Polymorphism

Dzmitry Sledneu

April 25, 2012

Motivation

Identity function

$$idNat = \lambda x : Nat.\ x$$

$$idBool = \lambda x : Bool.\ x$$

Abstraction principle

Each significant piece of functionality in a program should be implemented in just one place.
Abstracting out varying parts (varying parts are the types).

Polymorphism

Definition

Functions which can be applied to arguments of many types are called polymorphic (poly = many, morph = form).

Forms of polymorphism

- Parametric or universal polymorphism (generic types): The ability to instantiate type variables.
- Inclusion or subtype polymorphism: The ability to treat a value of subtype as a value of one of its supertypes.
- Ad-hoc polymorphism or overloading: The ability to define several versions of the same function name, with different types.

Universal polymorphism

Two forms:

1. Explicit or predicative (e.g. let-polymorphism): Type T containing a type variable X may not be used in such a way that X is instantiated to a polymorphic type.
2. Implicit or impredicative (e.g. System F): Type variable X in type T can be instantiated to any type (including T itself).

Let bindings

Definition

let $x = t_1$ in $t_2 \overset{def}{=} (\lambda x : T_1.\ t_2)\ t_1$

(Evaluate the expression t_1 and bind the name x to the resulting value while evaluating t_2).

$$\frac{\Gamma \vdash t_1 : T_1 \quad \Gamma, x : T_1 \vdash t_2 : T_2}{\Gamma \vdash \text{ let } x = t_1 \text{ in } t_2 : T_2} \quad \text{(T-LET)}$$

Identity function

This works:

$$\text{let } idNat = \lambda x : Nat \to Nat.\ x \text{ in}$$
$$\text{let } idBool = \lambda x : Bool \to Bool.\ x \text{ in}$$
$$\text{let } a = idNat\ 1 \text{ in}$$
$$\text{let } b = idBool\ \ True$$

This doesn't:

$$\text{let } id = \lambda x : X.\ x \text{ in}$$
$$\text{let } a = id\ 1 \text{ in}$$
$$\text{let } b = id\ \ True$$

Let-polymorphism

Associate a different variable X with each use of *id*:

$$\frac{\Gamma \vdash t_1 : T_1 \quad \Gamma \vdash [x \mapsto t_1]t_2 : T_2}{\Gamma \vdash \text{ let } x = t_1 \text{ in } t_2 : T_2} \quad \text{(T-LETPOLY)}$$

$$\frac{\Gamma \vdash [x \mapsto t_1]t_2 : T_2 \mid_{\mathcal{X}} \mathcal{C}}{\Gamma \vdash \text{ let } x = t_1 \text{ in } t_2 : T_2 \mid_{\mathcal{X}} \mathcal{C}} \quad \text{(CT-LETPOLY)}$$

$$\text{let } x = t_1 \text{ in } t_2 \rightarrow [x \mapsto t_1]t_2 \quad \text{(E-LET)}$$

Now this works:

let $id = \lambda x.\ x$ in

let $a = id\ 1$ in

let $b = id\ True$

System F

New form of abstraction:

$$\lambda X.\ t$$

New form of application:

$$t[T]$$

New reduction rules:

$$(\lambda X.\ t_{12})[T_2] \rightarrow [X \mapsto T_2]t_{12} \quad \text{(E-TAPPTABS)}$$

$$\frac{t_1 \rightarrow t_1'}{t_1[T_2] \rightarrow t_1'[T_2]} \quad \text{(E-TAPP)}$$

New typing rules

$$\frac{\Gamma, X \vdash t_2 : T_2}{\Gamma \vdash \lambda X.\ t_2 : \forall X. T_2} \quad \text{(T-TABS)}$$

$$\frac{\Gamma \vdash t_1 : \forall X. T_{12}}{\Gamma \vdash t_1[T_2] : [X \mapsto T_2]T_{12}} \quad \text{(T-TAPP)}$$

Example

Identity function

$$id = \lambda X.\ \lambda x : X.\ x$$

Typing

$$id : \forall X.\ X \to X$$

$$id[Nat] : Nat \to Nat$$

$$id[Bool] : Bool \to Bool$$

Evaluation

$$id[Nat]\ 0 \to 0$$

$$id[Bool]\ True \to True$$

Basic properties

Theorem (Preservation)

If $\Gamma \vdash t : T$ and $t \rightarrow t'$, then $\Gamma \vdash t' : T$.

Proof.

Left as an exercise. □

Theorem (Progress)

If t is a closed, well-typed term, then either t is a value or else there is some t' with $t \rightarrow t'$.

Proof.

Left as an exercise. □

Theorem (Strong normalization)

Every reduction path starting from a well-typed term is guaranteed to terminate.

Type erasure

Definition

$$
\begin{aligned}
erase(x) &= x \\
erase(\lambda x : T_1.\ t_2) &= \lambda x.\ erase(t_2) \\
erase(t_1\ t_2) &= erase(t_1)\ erase(t_2) \\
erase(\lambda X.\ t_2) &= erase(t_2) \\
erase(t_1[T_2]) &= erase(t_1)
\end{aligned}
$$

(Erase all type annotations).

Partial erasure

Definition

$$
\begin{aligned}
erase_p(x) &= x \\
erase_p(\lambda x : T_1.\ t_2) &= \lambda x : T_1.\ erase_p(t_2) \\
erase_p(t_1\ t_2) &= erase_p(t_1)\ erase_p(t_2) \\
erase_p(\lambda X.\ t_2) &= \lambda X.\ erase_p(t_2) \\
erase_p(t_1[T_2]) &= erase_p(t_1)[]
\end{aligned}
$$

(Erase all type applications arguments).

Type reconstruction undecidability

Type reconstruction

A term y in the untyped lambda-calculus is typable in System F if there is some well-typed term x such that $erase(x) = y$.

Theorem (Wells, 1994)

It is undecidable whether, given a closed term y of the untyped lambda-calculus, there is some well-typed term x in System F such that $erase(x) = y$.

Theorem (Boehm, 1985)

It is undecidable whether, given a closed term y in which type applications are marked but the arguments are omitted, there is some well-typed System F term x such that $erase_p(x) = y$.

THE END