Applied Functional Programming

USCS 2016

Johan Jeuring and Andres Löh (author)

Well-Typed

July 4-15, 2016

B2. Typed lambda calculus

Terms and Types

Terms

$e ::= x$		variables
	$\mid$ $e\ e$	application
	$\mid$ $\lambda(x : \tau)\ .\ e$	lambda abstraction

The only new thing is that lambda abstractions are annotated with a type.

Types

$\tau ::= \alpha$		type variables
	$\mid$ $\tau \to \tau$	function type

Types and free variables

Question

How do we assign a type to a term with free variables?

$\lambda(x : \mathsf{Nat})\ .\ \mathsf{plus}\ x\ \mathsf{one}$

Answer

We cannot unless we know the types of the free variables.

Environments

We therefore do not assign types to terms, but types to terms in a certain **environment** (also called **context**).

Environments

$$\begin{array}{lll} \Gamma ::= \varepsilon & & \text{empty environment} \\ \quad \mid & \Gamma, x : \tau & \text{binding} \end{array}$$

Later bindings for a variable always shadow earlier bindings.

The typing relation

A statement of the form

$$\Gamma \vdash e : \tau$$

can be read as follows:

"In environment Γ, term e has type τ."

Note that $\Gamma \vdash e : \tau$ is formally a ternary **relation** between an environment, a term and a type.

The $\vdash$ (called turnstile) and the colon are just notation for making the relation look nice but carry no meaning. We could have chosen the notation $T(\Gamma, e, \tau)$ for the relation as well, but $\Gamma \vdash e : \tau$ is commonly used.

Type rules

The relation is defined inductively in **natural deduction** style, using **inference rules**.

Variables

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

Above the bar are the **premises**.

Below the bar is the **conclusion**.

If the premises hold, we can infer the conclusion.

Type rules – contd.

Application

$$\frac{\Gamma \vdash e_1 : \tau_1 \to \tau_2 \qquad \Gamma \vdash e_2 : \tau_1}{\Gamma \vdash (e_1\ e_2) : \tau_2}$$

Variables scope over the entire inference rule. Multiple occurences of the same variable must be instantiated with the same expressions.

Type rules – contd.

Abstraction

$$\frac{\Gamma, x : \tau_1 \vdash e : \tau_2}{\Gamma \vdash \lambda(x : \tau_1) \,.\, e : \tau_1 \to \tau_2}$$

It pays off that we introduced the environments. The body of a lambda abstraction (viewed in isolation) can contain free occurrences of the bound variable.

Examples

$$\dfrac{\dfrac{x : \mathsf{Nat} \in x : \mathsf{Nat}}{x : \mathsf{Nat} \vdash x : \mathsf{Nat}}}{\varepsilon \vdash \lambda x : \mathsf{Nat} \,.\, x : \mathsf{Nat} \to \mathsf{Nat}}$$

Multiple applications of type rules can be stacked, leading (in general) to **proof trees** or **derivation trees**.

Examples – contd.

Let Γ abbreviate $\mathsf{neg} : \mathsf{Nat} \to \mathsf{Nat}, \mathsf{one} : \mathsf{Nat}$.

$$\dfrac{\dfrac{\mathsf{neg} : \mathsf{Nat} \to \mathsf{Nat} \in \Gamma}{\Gamma \vdash \mathsf{neg} : \mathsf{Nat} \to \mathsf{Nat}} \quad \dfrac{\mathsf{one} : \mathsf{Nat} \in \Gamma}{\Gamma \vdash \mathsf{one} : \mathsf{Nat}}}{\Gamma \vdash \mathsf{neg\ one} : \mathsf{Nat}}$$

Examples – contd.

Let Γ abbreviate $\text{plus} : \text{Nat} \to \text{Nat} \to \text{Nat}, \text{one} : \text{Nat}$.

Exercise

Try to complete the following proof tree:

$$\frac{\ldots}{\Gamma \vdash \lambda(x : \text{Nat}) \,.\, \text{plus}\ x\ \text{one} : \text{Nat} \to \text{Nat}}$$

Adding types

The simply typed lambda calculus itself knows only type variables and function types.

We can simulate constants of certain types by placing them in the environment.

We can also actually extend the calculus with new types. Then we can also add reduction rules.

Booleans

For simplicity, let us play the addition of a new type through with Booleans.

From the Monday lecture, we already know what syntactic constructs we need to add for Booleans:

- the **datatype** `Bool`
- the **constructors** `True` and `False`
- the **eliminator** **`if`** - **`then`** - **`else`**
- reduction rules

Now, we also add type rules.

Boolean syntax

Terms

e ::=	...	(as before)
	True	constructor
|	False	constructor
|	**if** e **then** e **else** e	eliminator

Types

τ ::=	...	(as before)
|	Bool	type of Booleans

Boolean reduction rules

$$\textbf{if}\ \text{True}\ \ \textbf{then}\ e_1\ \textbf{else}\ e_2 \rightsquigarrow e_1$$
$$\textbf{if}\ \text{False}\ \textbf{then}\ e_1\ \textbf{else}\ e_2 \rightsquigarrow e_2$$

Boolean type rules

$$\frac{}{\Gamma \vdash \mathsf{True} : \mathsf{Bool}} \qquad \frac{}{\Gamma \vdash \mathsf{False} : \mathsf{Bool}}$$

$$\frac{\Gamma \vdash e_1 : \mathsf{Bool} \qquad \Gamma \vdash e_2 : \tau \qquad \Gamma \vdash e_3 : \tau}{\Gamma \vdash \textbf{if } e_1 \textbf{ then } e_2 \textbf{ else } e_3 : \tau}$$

Other types

For other types (such as natural numbers or tuples or lists), we can also extend the simply typed lambda calculus.

We can also add additional constructs such as **let** or **case** to come closer to a full programming language.

However, there are still a few fundamental shortcomings . . .

Lack of polymorphism

In the simply typed lambda lambda calculus, there are no polymorphic functions.

We cannot type a term like

let id $= \lambda$(x : ?) . x
in **if** id True **then** id 0 **else** 1

because there is no valid type we can choose for ? .

Neither Bool nor Nat nor any other single type works.

We need multiple variants of the identity function for different types.

Introducing polymorphism

Idea

Allow to abstract from types.

let id $= \lambda\langle a \rangle\ (x : a)\ .\ x$
in **if** id $\langle$Bool$\rangle$ True **then** id $\langle$Nat$\rangle$ 0 **else** 1

We use angle brackets to syntactically distinguish type abstraction and application from term abstraction and application.

Polymorphic types

A type abstraction is reflected in the type:

$$\lambda\langle a\rangle\ (x : a)\ .\ x : \forall a.a \to a$$

Note that this is quite different from Haskell:

- In Haskell, type abstraction, type application and polymorphism is implicit.
- In (standard) Haskell, there is a difference between **let** -bound variables (can be polymorphic) and lambda-bound variables (always monomorphic).

System F

The typed lambda calculus based on the idea of type abstraction and application is called **System F** or **polymorphic lambda calculus**.

It was discovered (independently) by Girard and Reynolds.

System F grammar

Terms

e ::=	x	variables
|	e e	application
|	$\lambda(\mathrm{x} : \tau)$. e	term abstraction
|	$\lambda\langle\alpha\rangle$. e	type abstraction
|	(e $\langle\tau\rangle$)	type application

Types

τ ::=	α	type variables
|	$\tau \to \tau$	function type
|	$\forall\alpha.\tau$	polymorphic type

System F reduction

$$((\lambda(x : \tau)\ .\ e_1)\ e_2) \rightsquigarrow e_1\ [x \mapsto e_2]$$
$$(\lambda\langle\alpha\rangle\ .\ e)\ \langle\tau\rangle \rightsquigarrow e\ [\alpha \mapsto \tau]$$

The notions of free and bound variables are extended to type variables. Note that term and type variables live in different worlds.

System F environments

Mainly in preparation for future extensions, we include type variables in the environment.

$$
\begin{array}{lll}
\Gamma ::= \varepsilon & & \text{empty environment} \\
\quad \mid & \Gamma, x : \tau & \text{term variable binding} \\
\quad \mid & \Gamma, \alpha & \text{type variable binding}
\end{array}
$$

If a type variable occurs free in an expression, it should be included in the environment as well – we write:

$$\Gamma \vdash \tau : *$$

System F type rules

Rules for variables, term application are as in the simply typed lambda calculus.

Term abstraction

$$\frac{\Gamma, x : \tau_1 \vdash e : \tau_2 \qquad \Gamma \vdash \tau_1 : *}{\Gamma \vdash \lambda(x : \tau_1) \,.\, e : \tau_1 \to \tau_2}$$

Just a minor change: the type must be well-formed.

System F type rules – contd.

Type abstraction

$$\frac{\Gamma, \alpha \vdash \mathsf{e} : \tau}{\Gamma \vdash \lambda\langle\alpha\rangle \,.\, \mathsf{e} : \forall\alpha.\tau}$$

Type application

$$\frac{\Gamma \vdash \mathsf{e} : \forall\alpha.\tau_1 \qquad \Gamma \vdash \tau_2 : *}{\Gamma \vdash \mathsf{e}\ \langle\tau_2\rangle : \tau_1\ [\alpha \mapsto \tau_2]}$$

Higher-rank polymorphism

System F is – in some aspects – even more general than (standard) Haskell.

Consider the Haskell program:

λf $\rightarrow$ (f True, f 'x')

This expression results in a type error.

Higher-rank polymorphism – contd.

$$\lambda f \to (f\ \mathsf{True}, f\ \texttt{'x'})$$

The function `f` is applied to a `Bool` and a `Char`, i.e., it must be polymorphic.

The following expression is type-correct in System F (enriched by Booleans, characters, and pairs):

$$\lambda f : \forall a.a \to a\ .\ f\ \langle \mathsf{Bool} \rangle\ \mathsf{True}, f\ \langle \mathsf{Char} \rangle\ \texttt{'x'}$$

The type is

$$(\forall a.a \to a) \to (\mathsf{Bool}, \mathsf{Char})$$

Higher-rank polymorphism – contd.

The type

$$(\forall a.a \to a) \to (\mathsf{Bool}, \mathsf{Char})$$

is called a **rank-2** type, because a function argument is polymorphic.

If a function argument of a function argument is polymorphic, you get a rank-3 type and so forth.

Haskell and higher-rank types

Haskell allows arbitrary-rank types if you enable

```
{-# LANGUAGE RankNTypes #-}
```

But they cannot be inferred:

```
rank2f :: (∀a.a → a) → (Bool, Char)
rank2f = λf → (f True, f 'x')
```

We may see uses of such types later during the course.

What is missing compared to Haskell?

Parameterized types

System F types can be polymorphic, but there is no general mechanism to abstract from type constructors such as lists.

Inference

System F is so explicit that it is not usable as a programming language. We need to look at how to infer types.

Type checking vs. Type inference

- A **type system** is a proof system that specified the well-typed programs.
- It does not automatically provide you with an **algorithm** to check if a given program is correct.
- If a proof system is **syntax directed**, you can, however, read off an algorithm.
- A **type checking** algorithm decides whether a given program is well-typed or not.
- Typed lambda calculi are typically **explicitly typed**. User-level programming languages have some type information **implicit**.
- A **type inference** algorithm tries to reconstruct type information such that a well-typed explicitly typed program is produced.