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 generically 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 \textit{parametric polymorphism} to the language, which results in System F, introduced by Girand (1972) and Reynolds (1974).

\[
\text{Typ} \quad \tau \ ::= \quad t \quad \text{type variable} \\
\quad \tau_1 \to \tau_2 \quad \text{function} \\
\quad \forall (t.\tau) \quad \text{universal type}
\]

\[
\text{Exp} \quad e \ ::= \quad x \quad \text{variable} \\
\quad \lambda (x : \tau) \ e \quad \text{abstraction} \\
\quad e_1 (e_2) \quad \text{application} \\
\quad \Lambda (t) \ e \quad \text{type abstraction} \\
\quad e[\tau] \quad \text{type application}
\]

Take stock of what we’ve added since last time, and what we’ve removed. The familiar \texttt{type variables} are now baked into the language, along with the \texttt{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 $\Delta$, $\tau$ is a valid type. The type context $\Delta$ 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:

$$\begin{align*}
\Delta, \tau \text{ type} & \vdash \tau \text{ type} \\
\Delta & \vdash \tau_1 \text{ type} \quad \Delta & \vdash \tau_2 \text{ type} \\
\Delta & \vdash \tau_1 \rightarrow \tau_2 \text{ type} \\
\Delta & \vdash \forall t.\tau \text{ type}
\end{align*}$$

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 $\Delta$ along for the ride. There are two interesting new rules:

$$\begin{align*}
\Delta, t \text{ type}, \Gamma & \vdash e : \tau \\
\Delta & \vdash \Lambda(t) e : \forall t.\tau \\
\Delta, t \text{ type} & \vdash \tau \text{ type} \\
\Delta, \Gamma & \vdash e[\tau] : [\tau/t]\tau'
\end{align*}$$

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 $\tau$ is substituted for $t$ (provided that $\tau$ 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

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

is actually

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

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

$$\forall \alpha.\alpha \rightarrow \forall \beta.\beta$$

which System F easily can do.

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

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

This function is truly polymorphic, as we can apply $\text{id}[\text{nat}]$ to get the identity function on naturals, $\text{id}[\text{nat} \rightarrow \text{nat}]$ 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. $\text{id} 0$ and $\text{id} (\text{fn} (x:\text{nat}) \Rightarrow x)$ implicitly involve the specialization of the function $\text{id}$. 

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.

\[ \Lambda(t) e \rightarrow e'[\tau] \]

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 \textsf{nat} in System F?

\[ \textsf{nat} \triangleq \forall t.t \to (t \to t) \to t \]

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

\[ \texttt{z} \triangleq \Lambda(t) \lambda(b : t) \lambda(s : t \to t) b \]

\[ \texttt{s} \triangleq \lambda(x : \textsf{nat}) \Lambda(t) \lambda(b : t) \lambda(s : t \to t) s(e[t](b)(s)) \]

\[ \texttt{iter}\{\tau\}(e_1, x, e_2, c) \triangleq c[\tau](e_1)(\lambda(x : \tau) e_2) \]

As you can see in the definitions for \texttt{z}, \texttt{s}, and \texttt{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 \textsf{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 $\tau_1$ or something of type $\tau_2$ to be able to get something of type $\tau_1 + \tau_2$. By a similar argument, we need something of type $\tau_1$ and something of type $\tau_2$ to be able to construct a term of type $\tau_1 \times \tau_2$.

$$
1 \cdot e \triangleq \Lambda(t) \lambda(l : \tau_1 \rightarrow t) \lambda(r : \tau_2 \rightarrow t) l(e) \\
r \cdot e \triangleq \Lambda(t) \lambda(l : \tau_1 \rightarrow t) \lambda(r : \tau_2 \rightarrow t) r(e) \\
\langle e_1, e_2 \rangle \triangleq \Lambda(t) \lambda(h : \tau_1 \rightarrow \tau_2 \rightarrow t) h(e_1)(e_2) \\
e \cdot 1 \triangleq e[\tau_1](\lambda(l : \tau_1) \lambda(r : \tau_2) l) \\
e \cdot r \triangleq e[\tau_2](\lambda(l : \tau_1) \lambda(r : \tau_2) r)
$$

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:

$$
\alpha \text{ list} \triangleq \forall \alpha. \mu(t. \tau_1 + (\alpha \times t)) \\
\alpha \text{ stream} \triangleq \forall \alpha. \nu(t. \tau_1 + (\alpha \times t))
$$

Note that the thing inside of the $\forall$ is a type operator! However, $\forall \alpha. (\mu(t. \tau_1 + (\alpha \times t)))$ and $\forall \alpha. (\nu(t. \tau_1 + (\alpha \times t)))$ are not polynomial type operators since they contain inductive and coinductive types. We can still change our map 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.

$$
\text{Typ } \tau ::= \ldots \\
\exists(t.\tau) \quad \text{existential type}
$$

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

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.
2.1 Statics

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

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

\[
\frac{\Delta \vdash \rho \text{ type} \quad \Delta, t \text{ type} \quad \Delta, \Gamma \vdash e : [\rho/t]\tau}{\Delta, \Gamma \vdash \text{open}\{t.\tau\}\{\rho\}{e}[t, x, e_2] : \tau_2}
\]

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

\[
\frac{e \text{ val}}{\text{pack}\{t.\tau\}\{\rho\}{e} \text{ val}}
\]

\[
\frac{e \longmapsto e'}{\text{pack}\{t.\tau\}\{\rho\}{e} \longmapsto \text{pack}\{t.\tau\}\{\rho\}{e'}}
\]

\[
\frac{e_1 \longmapsto e'_1}{\text{open}\{t.\tau\}\{\rho\}{e_1}[t, x, e_2] \longmapsto \text{open}\{t.\tau\}\{\rho\}{e'_1}[t, x, e_2]}
\]

\[
\frac{e \text{ val}}{\text{open}\{t.\tau\}\{\rho\}{\text{pack}\{t.\tau\}\{\rho\}{e}}[t, x, e_2] \longmapsto [\rho, e/t, x]e_2}
\]

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.

\[
\tau \triangleq \langle \text{emp} \mapsto t, \text{enq} \mapsto (\text{nat} \times t) \mapsto t, \text{deq} \mapsto t \rightarrow 1 + (\text{nat} \times t)\rangle
\]

\[
\rho \triangleq \text{nat list}
\]

\[
\text{queue} \triangleq \text{pack}\{t.\tau\}\{\rho\}{e}
\]

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

\[
e \triangleq \langle \text{emp} \mapsto [],
\quad \text{enq} \mapsto \lambda (x : \text{nat} \times \text{nat list}) (x \cdot 1) :: (x \cdot r)
\quad \text{deq} \mapsto \lambda (q : \text{nat list}) \text{case rev}(q) \{ [] \mapsto \text{none} \mid f :: qr \mapsto \text{some}((f, \text{rev}(qr))) \}
\]
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 · deq(q)` since the thing we return must have extrinsic value.

```plaintext
open{t.τ}{nat option}(queue; t, x.
    let q = x · enq(⟨7, x · enq(⟨5, x · enq(⟨2, x · emp⟩)⟩)⟩)
    in case x · deq(q) {some(x) ↪ some(x · 1) | none ↪ none}
end)
```

### 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 $R$ over expressions of the abstract type. This relation will essentially convert one of the implementation types into the other implementation type.

#### 3.1 Dynamic Dispatch

As we have seen in the lecture, dynamic dispatch can be implemented in different ways. We leverage this idea to abstract types: we can see dynamic dispatch as an abstract type of objects supporting (1) creation of an object of a class, and (2) sending a message to an object to obtain a result. From the client side, there is no way to distinguish between different implementations.

Consider the following existential type that formalizes the idea:

$$\exists(t.\langle\text{new} \hookrightarrow \langle\tau \to t\rangle c \in C, \text{snd} \hookrightarrow \langle t \to \rho d\rangle d \in D\rangle)$$

The package supports and only supports creating an object of class $c \in C$ and invoking the method $d \in D$ on object of any class.

Given the dispatch matrix $e_{DM}$, we can implement this package in two natural ways:

1. Internal representation is the product of the results from all the methods invoked on the object created. Upon `new`, we compute the result for each method on the object and store all the results in a big tuple; when `snd` is called for some method, we simply project out the appropriate result to the client.

   $$\tau^I_{obj} \triangleq \langle d \hookrightarrow \rho_d\rangle d \in D$$
   $$e^I_{new} \triangleq (\lambda(x^c : \tau^c)(d \hookrightarrow e_{DM} \cdot c \cdot d(x^c))_{d \in D})_{c \in C}$$
   $$e^I_{snd} \triangleq (\lambda(x : t)(x \cdot d))_{d \in D}$$

2. Internal representation is the object class, represented as a sum type. Upon `new`, we only store which class the object was create for; when `snd` is called for some method, we case of the class of the object and execute the appropriate method from the dispatch matrix.

   $$\tau^I_{obj} \triangleq [c \hookrightarrow \tau^c]_{c \in C}$$
   $$e^I_{new} \triangleq (\lambda(x^c : \tau^c)c \cdot x^c)_{c \in C}$$
   $$e^I_{snd} \triangleq (\lambda(x : t)\text{ case } x\{c \cdot x^c \hookrightarrow e_{DM} \cdot c \cdot d(x^c)\}_{c \in C})_{d \in D}$$

It should be straightforward to convince ourselves that these two representations are equivalent: they are simply two different ways to organize the dispatch matrix; from the perspective of a
lawful client (one that does not write code that results in some type involving the internal representation type), there is no difference in terms of the computation result we get.

In other words, we want say there is some equivalence relation $R$ between the two representation type $\tau^I_{obj}$ and $\tau^H_{obj}$ that is respected by all the lawful operations given by the package. Precisely:

1. For all $c \in C$, if $e^I_c =_e e^H_c$ ("equivalent" expression of type $\tau^C$), then $R(new^I \cdot c(e^I_c), new^H \cdot c(e^H_c))$.
2. If $R(e^I, e^H)$, then for all $d \in D$, $snd^I \cdot d(e^I) =_{\rho_d} snd^H \cdot d(e^H)$ ("equivalent expression of type $\rho_d$").

$new^I, new^H, snd^I, snd^H$ are the implementations given by the two package respectively.

By formalizing our mental belief that the two representation types are morally the same – both are some way to organize the dispatch matrix, we can reach the following definition of the relation $R$.

$R(e^I, e^H)$ iff $e^I \mapsto^* \langle d \mapsto e^I \cdot d \mapsto^* e^H \rangle_{d \in D}$ and for all $d \in D, e^I_d =_{\rho_d} e^H_d$.

Then we can see the proof would go through nicely:

1. Given $e^I_c =_e e^H_c$;

   $$new^I \cdot c(e^I_c) \mapsto^* \langle d \mapsto e^I \cdot d \mapsto^* e^H \rangle_{d \in D}$$

   and for the second package,

   $$new^H \cdot c(e^H_c) \mapsto^* e^H_c$$

   It suffices to show

   $$e^H_c \mapsto^* e^H_c$$

   which is true since $e^I_c =_e e^H_c$.

2. Given $R(e^I, e^H)$,

   $$snd^I \cdot d(e^I) \mapsto^* e^I \cdot d \mapsto^* \langle d \mapsto e^I \cdot d \mapsto^* e^H \rangle_{d \in D}$$

   and for the second package,

   $$snd^H \cdot d(e^H) \mapsto^* casec^H \{ c \cdot x^C \mapsto e^H \cdot c \cdot d(x^C) \}_{c \in C} \mapsto^* casec^H \{ c \cdot x^C \mapsto e^H \cdot c \cdot d(x^C) \}_{c \in C} \mapsto^* e^H_d$$

   where

   $$e^I_d =_{\rho_d} e^H_d$$

   since $R(e^I, e^H)$. 

3.2 Queues

Suppose we have two implementations of queues $e_{\text{ref}}$ and $e_{\text{cand}}$. Let’s do some pattern-matching so that we can easily refer to each part of each implementation.

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

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

We want to show $e_{\text{ref}} \equiv_R e_{\text{cand}}$.

To show this, what we want to show is that $\equiv_R$ 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 $\equiv_R$ and prove the statements that result”:

$$\text{emp}_{\text{ref}} \equiv_R \text{emp}_{\text{cand}}$$

$$\text{enq}_{\text{ref}} (\text{nat} \times \equiv_R) \rightarrow \equiv_R \text{enq}_{\text{cand}}$$

$$\text{deq}_{\text{ref}} \equiv_R (\text{nat} \times \equiv_R) \text{deq}_{\text{cand}}$$

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

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

2. For all $n$,
   - Assume $q_{\text{ref}} \equiv_R q_{\text{cand}}$.
   - Prove $\text{enq}_{\text{ref}} (n) (q_{\text{ref}}) \equiv_R \text{enq}_{\text{cand}} (n) (q_{\text{cand}})$.

3. Assume $q_{\text{ref}} \equiv_R q_{\text{cand}}$. Want to show either:
   - $\text{deq}_{\text{ref}} (q_{\text{ref}}) \equiv \text{deq}_{\text{cand}} (q_{\text{cand}})$
   - $\text{deq}_{\text{ref}} (q_{\text{ref}}) \equiv \text{some} (\langle n, r_{\text{ref}} \rangle)$ and $\text{deq}_{\text{cand}} (q_{\text{cand}}) \equiv \text{some} (\langle n', r_{\text{cand}} \rangle)$ such that $n \equiv n'$ and $r_{\text{ref}} \equiv_R r_{\text{cand}}$.

So first we have to define our relation:

$$l \equiv_R \langle b, f \rangle \iff l \equiv b @ (\text{rev} f)$$

Now that we’ve defined our relation, showing everything remaining is just a matter of handwaving our way through some proofs. Note that proofs of bisimulations in this class are a rare exception to our previous rules of formality!
3.2 Queues

Let’s put the two implementations here so we can refer back to them later:

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

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

And now let’s show that \( R \) respects the relation:

1. \( e_{\text{ref}} R e_{\text{cand}} \)

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

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, \( q_{\text{ref}} \equiv b_{\text{cand}} @ (\text{rev } f_{\text{cand}}) \).

\[ \text{enq}_{\text{cand}}(n) (q_{\text{cand}}) \equiv (n :: b_{\text{cand}}) @ (\text{rev } f_{\text{cand}}) \]
\[ \equiv n :: (b_{\text{cand}} @ \text{rev } f_{\text{cand}}) \]
\[ \equiv n :: q_{\text{ref}} \]
\[ \equiv \text{enq}_{\text{ref}}(n) (q_{\text{ref}}) \]

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, \( q_{\text{ref}} \equiv b_{\text{cand}} @ (\text{rev } f_{\text{cand}}) \).
There are 3 cases:

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

\[ \text{deq}_{\text{ref}} (q_{\text{ref}}) \equiv \text{deq}_{\text{ref}} [] \]
\[ \equiv \ldots \]
\[ \equiv \text{none} \]

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

\[
q_{\text{ref}} \mathcal{R} q_{\text{cand}}
\]
\[
\mathcal{R} \langle b_{\text{cand}}, f :: f'_{\text{cand}} \rangle
\]
\[
\cong b_{\text{cand}} @ (\text{rev } f :: f'_{\text{cand}})
\]
\[
\cong b_{\text{cand}} @ (\text{rev } f'_{\text{cand}}) @ [f]
\]
\[
\text{rev } q_{\text{ref}} \cong \text{rev}(b_{\text{cand}} @ (\text{rev } f'_{\text{cand}}) @ [f])
\]
\[
\cong f :: (\text{rev}(b_{\text{cand}} @ (\text{rev } f'_{\text{cand}}))
\]

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

\[
\text{deq}_{\text{cand}}(q_{\text{cand}}) \cong \text{deq}_{\text{cand}} \langle b_{\text{cand}}, f :: f'_{\text{cand}} \rangle
\]
\[
\cong \text{some}(\langle f, \langle b_{\text{cand}}, f'_{\text{cand}} \rangle \rangle)
\]
\[
\cong \text{none}
\]

\[
\text{deq}_{\text{ref}}(q'_{\text{ref}}) \cong \text{case } \text{rev}(q_{\text{ref}}) \{ [] \leftarrow \text{none} | f :: qr \leftarrow \text{some}(\langle f, \text{rev}(qr) \rangle) \}
\]
\[
\cong \text{some}(\langle f, \text{rev}(\text{rev}(q'_{\text{ref}})) \rangle)
\]
\[
\cong \text{some}(\langle f, q'_{\text{ref}} \rangle)
\]

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

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.