, there is one case per construct in the syntax of terms.

In the pure λ-calculus, there are just three cases:

- variable;
- λ-abstraction;
- application.

Two of these are immediate...

Establishing progress

- The case of variables is void,

Establishing progress

- The case of variables is void,
  because *a variable is never closed*
  (it does not admit a type in the empty environment).

Establishing progress

- The case of variables is void,
  because *a variable is never closed*
  (it does not admit a type in the empty environment).
- The case of λ-abstractions is immediate,

Establishing progress

- The case of variables is void,
  because *a variable is never closed*
  (it does not admit a type in the empty environment).
- The case of λ-abstractions is immediate,
  because *a λ-abstraction is a value*.

Establishing progress

- The case of variables is void,
  because *a variable is never closed*
  (it does not admit a type in the empty environment).
- The case of λ-abstractions is immediate,
  because *a λ-abstraction is a value*.
- The only remaining case is that of applications.

Establishing progress

Let us show that a closed, well-typed, term $M_1\ M_2.$ is reducible.

By inversion of typing rules, there exist types τ_1 and τ_2 such that $\varnothing \vdash M_1 : \tau_2 \to \tau_1$ $\mathbf{(1)}$ and $\varnothing \vdash M_2 : \tau_2$ $\mathbf{(2)}$.

?

Establishing progress

Let us show that a closed, well-typed, term $M_1\ M_2.$ is reducible.

By inversion of typing rules, there exist types τ_1 and τ_2 such that $\varnothing \vdash M_1 : \tau_2 \rightarrow \tau_1$ $\mathbf{(1)}$ and $\varnothing \vdash M_2 : \tau_2$ $\mathbf{(2)}$.

By the I.H. applied to (1), M_1 is eiher reducible or a value V_1.

?

Establishing progress

Let us show that a closed, well-typed, term $M_1\ M_2$. is reducible.

By inversion of typing rules, there exist types τ_1 and τ_2 such that $\varnothing \vdash M_1 : \tau_2 \to \tau_1$ $\mathbf{(1)}$ and $\varnothing \vdash M_2 : \tau_2$ $\mathbf{(2)}$.

By the I.H. applied to (1), M_1 is eiher reducible or a value V_1.

If M_1 is reducible, so is M since $[]\ M_2$ is an evaluation context and we are done.

?

Establishing progress

Let us show that a closed, well-typed, term $M_1\ M_2.$ is reducible.

By inversion of typing rules, there exist types τ_1 and τ_2 such that $\varnothing \vdash M_1 : \tau_2 \to \tau_1$ $\mathbf{(1)}$ and $\varnothing \vdash M_2 : \tau_2$ $\mathbf{(2)}$.

By the I.H. applied to (1), M_1 is eiher reducible or a value V_1.

If M_1 is reducible, so is M since $[]\ M_2$ is an evaluation context and we are done.

By the I.H. applied to (2), M_2 is either reducible or a value V_2.

?

Establishing progress

Let us show that a closed, well-typed, term $M_1\ M_2$. is reducible.

By inversion of typing rules, there exist types τ_1 and τ_2 such that $\varnothing \vdash M_1 : \tau_2 \rightarrow \tau_1$ $\mathbf{(1)}$ and $\varnothing \vdash M_2 : \tau_2$ $\mathbf{(2)}$.

By the I.H. applied to (1), M_1 is eiher reducible or a value V_1.

If M_1 is reducible, so is M since $[]\ M_2$ is an evaluation context and we are done.

By the I.H. applied to (2), M_2 is either reducible or a value V_2.

If M_2 is reducible, so is M since $V_1\ []$ is an evaluation context and we are done.

?

Establishing progress

Let us show that a closed, well-typed, term $M_1\ M_2.$ is reducible.

By inversion of typing rules, there exist types τ_1 and τ_2 such that $\varnothing \vdash M_1 : \tau_2 \to \tau_1$ $\mathbf{(1)}$ and $\varnothing \vdash M_2 : \tau_2$ $\mathbf{(2)}$.

By the I.H. applied to (1), M_1 is eiher reducible or a value V_1.

If M_1 is reducible, so is M since $[]\ M_2$ is an evaluation context and we are done.

By the I.H. applied to (2), M_2 is either reducible or a value V_2.

If M_2 is reducible, so is M since $V_1\ []$ is an evaluation context and we are done.

Because V_1 is a value of type $\tau_1 \to \tau_2$, it must be a λ-abstraction (*see next slide*), so M is a β-redex, hence reducible. □

Establishing progress

Interestingly, the proof is constructive.

It corresponds to an algorithm that ...

?

Establishing progress

Interestingly, the proof is constructive.

It corresponds to an algorithm that
searches for the active redex in a well-typed term.

Classification of values

We have appealed to the following property:

Lemma (Classification)

Assume $\varnothing \vdash V : \tau$. *Then,*

- *if* τ *is an arrow type, then* V *is a* λ*-abstraction;*
- ... (e.g. *if* τ *is a product type, then* V *is product)*

Proof.

By cases over V:

- if V is a λ-abstraction, then τ must be an arrow type;
- ... (*e.g.* if V is product, then τ must be a product type)

Because different kinds of values receive types with different head constructors, this classification is injective, and can be inverted. □

Towards more complex type systems

In the pure λ-calculus, classification is trivial, because

?

Towards more complex type systems

In the pure λ-calculus, classification is trivial, because *every value is a λ-abstraction.*

Progress holds even in the absence of the well-typedness hypothesis, *i.e.* in the untyped λ-calculus, because *no term is ever stuck!*

As the programming language and its type system are extended with new features, however, type soundness is no longer trivial.

Warning!

Most type soundness proofs are shallow but large. Authors are tempted to skip the "easy" cases, but these may contain hidden traps!

Towards more complex type systems

Sometimes, the *combination* of two features is *unsound*, even though each feature, in isolation, is sound.

This is problematic, because researchers like studying each feature in isolation, and do not necessarily foresee problems with their combination.

This will be illustrated in this course by the interaction between references and polymorphism in ML.

In fact, a few such combinations have been implemented, deployed, and used for some time before they were found to be unsound!

- call/cc + polymorphism in SML/NJ [Harper and Lillibridge, 1991]
- mutable records + existential quantification in Cyclone [Grossman, 2006]

Soundness versus completeness

Becausew the λ-calculus is a Turing-complete programming language, whether a program goes wrong is an *undecidable* property.

As a result, *any sound, decidable type system must be incomplete*, that is, must reject some valid programs.

Type systems can be *compared* against one another via encodings, so it is sometimes possible to prove that one system is more expressive than another.

However, whether a type system is "sufficiently expressive in practice" can only be assessed via *empirical* means.

Contents

Normalization

In general, types also ensure termination of programs—as long as neither types nor term contain any form of recursion.

Normalization

In general, types also ensure termination of programs—as long as neither types nor term contain any form of recursion.

Even if one wishes to add recursion explicitly later on, it is an important property of the design that non-termination is originating from the constructions introduced especially for recursion and could not occur without them.

Normalization

In general, types also ensure termination of programs—as long as neither types nor term contain any form of recursion.

Even if one wishes to add recursion explicitly later on, it is an important property of the design that non-termination is originating from the constructions introduced especially for recursion and could not occur without them.

The simply-typed λ-calculus is also lifted at the level of types in richer type systems such as System F^{ω}; then, the decidability of type-equality depends on the termination of the reduction at the type level.

Normalization

In general, types also ensure termination of programs—as long as neither types nor term contain any form of recursion.

Even if one wishes to add recursion explicitly later on, it is an important property of the design that non-termination is originating from the constructions introduced especially for recursion and could not occur without them.

The simply-typed λ-calculus is also lifted at the level of types in richer type systems such as System F^ω; then, the decidability of type-equality depends on the termination of the reduction at the type level.

The proof of termination for the simply-typed λ-calculus is simple enough and interesting to be presented here.

Notice however, that our simply-typed λ-calculus is equipped with a call-by-value semantics. Proofs of termination are usually done with a strong evaluation strategy where reduction can occur in any context.

Normalization

Proving termination of reduction in fragments of the λ-calculus is often a difficult task because reduction may create new redexes or duplicate existing ones.

We follow the proof schema of Pierce [2002], which is a modern presentation in a call-by-value setting of an older proof by Hindley and Seldin [1986]. The proof method is due to [Tait, 1967]:

- build the set $\mathcal{T}_\tau$ of terminating terms of type τ and
- show that any term of type τ is in $\mathcal{T}_\tau$, by induction on terms.

This hypothesis is however too weak. The difficulty is as usual to find a strong enough induction hypothesis...

Normalization

Proving termination of reduction in fragments of the λ-calculus is often a difficult task because reduction may create new redexes or duplicate existing ones.

We follow the proof schema of Pierce [2002], which is a modern presentation in a call-by-value setting of an older proof by Hindley and Seldin [1986]. The proof method is due to [Tait, 1967]:

- build the set $\mathcal{T}_\tau$ of terminating terms of type τ and
- show that any term of type τ is in $\mathcal{T}_\tau$, by induction on terms.

This hypothesis is however too weak. The difficulty is as usual to find a strong enough induction hypothesis...

Terms of type $\tau_1 \rightarrow \tau_2$ should not only terminate but also terminate when applied to terms in $\mathcal{T}_{\tau_1}$,

Normalization

Definition

Let $\mathcal{T}_\tau$ be defined inductively on τ as follows:

- $\mathcal{T}_\alpha$ is the set of terms that terminates;
- $\mathcal{T}_{\tau_2 \to \tau_1}$ is the set of terms M_1 of type $\tau_2 \to \tau_1$ that terminates and such that $M_1\ M_2$ is in $\mathcal{T}_{\tau_1}$ for any term M_2 in $\mathcal{T}_{\tau_2}$.

The set $\mathcal{T}_\tau$ can be seen as a predicate, *i.e.* a unary relation. It is called a *logical relation* because it is defined inductively on the structure of types.

The following proofs is then schematic of the use of logical relations.

Normalization

Reduction of terms of type τ preserves membership in $\mathcal{T}_\tau$:

Lemma

If $\varnothing \vdash M : \tau$ and $M \longrightarrow M'$, then $M \in \mathcal{T}_\tau$ iff $M' \in \mathcal{T}_\tau$.

Proof.

By induction on the structure of the type τ □

Lemma

For any type τ, the reduction of any term in $\mathcal{T}_\tau$ terminates.

Immediate, by definition of $\mathcal{T}_\tau$.

Normalization

Therefore, it just remains to show that any term of type τ is in $\mathcal{T}_\tau$, *i.e.*:

Lemma

If $\varnothing \vdash M : \tau$, *then* $M \in \mathcal{T}_\tau$.

The proof is by induction on (the typing derivation of) M.

However, the case for abstraction requires some similar statement, but for open terms. We need to strengthen the Lemma.

A trick to avoid considering open terms is to require the statement to hold for all closed instances of an open term:

Lemma (strenghened)

If $(x_i : \tau_i)^{i \in I} \vdash M : \tau$, *then for any closed values* $(V_i)^{i \in I}$ *in* $(\mathcal{T}_{\tau_i})^{i \in I}$, *the term* $[(x_i \mapsto V_i)^{i \in I}]M$ *is in* $\mathcal{T}_\tau$.

Normalization

Proof. By structural induction on M.
We write Γ for $(x_i : \tau_i)^{i \in I}$ and θ for $[(x_i \mapsto V_i)^{i \in I}]$. Assume $\Gamma \vdash M : \tau$.

The only interesting case is when M is $\lambda x{:}\tau_1 .\, M_2$:

By inversion of typing, we know that $\Gamma, x : \tau_1 \vdash M_2 : \tau_2$ and $\tau_1 \rightarrow \tau_2$ is τ.

To show $\theta M \in \mathcal{T}_\tau$, we must show that it is terminating, which holds as it is a value, and that its application to any M_1 in $\mathcal{T}_{\tau_1}$ is in $\mathcal{T}_{\tau_2}$ $\mathbf{(1)}$.

Let $M_1 \in \mathcal{T}_{\tau_1}$. By definition $M_1 \longrightarrow^* V$ $\mathbf{(2)}$. We then have:

$$
\begin{array}{rcll}
(\theta M)\, M_1 & \triangleq & (\theta(\lambda x{:}\tau_1 .\, M_2))\, M_1 & \text{by definition of } M \\
& = & (\lambda x{:}\tau_1 .\, \theta M_2)\, M_1 & \text{choose } x \# \vec{x} \\
& \longrightarrow^* & (\lambda x{:}\tau_1 .\, \theta M_2)\, V & \text{by } (2) \\
& \longrightarrow & [x \mapsto V](\theta M_2) & \text{by } (\beta) \\
& = & ([x \mapsto V]\theta)(M_2) \;\in\; \mathcal{T}_{\tau_2} & \text{by induction hypothesis}
\end{array}
$$

This establishes (1) since $\mathcal{T}_{\tau_2}$ is closed by reduction.

Contents

Adding a unit

The simply-typed λ-calculus is modified as follows. Values and expressions are extended with "unit", written $()$:

$$M ::= \ldots \mid () \qquad\qquad V ::= \ldots \mid ()$$

No new reduction rule is introduced.

Types are extended with a "unit" and a new typing rule:

$$\tau ::= \ldots \mid \mathit{unit} \qquad\qquad \frac{\textsc{Unit}}{\Gamma \vdash () : \mathit{unit}}$$

Exercise

Check that type soundness is preserved by this extension.

Notice that the classification Lemma is no more degenerate.

Pairs

The simply-typed λ-calculus is modified as follows.

Values, expressions, evaluation contexts are extended:

$$\begin{array}{rcl} M & ::= & \ldots \mid (M, M) \mid \textit{proj}_i\ M \\ E & ::= & \ldots \mid ([], M) \mid (V, []) \mid \textit{proj}_i\ [] \\ V & ::= & \ldots \mid (V, V) \\ i & \in & \{1, 2\} \end{array}$$

A new reduction rule is introduced:

$$\textit{proj}_i\ (V_1, V_2) \longrightarrow V_i$$

Pairs

Types are extended:

$$\tau ::= \ldots \mid \tau \times \tau$$

Two new typing rules are introduced:

$$\textsc{Pair}\quad \frac{\Gamma \vdash M_1 : \tau_1 \qquad \Gamma \vdash M_2 : \tau_2}{\Gamma \vdash (M_1, M_2) : \tau_1 \times \tau_2} \qquad\qquad \textsc{Proj}\quad \frac{\Gamma \vdash M : \tau_1 \times \tau_2}{\Gamma \vdash \mathit{proj}_i\ M : \tau_i}$$

Exercise

Check that subject reduction is preserved by this extension.

Sums

Values, expressions, evaluation contexts are extended:

$$\begin{array}{rcl} M & ::= & \ldots \mid \textit{inj}_i\ M \mid \textit{case}\ M\ \textit{of}\ V \ [\!]\ V \\ E & ::= & \ldots \mid \textit{inj}_i\ [] \mid \textit{case}\ []\ \textit{of}\ V \ [\!]\ V \\ V & ::= & \ldots \mid \textit{inj}_i\ V \end{array}$$

A new reduction rule is introduced:

$$\textit{case}\ \textit{inj}_i\ V\ \textit{of}\ V_1 \ [\!]\ V_2 \longrightarrow V_i\ V$$

Sums

Types are extended:

$$\tau ::= \ldots \mid \tau + \tau$$

Two new typing rules are introduced:

$$\textsc{Inj}\quad \frac{\Gamma \vdash M : \tau_i}{\Gamma \vdash \mathit{inj}_i\ M : \tau_1 + \tau_2}$$

$$\textsc{Case}\quad \frac{\Gamma \vdash M : \tau_1 + \tau_2 \qquad \Gamma \vdash V_1 : \tau_1 \rightarrow \tau \qquad \Gamma \vdash V_2 : \tau_2 \rightarrow \tau}{\Gamma \vdash \mathit{case}\ M\ \mathit{of}\ V_1 \mathbin{[\!]} V_2 : \tau}$$

Sums with unique types

Notice that a property of simply-typed λ-calculus is lost: expressions do not have unique types anymore, *i.e.* the type of an expression is no longer determined by the expression.

Uniqueness of types can be recovered by using a type annotation in injections:

$$V ::= \ldots \mid \mathit{inj}_i\ V\ \mathit{as}\ \tau$$

and modifying the typing rules and reduction rules accordingly.

Exercise

Describe instead an extension with the option type.

Modularity of extensions

The three preceding extensions are very similar. Each one introduces:

- a new type constructor, to classify values of a new shape;
- new expressions, to *construct* and *destruct* values of a new shape.
- new typing rules for new forms of expressions;
- new reduction rules, to specify how values of the new shape can be destructed;
- new evaluation contexts, but just to propagate reduction under the new constructors.

Subject reduction is preserved because types of new redexes are preserved by the new reduction rules.

Progress is preserved because the type system ensures that the new destructors can only be applied to values such that at least one of the new reduction rules applies.

Modularity of extensions

These extensions are independent: they can be added to the λ-calculus alone or mixed altogether.

Indeed, no assumption about other extensions (the "$\ldots$") is ever made, except for the classification lemma which requires, informally, that "values of other shapes have types of other shapes".

This is indeed the case in the extensions we have presented: the unit has the Unit type, pairs have product types, sums have sum types.

In fact, these extensions could have been presented as several instances of a more general extension of the λ-calculus with constants, for which type soundness can be established uniformly under reasonable assumptions relating the given typing rules and reduction rules for constants [Pottier and Rémy, 2005].

(See also the treatment of *data types* in System F)

Recursive functions

The simply-typed λ-calculus is modified as follows.

Values and expressions are extended:

$$\begin{array}{rcl} M & ::= & \ldots \mid \mu f\!:\!\tau.\,\lambda x.M \\ V & ::= & \ldots \mid \mu f\!:\!\tau.\,\lambda x.M \end{array}$$

A new reduction rule is introduced:

$$(\mu f\!:\!\tau.\,\lambda x.M)\;V \longrightarrow [f \mapsto \mu f\!:\!\tau.\,\lambda x.M][x \mapsto V]M$$

Recursive functions

Types are *not* extended. We already have function types.

(Hence, types will not distinguish functions from recursive functions.)

A new typing rule is introduced:

$$\textsc{FixAbs}\quad \frac{\Gamma, f : \tau_1 \to \tau_2 \vdash \lambda x{:}\tau_1 . M : \tau_1 \to \tau_2}{\Gamma \vdash \mu f{:}\tau_1 \to \tau_2 . \lambda x . M : \tau_1 \to \tau_2}$$

In the premise, the type $\tau_1 \to \tau_2$ serves both as an assumption and a goal. This is a typical feature of recursive definitions.

A derived construct: let

The construct "*let* $x : \tau = M_1$ *in* M_2" can be viewed as syntactic sugar for the β-redex "$(\lambda x{:}\tau.\, M_2)\ M_1$".

The latter can be type-checked *only* by a derivation of the form:

$$\textsc{App}\ \dfrac{\textsc{Abs}\ \dfrac{\Gamma, x : \tau_1 \vdash M_2 : \tau_2}{\Gamma \vdash \lambda x{:}\tau_1.\, M_2 : \tau_1 \to \tau_2} \qquad \Gamma \vdash M_1 : \tau_1}{\Gamma \vdash (\lambda x{:}\tau_1.\, M_2)\ M_1 : \tau_2}$$

This means that the following *derived rule* is sound and *complete*:

$$\textsc{LetMono}\ \dfrac{\Gamma \vdash M_1 : \tau_1 \qquad \Gamma, x : \tau_1 \vdash M_2 : \tau_2}{\Gamma \vdash \textit{let}\ x : \tau_1 = M_1\ \textit{in}\ M_2 : \tau_2}$$

The construct "$M_1; M_2$" can in turn be viewed as syntactic sugar for ...

?

A derived construct: let

The construct "*let* $x : \tau = M_1$ *in* M_2" can be viewed as syntactic sugar for the β-redex "$(\lambda x{:}\tau.\, M_2)\ M_1$".

The latter can be type-checked *only* by a derivation of the form:

$$\dfrac{\textsc{Abs}\ \dfrac{\Gamma, x : \tau_1 \vdash M_2 : \tau_2}{\Gamma \vdash \lambda x{:}\tau_1.\, M_2 : \tau_1 \to \tau_2} \qquad \Gamma \vdash M_1 : \tau_1}{\Gamma \vdash (\lambda x{:}\tau_1.\, M_2)\ M_1 : \tau_2}\ \textsc{App}$$

This means that the following *derived rule* is sound and *complete*:

$$\textsc{LetMono}\quad \dfrac{\Gamma \vdash M_1 : \tau_1 \qquad \Gamma, x : \tau_1 \vdash M_2 : \tau_2}{\Gamma \vdash \textit{let}\ x : \tau_1 = M_1\ \textit{in}\ M_2 : \tau_2}$$

The construct "$M_1; M_2$" can in turn be viewed as syntactic sugar for *let* x : *unit* = M_1 *in* M_2 where $x \notin \mathrm{ftv}(M_2)$.

A derived construct: let

In the derived form *let* $x : \tau_1 = M_1$ *in* M_2 the type of M_1 must be explicitly given, although by uniqueness of types, it is entirely determined by the expression M_1 itself. Hence, it seems redundant.

Indeed, we can replace the derived form by a primitive form *let* $x = M_1$ *in* M_2 with the following primitive typing rule.

$$\textsc{LetMono}\quad \frac{\Gamma \vdash M_1 : \tau_1 \qquad \Gamma, x : \tau_1 \vdash M_2 : \tau_2}{\Gamma \vdash \textit{let}\ x = M_1\ \textit{in}\ M_2 : \tau_2}$$

This is fine, but not necessary, because removing redundant type annotations is the problem of type reconstruction and we should not bother about it in the explicitly-typed version of the language.

Minimizing the number of language constructs is at least as important as avoiding extra type annotations in an explicitly-typed language.

A derived construct: let rec

The construct "*let rec* $(f : \tau)\ x = M_1$ *in* M_2" can be viewed as syntactic sugar for "*let* $f = \mu f{:}\tau.\, \lambda x.M_1$ *in* M_2".

The latter can be type-checked *only* by a derivation of the form:

$$\textsc{LetMono}\ \dfrac{\textsc{FixAbs}\ \dfrac{\Gamma, f : \tau \to \tau_1; x : \tau \vdash M_1 : \tau_1}{\Gamma \vdash \mu f{:}\tau \to \tau_1.\, \lambda x.M_1 : \tau \to \tau_1} \qquad \Gamma, f : \tau \to \tau_1 \vdash M_2 : \tau_2}{\Gamma \vdash \textit{let } f = \mu f{:}\tau \to \tau_2.\, \lambda x.M_1 \textit{ in } M_2 : \tau_2}$$

This means that the following *derived rule* is sound *and* complete:

$$\textsc{LetRecMono}\ \dfrac{\Gamma, f : \tau \to \tau_1; x : \tau \vdash M_1 : \tau_1 \qquad \Gamma, f : \tau \to \tau_1 \vdash M_2 : \tau_2}{\Gamma \vdash \textit{let rec } (f : \tau \to \tau_1)\ x = M_1 \textit{ in } M_2 : \tau_2}$$

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 is reaches to 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:

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

$$\textsc{Handle-Val}\qquad \textit{try}\,V\ \textit{with}\ M \longrightarrow V$$

$$\textsc{Handle-Raise}\qquad \textit{try raise}\,V\ \textit{with}\ M \longrightarrow M\ V$$

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:

$$\textit{try}\; K\; (\textit{raise}\; M)\; \textit{with}\; \lambda x.\, x$$

Exceptions Example

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

$$\begin{array}{ll} & \textit{try}\ K\ (\textit{raise}\ M)\ \textit{with}\ \lambda x.\,x \qquad\qquad \text{by C}\small{\text{ONTEXT}} \\ \longrightarrow & \textit{try}\ K\ (\textit{raise}\ V)\ \textit{with}\ \lambda x.\,x \end{array}$$

Exceptions Example

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

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

Exceptions — Example

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

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

Exceptions Example

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

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

In particular, we do not have the following step,

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

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.

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

Exceptions on the type of exception

What can 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. ML declares a new type *exn* for exceptions which is a sum type, except that all cases are not declared in advance, but only as needed.

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

This is because $\mathit{raise}\,\cdot$ and $\mathit{try}\,\cdot\,\mathit{with}\,\cdot$ must agree beforehand on the type of exceptions as this type is not passed around by the typing rules.

Exceptions Type soundness

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

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*.*

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.

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 the semantics by replacing propagation of exceptions by deep handling of exceptions inside terms.

Replace the three 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 is perhaps a more intuitive, but equivalent, semantics for exceptions.

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

Exceptions small syntax 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 pattern in programming:

```
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 small syntax 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.

Why?

Exceptions — small syntax 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:

Can you find one?

Exceptions small syntax 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* $(Val : \lambda x.\, M_3 \mathrel{[]} Exc : M_2)$
- 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, none of them is very readable.

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 contents 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) 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 intuitively 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 $M \,/\, \mu$ 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:

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

References

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

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

References

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

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

Three new reduction rules are added:

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

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:

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

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 *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:

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

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 $\mu, \Gamma \vdash M : \tau$.

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?

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

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

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:

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

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.

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

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:

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

Establishing subject reduction

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

The hypotheses are:

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

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:

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

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:

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

Establishing subject reduction

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

The hypotheses are:

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

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:

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

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

(The 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:

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

Establishing subject reduction

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

The hypotheses are:

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

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:

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

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:

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

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 Mezo language [Pottier and Protzenko, 2012] designed especially for the explicit control of resources.

Further reading

For a textbook introduction to λ-calculus and simple types, see Pierce [2002].

For more details about syntactic type soundness proofs, see Wright and Felleisen [1994].

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.

▷ Adam Chlipala. A certified type-preserving compiler from lambda calculus to assembly language. In *ACM Conference on Programming Language Design and Implementation (PLDI)*, pages 54–65, June 2007.

▷ Dan Grossman. Quantified types in an imperative language. *ACM Transactions on Programming Languages and Systems*, 28(3): 429–475, May 2006.

Bibliography II

▷ Bob Harper and Mark Lillibridge. ML with callcc is unsound. Message to the TYPES mailing list, July 1991.

J. Roger Hindley and Jonathan P. Seldin. *Introduction to Combinators and Lambda-Calculus.* Cambridge University Press, 1986.

▷ John Hughes. Why functional programming matters. *Computer Journal*, 32(2):98–107, 1989.

▷ Peter J. Landin. Correspondence between ALGOL 60 and Church's lambda-notation: part I. *Communications of the ACM*, 8(2):89–101, 1965.

▷ 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.

▷ Robin Milner. A theory of type polymorphism in programming. *Journal of Computer and System Sciences*, 17(3):348–375, December 1978.

Bibliography III

- Greg Morrisett, David Walker, Karl Crary, and Neal Glew. From system F to typed assembly language. *ACM Transactions on Programming Languages and Systems*, 21(3):528–569, May 1999.
- 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.

Benjamin C. Pierce. *Types and Programming Languages.* MIT Press, 2002.

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

Bibliography IV

- François Pottier and Didier Rémy. The essence of ML type inference. In Benjamin C. Pierce, editor, *Advanced Topics in Types and Programming Languages*, chapter 10, pages 389–489. MIT Press, 2005.
- John C. Reynolds. Types, abstraction and parametric polymorphism. In *Information Processing 83*, pages 513–523. Elsevier Science, 1983.
- John C. Reynolds. Three approaches to type structure. In *International Joint Conference on Theory and Practice of Software Development (TAPSOFT)*, volume 185 of *Lecture Notes in Computer Science*, pages 97–138. Springer, March 1985.
- W. W. Tait. Intensional interpretations of functionals of finite type i. *The Journal of Symbolic Logic*, 32(2):pp. 198–212, 1967. ISSN 00224812.

Bibliography V

- 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 and Matthias Felleisen. A syntactic approach to type soundness. *Information and Computation*, 115(1):38–94, November 1994.