System F: A Calculus for Modeling Generics

Syntax:

$$\begin{array}{rcl} T & ::= & \ldots \mid \alpha \mid \forall \alpha.\ T \\ e & ::= & x \mid \lambda x : T.\ e \mid (e\ e) \\ & & \Lambda \alpha.\ e \mid e[T] \end{array}$$

Evaluation contexts:

$$E ::= \ldots \mid E[T]$$

Reduction rules:

$$(\Lambda \alpha.\ e)[T] \longrightarrow [\alpha \mapsto T]e$$

Type rules:

$$\frac{\Gamma, \alpha \vdash e : T}{\Gamma \vdash \Lambda \alpha.\ e : \forall \alpha.\ T} \qquad \frac{\Gamma \vdash e : \forall \alpha.\ T_1 \quad \Gamma \vdash T_2\ ok}{\Gamma \vdash e[T_2] : [\alpha \mapsto T_2]T_1}$$

$$\frac{\Gamma \vdash T_1\ ok \quad \Gamma, x : T_1 \vdash e : T_2}{\Gamma \vdash \lambda x : T_1.\ e : T_1 \to T_2}$$

Well-formed Types

$$\frac{\alpha \in \Gamma}{\Gamma \vdash \alpha \ ok} \qquad \frac{\Gamma, \alpha \vdash T \ ok}{\Gamma \vdash \forall \alpha.\ T \ ok}$$

$$\frac{\Gamma \vdash T_1 \ ok \quad \Gamma \vdash T_2 \ ok}{\Gamma \vdash T_1 \to T_2 \ ok}$$

Examples

id $= \Lambda\alpha.\ \lambda$x:$\alpha$. x
$>$ id : $\forall\alpha.\ \alpha \to \alpha$

id[int]
$>$ (λ x:int. x) : int $\to$ int

id[int](2)
$>$ 2 : int

id[bool]
$>$ (λ x:bool. x) : bool $\to$ bool

double $= \Lambda\alpha.\ \lambda$f: $\alpha \to \alpha$. λa: α. f (f a)
$>$ double : $\forall\alpha.\ (\alpha \to \alpha) \to \alpha \to \alpha$

double[int] succ 2
$>$ 4 : int

Polymorphic Functions on Lists

Suppose that operations on lists have the following types:

nil : $\forall$a. list a
cons : $\forall$a. a $\rightarrow$ list a $\rightarrow$ list a
isnil : $\forall$a. list a $\rightarrow$ bool
hd : $\forall$a. list a $\rightarrow$ a
tl : $\forall$a. list a $\rightarrow$ list a

The map function applies a function to each element in a list, creating a new list.

```
map : ∀a. ∀b. (a → b) → list a → list b
map = Λa. Λb. λf : a→ b.
  (fix (λ m: (list a) → (list b).
    λls : list a.
      if isnil[a] ls then
        nil[b]
      else
        cons[b](f (head[a] ls), m (tail [a] ls))))
```

Type Safety

Theorem

Preservation *If* $\Gamma \vdash e : T$ *and* $e \longrightarrow e'$ *then* $\Gamma \vdash e' : T$.

Theorem

Progress *If* $\emptyset \vdash e : T$ *then either* e *is a value or there is some* e' *such that* $e \longrightarrow e'$.

Theorem

Normalization *If* $\emptyset \vdash e : T$ *then for some* v, $e \longrightarrow^* v$ *and* v *is a value.*

Erasure, Typability, Type Inference

$$
\begin{aligned}
\mathit{erase}(x) &= x \\
\mathit{erase}(\lambda x : T.\ e) &= \lambda x.\ \mathit{erase}(e) \\
\mathit{erase}(e_1\ e_2) &= (\mathit{erase}(e_1)\ \mathit{erase}(e_2)) \\
\mathit{erase}(\Lambda\alpha.\ e) &= \lambda_{\cdot}\ \mathit{erase}(e) \\
\mathit{erase}(e[T]) &= (\mathit{erase}(e)\ \mathit{unit})
\end{aligned}
$$

Theorem

Wells It is undecidable whether, for a given closed term e in the untyped lambda calculus, there is some term e' in System F such that $\mathit{erase}(e') = e$.

Parametricity

- The type of a polymorphic function constrains the kinds of behavior that function.
- For example, the following is the *only* function that has type $\forall\alpha.\ \alpha \to \alpha$:

$$\Lambda\alpha.\ \lambda x{:}\alpha.\ x$$

- Can you think of all the functions that implement the type $\forall\alpha.(\alpha \to \alpha) \to \alpha \to \alpha$?
- Can you think of all the functions that implement $\forall\alpha\beta.\ (\alpha \to \beta) \to \mathit{list}\ \alpha \to \mathit{list}\ \beta$.

Type passing versus type erasure

- The run-time semantics presented earlier is called "type passing" because type application (e.g., $(\Lambda\alpha.\lambda x : \alpha.\ x)[\mathit{int}]$) substites a type for a type variable (producing $\lambda x : \mathit{int}.\ x$).
- However, the types play no important role during run-time, so performing this substitution is useless work.
- Alternatively, we can erase types and just evaluate programs using the run-time semantics of the untyped lambda calculus.
- However, to preserve evaluation order, we have to be careful with the erasure of type abstraction and type application.

$$\mathit{erase}(\Lambda\alpha.\ e) = \lambda\_.\ \mathit{erase}(e)$$
$$\mathit{erase}(e[T]) = (\mathit{erase}(e)\ \mathit{unit})$$

First-class Polymorphism

- System F is an example of a language with first-class polymorphism.
- Polymorphic values can be passed as arguments to functions, returned from functions, and stored in memory (e.g., as an element of a list).
- First-class polymorphism is rather unusual in programming languages. C++ doesn't have it, ML doesn't have it, Java doesn't have it.
- The following is a simple example of a function that uses first-class polymorphism:

$$\Lambda\alpha.\ \lambda \mathsf{f} : \forall\alpha.\ \alpha \to \alpha.$$
$$\quad (\mathsf{f}[\mathsf{int}]\ 1,\ \mathsf{f}[\mathsf{bool}]\ \mathsf{true})$$

Compiling Generics

- There are two common approaches to compiling generics:
  1. One uniform implementation with boxed representations.
  2. Type-specialized implementations generated for each instantiation.
- With first-class polymorphism, if you want to use type-specialization, you have to do run-time code generation. (Though whole-program analyses can reduce the need for this.)