System F and Parametricity

Stefan Muller

CS 534: Types and Programming Languages, Spring 2024
Lecture 18

1 Parametricity

Q: How many distinct[1] values are there of type $\forall\alpha.\alpha \rightarrow \alpha$?
A: Just one (id from last class).

Intuitively, *parametricity* is the idea that, in a function $\Lambda\alpha.e$, e can't "inspect" α and can't do something different depending on what α is. For a function $\Lambda\alpha.\lambda x : \alpha.e$, this means that e can't do anything with x other than pass it around and (in this case) return it. It can't add to it because it doesn't know it's int, can't project out of it, etc.

How many values are there of type:

$$\begin{array}{ll} \forall\alpha.\forall\beta.(\alpha \times \beta) \rightarrow (\beta \times \alpha)? & 1 \\ \forall\alpha.(\alpha \times \alpha) \rightarrow (\alpha \times \alpha)? & 4 \\ \forall\alpha.\forall\beta.(\alpha \times \beta) \rightarrow (\beta + \alpha)? & 2 \\ \forall\alpha.\alpha? & 0 \end{array}$$

This all falls out of a result called the Parametricity Theorem (which we won't prove or even state formally in this class; this is just a taste). These results for particular types are dubbed "theorems for free", a term coined by Phil Wadler.

2 System F

We saw last time the power of adding parametricity to STLC It turns out that we don't actually lose any power by *only* having these constructs and normal functions. This language is called System F, and it was discovered (in slightly different variations) by Jean-Yves Girard and John Reynolds.

$$\begin{array}{lll} \tau & ::= & \alpha \mid \tau \rightarrow \tau \mid \forall\alpha.\tau \\ e & ::= & x \mid \lambda x : \tau.e \mid \Lambda\alpha.e \mid e[\tau] \end{array}$$

Remember our definition of Booleans in the untyped lambda calculus:

$$\begin{array}{rcl} \textsf{if } e_1 \textsf{ then } e_2 \textsf{ else } e_3 & \triangleq & e_1\ e_2\ e_3 \\ \textsf{true} & \triangleq & \lambda x.\lambda y.x \\ \textsf{false} & \triangleq & \lambda x.\lambda y.x \end{array}$$

If we add types, what then is the type corresponding to Booleans? It's clearly something of the form $\tau \rightarrow \tau \rightarrow \tau$, but what's τ? We don't know. It's the type of the branches of your conditional. If we pick one, then these Booleans are only useful in situations where that's the type you want for the branches of the conditional. But now with polymorphism, we can generalize over all such uses. The key to defining STLC constructs in System F is thinking about what the "user" of such a construct looks like and then parameterizing over all "users."

[1]By this we mean not *observationally equivalent*, where observational equivalence ($\cong$) for functions means (roughly) that $\lambda x.e_1 \cong \lambda x.e_2$ if for all e, $(\lambda x.e_1)\ e$ and $(\lambda x.e_2)\ e$ evaluate to the same value (or observationally equivalent functions).

$$
\begin{aligned}
\mathsf{unit} &\triangleq \forall\alpha.\alpha \to \alpha \\
() &\triangleq \Lambda\alpha.\lambda x:\alpha.x \\
\mathsf{void} &\triangleq \forall\alpha.\alpha \\
\mathsf{abort}_\tau\ e &\triangleq e[\tau] \\
\tau_1 \times \tau_2 &\triangleq \forall\alpha.(\tau_1 \to \tau_2 \to \alpha) \to \alpha \\
(e_1, e_2) &\triangleq \Lambda\alpha.\lambda s:\tau_1 \to \tau_2 \to \alpha.s\ e_1\ e_2 \\
\mathsf{fst}\ e &\triangleq e[\tau]\ (\lambda x:\tau_1.\lambda y:\tau_2.x) \\
\mathsf{snd}\ e &\triangleq e[\tau]\ (\lambda x:\tau_1.\lambda y:\tau_2.y) \\
\mathsf{bool} &\triangleq \forall\alpha.\alpha \to \alpha \to \alpha \\
\mathsf{if}\ e_1\ \mathsf{then}\ e_2\ \mathsf{else}\ e_3 &\triangleq e_1[\tau]\ e_2\ e_3 \\
\mathsf{true} &\triangleq \Lambda\alpha.\lambda x:\alpha.\lambda y:\alpha.x \\
\mathsf{false} &\triangleq \Lambda\alpha.\lambda x:\alpha.\lambda y:\alpha.y \\
\tau_1 + \tau_2 &\triangleq \forall\alpha.(\tau_1 \to \alpha) \to (\tau_2 \to \alpha) \to \alpha \\
\mathsf{inl}\ e &\triangleq \Lambda\alpha.\lambda f:\tau_1 \to \alpha.\lambda g:\tau_2 \to \alpha.f\ e \\
\mathsf{inr}\ e &\triangleq \Lambda\alpha.\lambda f:\tau_1 \to \alpha.\lambda g:\tau_2 \to \alpha.g\ e \\
\mathsf{case}\ e\ \mathsf{of}\ \{x.e_1; y.e_2\} &\triangleq e_1[\tau]\ (\lambda x:\tau_1.e_1)\ (\lambda y:\tau_2.e_2)
\end{aligned}
$$