MPRI, Typage

Didier Rémy
(With much course meterial from François Pottier)

September 14, 2010

Plan of the course

Introduction

Simply-typed λ-calculus

Introduction

Contents

Online material

Course material (notes, slides and other information) are available at http://gallium.inria.fr/~remy/mpri/.

You are very welcome to ask questions!

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

Please don't wait the end of the course to raise problems.

You may send me mail at Didier.Remy@inria.fr

What is functional programming?

The term "functional programming" means various things. *Functional programming* views functions as ordinary data—which, in particular can be passed as argument to other functions and stored in data structures.

Functional programming often loosely or strongly discourages the use of modifiable data, in favor of effect-free transformations of data.

(In contrast, the mainstream object-oriented programming languages, view objects as the primary kind of data, and encourage the use of modifiable data.)

A common idea behind functional programming is that repetitive patterns can be abstracted away as functions that are called multiple times so avoidnig code duplication.

What are functional programming?

Functional programming languages are traditionally *typed* (Scheme and Erlang are exceptions) and have close connections with logic. We will focus on typed langages. In fact types will play a central role as explained below.

Functional programming languages are usually given a precise and formal semantics derived from the one of the lambda-calculus.

Functional programming languages differ in that 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.

Functional languages are usually *sequential* languages, whose model of evaluation is not concurrent, even if core languages may then be extended with primitives to support concurrency.

Contents

What are types?

A *type* is a concise, formal description of the behavior of a program fragment.

For instance, the following are types:

int	an integer
int → bool	a function that maps an integer argument to a Boolean result
(int → bool) → (list int → 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 are useful for several reasons:

- 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 at the machine level)

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."*

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.

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.

Because type systems are *compositional*, a type inference problem can often be expressed as a *constraint solving* problem, where constraints are made up of predicates about types, conjunction, and existential quantification.

Full type reconstruction is undecidable for expressive type systems.

Outline of the course

This course is structured in seven $2^{1/2}$-hour 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

4,5 Extistential types. Type-preserving closure conversion.

6,7 Overloading. Type classes

Simply-typed lambda-calculus

Contents

Why λ-calculus?

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

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 given by

$$T ::= X \mid T \to T \mid \ldots$$

where X denotes a type variable.

λ-terms, also known as *terms* and *expressions*, are given by:

$$t ::= x \mid \lambda x{:}T.\, t \mid t\ t \mid \ldots$$

where x denotes a (value) variable.

We assume given denumerable sets of type variables and value variables.

More term- and type-level constructs will be introduced later on, as suggested by the "..." in the definitions.

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:

$$v ::= \lambda x{:}T.\, t \mid \ldots$$

The *reduction relation* $t_1 \longrightarrow t_2$ is inductively defined:

$$\overset{\beta_v}{(\lambda x{:}T.\, t)\; v \longrightarrow [x \mapsto v]t} \qquad \overset{\text{Context}}{\frac{t \longrightarrow t'}{E[t] \longrightarrow E[t']}}$$

Evaluation contexts are defined as follows:

$$E ::= []\; t \mid v\; [] \mid \ldots$$

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

An evaluation context of arbitrary depth can be defined as:

$$\mathcal{E} ::= [] \mid E[\mathcal{E}]$$

Static semantics

Technically, the type system is a 3-place predicate, whose instances are called *judgments*. Judgments take the form:

$$\Gamma \vdash t : T$$

where a *typing context* Γ is a finite sequence of bindings of variables to types.

Typing context

A typing context (also called a *type environment*) Γ binds program variables to types. It can be extended with the notation $\Gamma, x : T$.

To avoid confusion between the new binding and any other binding that may appear in Γ, we disallow type environments to bind the same variable several times. This is not restrictive because bound variables can be renamed in source programs 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.

Static semantics

Judgments are defined inductively:

$$\text{Var}\ \ \Gamma \vdash x : \Gamma(x) \qquad \text{Abs}\ \ \frac{\Gamma, x : T_1 \vdash t : T_2}{\Gamma \vdash \lambda x{:}T_1.\, t : T_1 \to T_2} \qquad \text{App}\ \ \frac{\Gamma \vdash t_1 : T_1 \to T_2 \qquad \Gamma \vdash t_2 : T_1}{\Gamma \vdash t_1\ t_2 : T_2}$$

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 *type derivation*:

$$\dfrac{\dfrac{\dfrac{\dfrac{}{\Gamma \vdash f : T_1 \to T_2}\,\text{Var} \quad \dfrac{}{\Gamma \vdash x : T_1}\,\text{Var}}{\Gamma \vdash f\ x : T_2}\,\text{App} \quad \dfrac{\dfrac{}{\Gamma \vdash f : T_1 \to T_2}\,\text{Var} \quad \dfrac{}{\Gamma \vdash y : T_1}\,\text{Var}}{\Gamma \vdash f\ y : T_2}\,\text{App}}{f : T_1 \to T_2; x, y : T_1 \vdash (f\ x, f\ y) : T_2 \times T_2}\,\text{Pair}}{\varnothing \vdash \lambda f{:}T_1 \to T_2.\, \lambda x{:}T_1.\, \lambda y{:}T_1.\, (f\ x, f\ y) : (T_1 \to T_2) \to T_1 \to T_1 \to (T_2 \times T_2)}\,\text{Abs}^3$$

Γ stands for $(f : T_1 \to T_2; x, y : T_1)$. Rule Pair is introduced later on.

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

Inversion of typing rules

This informal reasoning that was used to build the previous typing derivation is stated formally in the inversion Lemma, which describes how the subterm of a well-typed term can be typed.

Lemma (Inversion of typing rules)

Assume $\Gamma \vdash t : T$.

- *If* t *is a variable* x, *then* $x \in \mathrm{dom}(\Gamma)$ *and* $\Gamma(x) = T$.
- *If* t *is* $t_1\ t_2$ *then* $\Gamma \vdash t_1 : T_1 \to T$ *and* $\Gamma \vdash t_2 : T_1$ *for some type* T_1.
- *If* t *is* $\lambda x{:}T_1.t_1$, *then* T *is of the form* $T_1 \to T_2$ *and* $\Gamma, x : T_1 \vdash t_1 : T_2 \to T_1$.

The inversion lemma is a basic property that is 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 (weakening of) 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 weakening of the typing context) in a precise and concise way, because terms of the language have a concrete syntax.

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

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

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

$$\begin{array}{rcl} \lceil x \rceil & \triangleq & x \\ \lceil \lambda x{:}T.\, t \rceil & \triangleq & \lambda x.\, \lceil t \rceil \\ \lceil t_1\ t_2 \rceil & \triangleq & \lceil t_1 \rceil\ \lceil t_2 \rceil \end{array}$$

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.)

Type erasure 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. This is an important property for a language to have, called *type-erasing semantics*.

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

Type erasure 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. This is an important property for a language to have, called *type-erasing semantics*.

However, 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 in correspondence.

Type-erasing semantics

On the one hand, type erasure preserves reduction.

Lemma

If $t_1 \longrightarrow t_2$ *then* $\lceil t_1 \rceil \longrightarrow \lceil t_2 \rceil$.

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

Lemma

If $\lceil t \rceil \longrightarrow a$ *then there exists* t' *such that* $t \longrightarrow t'$ *and* $\lceil t' \rceil = a$.

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

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

Type-erasing semantics

Whether a language has a *type-erasing semantics* is an important property about the language.

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

the properties of an implicitly-typed language 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 t is *well-typed* if it admits some type T 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

Type soundness follows from two properties:

Theorem (Subject reduction)

Reduction preserves types: if $t_1 \longrightarrow t_2$ then for any type T such that $\emptyset \vdash t_1 : T$, we also have $\emptyset \vdash t_2 : T$.

Theorem (Progress)

A well-typed, irreducible term is a value: if $\emptyset \vdash t : T$ and $t \not\longrightarrow$, then t is a value.

This syntactic proof method is due to Wright and Felleisen [1994].

Establishing subject reduction

Subject reduction is proved by *induction* over the hypothesis $t_1 \longrightarrow t_2$. 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.

$$\beta_v \quad (\lambda x{:}T.\,t)\; v \longrightarrow [x \mapsto v]t$$

$$\text{Context} \quad \frac{t \longrightarrow t'}{E[t] \longrightarrow E[t']}$$

Establishing subject reduction

Case β

In the β-reduction case, the first hypothesis is

$$(\lambda x{:}T.\,t)\ v \longrightarrow [x \mapsto v]t \tag{1}$$

the second hypothesis is

$$\varnothing \vdash (\lambda x{:}T.\,t)\ v : T_0 \tag{2}$$

and the goal is

$$\varnothing \vdash [x \mapsto v]t : T_0 \tag{3}$$

How do we proceed?

Establishing subject reduction — Case β

We *decompose* the hypothesis.

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

$$\text{App}\ \dfrac{\text{Abs}\ \dfrac{x:T \vdash t:T_0\ \ (\mathbf{4})}{\varnothing \vdash (\lambda x{:}T.\,t) : T \to T_0} \qquad \varnothing \vdash v : T\ \ (\mathbf{5})}{\varnothing \vdash (\lambda x{:}T.\,t)\ v : T_0\ \ (2)}$$

Where next?

Establishing subject reduction — Value substitution

To conclude, we only need the following lemma:

Lemma (Value substitution)

If $x : T \vdash t : T_0$ *and* $\varnothing \vdash v : T$, *then* $\varnothing \vdash [x \mapsto v]t : T_0$.

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

How do we prove this lemma?

Establishing subject reduction — Value substitution

The lemma must be suitably *generalized* so that it can be proved by *structural induction* over the typing derivation for t:

Lemma (Value substitution, strengthened)

If $x : T, \Gamma \vdash t : T_0$ *and* $\varnothing \vdash v : T$, *then* $\Gamma \vdash [x \mapsto v]t : T_0$.

The proof is now straightforward, and, at variables, exploits another lemma:

Lemma (Weakening)

If $\varnothing \vdash v : T_1$ *then* $\Gamma \vdash v : T_1$.

This closes the case of the β-reduction rule.

Establishing subject reduction

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 $\Gamma \vdash t : T$ *and* $x \notin \mathrm{dom}(\Gamma)$, *then* $\Gamma, y : T' \vdash t : T$.

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

Case t *is* x: Then x must be bound to T in Γ. Hence, it is also bound to T in $(\Gamma, y : T'$. We conclude by rule Var.

Case t *is* $\lambda x{:}T_2.t_1$: *W.l.o.g*, we may choose $x \notin \mathrm{dom}(\Gamma)$ and $x \neq y$. We have $\Gamma, x : T_2 \vdash t_1 : T_1$ with $T_2 \to T_1$ equal to T. By induction hypothesis, we have $\Gamma, x : T_2, y : T' \vdash t_1 : T_1$. Thanks to a *permutation* lemma, we have $\Gamma, y : T', x : T_2 \vdash t_1 : T_1$ and we conclude by Rule Abs.

Case t *is* $t_1\ t_2$: easy.

Establishing subject reduction — Permutation

Lemma (Permutation lemma)

If $\Gamma \vdash t : T$ *and* Γ' *is a permutation of* Γ*, then* $\Gamma' \vdash t : T$*.*

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 variables declarations).

Formally, the proof is by induction on t.

Establishing subject reduction — Case Context

In the context case, the first hypothesis is

$$t \longrightarrow t' \tag{1}$$

where, by induction hypothesis, this reduction preserves types (2).

The second hypothesis is

$$\varnothing \vdash E[t] : T \tag{3}$$

where E is an *evaluation context* ($E ::= []\ t \mid v\ [] \mid \ldots$).

The goal is

$$\varnothing \vdash E[t'] : T \tag{4}$$

How do we proceed?

Establishing subject reduction

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 $\varnothing \vdash E[t] : T$, *then, there exists* T' *such that:*

- $\varnothing \vdash t : T'$,
- *for every* t', $\varnothing \vdash t' : T'$ *implies* $\varnothing \vdash E[t'] : T$.

The proof is straightforward, by cases over E.

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

Establishing progress

Progress ("A well-typed term t is either reducible or a value") is proved by *structural induction* over the term t. 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, because *a variable is never well-typed* (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 and proceeds as follows.

Establishing progress

Let us consider a well-typed term $t_1\ t_2$.

By inversion of typing rules, there exist types T_1 and T_2 such that $\varnothing \vdash t_1 : T_2 \to T_1$ and $\varnothing \vdash t_2 : T_2$. In particular, both t_1 and t_2 are well-typed. By the induction hypothesis, t_1 is either reducible or a value v_1.

Establishing progress

Let us consider a well-typed term $t_1\ t_2$.

By inversion of typing rules, there exist types T_1 and T_2 such that $\varnothing \vdash t_1 : T_2 \to T_1$ and $\varnothing \vdash t_2 : T_2$. In particular, both t_1 and t_2 are well-typed. By the induction hypothesis, t_1 is either reducible or a value v_1.

If t_1 is reducible, then, because $[]\ t_2$ is an evaluation context, $t_1\ t_2$ is reducible as well, and we are done.

Establishing progress

Let us consider a well-typed term $t_1\ t_2$.

By inversion of typing rules, there exist types T_1 and T_2 such that $\varnothing \vdash t_1 : T_2 \to T_1$ and $\varnothing \vdash t_2 : T_2$. In particular, both t_1 and t_2 are well-typed. By the induction hypothesis, t_1 is either reducible or a value v_1.

If t_1 is reducible, then, because $[]\ t_2$ is an evaluation context, $t_1\ t_2$ is reducible as well, and we are done.

Otherwise, by the induction hypothesis, t_2 is either reducible or a value v_2. If t_2 is reducible, then, because $v_1\ []$ is an evaluation context, $v_1\ t_2$ is reducible as well, and we are done.

Establishing progress

Let us consider a well-typed term $t_1\ t_2$.

By inversion of typing rules, there exist types T_1 and T_2 such that $\varnothing \vdash t_1 : T_2 \to T_1$ and $\varnothing \vdash t_2 : T_2$. In particular, both t_1 and t_2 are well-typed. By the induction hypothesis, t_1 is either reducible or a value v_1.

If t_1 is reducible, then, because $[]\ t_2$ is an evaluation context, $t_1\ t_2$ is reducible as well, and we are done.

Otherwise, by the induction hypothesis, t_2 is either reducible or a value v_2. If t_2 is reducible, then, because $v_1\ []$ is an evaluation context, $v_1\ t_2$ is reducible as well, and we are done.

Otherwise, because v_1 is a value of type $T_1 \to T_2$, it must be a λ-abstraction (*see next slide*), so $v_1\ v_2$ is a β-redex, and we are done. □

Classification of values

We have appealed to the following property:

Lemma (Classification)

Assume $\varnothing \vdash v : T$. *Then,*

- *if* T *is an arrow type, then* v *is a* λ*-abstraction;*
- ...

Proof.

By *cases over* v:

- if v is a λ-abstraction, then T must be an arrow 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 *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.

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 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

Because 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 no form of recursion in types or terms has been added.

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 new construction and could not occur without it.

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 $\mathcal{T}_T$ of terminating terms of type T and
- show that any term of type T is actually in $\mathcal{T}_T$, 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 $\mathcal{T}_T$ of terminating terms of type T and
- show that any term of type T is actually in $\mathcal{T}_T$, 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 $T_1 \to T_2$ should not only terminate but also terminate when applied to terms in $\mathcal{T}_{T_1}$,

Normalization

Definition

Let $\mathcal{T}_T$ be defined inductively on T as follows: let $\mathcal{T}_X$ be the set of terms that terminates; let $\mathcal{T}_{T_1 \to T_2}$ be the set of terms t_2 of type T_2 that terminates and such that $t_1\ t_2$ is in $\mathcal{T}_{T_2}$ for any term t_1 in $\mathcal{T}_{T_1}$.

The set $\mathcal{T}_T$ can be seen as a predicate, *i.e.* a unary relation. It is called a (unary) *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 T preserves membership in $\mathcal{T}_T$:

Lemma

If $\varnothing \vdash t : T$ and $t_1 \longrightarrow t_2$, then $t_1 \in \mathcal{T}_T$ iff $t_2 \in \mathcal{T}_T$.

Proof.

By induction on the structure of the type T □

All terms in $\mathcal{T}_T$ are terminating.

Lemma

For any type T, the reduction of any term in $\mathcal{T}_T$ halts.

The proof is immediate.

Normalization

Therefore, it just remains to show that any term of type T is in $\mathcal{T}_T$:

Lemma

If $\varnothing \vdash t : T$, *then* $t \in \mathcal{T}_T$.

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

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

If $\vec{x} : \vec{T} \vdash t : T$ *and* $\vec{v}_i \in \mathcal{T}_{T_i}$ *for all* v_i *in* $\vec{v}$, *then* $[\vec{x} \mapsto \vec{v}]t \in \mathcal{T}_T$.

Normalization

Proof.

By structural induction on t. Assume $\vec{x}:\vec{T} \vdash t : T$.

The only interesting case is when t is $\lambda x{:}T_1.t_2$: By inversion of typing, we know that $\vec{x}:\vec{T}, x:T_1 \vdash t_2 : T_2$ where $T_1 \to T_2$ is T. To show that $[\vec{x} \mapsto \vec{v}]t$ is in $\mathcal{T}_T$, we must show that t is terminating, which is obvious, as it is a value, and also that its application to any t_1 in $\mathcal{T}_{T_1}$ is in $\mathcal{T}_{T_2}$ (**1**). Let $t_1 \in \mathcal{T}_{T_1}$. By definition $t_1 \longrightarrow^* v$ (**2**). We have:

$$
\begin{array}{rcll}
([\vec{x} \mapsto \vec{v}]t)\ t_1 & \triangleq & ([\vec{x} \mapsto \vec{v}](\lambda x{:}T_1.t_2))\ t_1 & \text{definition of } t \\
& = & (\lambda x{:}T_1.[\vec{x} \mapsto \vec{v}]t_2)\ t_1 & \text{choose } x \# \vec{x} \\
& \longrightarrow^* & (\lambda x{:}T_1.[\vec{x} \mapsto \vec{v}]t_2)\ v & \text{by (2)} \\
& \longrightarrow & [x \mapsto v]([\vec{x} \mapsto \vec{v}]t_2) & \text{by } (\beta) \\
& \in & \mathcal{T}_{T_2} & \text{by induction hypothesis}
\end{array}
$$

which establishes (1) since $\mathcal{T}_{T_2}$ is closed by reduction. □

Contents

Unit

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

$$\begin{array}{lcl} v & ::= & \ldots \mid () \\ t & ::= & \ldots \mid () \end{array}$$

No new reduction rule is introduced.

Unit

Types are extended with a "unit":

$$T ::= \ldots \mid \mathsf{unit}$$

A typing rule is introduced:

$$\frac{}{\Gamma \vdash () : \mathsf{unit}}\ \text{Unit}$$

Exercise

Check that type soundness is preserved.

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:

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

A new reduction rule is introduced:

$$\mathsf{proj}_i\ (v_1, v_2) \longrightarrow v_i$$

Pairs

Types are extended:

$$T ::= \ldots \mid T \times T$$

Two new typing rules are introduced:

$$\text{Pair}\quad \frac{\Gamma \vdash t_1 : T_1 \qquad \Gamma \vdash t_2 : T_2}{\Gamma \vdash (t_1, t_2) : T_1 \times T_2} \qquad \text{Proj}\quad \frac{\Gamma \vdash t : T_1 \times T_2}{\Gamma \vdash \mathrm{proj}_i\, t : T_i}$$

Exercise

Check that type soundness is preserved when adding pairs.

Sums

Values, expressions, evaluation contexts are extended:

$$\begin{array}{lcl} v & ::= & \ldots \mid \mathsf{inj}_i\ v \\ t & ::= & \ldots \mid \mathsf{inj}_i\ t \mid \mathit{case}\ t\ \mathit{of}\ v \parallel v \\ E & ::= & \ldots \mid \mathsf{inj}_i\ [] \mid \mathit{case}\ []\ \mathit{of}\ v \parallel v \end{array}$$

A new reduction rule is introduced:

$$\mathit{case}\ \mathsf{inj}_i\ v\ \mathit{of}\ v_1 \parallel v_2 \longrightarrow v_i\ v$$

Sums

Types are extended:

$$T ::= \ldots \mid T + T$$

Two new typing rules are introduced:

Inj

$$\frac{\Gamma \vdash t : T_i}{\Gamma \vdash \mathrm{inj}_i\ t : T_1 + T_2}$$

Case

$$\frac{\Gamma \vdash t : T_1 + T_2 \qquad \Gamma \vdash v_1 : T_1 \to T \qquad \Gamma \vdash v_2 : T_2 \to T}{\Gamma \vdash \mathrm{case}\ t\ \mathrm{of}\ v_1 \,[]\, v_2 : T}$$

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:

$$v ::= \ldots \mid \mathsf{inj}_i\ v\ \mathsf{as}\ T$$

and modifying the typing rules and reduction rules accordingly.

Modularity of extensions

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

- a new type constructor, to form types 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 "...") 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 different 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:

$$\begin{array}{rcl} v & ::= & \ldots \mid \mu f{:}T.\,\lambda x.t \\ t & ::= & \ldots \mid \mu f{:}T.\,\lambda x.t \end{array}$$

A new reduction rule is introduced:

$$(\mu f{:}T.\,\lambda x.t)\; v \longrightarrow [f \mapsto \mu f{:}T.\,\lambda x.t][x \mapsto v]t$$

Recursive functions

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

A new typing rule is introduced:

$$\text{FixAbs}\quad \frac{\Gamma, f : T_1 \to T_2 \vdash \lambda x{:}T_1.\, t : T_1 \to T_2}{\Gamma \vdash \mu f{:}T_1 \to T_2.\, \lambda x.t : T_1 \to T_2}$$

In the premise, the type $T_1 \to T_2$ serves both as an assumption and a goal. This is a typical feature of recursive definitions.

A derived construct: let

The construct "let $x : T = t_1$ in t_2" can be viewed as syntactic sugar for the β-redex "$(\lambda x{:}T.\, t_2)\ t_1$".

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

$$\text{App}\ \frac{\text{Abs}\ \dfrac{\Gamma, x : T_1 \vdash t_2 : T_2}{\Gamma \vdash \lambda x{:}T_1.\, t_2 : T_1 \to T_2} \qquad \Gamma \vdash t_1 : T_1}{\Gamma \vdash (\lambda x{:}T_1.\, t_2)\ t_1 : T_2}$$

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

$$\text{LetMono}\ \frac{\Gamma \vdash t_1 : T_1 \qquad \Gamma, x : T_1 \vdash t_2 : T_2}{\Gamma \vdash \text{let } x : T_1 = t_1 \text{ in } t_2 : T_2}$$

The construct "$t_1; t_2$" can in turn be viewed as syntactic sugar for let x : unit $= t_1$ in t_2 where $x \# t_2$.

A derived construct: let

In the derived form let $x : T_1 = t_1$ in t_2 the type of t_1 must be explicitly given, although by uniqueness of types, it is entirely determined by the expression t_1 itself.

Hence, it seems redundant.

Indeed, we can replace the derived form by a primitive form let $x = t_1$ in t_2 with the following primitive typing rule.

$$\text{LetMono}\quad \frac{\Gamma \vdash t_1 : T_1 \qquad \Gamma, x : T_1 \vdash t_2 : T_2}{\Gamma \vdash \text{let } x = t_1 \text{ in } t_2 : T_2}$$

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 $(f : T)\ x = t_1$ in t_2" can be viewed as syntactic sugar for "let $f = \mu f{:}T.\lambda x.t_1$ in t_2". The latter can be type-checked *only* by a derivation of the form:

$$\text{LetMono}\ \frac{\text{FixAbs}\ \dfrac{\Gamma, f : T \to T_1; x : T \vdash t_1 : T_1}{\Gamma \vdash \mu f{:}T \to T_1.\lambda x.t_1 : T \to T_1} \qquad \Gamma, f : T \to T_1 \vdash t_2 : T_2}{\Gamma \vdash \text{let } f = \mu f{:}T \to T_2.\lambda x.t_1 \text{ in } t_2 : T_2}$$

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

$$\text{LetRecMono}\ \frac{\Gamma, f : T \to T_1; x : T \vdash t_1 : T_1 \qquad \Gamma, f : T \to T_1 \vdash t_2 : T_2}{\Gamma \vdash \text{let rec } (f : T \to T_1)\ x = t_1 \text{ in } t_2 : T_2}$$

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:

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

Exceptions

Semantics

However, we do not treat raise v as a value.

This stops the normal order of evaluation,

Instead, three reduction rules propagate and handle exceptions.

Raise
$$F[\text{raise } v] \longrightarrow \text{raise } v$$

Handle-Val
$$\text{try } v \text{ with } t \longrightarrow v$$

Handle-Raise
$$\text{try raise } v \text{ with } t \longrightarrow t\ v$$

Rule Raise propagates an exception up the evaluation contexts, but not through a handler. This is why the rule uses an evaluation context F which stands for any E other than try [] with t.

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 $t \longrightarrow v$, we have the following reduction:

$$\begin{array}{rll}
 & \mathsf{try}\ K\ (\mathsf{raise}\ t)\ \mathsf{with}\ \lambda x.\, x & \text{by Context} \\
\longrightarrow & \mathsf{try}\ K\ (\mathsf{raise}\ v)\ \mathsf{with}\ \lambda x.\, x & \text{by Raise} \\
\longrightarrow & \mathsf{try}\ \mathsf{raise}\ v\ \mathsf{with}\ \lambda x.\, x & \text{by Handle-Raise} \\
\longrightarrow & (\lambda x.\, x)\ v & \text{by } \beta \\
\longrightarrow & v &
\end{array}$$

In particular, we do not have the following step,

$$\begin{array}{rll}
 & \mathsf{try}\ K\ (\mathsf{raise}\ v)\ \mathsf{with}\ \lambda x.\, x & \text{by } \beta \\
\not\longrightarrow & \mathsf{try}\ \lambda y.\, y\ \mathsf{with}\ \lambda x.\, x \longrightarrow \lambda y.\, y &
\end{array}$$

since $\mathsf{raise}\ v$ is *not* a value.

Exceptions

Typing rules

We need not add a new type but instead assume given a fixed type T_{exn} for exceptional values.

Typing rules are:

$$\text{Raise}\quad \frac{\Gamma \vdash t : T_{exn}}{\Gamma \vdash \text{raise}\, t : T} \qquad \text{Try}\quad \frac{\Gamma \vdash t_1 : T \qquad \Gamma \vdash t_2 : T_{exn} \to T}{\Gamma \vdash \text{try}\, t_1 \text{ with } t_2 : T}$$

There are some subtleties:

- Raise turns an expression of type T_{exn} into an exception.
- Consistently, the handler has type $T_{exn} \to T$, since it receives the exception value of type exn as argument;
- Both premises of Rule Try must return values of the same type T.
- raise t can have any type, as the current computation is aborted.

Exceptions — on the type of exception

What can we choose for T_{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 when needed.

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

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

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 $\varnothing \vdash t : T$ *and* $t \not\longrightarrow$, *then* t *is either* v *or* $\mathsf{raise}\, v$ *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.

Exception 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.

Repalce the three reduction by:

Handle-Val'
try $G[v]$ with $t \longrightarrow v$

Handle-Raise'
try $G[\text{raise } v]$ with $t \longrightarrow t\ v$

where G is an evaluation context without handlers of arbitrary depth:

$$G ::= [\,] \mid G\ t \mid v\ G \mid \text{raise } G$$

This is perhaps a more intuitive, but equivalent, semantics for exceptions.

In this case, uncaught exceptions are of the form $G[v]$.

Exceptions

small syntax variation

Benton and Kennedy [2001] have argued for merging let and try constructs into a unique form let $x = t_1$ with t_2 in t_3.

The expression t_1 is evaluated first and if it returns a value it is substituted for x in t_3, as if we had evaluated let $x = t_1$ in t_3; otherwise, *i.e.*, if it raises an exception raise v, then the exception is handled by t_2, as if we had evaluated try t_1 with t_2.

The main advantage of this combined form is that it captures a common pattern in programming with no elegant workaround

```
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
```

This form is also better suited for program transformations as argued by Benton and Kennedy [2001].

Exceptions

small syntax variation

Encoding the new form let $x = t_1$ with t_2 in t_3 with "let" and "try" is not easy:

In particular, it is not equivalent to: try let $x = t_1$ in t_2 with t_3. Why?

Exceptions

small syntax variation

Encoding the new form let $x = t_1$ with t_2 in t_3 with "let" and "try" is not easy:

In particular, it is not equivalent to: try let $x = t_1$ in t_2 with t_3. Why?

The continuation t_2 could raise an exception that would then be handled by t_2, which is incorrect.

Exceptions

small syntax variation

Encoding the new form let $x = t_1$ with t_2 in t_3 with "let" and "try" is not easy:

In particular, it is not equivalent to: try let $x = t_1$ in t_2 with t_3. Why?

The continuation t_2 could raise an exception that would then be handled by t_2, which is incorrect.

There are several encodings:

- Use a sum type to know whether t_1 raised an exception:
  $\mathit{case}\ (\text{try}\ \mathit{Val}\ t_1\ \text{with}\ \lambda y.\ \mathit{Exc}\ y)\ \mathit{of}\ (\mathit{Val} : \lambda x.\ t_3\ []\ \mathit{Exc} : t_2)$
- Freeze the continuation t_3 while handling the exception:
  $(\text{try let}\ x = t_1\ \text{in}\ \lambda().\ t_3\ \text{with}\ \lambda y.\ \lambda().\ t_2\ y)\ ()$

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:

$$\begin{array}{rcl} t & ::= & \ldots \mid \mathsf{ref}\ t \mid t := t \mid !t \\ E & ::= & \ldots \mid \mathsf{ref}\ [] \mid [] := t \mid v := [] \mid ![] \end{array}$$

References

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

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

(which intuitively should yield 1) would reduce to:

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

(which intuitively yields 3).

How shall we solve this problem?

References

(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*:

$$\begin{array}{lcl} v & ::= & \ldots \mid \ell \\ t & ::= & \ldots \mid \ell \end{array}$$

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 t/μ 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 t 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:

$$\begin{aligned}(\lambda x{:}T.\,t)\ v/\mu &\longrightarrow [x \mapsto v]t/\mu \\ E[t]/\mu &\longrightarrow E[t']/\mu' \qquad \text{if } t/\mu \longrightarrow t'/\mu'\end{aligned}$$

Three new reduction rules are added:

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

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:

$$T ::= \ldots \mid \mathsf{ref}\ T$$

Three new typing rules are introduced:

Ref
$$\frac{\Gamma \vdash t : T}{\Gamma \vdash \mathsf{ref}\ t : \mathsf{ref}\ T}$$

Set
$$\frac{\Gamma \vdash t_1 : \mathsf{ref}\ T \qquad \Gamma \vdash t_2 : T}{\Gamma \vdash t_1 := t_2 : \mathsf{unit}}$$

Get
$$\frac{\Gamma \vdash t : \mathsf{ref}\ T}{\Gamma \vdash !t : T}$$

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 t/μ well-typed?

References

When does a location ℓ have type ref T?

References

When does a location ℓ have type ref T? A possible answer is, *when it points to some value of type T*. Intuitively, this could be formalized by a typing rule of the form:

$$\frac{\mu, \varnothing \vdash \mu(\ell) : T}{\mu, \Gamma \vdash \ell : \mathsf{ref}\ T}$$

Comments?

References

When does a location ℓ have type ref T? A possible answer is, *when it points to some value of type T*. Intuitively, this could be formalized by a typing rule of the form:

$$\frac{\mu, \varnothing \vdash \mu(\ell) : T}{\mu, \Gamma \vdash \ell : \text{ref } T}$$

Comments?

- typing judgments would have the form $\mu, \Gamma \vdash t : T$.

References

When does a location ℓ have type ref T? A possible answer is, *when it points to some value of type T*. Intuitively, this could be formalized by a typing rule of the form:

$$\frac{\mu, \varnothing \vdash \mu(\ell) : T}{\mu, \Gamma \vdash \ell : \text{ref } T}$$

Comments?

- typing judgments would have the form $\mu, \Gamma \vdash t : T$.
- 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 T? A possible answer is, *when it points to some value of type T*. Intuitively, this could be formalized by a typing rule of the form:

$$\frac{\mu, \varnothing \vdash \mu(\ell) : T}{\mu, \Gamma \vdash \ell : \mathsf{ref}\, T}$$

Comments?

- typing judgments would have the form $\mu, \Gamma \vdash t : T$.
- 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 T_1 and T_2, then ℓ admits types ref T_1 and ref T_2. So, one can write at type T_1 and read at type T_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* M, a finite mapping of locations to types.

So, when does a location ℓ have type ref T? "When M says so."

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

Comments:

- Tping judgments now have the form $M, \Gamma \vdash t : T$.

References

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

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

$$\text{Store}\quad \frac{\forall \ell \in dom(\mu),\quad M, \varnothing \vdash \mu(\ell) : M(\ell)}{\vdash \mu : M} \qquad \text{Config}\quad \frac{M, \varnothing \vdash t : T \qquad \vdash \mu : M}{\vdash t/\mu : T}$$

Comments:

- This is an *inductive* definition. The store typing M 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 $t/\mu \longrightarrow t'/\mu'$ *and* $\vdash t/\mu : T$, *then* $\vdash t'/\mu' : T$.

Theorem (Progress)

If t/μ *is a well-typed, irreducible configuration, then* t *is a value.*

Restating subject reduction

Inlining Config, subject reduction can also be restated as:

Theorem (Subject reduction, expanded)

If $t/\mu \longrightarrow t'/\mu'$ *and* $M, \varnothing \vdash t : T$ *and* $\vdash \mu : M$, *then there exists* M' *such that* $M', \varnothing \vdash t' : T$ *and* $\vdash \mu' : M'$.

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

Establishing subject reduction

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

The hypotheses are:

$$t/\mu \longrightarrow t'/\mu' \quad \text{and} \quad M, \varnothing \vdash E[t] : T \quad \text{and} \quad \vdash \mu : M$$

Establishing subject reduction

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

The hypotheses are:

$$t/\mu \longrightarrow t'/\mu' \quad \text{and} \quad M, \varnothing \vdash E[t] : T \quad \text{and} \quad \vdash \mu : M$$

Assuming compositionality, there exists T' such that:

$$M, \varnothing \vdash t : T' \quad \text{and} \quad \forall t', \quad (M, \varnothing \vdash t' : T') \Rightarrow (M, \varnothing \vdash E[t'] : T)$$

Establishing subject reduction

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

The hypotheses are:

$$t/\mu \longrightarrow t'/\mu' \quad \text{and} \quad M, \varnothing \vdash E[t] : T \quad \text{and} \quad \vdash \mu : M$$

Assuming compositionality, there exists T' such that:

$$M, \varnothing \vdash t : T' \quad \text{and} \quad \forall t', \quad (M, \varnothing \vdash t' : T') \Rightarrow (M, \varnothing \vdash E[t'] : T)$$

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

$$M', \varnothing \vdash t' : T' \quad \text{and} \quad \vdash \mu' : M'$$

Establishing subject reduction

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

The hypotheses are:

$$t/\mu \longrightarrow t'/\mu' \quad \text{and} \quad M, \varnothing \vdash E[t] : T \quad \text{and} \quad \vdash \mu : M$$

Assuming compositionality, there exists T' such that:

$$M, \varnothing \vdash t : T' \quad \text{and} \quad \forall t', \quad (M, \varnothing \vdash t' : T') \Rightarrow (M, \varnothing \vdash E[t'] : T)$$

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

$$M', \varnothing \vdash t' : T' \quad \text{and} \quad \vdash \mu' : M'$$

Here, *we are stuck*. The context E is well-typed under M, but the term t' is well-typed under M', 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 $t/\mu \longrightarrow t'/\mu'$ *and* $M, \varnothing \vdash t : T$ *and* $\vdash \mu : M$, *then there exists* M' *such that* $M', \varnothing \vdash t' : T$ *and* $\vdash \mu' : M'$ *and* $M \subseteq M'$.

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

Establishing subject reduction

Growing the store typing preserves well-typedness:

Lemma (Stability under memory allocation)

If $M \subseteq M'$ *and* $M, \Gamma \vdash t : T$, *then* $M', \Gamma \vdash t : T$.

(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 $M, \varnothing \vdash E[t] : T$. *Then, there exists* T' *such that:*

- $M, \varnothing \vdash t : T'$,
- *for every* M' *such that* $M \subseteq M'$, *for every* t', $M', \varnothing \vdash t' : T'$ *implies* $M', \varnothing \vdash E[t'] : T$.

Establishing subject reduction

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

The hypotheses are:

$$M, \varnothing \vdash E[t] : T \quad \text{and} \quad \vdash \mu : M \quad \text{and} \quad t/\mu \longrightarrow t'/\mu'$$

Establishing subject reduction

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

The hypotheses are:

$$M, \varnothing \vdash E[t] : T \quad \text{and} \quad \vdash \mu : M \quad \text{and} \quad t/\mu \longrightarrow t'/\mu'$$

By compositionality, there exists T' such that:

$$M, \varnothing \vdash t : T'$$
$$\forall M', \forall t', \quad (M \subseteq M') \Rightarrow (M', \varnothing \vdash t' : T') \Rightarrow (M', \varnothing \vdash E[t'] : T')$$

Establishing subject reduction

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

The hypotheses are:

$$M, \varnothing \vdash E[t] : T \quad \text{and} \quad \vdash \mu : M \quad \text{and} \quad t/\mu \longrightarrow t'/\mu'$$

By compositionality, there exists T' such that:

$$M, \varnothing \vdash t : T'$$
$$\forall M', \forall t', \quad (M \subseteq M') \Rightarrow (M', \varnothing \vdash t' : T') \Rightarrow (M', \varnothing \vdash E[t'] : T')$$

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

$$M', \varnothing \vdash t' : T' \quad \text{and} \quad \vdash \mu' : M' \quad \text{and} \quad M \subseteq M'$$

The goal follows immediately.

Exercise

Exercise (Recommended)

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

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.

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 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.

Bibliography IV

- 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.
- 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.