

Programming Languages, CPSC 604—Slides 7

Lambda-Calculus and Types

Jaakko Järvi

February 12, 2009

Outline

- 1 Adding types to lambda-calculus
- 2 Type safety
- 3 Warmup — simple extensions

Simply typed lambda calculus

- We'll extend lambda calculus with one type `bool`
 - This is just for convenience to help understanding
- For this formalism, we define a type system
- and look at how to establish type safety for this language

Type of lambda abstractions

- Type of a lambda abstraction includes the types of the parameter and result
- Grammar of types:

$$T ::= \text{bool}$$
$$T \rightarrow T$$

- \rightarrow is a **type constructor**
- **Simply typed lambda calculus:** each binding occurrence of a variable is annotated with a type

$$\lambda x : T. t$$

Examples

- What are the types of these terms?

$$\lambda x : \text{bool}.x$$

$$\lambda f : \text{bool} \rightarrow \text{bool}.f \ x$$

- Compare this **explicit typing** to the **implicit typing** (also type *deduction/inference/reconstruction*) in our language \mathcal{NB}
- Or in untyped lambda-calculus:

$$\lambda x.x$$

$$\lambda f.f \ x$$

Purpose of the type system

- From last time: some lambda terms diverge
- What does this term mean?
 $(\lambda f : \text{bool} \rightarrow \text{bool}. f\ x)\ \text{true}$

⇒ reject such cases

- Now the answer to the following question should be clear: “What’s the easiest type system to accomplish this”?

Purpose of the type system

- Again, the goal of the type system:
Reject unsafe programs, but don't reject too many safe programs
- **Exact** type analysis for lambda-calculus undecidable
- We have to settle with a conservative approximation

Quote

The fundamental problem addressed by a type theory is to insure that programs have meaning. The fundamental problem caused by a type theory is that meaningful programs may not have meanings ascribed to them. The quest for richer type systems results from this tension. [Mark Manasse]

Typing relation with context

- Now our typing relation (with context/environment information) is a three-place relation (Γ, t, T) , written as:

$$\Gamma \vdash t : T$$

- and read as:

Term t has type T in the typing context Γ

- If $\Gamma = \emptyset$, it is usually omitted

$$\vdash t : T$$

Typing context

- Also, **type environment**
- Closed terms can be typed without an environment, but where does one get the types of free variables?
- For example, if the type of f is not known, how to type this term:

$$\lambda x : \text{bool}. f \ x$$

- Typing environment is a sequence of bindings (variable-type pairs) written as $x : T$
- Comma operator adds bindings (to the right): $\Gamma, x : T$
- Commonly assumed that variable names are unique
- Typing context can be viewed as a function from variables to their types, thus $\text{dom}(\Gamma)$ is the set of variables bound by Γ

Typing rules for simply-typed lambda calculus

- x, y, z, f, g range over variables
- s, t, u range over terms
- S, T, U range over types

$$\text{T-Variable} \\ \frac{x : T \in \Gamma}{\Gamma \vdash x : T}$$

$$\text{T-Abstraction} \\ \frac{\Gamma, x : T \vdash u : U}{\Gamma \vdash \lambda x : T. u : T \rightarrow U}$$

$$\text{T-Application} \\ \frac{\Gamma \vdash t : U \rightarrow T \quad \Gamma \vdash u : U}{\Gamma \vdash t u : T}$$

Rules for bool

T-True

$\vdash \text{true} : \text{bool}$

T-False

$\vdash \text{false} : \text{bool}$

Typing derivations

- We can construct derivations (proofs) that explain why a given term has a certain type
- Example: $(\lambda f : \text{bool} \rightarrow \text{bool}. f \text{ false}) \lambda g : \text{bool}. g$ has type bool (under empty environment)

Typing derivations

- We can construct derivations (proofs) that explain why a given term has a certain type
- Example: $(\lambda f : \text{bool} \rightarrow \text{bool}. f \text{ false}) \lambda g : \text{bool}. g$ has type bool (under empty environment)

$$\frac{
 \frac{
 \frac{f : \text{bool} \rightarrow \text{bool} \in f : \text{bool} \rightarrow \text{bool}}{f : \text{bool} \rightarrow \text{bool} \vdash f : \text{bool} \rightarrow \text{bool}}
 \quad
 \frac{f : \text{bool} \rightarrow \text{bool} \vdash \text{false} : \text{bool}}{f : \text{bool} \rightarrow \text{bool} \vdash f \text{ false} : \text{bool}}
 }{f : \text{bool} \rightarrow \text{bool} \vdash (\lambda f : \text{bool} \rightarrow \text{bool}. f \text{ false}) : (\text{bool} \rightarrow \text{bool}) \rightarrow \text{bool}}
 \quad
 \frac{
 \frac{g : \text{bool} \in g : \text{bool}}{g : \text{bool} \vdash g : \text{bool}}
 }{g : \text{bool} \vdash \lambda g : \text{bool}. g : \text{bool} \rightarrow \text{bool}}
 }{
 \vdash (\lambda f : \text{bool} \rightarrow \text{bool}. f \text{ false}) \lambda g : \text{bool}. g : \text{bool}
 }$$

Evaluation rules

- Syntax

$$t ::= x \mid v \mid t t$$

$$v ::= \lambda x : T. t \mid \text{true} \mid \text{false}$$

$$T ::= \text{bool} \mid T \rightarrow T$$

- Evaluation rules

$$\frac{t_1 \rightarrow t_1'}{t_1 t_2 \rightarrow t_1' t_2} \qquad \frac{t \rightarrow t'}{v t \rightarrow v t'}$$

$$(\lambda x : T. t) v \rightarrow [v/x]t$$

- t = terms, v = value, T = type
- \rightarrow is the smallest binary relation on terms satisfying the rules
- Note again that evaluation rules do not bother with types!
 - Type checker's task to reject programs that try to apply rules for meaningless types
 - This corresponds (to some extent) to language implementations

Outline

- 1 Adding types to lambda-calculus
- 2 Type safety**
- 3 Warmup — simple extensions

Type safety

- We again define type safety of simply typed lambda-calculus via the progress and preservation theorems
- Reminder:
 - Type safety = progress + preservation
- **Progress:** If t is a closed, well-typed term, then either t is a value, or there exists some u , such that $t \rightarrow u$.
- **Preservation:** If $\Gamma \vdash t : T$ and $t \rightarrow u$, then $\Gamma \vdash u : T$

Simply-typed lambda-calculus on one page

• Syntax

$$t ::= x \mid v \mid t t$$

$$v ::= \lambda x : T . t \mid \text{true} \mid \text{false}$$

$$T ::= \text{bool} \mid T \rightarrow T$$

• Evaluation rules

E-AppFun

$$\frac{t_1 \rightarrow t_1'}{t_1 t_2 \rightarrow t_1' t_2}$$

E-AppArg

$$\frac{t \rightarrow t'}{v t \rightarrow v t'}$$

E-AppAbs

$$(\lambda x : T . t) v \rightarrow [v/x]t$$

• Typing rules

T-Variable

$$\frac{x : T \in \Gamma}{\Gamma \vdash x : T}$$

T-Abstraction

$$\frac{\Gamma, x : T \vdash u : U}{\Gamma \vdash \lambda x : T . u : T \rightarrow U}$$

T-Application

$$\frac{\Gamma \vdash t : U \rightarrow T \quad \Gamma \vdash u : U}{\Gamma \vdash t u : T}$$

T-True

$$\vdash \text{true} : \text{bool}$$

T-False

$$\vdash \text{false} : \text{bool}$$

A few lemmas

- We need a few lemmas, which are utterly trivial in our simple language
- The following lemma allows us to read the type rules backwards:
 - For each syntactic form, we can deduce the types of the subterms

Lemma (Inversion of typing relation)

- 1 If $\Gamma \vdash x : T$, then $x : T \in \Gamma$.
- 2 If $\Gamma \vdash \lambda x : T. t : U$, then for some S with $\Gamma, x : T \vdash t : S$, $U = T \rightarrow S$.
- 3 If $\Gamma \vdash t u : S$, then for some type U , $\Gamma \vdash t : U \rightarrow S$, and $\Gamma \vdash u : U$.
- 4 If $\Gamma \vdash \text{true} : T$, $T = \text{bool}$
- 5 If $\Gamma \vdash \text{false} : T$, $T = \text{bool}$

Proof.

Follows directly from definition of typing relation □

Uniqueness of types

Lemma (Uniqueness of types)

A term t in a typing context Γ , where $FV(t) \in \text{dom}(\Gamma)$, has at most one type.

Proof.

Structural induction on t using the inversion lemma. □

- Does not necessarily hold in more complicated systems, such as ones that support subtyping

Lemma (Canonical forms)

- 1 If v a value of type `bool`, then $v = \text{true}$ or $v = \text{false}$
- 2 If v a value of type $T \rightarrow U$, then $v = \lambda x:T : t$

Proof.

Follows from inversion and uniqueness lemmas. □

Progress

Theorem (Progress)

If t is a closed, well-typed term, then either t is a value, or there exists some s , such that $t \rightarrow s$.

Proof

Induction on typing derivation $\vdash t : T$. For the cases T-True and T-False, t is a value and thus the property holds. Similarly for T-Abstraction. The last derivation cannot be T-Variable, since the theorem requires a closed term.

Proof continues...

...continues.

Finally the case where T-Application is the last rule used. Let $t = u s$.
From inversion lemma $\vdash u : U_1 \rightarrow U_2$ and $\vdash s : U_1$.

By induction hypothesis, u is a value or for some u_1 , $u \rightarrow u_1$. Similarly, s is a value or for some s_1 , $s \rightarrow s_1$.

If $u \rightarrow u_1$, then E-AppFun applies to $u s$.

If u is a value and if $s \rightarrow s_1$, then E-AppArg applies to $u s$.

If both u and s are values, then by the type of u ($U_1 \rightarrow U_2$) and the canonical forms lemma, u is of the form $\lambda x : U_1. r$, to which E-AppAbs applies.



Preservation

- First some lemmas

Lemma (Permutation)

If $\Gamma \vdash t : T$ and Δ a permutation of Γ , then $\Delta \vdash t : T$, with both derivations of the same depth.

Proof.

By induction on typing derivations. □

Lemma (Weakening)

If $\Gamma \vdash t : T$ and $x \notin \text{dom}(\Gamma)$, then $\Gamma, x : U \vdash t : T$, with both derivations of the same depth.

Proof.

By induction on typing derivations. □

Preservation

- More lemmas

Lemma (Substitution preserves types)

If $\Gamma, x: U \vdash t: T$ and $\Gamma \vdash u: U$, then $\Gamma \vdash [u/x]t: T$.

Proof

By induction on derivation of $\Gamma, x: U \vdash t: T$, doing case analysis on the final rule used.

In the case of T-Var, we have $t = y$ for some y , such that $y: T \in (\Gamma, x: U)$. If $y \neq x$ then $[u/x]y = y$ and clearly $\Gamma \vdash [u/x]y: T$. If $y = x$ then $[u/x]x = u$ and from T-Var we can see that $T = U$. Thus we must show that $\Gamma \vdash u: U$, which is an assumption.

Preservation

Lemma (Substitution preserves types)

If $\Gamma, x: U \vdash t: T$ and $\Gamma \vdash u: U$, then $\Gamma \vdash [u/x]t: T$.

Proof continues...

By induction on derivation of $\Gamma, x: U \vdash t: T$, doing case analysis on the final rule used.

In the case of T-Abstraction

$t = \lambda y: T_2. t_1$, $T = T_2 \rightarrow T_1$ and

$\Gamma, x: U, y: T_2 \vdash t_1: T_1$.

It is ok to assume that $x \neq y$ and that $y \notin FV(u)$. Permutation lemma gives us $\Gamma, y: T_2, x: U \vdash t_1: T_1$. By weakening lemma applied to $\Gamma \vdash u: U$, we get $\Gamma, y: T_2 \vdash u: U$. By induction hypothesis, $\Gamma, y: T_2 \vdash [u/x]t_1: T_1$, and by T-Abstraction $\Gamma \vdash \lambda y: T_2. [u/x]t_1: T_2 \rightarrow T_1$. From this, the result follows, as the definition of substitution gives $[u/x]t = \lambda y: T_2. [u/x]t_1$.

Preservation

Lemma (Substitution preserves types)

If $\Gamma, x: U \vdash t: T$ and $\Gamma \vdash u: U$, then $\Gamma \vdash [u/x]t: T$.

Proof continues...

By induction on derivation of $\Gamma, x: U \vdash t: T$, doing case analysis on the final rule used.

Cases of T-True and T-False. Substitution has no affect on the term, so it immediately follows that type is preserved.

Preservation

Lemma (Substitution preserves types)

If $\Gamma, x: U \vdash t: T$ and $\Gamma \vdash u: U$, then $\Gamma \vdash [u/x]t: T$.

... proof continues.

By induction on derivation of $\Gamma, x: U \vdash t: T$, doing case analysis on the final rule used.

In the case of T-Application

$$t = t_1 t_2$$

$$\Gamma, x: U \vdash t_1: T_2 \rightarrow T_1,$$

$$\Gamma, x: U \vdash t_2: T_2, \text{ and}$$

$$T = T_1.$$

Induction hypothesis gives $\Gamma \vdash [u/x]t_1: T_2 \rightarrow T_1$, and $\Gamma \vdash [u/x]t_2: T_2$.

Then, by T-Application, $\Gamma \vdash [u/x]t_1 [u/x]t_2: T$, which by the definition of substitution becomes $\Gamma \vdash [u/x](t_1 t_2): T$ □

Preservation

Theorem (Preservation)

If $\Gamma \vdash t : T$ and $t \rightarrow u$, then $\Gamma \vdash u : T$

Proof.

By induction on typing derivation $\Gamma \vdash t : T$. We must analyze each type rule.

First, none of the cases T-Var, T-Abstraction, T-True, or T-False apply.

Why?

Preservation

Theorem (Preservation)

If $\Gamma \vdash t : T$ and $t \rightarrow u$, then $\Gamma \vdash u : T$

Proof.

By induction on typing derivation $\Gamma \vdash t : T$. We must analyze each type rule.

First, none of the cases T-Var, T-Abstraction, T-True, or T-False apply.

Why?

The terms in these rules are values, and do not reduce to anything. Thus the theorem, which is an implication, is true since the assumptions are not.

Preservation

Theorem (Preservation)

If $\Gamma \vdash t : T$ and $t \rightarrow u$, then $\Gamma \vdash u : T$

Proof continues...

In the case of T-Application, we have the following:

$$t = r s \quad \Gamma \vdash r : R_1 \rightarrow R_2 \quad \Gamma \vdash s : R_1 \quad T = R_2$$

Now look at evaluation rules which can reduce an application.

How many are there?

Preservation

Theorem (Preservation)

If $\Gamma \vdash t : T$ and $t \rightarrow u$, then $\Gamma \vdash u : T$

Proof continues...

In the case of T-Application, we have the following:

$$t = r s \quad \Gamma \vdash r : R_1 \rightarrow R_2 \quad \Gamma \vdash s : R_1 \quad T = R_2$$

Now look at evaluation rules which can reduce an application.

How many are there?

3, and they are E-AppFun, E-AppArg, and E-AppAbs

Preservation

Theorem (Preservation)

If $\Gamma \vdash t : T$ and $t \rightarrow u$, then $\Gamma \vdash u : T$

Proof continues...

$$t = r s \quad \Gamma \vdash r : R_1 \rightarrow R_2 \quad \Gamma \vdash s : R_1 \quad T = R_2$$

In the subcase of E-AppFun we have

$$r \rightarrow r' \quad u = r' s$$

From T-Application premises we get $\Gamma \vdash r : R_1 \rightarrow R_2$, to which we can apply induction hypothesis, and conclude that $\Gamma \vdash r' : R_1 \rightarrow R_2$.
 Now, from T-Application we also know that $\Gamma \vdash s : R_1$, thus combining these to another application of T-Application gives us $\Gamma \vdash r' s : R_2$, which is the same as $\Gamma \vdash u : T$.

Preservation

Theorem (Preservation)

If $\Gamma \vdash t : T$ and $t \rightarrow u$, then $\Gamma \vdash u : T$

Proof continues...

In the subcase of E-AppArg we have

$$t = v s \qquad s \rightarrow s' \qquad u = v s'$$

Similar to the previous subcase.

Preservation

Theorem (Preservation)

If $\Gamma \vdash t : T$ and $t \rightarrow u$, then $\Gamma \vdash u : T$

...proof continues.

$$t = r s \quad \Gamma \vdash r : R_1 \rightarrow R_2 \quad \Gamma \vdash s : R_1 \quad T = R_2$$

In the subcase of E-AppAbs we have

$$t = (\lambda x : R_1. p) v \quad u = [v/x]p$$

From $\Gamma \vdash \lambda x : R_1. p : R_1 \rightarrow R_2$, inversion lemma gives:

$$\Gamma, x : R_1 \vdash p : R_2$$

Substitution lemma gives the desired result $\Gamma \vdash [v/x]p : R_2$. □

Summarizing

- This is what we proved:

Theorem (Progress)

If t is a closed, well-typed term, then either t is a value, or there exists some s , such that $t \rightarrow s$.

Theorem (Preservation)

If $\Gamma \vdash t : T$ and $t \rightarrow u$, then $\Gamma \vdash u : T$.

- As we remember from the \mathcal{NB} language, together they give **type safety**.

Outline

- 1 Adding types to lambda-calculus
- 2 Type safety
- 3 Warmup — simple extensions**

Plan

- We'll eventually get up to simple typed lambda calculus with subtyping
- To gain confidence with dealing with type systems, we'll look at a few simple extensions first

About base types

- We used the `bool` as a base type in our language
- Notice, no operations were defined for them
- The role of `bool` was thus just “some type”
- Commonly, all primitive types can be abstracted away to just one base type, without affecting the formalism in any essential way

Unit type and sequencing

- New syntax: $t ::= \dots \text{unit} \mid t; t$
- New value: $v ::= \dots \text{unit}$
- New type: $T ::= \dots \text{unit}$
- Typing of unit and sequencing:

$$\Gamma \vdash \text{unit} : \text{unit}$$

$$\frac{\Gamma \vdash t : \text{unit} \quad \Gamma \vdash u : U}{\Gamma \vdash t; u : U}$$

- Evaluation of sequencing:

$$\frac{t \rightarrow u}{t; s \rightarrow u; s}$$

$$\text{unit}; u \rightarrow u$$

Ascription

- New syntax: $t ::= \dots t \text{ as } T$
- Typing of ascription:

$$\frac{\Gamma \vdash t : T}{\Gamma \vdash t \text{ as } T : T}$$

- Evaluation rules:

$$\frac{t \rightarrow u}{t \text{ as } T \rightarrow u \text{ as } T} \qquad v \text{ as } T \rightarrow v$$

- Notice how we seem to always have **congruence rules**, that guide where reduction takes place, and rules that describe the “real” action

Ascription

- Why ascription?

Ascription

- Why ascription?
- Form of documentation
- Finding type errors
- Abstraction (cf. subtyping)
- Aiding type inference

Pairs

- New syntax: $t ::= \dots \{t, t\} \mid t.1 \mid t.2$
- Typing of pairs:

$$\frac{\Gamma \vdash t : T \quad \Gamma \vdash u : U}{\Gamma \vdash \{t, u\} : T \times U}$$

$$\frac{\Gamma \vdash t : T \times U}{\Gamma \vdash t.1 : T}$$

$$\frac{\Gamma \vdash t : T \times U}{\Gamma \vdash t.2 : U}$$

- Evaluation rules:

$$\{v_1, v_2\}.1 \rightarrow v_1$$

$$\{v_1, v_2\}.2 \rightarrow v_2$$

$$\frac{t \rightarrow t'}{\{t, u\} \rightarrow \{t', u\}}$$

$$\frac{u \rightarrow u'}{\{v, u\} \rightarrow \{v, u'\}}$$

$$\frac{t \rightarrow t'}{t.1 \rightarrow t'.1}$$

$$\frac{t \rightarrow t'}{t.2 \rightarrow t'.2}$$

Records

- Pairs generalize easily to tuples (try this at home)
- ... and tuples to records
- We use the syntax:

```
{age=44, name="Smith"}           // record value  
{age=44, name="Smith"}.name    // field access
```

and write the types as:

```
{age=44, name="Smith"} : {age:Int, name:String}
```

- Of course we haven't defined `Ints` or `Strings`, but we can just assume more base types

Records

- New syntax: $t ::= \dots \{l_i = t_i^{i \in 1 \dots n}\} \mid t.l$
- New values: $v ::= \dots \{l_i = v_i^{i \in 1 \dots n}\}$
- New types: $T ::= \dots \{l_i : T_i^{i \in 1 \dots n}\}$
- Typing of records:

$$\frac{\text{for each } i, \Gamma \vdash t_i : T_i}{\Gamma \vdash \{l_i = t_i^{i \in 1 \dots n}\} : \{l_i : T_i^{i \in 1 \dots n}\}}$$

$$\frac{\Gamma \vdash t : \{l_i : T_i^{i \in 1 \dots n}\}}{\Gamma \vdash t.l_j : T_j}$$

- Evaluation rules:

$$\{l_i = v_i^{i \in 1 \dots n}\}.l_j \rightarrow v_j$$

$$\frac{t \rightarrow t'}{t.l \rightarrow t'.l}$$

$$\frac{t_j \rightarrow t_j'}{\{l_i = v_i^{i \in 1 \dots j-1}, l_j = t_j, l_k = t_k^{k \in j+1 \dots n}\} \rightarrow \{l_i = v_i^{i \in 1 \dots j-1}, l_j = t_j', l_k = t_k^{k \in j+1 \dots n}\}}$$

Summary

- We defined the operational semantics of a simple language
- and its type system as a formal proof system
- and established type safety
- We added a bunch of extensions, and saw how those can be modeled in the proof system
 - Extend the safety proofs at home (well, maybe just one extension)
- All rules have been **syntax directed**, so it is easy to see how the formalization gives rise to a type checker program
 - more expressive the system, more difficult this will be
 - in many practical languages, typing is undecidable