Recitation 6:
System F and Existential Types

15-312: Foundations of Programming Languages

Serena Wang, Charles Yuan

February 21, 2018

We saw how to use inductive and coinductive types last recitation. We can also add polymorphic types to our type system, which leads us to System **F**. With polymorphic types, we can have functions that work the same regardless of the types of some parts of the expression.[1]

1 System F

Inductive and coinductive types expand the expressivity of **T** considerably. The power of type operators allows us to generiсly manipulate data of heterogeneous types, building new types from old ones. But to write truly "generic" programs, we want truly polymorphic expressions—functions that operate on containers of some arbitrary type, for example. To gain this power, we add *parametric polymorphism* to the language, which results in System **F**, introduced by Girand (1972) and Reynolds (1974).

Typ	τ	::=	t	type variable
			$\tau_1 \to \tau_2$	function
			$\forall(t.\tau)$	universal type
Exp	e	::=	x	variable
			$\lambda\,(x:\tau)\,e$	abstraction
			$e_1(e_2)$	application
			$\Lambda(t)\,e$	type abstraction
			$e[\tau]$	type application

Take stock of what we've added since last time, and what we've removed. The familiar **type variables** are now baked into the language, along with the **universal type**. We also have a new form of lambda expression, one that works over type variables rather than expression variables.

What's missing? Nearly every other construct we've come to know and love! As will be the case repeatedly in the course, our tools such as products, sums, and inductive types are subsumed by the new polymorphic types. The result is an extremely simple System **F** that is actually even more powerful.

[1] These notes are derived from 15-312 Spring 2017 course notes by Jake Zimmerman.

1.1 Statics

Now that types have variables, we need to decide which type abt's are considered valid. We introduce the following judgment:

$$\Delta \vdash \tau \text{ type}$$

meaning that in the type context Δ, τ is a valid type. The type context Δ contains the type variables that we have seen so far.

We also attach the type context to the typing judgment, which now looks like:

$$\Delta, \Gamma \vdash e : \tau$$

To define what types are valid, we essentially just want to state that closed types (ones with no free variables) are valid, and open types are invalid. These rules express that fact:

$$\frac{}{\Delta, \tau \text{ type} \vdash \tau \text{ type}} \qquad \frac{\Delta \vdash \tau_1 \text{ type} \quad \Delta \vdash \tau_2 \text{ type}}{\Delta \vdash \tau_1 \to \tau_2 \text{ type}} \qquad \frac{\Delta, t \text{ type} \vdash \tau \text{ type}}{\Delta \vdash \forall t.\tau \text{ type}}$$

And now we may define the typing judgment. The cases for variable, lambda, and application are as they were in System **T**; we simply carry Δ along for the ride. There are two interesting new rules:

$$\frac{\Delta, t \text{ type}, \Gamma \vdash e : \tau}{\Delta, \Gamma \vdash \Lambda(t)\, e : \forall t.\tau} \qquad \frac{\Delta, \Gamma \vdash e : \forall t.\tau' \quad \Delta \vdash \tau \text{ type}}{\Delta, \Gamma \vdash e[\tau] : [\tau/t]\tau'}$$

Type lambdas are the introduction of universal types, and type applications are their elimination. The type application rule is saying that if some expression e is valid for all choices of t, then it will also be valid when the actual type τ is substituted for t (provided that τ is a valid type).

This is very similar to polymorphic types in ML, where types may contain type variables. Be aware that ML usually leaves the type lambda implicit. That is, the ML type

```
('a -> 'b) -> 'c
```

is actually

$$\forall\alpha.\forall\beta.\forall\gamma.(\alpha \to \beta) \to \gamma$$

o in System **F**. Observe that ML implicitly places the type lambdas at the front of the type. As we will soon see, this is an important distinction between ML and System **F**. ML cannot directly express a type like

$$\forall\alpha.\alpha \to \forall\beta.\beta$$

which System **F** easily can do.

ML also does not explicitly apply types. Consider the polymorphic identity function in **F**:

$$\texttt{id} \triangleq \forall\alpha.\lambda\,(x : \alpha)\, x$$

This function is truly polymorphic, as we can apply `id`[`nat`] to get the identity function on naturals, `id`[`nat` $\to$ `nat`] to get the identity function on functions from naturals to naturals, etc. However, in ML, the type checker automatically applies the appropriate type argument to its type abstractions. `id 0` and `id (fn (x:nat) => x)` implicitly involve the specialization of the function `id`.

1.2 Dynamics

System **F** also has a remarkably simple dynamics. The rules for lambda and application remain the same as in lazy/eager System **T**, and we need only introduce the rules for type lambda and type application.

$$\frac{}{\Lambda(t)\, e\ \mathsf{val}} \qquad \frac{e \longmapsto e'}{e[\tau] \longmapsto e'[\tau]} \qquad \frac{}{\Lambda(t)\, e[\tau] \longmapsto [\tau/t]e}$$

That's it! Type functions are values, type applications are eager, and they eventually substitute a type for a variable in a type abstraction.

Examples:

$$\Lambda(\alpha)\, \lambda\, (x : \alpha)\, x \text{ is the polymorphic identity function}$$
$$\Lambda(\alpha)\, \Lambda(\beta)\, \lambda\, (f : \alpha \to \beta)\, \lambda\, (x : \alpha)\, f(x) \text{ is the polymorphic applicator function}$$

1.3 Church Encodings

System **F** is great, but we're still pining for all our old types like natural numbers, lists, etc. But what if I told you that universal types could replace all of them? As it turns out, we can construct products, sums, inductive types, etc. in System **F**, using a scheme called **Church encodings**.

How would we express `nat` in System **F**?

$$\texttt{nat} \triangleq \forall t.t \to (t \to t) \to t$$

It may be difficult at first glance to see why this polymorphic type expresses everything we need for `nat`. Using this definition of `nat`, how would we write `z`, `s`, and `rec` - all the stuff we used to have for `nat` in System **T**?

$$\texttt{z} \triangleq \Lambda(t)\, \lambda\, (b : t)\, \lambda\, (s : t \to t)\, b$$
$$\texttt{s} \triangleq \lambda\, (x : \texttt{nat})\, \Lambda(t)\, \lambda\, (b : t)\, \lambda\, (s : t \to t)\, s(e[t](b)(s))$$
$$\texttt{iter}\{\tau\}(e_1, x.e_2, e) \triangleq e[\tau](e_1)(\lambda\, (x : \tau)\, e_2)$$

As you can see in the definitions for `z`, `s`, and `rec`, the first polymorphic term taken in to the function represents the zero base case, and the second polymorphic term represents the successor case. We only need a way for us to define zero and the successor for us to be able to construct a natural number, and so we only need these two arguments before the polymorphic function gives us something of type `nat`.

In the Church encoding, the number is its own recursor! That is a powerful idea. A number is only as meaningful as the ability to count with it, and so it is fitting that numbers be represented using their recursor.

How would we express sum and product types in System **F**?

$$\tau_1 + \tau_2 \triangleq \forall t.(\tau_1 \to t) \to (\tau_2 \to t) \to t$$
$$\tau_1 \times \tau_2 \triangleq \forall t.(\tau_1 \to \tau_2 \to t) \to t$$

For sum types, we simply need something of type τ_1 or something of type τ_2 to be able to get something of type $\tau_1 + \tau_2$. By a similar argument, we need something of type τ_1 and something of type τ_2 to be able to construct a term of type $\tau_1 \times \tau_2$.

$$\begin{aligned}
\texttt{l} \cdot e &\triangleq \Lambda(t)\, \lambda\,(l : \tau_1 \to t)\, \lambda\,(r : \tau_2 \to t)\, l(e) \\
\texttt{r} \cdot e &\triangleq \Lambda(t)\, \lambda\,(l : \tau_1 \to t)\, \lambda\,(r : \tau_2 \to t)\, r(e) \\
\langle e_1, e_2 \rangle &\triangleq \Lambda(t)\, \lambda\,(h : \tau_1 \to \tau_2 \to t)\, h(e_1)(e_2) \\
e \cdot \texttt{l} &\triangleq e[\tau_1](\lambda\,(l : \tau_1)\, \lambda\,(r : \tau_2)\, l) \\
e \cdot \texttt{r} &\triangleq e[\tau_2](\lambda\,(l : \tau_1)\, \lambda\,(r : \tau_2)\, r)
\end{aligned}$$

Each function argument tells us how to interact with the sum or product type internally. As an exercise, try to define `case` in the encoding.

We can also now create polymorphic data structures, which you've seen in SML in 15-150:

$$\begin{aligned}
\alpha\ \texttt{list} &\triangleq \forall\alpha.\mu(t.1 + (\alpha \times t)) \\
\alpha\ \texttt{stream} &\triangleq \forall\alpha.\nu(t.1 + (\alpha \times t))
\end{aligned}$$

Note that the thing inside of the $\forall$ is a type operator! However, $\alpha.\mu(t.1 + (\alpha \times t))$ and $\alpha.\nu(t.1 + (\alpha \times t))$ are not polynomial type operators since they contain inductive and coinductive types. We can still change our $\texttt{map}\{t.\tau\}$ to work with these type operators, though, and you'll see how to do this in Assignment 3.

2 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	τ	::=	...	
			$\exists(t.\tau)$	existential type
Exp	e	::=	...	
			$\texttt{pack}\{t.\tau\}[\rho](e)$	existental pack
			$\texttt{open}\{t.\tau\}\{\rho\}(e_1; t, x.e_2)$	existential unpack

`pack` introduces an existential type, where ρ 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 τ 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.

2.1 Statics

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

$$\frac{\Delta, t\ \mathsf{type} \vdash \tau\ \mathsf{type}}{\Delta \vdash \exists(t.\tau)\ \mathsf{type}}$$

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

$$\frac{\Delta, \Gamma \vdash e_1 : \exists(t.\tau) \quad \Delta, t\ \mathsf{type}, \Gamma, x : \tau \vdash e_2 : \tau_2 \quad \Delta \vdash \tau_2\ \mathsf{type}}{\Delta, \Gamma \vdash \mathtt{open}\{t.\tau\}\{\tau_2\}(e_1; t, x.e_2) : \tau_2}$$

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.

2.2 Dynamics

$$\frac{e\ \mathsf{val}}{\mathtt{pack}\{t.\tau\}[\rho](e)\ \mathsf{val}} \qquad \frac{e \longmapsto e'}{\mathtt{pack}\{t.\tau\}[\rho](e) \longmapsto \mathtt{pack}\{t.\tau\}[\rho](e')}$$

$$\frac{e_1 \longmapsto e_1'}{\mathtt{open}\{t.\tau\}\{\rho\}(e_1; t, x.e_2) \longmapsto \mathtt{open}\{t.\tau\}\{\rho\}(e_1'; t, x.e_2)}$$

$$\frac{e\ \mathsf{val}}{\mathtt{open}\{t.\tau\}\{\rho\}(\mathtt{pack}\{t.\tau\}[\rho](e); t, x.e_2) \longmapsto [\rho, e/t, x]e_2}$$

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!

2.3 Examples with Queues

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

$$\begin{aligned} \tau &\triangleq \langle \mathtt{emp} \hookrightarrow t, \mathtt{enq} \hookrightarrow (\mathtt{nat} \times t) \to t, \mathtt{deq} \hookrightarrow t \to 1 + (\mathtt{nat} \times t) \rangle \\ \rho &\triangleq \mathtt{nat\ list} \\ \mathtt{queue} &\triangleq \mathtt{pack}\{t.\tau\}[\rho](e) \end{aligned}$$

The e that we use to define `queue` is below. We'll use some syntax from SML in the example code below.

$$\begin{aligned} e \triangleq \langle & \mathtt{emp} \hookrightarrow [], \\ & \mathtt{enq} \hookrightarrow \lambda\,(x : \mathtt{nat} \times (\mathtt{nat\ list}))\,(x \cdot \mathtt{l}) :: (x \cdot \mathtt{r}) \\ & \mathtt{deq} \hookrightarrow \lambda\,(q : \mathtt{nat\ list})\,\mathtt{case}\ \mathtt{rev}(q)\ \{[] \hookrightarrow \mathtt{none} \mid f :: qr \hookrightarrow \mathtt{some}(\langle f, \mathtt{rev}(qr) \rangle)\} \rangle \end{aligned}$$

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

$$\begin{aligned}&\texttt{open}\{t.\tau\}\{\texttt{nat option}\}(\texttt{queue}; t, x.\\ &\qquad\qquad \texttt{let } q = x \cdot \texttt{enq}(\langle 7, x \cdot \texttt{enq}(\langle 5, x \cdot \texttt{enq}(\langle 2, x \cdot \texttt{emp}\rangle)\rangle)\rangle)\\ &\qquad\qquad \texttt{in case } x \cdot \texttt{deq}(q)\ \{\texttt{some}(x) \hookrightarrow \texttt{some}(x \cdot \texttt{1}) \mid \texttt{none} \hookrightarrow \texttt{none}\}\\ &\qquad\qquad \texttt{end})\end{aligned}$$

3 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 $e_{\texttt{ref}}$ and $e_{\texttt{cand}}$. Let's do some pattern-matching so that we can easily refer to each part of each implementation.

$$\begin{aligned}e_{\texttt{ref}} = \langle &\texttt{emp} \hookrightarrow \texttt{emp}_{\texttt{ref}},\\ &\texttt{enq} \hookrightarrow \texttt{enq}_{\texttt{ref}},\\ &\texttt{deq} \hookrightarrow \texttt{deq}_{\texttt{ref}}\rangle\\ e_{\texttt{cand}} = \langle &\texttt{emp} \hookrightarrow \texttt{emp}_{\texttt{cand}},\\ &\texttt{enq} \hookrightarrow \texttt{enq}_{\texttt{cand}},\\ &\texttt{deq} \hookrightarrow \texttt{deq}_{\texttt{cand}}\rangle\end{aligned}$$

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

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:

$$\exists(t.\langle \texttt{emp} \hookrightarrow t, \texttt{enq} \hookrightarrow (\texttt{nat} \times t) \to t, \texttt{deq} \hookrightarrow t \to 1 + (\texttt{nat} \times t)\rangle)$$

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

$$\begin{aligned}\texttt{emp}_{\texttt{ref}}\ &\mathcal{R}\ \texttt{emp}_{\texttt{cand}}\\ \texttt{enq}_{\texttt{ref}}\ (\texttt{nat} \times \mathcal{R}) &\to \mathcal{R}\ \texttt{enq}_{\texttt{cand}}\\ \texttt{deq}_{\texttt{ref}}\ \mathcal{R} &\to (\texttt{nat} \times \mathcal{R})\ \texttt{deq}_{\texttt{cand}}\end{aligned}$$

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

1. $\texttt{emp}_{\texttt{ref}}\ \mathcal{R}\ \texttt{emp}_{\texttt{cand}}$
2. For all n,
   - Assume $q_{\texttt{ref}}\ \mathcal{R}\ q_{\texttt{cand}}$.
   - Prove $\texttt{enq}_{\texttt{ref}}\,(n)\,(q_{\texttt{ref}})\ \mathcal{R}\ \texttt{enq}_{\texttt{cand}}\,(n)\,(q_{\texttt{cand}})$.
3. Assume $q_{\texttt{ref}}\ \mathcal{R}\ q_{\texttt{cand}}$. Want to show either:

- $\texttt{deq}_{\texttt{ref}}\,(q_{\texttt{ref}}) \cong \texttt{deq}_{\texttt{cand}}\,(q_{\texttt{cand}})$
- $\texttt{deq}_{\texttt{ref}}\,(q_{\texttt{ref}}) \cong \texttt{some}(\langle n, r_{\texttt{ref}}\rangle)$ and $\texttt{deq}_{\texttt{cand}}\,(q_{\texttt{cand}}) \cong \texttt{some}(\langle n', r_{\texttt{cand}}\rangle)$ such that $n \cong n'$ and $r_{\texttt{ref}}\ \mathcal{R}\ r_{\texttt{cand}}$.

So first we have to define our relation:

$$l\ \mathcal{R}\ \langle b, f\rangle \qquad \Longleftrightarrow \qquad l\ \cong\ b \mathbin{@} (\texttt{rev}\ \ f)$$

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:

$$\begin{aligned} e_{\texttt{ref}} \triangleq \langle & \texttt{emp} \hookrightarrow \texttt{[]}, \\ & \texttt{enq} \hookrightarrow \lambda\,(x : \texttt{nat} \times (\texttt{nat list}))\,(x \cdot \texttt{l}) :: (x \cdot \texttt{r}) \\ & \texttt{deq} \hookrightarrow \lambda\,(q : \texttt{nat list})\,\texttt{case rev}(q)\ \{\texttt{[]} \hookrightarrow \texttt{none} \mid f :: qr \hookrightarrow \texttt{some}(\langle f, \texttt{rev}(qr)\rangle)\}\rangle \end{aligned}$$

$$\begin{aligned} e_{\texttt{cand}} \triangleq \langle & \texttt{emp} \hookrightarrow \langle \texttt{[]}, \texttt{[]}\rangle, \\ & \texttt{enq} \hookrightarrow \lambda\,(x : \texttt{nat} \times (\texttt{nat list} \times \texttt{nat list}))\,\langle (x \cdot \texttt{l}) :: (x \cdot \texttt{r} \cdot \texttt{l}), x \cdot \texttt{r} \cdot \texttt{r}\rangle \\ & \texttt{deq} \hookrightarrow \lambda(q : \texttt{nat list} \times \texttt{nat list}) \\ & \qquad \texttt{case}(x \cdot \texttt{r})\{\texttt{[]} \hookrightarrow \texttt{case rev}(bs)\ \{\texttt{[]} \hookrightarrow \texttt{none} \mid b :: bs' \hookrightarrow \texttt{some}(\langle b, \langle \texttt{[]}, bs'\rangle\rangle)\} \\ & \qquad\qquad \mid f :: fs' \hookrightarrow \texttt{some}(\langle f, \langle x \cdot \texttt{l}, fs'\rangle\rangle)\rangle \end{aligned}$$

And now let's show that $\mathcal{R}$ respects the relation:

1. $\texttt{emp}_{\texttt{ref}}\ \mathcal{R}\ \texttt{emp}_{\texttt{cand}}$

$$\begin{aligned} \texttt{emp}_{\texttt{cand}} &\cong \texttt{[]}\ @\ (\texttt{rev}\ \ \texttt{[]}) \\ &\cong \texttt{[]}\ @\ \texttt{[]} \\ &\cong \texttt{[]} \\ &\cong \texttt{emp}_{\texttt{ref}} \end{aligned}$$

2. $\texttt{enq}_{\texttt{ref}}\,(n)\,(q_{\texttt{ref}})\ \mathcal{R}\ \texttt{enq}_{\texttt{cand}}\,(n)\,(q_{\texttt{cand}})$

   Let $q_{\texttt{cand}} = \langle b_{\texttt{cand}}, f_{\texttt{cand}}\rangle$.
   Assume $q_{\texttt{ref}}\ \mathcal{R}\ q_{\texttt{cand}}$.
   Thus, $q_{\texttt{ref}}\ \cong\ b_{\texttt{cand}} \mathbin{@} (\texttt{rev}\ \ f_{\texttt{cand}})$.

$$\begin{aligned} \texttt{enq}_{\texttt{cand}}\,(n)\,(q_{\texttt{cand}}) &\cong (n :: b_{\texttt{cand}})\ @\ (\texttt{rev}\ \ f_{\texttt{cand}}) \\ &\cong n :: (b_{\texttt{cand}}\ @\ \texttt{rev}\ \ f_{\texttt{cand}}) \\ &\cong n :: q_{\texttt{ref}} \\ &\cong \texttt{enq}_{\texttt{ref}}\,(n)\,(q_{\texttt{ref}}) \end{aligned}$$

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

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

a) $q_{\texttt{ref}} = \texttt{[]}$, $q_{\texttt{cand}} = \langle \texttt{[]}, \texttt{[]} \rangle$

$$\begin{aligned} \texttt{deq}_{\texttt{ref}}\,(q_{\texttt{ref}}) &\cong \texttt{deq}_{\texttt{ref}}\ \texttt{[]} \\ &\cong \ldots \\ &\cong \texttt{none} \\ \texttt{deq}_{\texttt{cand}}\,(q_{\texttt{cand}}) &\cong \texttt{deq}_{\texttt{ref}}\ \langle \texttt{[]}, \texttt{[]} \rangle \\ &\cong \ldots \\ &\cong \texttt{none} \end{aligned}$$

b) $q_{\texttt{ref}} = n :: q'_{\texttt{ref}}$, $q_{\texttt{cand}} = \langle b_{\texttt{cand}}, f :: f'_{\texttt{cand}} \rangle$

$$\begin{aligned} q_{\texttt{ref}} \ \mathcal{R}\ q_{\texttt{cand}} & \\ \mathcal{R}\ \langle b_{\texttt{cand}}, f :: f'_{\texttt{cand}} \rangle & \\ \cong b_{\texttt{cand}} \mathbin{@} (\texttt{rev}\ \ f :: f'_{\texttt{cand}}) & \\ \cong b_{\texttt{cand}} \mathbin{@} (\texttt{rev}\ \ f'_{\texttt{cand}}) \mathbin{@} \texttt{[}f\texttt{]} & \\ \texttt{rev}\ \ q_{\texttt{ref}} \cong \texttt{rev}(b_{\texttt{cand}} \mathbin{@} (\texttt{rev}\ \ f'_{\texttt{cand}}) \mathbin{@} \texttt{[}f\texttt{]}) & \\ \cong f :: (\texttt{rev}(b_{\texttt{cand}} \mathbin{@} (\texttt{rev}\ \ f'_{\texttt{cand}})) & \end{aligned}$$

Thus, $\texttt{rev}\ \ q_{\texttt{ref}} \cong f :: (\texttt{rev}(q'_{\texttt{ref}}))$, where $q'_{\texttt{ref}} \cong b_{\texttt{cand}} \mathbin{@} (\texttt{rev}\ \ f'_{\texttt{cand}})$. Therefore, $q'_{\texttt{ref}}\ \mathcal{R}\ \langle b_{\texttt{cand}}, f'_{\texttt{cand}} \rangle$.

$$\begin{aligned} \texttt{deq}_{\texttt{cand}}\,(q_{\texttt{cand}}) &\cong \texttt{deq}_{\texttt{cand}}\,\langle b_{\texttt{cand}}, f :: f'_{\texttt{cand}} \rangle \\ &\cong \texttt{some}(\langle f, \langle b_{\texttt{cand}}, f'_{\texttt{cand}} \rangle \rangle) \\ &\cong \texttt{none} \\ \texttt{deq}_{\texttt{ref}}\,(q_{\texttt{ref}}) &\cong \texttt{case}\ \texttt{rev}(q_{\texttt{ref}})\ \{\texttt{[]} \hookrightarrow \texttt{none} \mid f :: qr \hookrightarrow \texttt{some}(\langle f, \texttt{rev}(qr) \rangle)\} \\ &\cong \texttt{some}(\langle f, \texttt{rev(rev(}q'_{\texttt{ref}}\texttt{))} \rangle) \\ &\cong \texttt{some}(\langle f, q'_{\texttt{ref}} \rangle) \end{aligned}$$

c) $q_{\texttt{cand}} \cong \langle n :: b_{\texttt{cand}}, \texttt{[]} \rangle$
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.

4 Definability in System F

We introduced the new language System **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 τ that uses the representation type t, such that we may perform computation using τ 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 $\tau \to \tau_2$ where t *is not allowed to appear* in τ_2.

Think about it this way: we defined two alternative queue implementations $e_{\texttt{ref}}$ and $e_{\texttt{cand}}$. Let's denote their representation types (recall, one list and two lists, respectively) as $\rho_{\texttt{ref}}$ and $\rho_{\texttt{cand}}$. A client may use the queue data structure to perform computation, but cannot *return a value containing type* $\rho_{\texttt{ref}}$ or $\rho_{\texttt{cand}}$ 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 τ 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 `pack` and `open`. Based on the definition of the existential type, defining these two is pretty much a game of matching up types. The definitions are:

$$\texttt{pack}\{t.\tau\}[\rho](e) \triangleq \Lambda(u)\,\lambda\,(x : \forall(t.\tau \to u))\,x[\rho](e)$$
$$\texttt{open}\{t.\tau\}\{\tau_2\}(e_1; t, x.e_2) \triangleq e_1[\tau_2](\Lambda(t)\,\lambda\,(x : \tau)\,e_2)$$

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