System F: A Calculus for Modeling Generics

Syntax:

\[
T ::= \ldots \mid \alpha \mid \forall \alpha. \ T
\]

\[
e ::= x \mid \lambda x : T. \ e \mid (e \ e) \mid \Lambda \alpha. \ e \mid e[T]
\]

Evaluation contexts:

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

Reduction rules:

\[
(\Lambda \alpha. \ e)[T] \rightarrow [\alpha \mapsto T]e
\]

Type rules:

\[
\frac{\Gamma, \alpha \vdash e : T}{\Gamma \vdash \Lambda \alpha. \ e : \forall \alpha. \ T}
\]

\[
\frac{\Gamma \vdash e : \forall \alpha. \ T_1 \quad \Gamma \vdash T_2 \ ok}{\Gamma \vdash e[T_2] : [\alpha \mapsto T_2]T_1}
\]

\[
\frac{\Gamma \vdash T_1 \ ok \quad \Gamma, x : T_1 \vdash e : T_2}{\Gamma \vdash \lambda x : T_1. \ e : T_1 \rightarrow T_2}
\]
Well-formed Types

$$\alpha \in \Gamma \quad \Rightarrow \quad \Gamma \vdash \alpha \ ok$$

$$\Gamma, \alpha \vdash T \ ok \quad \Rightarrow \quad \Gamma \vdash \forall \alpha. \ T \ ok$$

$$\Gamma \vdash T_1 \ ok \quad \Gamma \vdash T_2 \ ok \quad \Rightarrow \quad \Gamma \vdash T_1 \rightarrow \ T_2 \ ok$$
Examples

\[ \text{id} = \Lambda \alpha. \lambda x: \alpha. \ x \]
\[ \text{id} : \forall \alpha. \ \alpha \rightarrow \alpha \]

\[ \text{id}[\text{int}] \]
\[ (\lambda x: \text{int}. \ x) : \text{int} \rightarrow \text{int} \]

\[ \text{id}[\text{int}](2) \]
\[ 2 : \text{int} \]

\[ \text{id}[\text{bool}] \]
\[ (\lambda x: \text{bool}. \ x) : \text{bool} \rightarrow \text{bool} \]

\[ \text{double} = \Lambda \alpha. \ \lambda f: \alpha \rightarrow \alpha. \ \lambda a: \alpha. \ f (f \ a) \]
\[ \text{double} : \forall \alpha. \ (\alpha \rightarrow \alpha) \rightarrow \alpha \rightarrow \alpha \]

\[ \text{double}[\text{int}] \ \text{succ} \ 2 \]
\[ 4 : \text{int} \]
Suppose that operations on lists have the following types:

- `nil : ∀a. list a`
- `cons : ∀a. a → list a → list a`
- `isnil : ∀a. list a → bool`
- `hd : ∀a. list a → a`
- `tl : ∀a. list a → 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 $e \rightarrow e'$ 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 $e \rightarrow e'$.

Theorem

*Normalization* If $\emptyset \vdash e : T$ then for some $v$, $e \rightarrow^* v$ and $v$ is a value.
Erasure, Typability, Type Inference

\[
erase(x) = x \\
erase(\lambda x : T. e) = \lambda x. \ erase(e) \\
erase(e_1 \ e_2) = (\ erase(e_1) \ erase(e_2)) \\
erase(\Lambda \alpha. e) = \lambda. \ erase(e) \\
erase(e[T]) = (\ erase(e) \ unit)
\]

**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 \(\ erase(e') = e\).
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 $\forall \alpha. \alpha \rightarrow \alpha$:

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

- Can you think of all the functions that implement the type $\forall \alpha. (\alpha \rightarrow \alpha) \rightarrow \alpha \rightarrow \alpha$?

- Can you think of all the functions that implement $\forall \alpha \beta. (\alpha \rightarrow \beta) \rightarrow \text{list } \alpha \rightarrow \text{list } \beta$.?
The run-time semantics presented earlier is called “type passing” because type application (e.g., \((\Lambda \alpha. \lambda x : \alpha. x)[\text{int}])\) substitutes a type for a type variable (producing \(\lambda x : \text{int}. x\)).

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.

\[
\begin{align*}
erase(\Lambda \alpha. e) &= \lambda _. erase(e) \\
erase(e[T]) &= (erase(e) \text{ unit})
\end{align*}
\]
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:

\[
\Lambda \alpha. \lambda f: \forall \alpha. \alpha \to \alpha.
\]

(f[int] 1, f[bool] 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.)