Recitation 7:
Existential Types and Partiality
15-312: Foundations of Programming Languages
Serena Wang, Charles Yuan
February 27, 2019

1 Existential Types in System FE

Existential types are the foundation of modularity. The main idea of modularity is to separate the client from the implementation. Let’s see whether adding existential types actually gives us the ability to express more than we could with just polymorphic types before. We can add existential types to System F using the primitives below, leading to System FE.

Typ \( \tau ::= \ldots \exists (t.\tau) \) existential type

Exp \( e ::= \ldots \)
pack\( \{t.\tau\}[\rho](e) \) existential pack
open\( \{t.\tau\}\{\tau_2\}(e_1; t, x.e_2) \) existential unpack

pack introduces an existential type, where \( \rho \) is the concrete implementation type that won’t be visible outside the package, and where \( e \) is the implementation of the existential type.

open eliminates an existential type by substituting \( e_1 \) for \( x \) in \( e_2 \). Here, \( e_1 \) is the packed-up library, which has some existential type, and \( \tau \) is the interface type, which uses \( t \) somewhere within it. \( x \) is the interface of the library that the client can use, and \( e_2 \) is the client’s code, which uses the library.

1.1 Statics

Remember that we now have a type context \( \Delta \) for our typing judgments, and a judgment for checking validity of types.

\[
\frac{\Delta, t \vdash \tau \text{ type}}{\Delta \vdash \exists (t.\tau) \text{ type}}
\]

\[
\frac{\Delta \vdash \rho \text{ type} \quad \Delta, t \vdash \tau \text{ type} \quad \Delta, \Gamma \vdash e : [\rho/t]\tau}{\Delta, \Gamma \vdash \text{pack}\{t.\tau\}[\rho](e) : \exists (t.\tau)}
\]

\[
\frac{\Delta, \Gamma \vdash e_1 : \exists (t.\tau) \quad \Delta, t \vdash \tau, x : \tau \vdash e_2 : \tau_2 \quad \Delta \vdash \tau_2 \text{ type}}{\Delta, \Gamma \vdash \text{open}\{t.\tau\}\{\tau_2\}(e_1; t, x.e_2) : \tau_2}
\]

As you can see in the statics rule for open, abstraction is enforced statically. The client code simply doesn’t have the implementation type in scope.
1.2 Dynamics

\[
\begin{align*}
&\text{pack}\{t.\tau\}|\rho|(e) \rightarrow \text{pack}\{t.\tau\}|\rho|(e') \\
&e \rightarrow e' \\
&\text{open}\{t.\tau\}\{e_1; t, x, e_2\} \rightarrow \text{open}\{t.\tau\}\{e'_1; t, x, e_2\} \\
&\text{open}\{t.\tau\}\{\text{pack}\{t.\tau\}|\rho|(e); t, x, e_2\} \rightarrow [\rho, e/t, x] e_2
\end{align*}
\]

The only that’s actually interesting is the last one, which tells us that there are no secrets at runtime. We get direct access to the implementation type, which we can use for whatever we want (i.e., optimizations). Thus, data abstraction is a compile-time discipline, and there is no boundary between the client and implementation at execution time. Using the protections of abstract data structures comes at zero cost to the program when it runs!

1.3 Examples with Queues

So how do we actually use existential types? Let’s look at how we would implement queues in System FE.

\[
\begin{align*}
\tau &\triangleq \langle \text{emp} \mapsto t, \text{enq} \mapsto (\text{nat} \times t) \rightarrow t, \text{deq} \mapsto t \rightarrow 1 + (\text{nat} \times t) \rangle \\
\rho &\triangleq \text{nat list} \\
\text{queue} &\triangleq \text{pack}\{t.\tau\}|\rho|(e)
\end{align*}
\]

The \(e\) that we use to define \text{queue} is below. We’ll use some syntax from SML in the example code below.

\[
\begin{align*}
&\text{enq} \rightarrow \lambda (x : \text{nat} \times (\text{nat list})) (x \cdot l) :: (x \cdot r) \\
&\text{deq} \rightarrow \lambda (q : \text{nat list}) \text{ case } \text{rev}(q) \{ [] \mapsto \text{none} \mid f :: qr \mapsto \text{some}((f, \text{rev}(qr))) \}
\end{align*}
\]

When we try to get the head of the queue, we can use \text{open} as in the code below. Note, however, that we cannot return \(x \cdot \text{deq}(q)\) since the thing we return must have extrinsic value.

\[
\begin{align*}
&\text{open}\{t.\tau\}\{\text{nat option}\}(\text{queue}; t, x). \\
&\text{let } q = x \cdot \text{enq}(\langle 7, x \cdot \text{enq}(\langle 5, x \cdot \text{enq}(\langle 2, x \cdot \text{emp} \rangle) \rangle)\rangle) \\
&\text{in case } x \cdot \text{deq}(q) \{ \text{some}(x) \mapsto \text{some}(x \cdot l) \mid \text{none} \mapsto \text{none} \}
\end{align*}
\]

2 Bisimulations

Bisimulations allow us to compare two implementations of an abstract type and see whether they are equivalent. To do so, we define a relation \(\mathcal{R}\) over expressions of the abstract type. This relation will essentially convert one of the implementation types into the other implementation type.
Suppose we have two implementations of queues $e_{ref}$ and $e_{cand}$. Let’s do some pattern-matching so that we can easily refer to each part of each implementation.

$$
e_{ref} = \langle \text{emp} \mapsto \text{emp}_{ref}, \text{enq} \mapsto \text{enq}_{ref}, \text{deq} \mapsto \text{deq}_{ref}\rangle$$

$$
e_{cand} = \langle \text{emp} \mapsto \text{emp}_{cand}, \text{enq} \mapsto \text{enq}_{cand}, \text{deq} \mapsto \text{deq}_{cand}\rangle$$

We want to show $e_{ref} \mathcal{R} e_{cand}$.

To show this, what we want to show is that $\mathcal{R}$ respects the operations of our existential type.

Recall that our existential type was this:

$$\exists (t. \langle \text{emp} \mapsto t, \text{enq} \mapsto (\text{nat} \times t) \rightarrow t, \text{deq} \mapsto t \rightarrow 1 + (\text{nat} \times t) \rangle)$$

Respecting the operations means that we want to “replace $t$ with $\mathcal{R}$ and prove the statements that result”:

$$
\text{emp}_{ref} \mathcal{R} \text{emp}_{cand}
\text{enq}_{ref} (\text{nat} \times \mathcal{R}) \rightarrow \mathcal{R} \text{enq}_{cand}
\text{deq}_{ref} \mathcal{R} \rightarrow (\text{nat} \times \mathcal{R}) \text{deq}_{cand}
$$

It’s not exactly obvious what these mean, so let’s write them out more elaborately. We call these our **proof obligations**:

1. $\text{emp}_{ref} \mathcal{R} \text{emp}_{cand}$
2. For all $n$,
   - Assume $q_{ref} \mathcal{R} q_{cand}$.
   - Prove $\text{enq}_{ref} (n) (q_{ref}) \mathcal{R} \text{enq}_{cand} (n) (q_{cand})$.
3. Assume $q_{ref} \mathcal{R} q_{cand}$. Want to show either:
   - $\text{deq}_{ref} (q_{ref}) \simeq \text{deq}_{cand} (q_{cand})$
   - $\text{deq}_{ref} (q_{ref}) \simeq \text{some} (\langle n, r_{ref} \rangle)$ and $\text{deq}_{cand} (q_{cand}) \simeq \text{some} (\langle n', r_{cand} \rangle)$ such that $n \simeq n'$ and $r_{ref} \mathcal{R} r_{cand}$.

So first we have to define our relation:

$$l \mathcal{R} \langle b, f \rangle \iff l \simeq b @ (\text{rev} f)$$

Now that we’ve defined our relation, showing everything remaining is just a matter of hand-waving our way through some proofs. Note that proofs of bisimulations in this class are a **rare exception** to our previous rules of formality!
Let’s put the two implementations here so we can refer back to them later:

$$e_{\text{ref}} \triangleq (\text{emp} \mapsto \text{[]} \),$$
$$\text{enq} \mapsto \lambda (x : \text{nat} \times \text{(nat list)})(x \cdot l :: (x \cdot r)$$
$$\text{deq} \mapsto \lambda (q : \text{nat list}) \text{case rev}(q) \{ [ ] \mapsto \text{none} \mid f :: qr \mapsto \text{some}((f, \text{rev}(qr))) \} \}$$

$$e_{\text{cand}} \triangleq (\text{emp} \mapsto ([], []),$$
$$\text{enq} \mapsto \lambda (x : \text{nat} \times \text{(nat list} \times \text{nat list)})(x \cdot l :: (x \cdot r \cdot l), x \cdot r \cdot r)$$
$$\text{deq} \mapsto \lambda (q : \text{nat list} \times \text{nat list})$$
$$\text{case}(x \cdot r)\{ [ ] \mapsto \text{case rev}(bs) \{ [ ] \mapsto \text{none} \mid b :: bs' \mapsto \text{some}((b, ([], bs'))) \}$$
$$\mid f :: fs' \mapsto \text{some}((f, (x \cdot l, fs'))) \}$$

And now let’s show that $R$ respects the relation:

1. $\text{emp}_{\text{ref}} R \text{emp}_{\text{cand}}$

$$\text{emp}_{\text{cand}} \cong [ ] \oplus (\text{rev} [ ])$$
$$\cong [ ] \oplus [ ]$$
$$\cong [ ]$$
$$\cong \text{emp}_{\text{ref}}$$

2. $\text{enq}_{\text{ref}} (n) (q_{\text{ref}}) R \text{enq}_{\text{cand}} (n) (q_{\text{cand}})$

Let $q_{\text{cand}} = \langle b_{\text{cand}}, f_{\text{cand}} \rangle$.
Assume $q_{\text{ref}} R q_{\text{cand}}$.
Thus, $q_{\text{ref}} \cong b_{\text{cand}} \oplus (\text{rev} f_{\text{cand}})$.

$$\text{enq}_{\text{cand}} (n) (q_{\text{cand}}) \cong (n :: b_{\text{cand}}) \oplus (\text{rev} f_{\text{cand}})$$
$$\cong n :: (b_{\text{cand}} \oplus \text{rev} f_{\text{cand}})$$
$$\cong n :: q_{\text{ref}}$$
$$\cong \text{enq}_{\text{ref}} (n) (q_{\text{ref}})$$

3. Assume $q_{\text{ref}} R q_{\text{cand}}$. Want to show either:

Let $q_{\text{cand}} = \langle b_{\text{cand}}, f_{\text{cand}} \rangle$.
Assume $q_{\text{ref}} R q_{\text{cand}}$.
Thus, $q_{\text{ref}} \cong b_{\text{cand}} \oplus (\text{rev} f_{\text{cand}})$.
There are 3 cases:

a) $q_{\text{ref}} = [ ], q_{\text{cand}} = [ [], [] ]$

$$\text{deq}_{\text{ref}} (q_{\text{ref}}) \cong \text{deq}_{\text{ref}} [ ]$$
$$\cong \ldots$$
$$\cong \text{none}$$
$$\text{deq}_{\text{cand}} (q_{\text{cand}}) \cong \text{deq}_{\text{ref}} ([ ], [ ])$$
$$\cong \ldots$$
$$\cong \text{none}$$
b) \( q_{\text{ref}} = n :: q'_{\text{ref}} \), \( q_{\text{cand}} = \langle b_{\text{cand}}, f :: f'_{\text{cand}} \rangle \)

\[
\begin{align*}
q_{\text{ref}} & \Rightarrow q_{\text{cand}} \\
R (b_{\text{cand}}, f :: f'_{\text{cand}}) & \equiv b_{\text{cand}} @ (\text{rev} f :: f'_{\text{cand}}) \\
& \equiv b_{\text{cand}} @ (\text{rev} f'_{\text{cand}}) @ [f] \\
\text{rev} q_{\text{ref}} & \equiv \text{rev}(b_{\text{cand}} @ (\text{rev} f'_{\text{cand}}) @ [f]) \\
& \equiv f :: (\text{rev}(b_{\text{cand}} @ (\text{rev} f'_{\text{cand}})))
\end{align*}
\]

Thus, \( \text{rev} q_{\text{ref}} \equiv f :: (\text{rev}(q'_{\text{ref}})) \), where \( q'_{\text{ref}} \equiv b_{\text{cand}} @ (\text{rev} f'_{\text{cand}}) \). Therefore, \( q'_{\text{ref}} \Rightarrow \langle b_{\text{cand}}, f :: f'_{\text{cand}} \rangle \).

\[
\begin{align*}
\text{deq}_{\text{cand}} (q_{\text{cand}}) & \equiv \text{deq}_{\text{cand}} \langle b_{\text{cand}}, f :: f'_{\text{cand}} \rangle \\
& \equiv \text{some}((f, \langle b_{\text{cand}}, f'_{\text{cand}} \rangle)) \\
& \equiv \text{none} \\
\text{deq}_{\text{ref}} (q_{\text{ref}}) & \equiv \text{case} \text{rev}(q_{\text{ref}}) [[] \mapsto \text{none} | f :: qr \mapsto \text{some}(\langle f, \text{rev}(qr) \rangle)) \\
& \equiv \text{some}((f, \text{rev}(\text{rev}(q'_{\text{ref}})))) \\
& \equiv \text{some}((f, q'_{\text{ref}}))
\end{align*}
\]

c) \( q_{\text{cand}} \equiv \langle n :: b_{\text{cand}}, [] \rangle \)

This proof is just as tedious and equally doable as the previous case with hand waving.

Since this example was simple, we were able to do everything in symbols, with only a few assumptions (like associativity, reversing lists, etc.).

For your homework, you may not be able to formalize your bisimulation proofs all that rigorously. You’ll probably have a paragraph of prose for proof obligation.

3 Definability in System F

We introduced the new language System \( \text{FE} \) to add existential types. But is it really necessary to bake in existentials as part of the core language? Could universal types be sufficient to convey the data abstraction that defines an existential type?

What does it mean to have a value of the type \( \exists (t.\tau) \)? It means that we have some type \( \tau \) that uses the representation type \( t \), such that we may perform computation using \( \tau \) in a way that is agnostic of the true identity of \( t \). Suppose we have such a representation \( t \). Such a computation might then look like \( \tau \rightarrow \tau_2 \) where \( t \) is not allowed to appear in \( \tau_2 \).

Think about it this way: we defined two alternative queue implementations \( e_{\text{ref}} \) and \( e_{\text{cand}} \). Let’s denote their representation types (recall, one list and two lists, respectively) as \( \rho_{\text{ref}} \) and \( \rho_{\text{cand}} \). A client may use the queue data structure to perform computation, but cannot return a value containing type \( \rho_{\text{ref}} \) or \( \rho_{\text{cand}} \) explicitly. That would break the data abstraction, as somehow the user would now be able to distinguish the two queue implementations from each other!

So what we really want is to say, for all representation types, a client may use the interface functions to perform a computation, then must discard the actual representation type in their output. That seems like a job for universal types...
Let’s try it out. Suppose the client has a type $u$ over which they need to do computation using the abstract data type. That abstract type has type $\exists(t.\tau)$, so we can say that from $\tau$ we wish to derive $u$, and we want to be able to do this over all representation types $t$. That’s $\forall(t.\tau \to u)$.

Only one step remains. An existential type, in turn, cannot know about its client, so we must quantify over $u$. The result is this definition of existential types:

$$\exists(t.\tau) \triangleq \forall(u.\forall(t.\tau \to u) \to u)$$

How do we prevent the type variable $t$ from appearing in $u$? Observe that in the final occurrence of $u$, $t$ is not in scope at all! That’s the beauty of this encoding of existential types. The fact that the representation type cannot be referred to in the output of the computation is enforced by the type itself. It’s impossible to write a type that violates the data abstraction!

And finally, we wish to derive the definitions of $\text{pack}$ and $\text{open}$. Based on the definition of the existential type, defining these two is pretty much a game of matching up types. The definitions are:

$$\text{pack}\{t.\tau\}[\rho](e) \triangleq \Lambda(u) \lambda(x : \forall(t.\tau \to u)) x[\rho](e)$$

$$\text{open}\{t.\tau\}\{\tau_2\}(e_1 ; t, x.e_2) \triangleq e_1[\tau_2](\Lambda(t) \lambda(x : \tau) e_2)$$

Check that these definitions have the correct types, and align with our understanding of data abstraction. Congrats! You’ve successfully shown that $\text{FE}$ is only as strong as $\text{F}$. In practice explicit existentials are convenient, so we’ll keep them around, but this is another testament to the power of pure System $\text{F}$.

### 4 Partiality

So far, every system we have discussed in this course is strongly normalizing; that is, System $\text{E}$, System $\text{T}$, System $\text{F}$, and System $\text{FE}$ may only express computations that provably terminate and evaluate to a value—they are total. It was impossible to express infinite loops or divergent programs in any of those systems since they do not support general recursion. Living in the garden of total languages has allowed us to disregard the possibility that a program does not terminate, so why should we leave?

Recall from your theory course that since the halting problem is undecidable, the set of Turing machine programs that terminate is uncomputable. Therefore no total language can express every total program. By contrast, a partial programming language, one that allows for the possibility of divergence, can possibly express all programs. We will introduce a partial programming language, System $\text{PCF}$ (Plotkin, 1977), which is the first of this course to be Turing-complete. It does so by introducing fixed points, a means of general recursion.

$\text{PCF}$ is a very simple system and writing programs in it is much easier than in System $\text{T}$ or $\text{F}$. As we’ve mentioned before in the class, total programming languages also incur a blowup in the size of their programs, since they must embed the proof of the program’s correctness. That issue will now disappear, and $\text{PCF}$ programs look very much like ML programs.\(^1\)

---

\(^1\)These notes are partially derived from 15-312 Spring 2017 course notes by Jake Zimmerman.
5 System PCF

PCF is derived from T by adding the fixed-point recursion operator. It and the new zero test operator replace the System T primitive recursor.

\[
\text{Typ } \tau ::= \text{nat} \quad \text{natural number} \\
\tau_1 \to \tau_2 \quad \text{function}^2
\]

\[
\text{Exp } e ::= x \quad \text{variable} \\
\lambda (x : \tau) e \quad \text{abstraction} \\
e_1(e_2) \quad \text{application} \\
z \quad \text{zero} \\
s(e) \quad \text{successor} \\
\text{ifz } e \{ z \mapsto e_0 \mid s(x) \mapsto e_1 \} \quad \text{zero test} \\
\text{fix } x : \tau \text{ is } e \quad \text{recursion}
\]

The natural numbers, along with zero and successor, are exactly as we expect them to be. The zero test is analogous to a case on an expression, branching on whether it is zero or not. Unlike the System T recursor, it does not recursively compute on the predecessor.

Recursion is the job of the fixed-point operator fix, which defines a self-referential expression of some type. fix \( x : t \text{ is } e \) is roughly equivalent to the ML expression

\[
\text{val rec } x : t = e
\]

except that this fixed-point construct is much more powerful than recursive values in ML! In particular, \( e \) does not have to be a lambda expression.

5.1 Statics

There is only one interesting rule, that of the fixed-point operator. You should be able to infer how the typing of the zero test works; it’s based on \( e_0 \) and \( e_1 \). Everything else is exactly as it was in System T.

\[
\Gamma, x : \tau \vdash e : \tau \\
\Gamma \vdash \text{fix } x : \tau \text{ is } e : \tau
\]

Since \( x \) is a self-reference in \( e \), we expect that they have the same type.

5.2 Dynamics

Again, the only interesting rule is that of the fixed-point operator. Most rules are as they were in lazy/eager System T, with the zero test either yielding \( e_0 \) or \( e_1 \) with \( e \) substituted for \( x \) depending on whether \( e \) is zero.

\[
\text{fix } x : \tau \text{ is } e \mapsto [\text{fix } x : \tau \text{ is } e/x]e
\]

That’s it! To evaluate a fixed-point expression, we merely substitute an instance of the fixed-point for the self-reference in that expression. Intuitively, this satisfies exactly what recursion should be. Every time \( e \) refers to itself, we substitute in an instance of itself there.

\[2\text{The “partial function” arrow } \mapsto \text{ is often used for partial functions.}\]
Certainly, we may write trivial programs in PCF using numbers and functions. They correspond to basic ML:

\[
\begin{align*}
  z & \equiv 0 \\
  s(s(z)) & \equiv 2 \\
  (\lambda (x : \text{nat} \to \text{nat}) x)(\lambda (x : \text{nat}) x) & \equiv (\text{fn} \ (x : \text{int} \to \text{int}) \to x)(\text{fn} \ (x : \text{int}) \to x)
\end{align*}
\]

\(\text{ifz} \ s(s(z)) \{ z \mapsto z \mid s(x) \mapsto s(x) \} \equiv \text{if} \ 2 = 0 \ \text{then} \ 0 \ \text{else} \ (2 - 1) + 1 \)

What about a recursive ML function, like this?

\[
\text{fun fact 0 = 1} \\
| \text{fact n = n * fact (n - 1)}
\]

We need to define multiplication in PCF first. In ML:

\[
\begin{align*}
  \text{fun mult 0 _ = 0} \\
  | \text{mult m n = n + mult (m - 1) n}
\end{align*}
\]

OK, we need addition too. In ML:

\[
\begin{align*}
  \text{fun plus 0 n = n} \\
  | \text{plus m n = 1 + plus (m - 1) n}
\end{align*}
\]

Now in PCF, we can translate directly.

\[
\begin{align*}
  \text{plus} & \triangleq \text{fix} \ x : \text{nat} \to \text{nat} \to \text{nat} \ is \\
  & \lambda (m : \text{nat}) \ \lambda (n : \text{nat}) \\
  & \text{ifz} \ n \ \{ z \mapsto m \mid s(n') \mapsto s(x(n')(m)) \}
\end{align*}
\]

\[
\begin{align*}
  \text{mult} & \triangleq \text{fix} \ x : \text{nat} \to \text{nat} \to \text{nat} \ is \\
  & \lambda (m : \text{nat}) \ \lambda (n : \text{nat}) \\
  & \text{ifz} \ m \ \{ z \mapsto z \mid s(m') \mapsto \text{plus}(n)(x(m')(n)) \}
\end{align*}
\]

\[
\begin{align*}
  \text{fact} & \triangleq \text{fix} \ x : \text{nat} \to \text{nat} \to \text{nat} \ is \\
  & \lambda (n : \text{nat}) \\
  & \text{ifz} \ n \ \{ z \mapsto s(z) \mid s(n') \mapsto \text{mult}(n)(x(n')) \}
\end{align*}
\]

Note the structure of each expression: a fixed-point declaration with the type, then a body that refers back to \(x\) which is the fixed point. We could even have renamed \(x\) to \(\text{plus, mult, fact}\) respectively to make it even more like ML.

Now, what about our good friend, the Ackermann function? It was a mess to encode it in System T, while it’s trivial in ML:

\[
\begin{align*}
  \text{fun A 0 n = n + 1} \\
  | \text{A m 0 = A (m - 1) 1} \\
  | \text{A m n = A (m - 1) (A m (n - 1))}
\end{align*}
\]

Guess what? It’s trivial in PCF too!

\[
\begin{align*}
  A & \triangleq \text{fix} \ x : \text{nat} \to \text{nat} \to \text{nat} \ is \\
  & \lambda (m : \text{nat}) \ \lambda (n : \text{nat}) \\
  & \text{ifz} \ m \ \{ z \mapsto s(n) \mid s(m') \mapsto \text{ifz} \ n \ \{ z \mapsto x(m')(s(z)) \mid s(n') \mapsto x(m')(x(m')(n')) \} \}
\end{align*}
\]

Observe how easy PCF is to work with relative to T. Programs are shorter and more intuitive, and having full recursion is nice.