Assignment 5: Universal, Existential, and Recursive Types

15-312: Foundations of Programming Languages
Kevin Watkins ⟨kw@cmu.edu⟩

Out: Tuesday, March 15, 2005
Due: Tuesday, March 22, 2005 (10:30 a.m.)

50 points total

§ 1. Encodings in System F It turns out that (if we ignore side effects for a moment), there are encodings of all the type constructs we've seen so far in terms of just function ($\rightarrow$) and universal ($\forall$) types. This was first noticed by a logician, Jean-Yves Girard, in his 1972 Ph.D. thesis, where he proposed a pure language based on just these two type constructors, called System F. (Actually, his thesis has many different variations in it, some of which have other type constructors, and the language wasn't originally called System F, although everyone calls it that now.) Girard wanted to use his language to investigate certain questions in constructive logic, although he also pointed out that it could be thought of as a programming language. Girard was the first to prove that every well-typed program in System F terminates.

At nearly the same time, John Reynolds was investigating polymorphism in programming languages, and came up with a system identical to one of Girard's, which he published in 1974 in the paper "Towards a theory of type structure." In that paper Reynolds also investigated data abstraction and showed how polymorphic types could be used to enforce it.

The point of this question is to investigate some of these encodings of various type constructors in terms of functions and universal types. For our purposes, "System F" will mean the fragment of MinML with just the following type constructors:

$$\tau ::= \alpha \mid \tau_1 \rightarrow \tau_2 \mid \forall\alpha.\, \tau$$

and without any side effects or recursive functions. It turns out then that every well typed expression evaluates to a value in finitely many steps.

§ 1.1. Encoding Pairs Still working in System F, let's introduce the type $\tau_1 \times \tau_2$ by *defining* it to be

$$\tau_1 \times \tau_2 = \forall\alpha.\, (\tau_1 \rightarrow \tau_2 \rightarrow \alpha) \rightarrow \alpha$$

where α is some type variable not free in τ_1 or τ_2 (to avoid capture).

Think about what the values of type $\tau_1 \times \tau_2$ could be. Since it is a $\forall$-type, a value of this type must have the form $\Lambda\alpha.\, e$ where e is an expression of type $(\tau_1 \rightarrow \tau_2 \rightarrow \alpha) \rightarrow \alpha$ for *arbitrary* α. So the value of e has to be a function $\lambda x : (\tau_1 \rightarrow \tau_2 \rightarrow \alpha).\, e'$, where e' has type α. But the only way we have of making a value of type α is by applying x to a couple of arguments. So the value of e' must be of the form $x\, e_1\, e_2$, where e_1 has type τ_1 and e_2 has type τ_2. So inside any value of type $\tau_1 \times \tau_2$ there will be embedded expressions of type τ_1 and τ_2, which is why it acts like a product type.

Q 1.1. [5 pts] Write System F functions having the types

$$\begin{array}{rcl} \textbf{mkpair} & : & \forall\beta_1.\,\forall\beta_2.\,\beta_1 \to \beta_2 \to \beta_1 \times \beta_2 \\ \textbf{fst} & : & \forall\beta_1.\,\forall\beta_2.\,\beta_1 \times \beta_2 \to \beta_1 \\ \textbf{snd} & : & \forall\beta_1.\,\forall\beta_2.\,\beta_1 \times \beta_2 \to \beta_2 \end{array}$$

that implement the basic operations on these encoded pairs. The function **mkpair** should take a value of type β_1 and a value of type β_2 and return something in type $\beta_1 \times \beta_2$ that acts like a pair. (Remember that $\beta_1 \times \beta_2$ means the definition above!) The functions **fst** and **snd** should extract the components of a simulated pair.

Q 1.2. [5 pts] (a) We can make an analogy from pair types to unit types by thinking of pairs as the 2-ary and units as the 0-ary case of the same idea. Following this analogy, propose a definition for unit types

$$\textbf{unit} = ?$$

in System F.

(b) Your encoding of the unit type should contain *exactly* one System F value. What is it?

§ 1.2. Encoding Existential Types In lecture, we saw a similar encoding of existential types:

$$\exists\alpha.\,\tau = \forall\beta.\,(\forall\alpha.\,\tau \to \beta) \to \beta$$

where β is some type variable not free in τ (to avoid capture).

Q 1.3. [5 pts] Figure out and describe how to translate the constructs **pack** (τ', e) **as** $\exists\alpha.\,\tau$ and **unpack** $(\alpha, x) = e_1$ **in** e_2 from lecture into System F programs that have the same meaning, but use the *definition* of $\exists\alpha.\,\tau$ given above. That is, you should give definitions for **pack** and **unpack** that expand them into the constructs of System F. Your translation should have the property that if a **pack** or **unpack** is well-typed in MinML, its encoding into System F should be well-typed in System F and "do the same thing".

§ 1.3. A Mystery Encoding Consider the following definition:

$$\tau_1 \heartsuit \tau_2 = \forall\alpha.\,(\tau_1 \to \alpha) \to (\tau_2 \to \alpha) \to \alpha$$

where again α is not free in τ_1 or τ_2. This is an encoding into System F of a very familiar type constructor from MinML.

Q 1.4. [5 pts] Which MinML type constructor is it? For all of the MinML expression constructs involving this type (there are three of them), give encodings into System F in the style of Question 1.1.

Q 1.5. [extra credit] Discover which MinML types are represented by the following System F types:

$$\textbf{foo} = \forall\alpha.\,\alpha$$

$$\tau\ \textbf{bar} = \forall\alpha.\,\tau \to \alpha$$

§ 1.4. Simulation Theorem Actually proving that the encoding of MinML constructs into System F works is a bit tricky. For example, we might guess the following correctness theorem:

> Let e be a MinML expression involving product types, and let $\ulcorner e \urcorner$ be its translation into System F. Then if $e \mapsto e'$ in the M-machine for MinML, we have $\ulcorner e \urcorner \mapsto \ulcorner e' \urcorner$ in the M-machine for System F, where $\ulcorner e' \urcorner$ is the System F translation of e'.

Q 1.6. [5 pts] Find a counterexample to the supposed theorem.

§ 2. Existential and Recursive Types For this part of the assignment, we are going back to using MinML constructs and syntax, not System F.

Consider the following SML datatype for bitstrings:

```
datatype bits = Empty | Zero of bits | One of bits
```

The idea is that a bitstring such as `1011` will be represented as `One (Zero (One (One Empty)))` (so the representation is big-endian).

Q 2.1. [5 pts] Represent the `bits` datatype in MinML as a recursive (μ) type.

Q 2.2. [5 pts] Write a MinML function **or** : **bits** $\times$ **bits** $\rightarrow$ **bits** using your definition of **bits** as a recursive type. Your **or** function should compute the bitwise OR of its two arguments; for example, the bitwise OR of `11000` and `01010` is `11010`. If the two arguments to **or** do not have the same length, it should use MinML's **fail** operation to raise an exception.

Q 2.3. [5 pts] Now use MinML's existential type ($\exists$) to represent the `bits` datatype as an *abstract* type, exposing just the three constructors and the destructor. (The destructor is just an operation that lets you do case analysis on values of type **bits**.)

Q 2.4. [10 pts] Revise your **or** function from Question 2.2 to work with the abstract type instead of operating on the recursive type directly.