Part VIII

Data Structures and Abstraction

Chapter 19

Aggregate Data Structures

It is interesting to add to MinML support for programming with aggregate data structures such as n-tuples, lists, and tree structures. We will decompose these familiar data structures into three types:

1. Product (or tuple) types. In general these are types whose values are n-tuples of values, with each component of a specified type. We will study two special cases that are sufficient to cover the general case: 0-tuples (also known as the unit type) and 2-tuples (also known as ordered pairs).

2. Sum (or variant or union) types. These are types whose values are values of one of n specified types, with an explicit "tag" indicating which of the n choices is made.

3. Recursive types. These are "self-referential" types whose values may have as constituents values of the recursive type itself. Familiar examples include lists and trees. A non-empty list consists of a value at the head of the list together with another value of list type.

19.1 Products

The first-order abstract syntax associated with nullary and binary product types is given by the following grammar:

Types	τ	$::=$	`unit` $\mid \tau_1$ `*` τ_2
Expressions	e	$::=$	`()` $\mid$ `check` e_1 `is ()` `in` e_2 `end` $\mid$ `(`e_1`,`e_2`)` $\mid$ `split` e_1 `as (`x`,`y`)` `in` e_2 `end`
Values	v	$::=$	`()` $\mid$ `(`v_1`,`v_2`)`

The higher-order abstract syntax is given by stipulating that in the expression $\texttt{split}\,e_1\,\texttt{as}\,(x,y)\,\texttt{in}\,e_2\,\texttt{end}$ the variables x and y are bound within e_2, and hence may be renamed (consistently, avoiding capture) at will without changing the interpretation of the expression.

The static semantics of these constructs is given by the following typing rules:

$$\frac{}{\Gamma \vdash \texttt{()} : \texttt{unit}} \tag{19.1}$$

$$\frac{\Gamma \vdash e_1 : \texttt{unit} \quad \Gamma \vdash e_2 : \tau_2}{\Gamma \vdash \texttt{check}\,e_1\,\texttt{is}\,\texttt{()}\,\texttt{in}\,e_2\,\texttt{end} : \tau_2} \tag{19.2}$$

$$\frac{\Gamma \vdash e_1 : \tau_1 \quad \Gamma \vdash e_2 : \tau_2}{\Gamma \vdash \texttt{(}e_1\texttt{,}\,e_2\texttt{)} : \tau_1\texttt{*}\tau_2} \tag{19.3}$$

$$\frac{\Gamma \vdash e_1 : \tau_1\texttt{*}\tau_2 \quad \Gamma, x{:}\tau_1, y{:}\tau_2 \vdash e_2 : \tau}{\Gamma \vdash \texttt{split}\,e_1\,\texttt{as}\,\texttt{(}x\texttt{,}\,y\texttt{)}\,\texttt{in}\,e_2\,\texttt{end} : \tau} \tag{19.4}$$

The dynamic semantics is given by these rules:

$$\frac{}{\texttt{check}\,\texttt{()}\,\texttt{is}\,\texttt{()}\,\texttt{in}\,e\,\texttt{end} \mapsto e} \tag{19.5}$$

$$\frac{e_1 \mapsto e_1'}{\texttt{check}\,e_1\,\texttt{is}\,\texttt{()}\,\texttt{in}\,e_2\,\texttt{end} \mapsto \texttt{check}\,e_1'\,\texttt{is}\,\texttt{()}\,\texttt{in}\,e_2\,\texttt{end}} \tag{19.6}$$

$$\frac{e_1 \mapsto e_1'}{\texttt{(}e_1\texttt{,}\,e_2\texttt{)} \mapsto \texttt{(}e_1'\texttt{,}\,e_2\texttt{)}} \tag{19.7}$$

$$\frac{e_2 \mapsto e_2'}{\texttt{(}v_1\texttt{,}\,e_2\texttt{)} \mapsto \texttt{(}v_1\texttt{,}\,e_2'\texttt{)}} \tag{19.8}$$

$$\frac{}{\texttt{split}\,\texttt{(}v_1\texttt{,}\,v_2\texttt{)}\,\texttt{as}\,\texttt{(}x\texttt{,}\,y\texttt{)}\,\texttt{in}\,e\,\texttt{end} \mapsto \{v_1, v_2/x, y\}e} \tag{19.9}$$

$$\frac{e_1 \mapsto e_1'}{\texttt{split}\,e_1\,\texttt{as}\,\texttt{(}x\texttt{,}\,y\texttt{)}\,\texttt{in}\,e_2\,\texttt{end} \mapsto \texttt{split}\,e_1'\,\texttt{as}\,\texttt{(}x\texttt{,}\,y\texttt{)}\,\texttt{in}\,e_2\,\texttt{end}} \tag{19.10}$$

Exercise 48
State and prove the soundness of this extension to MinML.

Exercise 49
A variation is to treat any pair (e_1, e_2) *as a value, regardless of whether or not* e_1 *or* e_2 *are values. Give a precise formulation of this variant, and prove it sound.*

Exercise 50
It is also possible to formulate a direct treatment of n*-ary product types (for* $n \geq 0$*), rather than to derive them from binary and nullary products. Give a direct formalization of* n*-ary products. Be careful to get the cases* $n = 0$ *and* $n = 1$ *right!*

Exercise 51
Another variation is to considered labelled products *in which the components are accessed directly by referring to their labels (in a manner similar to C* `struct`*'s). Formalize this notion.*

19.2 Sums

The first-order abstract syntax of nullary and binary sums is given by the following grammar:

$$\begin{array}{llll} \textit{Types} & \tau & ::= & \mathtt{void} \mid \tau_1\mathtt{+}\tau_2 \\ \textit{Expressions} & e & ::= & \mathtt{inl}_{\tau_1\mathtt{+}\tau_2}\mathtt{(}e_1\mathtt{)} \mid \mathtt{inr}_{\tau_1\mathtt{+}\tau_2}\mathtt{(}e_2\mathtt{)} \mid \\ & & & \mathtt{case}_\tau\, e_0\, \mathtt{of}\, \mathtt{inl(}x\mathtt{:}\tau_1\mathtt{)}\, \mathtt{=>}\, e_1 \mid \mathtt{inr(}y\mathtt{:}\tau_2\mathtt{)}\, \mathtt{=>}\, e_2\, \mathtt{end} \\ \textit{Values} & v & ::= & \mathtt{inl}_{\tau_1\mathtt{+}\tau_2}\mathtt{(}v_1\mathtt{)} \mid \mathtt{inr}_{\tau_1\mathtt{+}\tau_2}\mathtt{(}v_2\mathtt{)} \end{array}$$

The higher-order abstract syntax is given by noting that in the expression $\mathtt{case}_\tau\, e_0\, \mathtt{of}\, \mathtt{inl(}x\mathtt{:}\tau_1\mathtt{)}\, \mathtt{=>}\, e_1 \mid \mathtt{inr(}y\mathtt{:}\tau_2\mathtt{)}\, \mathtt{=>}\, e_2\, \mathtt{end}$, the variable x is bound in e_1 and the variable y is bound in e_2.

The typing rules governing these constructs are given as follows:

$$\frac{\Gamma \vdash e_1 : \tau_1}{\Gamma \vdash \mathtt{inl}_{\tau_1\mathtt{+}\tau_2}\mathtt{(}e_1\mathtt{)} : \tau_1\mathtt{+}\tau_2} \tag{19.11}$$

$$\frac{\Gamma \vdash e_2 : \tau_2}{\Gamma \vdash \mathtt{inl}_{\tau_1\mathtt{+}\tau_2}\mathtt{(}e_2\mathtt{)} : \tau_1\mathtt{+}\tau_2} \tag{19.12}$$

$$\frac{\Gamma \vdash e_0 : \tau_1\mathtt{+}\tau_2 \quad \Gamma, x_1\mathtt{:}\tau_1 \vdash e_1 : \tau \quad \Gamma, x_2\mathtt{:}\tau_2 \vdash e_2 : \tau}{\Gamma \vdash \mathtt{case}_\tau\, e_0\, \mathtt{of}\, \mathtt{inl(}x_1\mathtt{:}\tau_1\mathtt{)}\, \mathtt{=>}\, e_1 \mid \mathtt{inr(}x_2\mathtt{:}\tau_2\mathtt{)}\, \mathtt{=>}\, e_2\, \mathtt{end} : \tau} \tag{19.13}$$

The evaluation rules are as follows:

$$\frac{e \mapsto e'}{\texttt{inl}_{\tau_1+\tau_2}(e) \mapsto \texttt{inl}_{\tau_1+\tau_2}(e')} \tag{19.14}$$

$$\frac{e \mapsto e'}{\texttt{inr}_{\tau_1+\tau_2}(e) \mapsto \texttt{inr}_{\tau_1+\tau_2}(e')} \tag{19.15}$$

$$\texttt{case}_{\tau}\ \texttt{inl}_{\tau_1+\tau_2}(v)\ \texttt{of}\ \texttt{inl}(x_1:\tau_1) \Rightarrow e_1 \mid \texttt{inr}(x_2:\tau_2) \Rightarrow e_2\ \texttt{end} \mapsto \{v/x_1\}e_1 \tag{19.16}$$

$$\texttt{case}_{\tau}\ \texttt{inr}_{\tau_1+\tau_2}(v)\ \texttt{of}\ \texttt{inl}(x_1:\tau_1) \Rightarrow e_1 \mid \texttt{inr}(x_2:\tau_2) \Rightarrow e_2\ \texttt{end} \mapsto \{v/x_2\}e_2 \tag{19.17}$$

Exercise 52
State and prove the soundness of this extension.

Exercise 53
Consider these variants: $\texttt{inl}_{\tau_1+\tau_2}(e)$ and $\texttt{inr}_{\tau_1+\tau_2}(e)$ are values, regardless of whether or not e is a value; n-ary sums; labelled sums.

19.3 Recursive Types

Recursive types are somewhat less familiar than products and sums. Few well-known languages provide direct support for these. Instead the programmer is expected to simulate them using pointers and similar low-level representations. Here instead we'll present them as a fundamental concept.

As mentioned in the introduction, the main idea of a recursive type is similar to that of a recursive function — self-reference. The idea is easily illustrated by example. Informally, a list of integers may be thought of as either the empty list, $\texttt{nil}$, or a non-empty list, $\texttt{cons}(h,t)$, where h is an integer and t is another list of integers. The operations $\texttt{nil}$ and $\texttt{cons}(-,-)$ are *value constructors* for the type $\texttt{ilist}$ of integer lists. We may program with lists using a form of case analysis, written

$$\texttt{listcase}\ e\ \texttt{of}\ \texttt{nil} \Rightarrow e_1 \mid \texttt{cons}(x,y) \Rightarrow e_2\ \texttt{end},$$

where x and y are bound in e_2. This construct analyses whether e is the empty list, in which case it evaluates e_1, or a non-empty list, with head x and tail y, in which case it evaluates e_2 with the head and tail bound to these variables.

Exercise 54
Give a formal definition of the type `ilist`*.*

Rather than take lists as a primitive notion, we may define them from a combination of sums, products, and a new concept, recursive types. The essential idea is that the types `ilist` and `unit+(int*ilist)` are *isomorphic*, meaning that there is a one-to-one correspondence between values of type `ilist` and values of the foregoing sum type. In implementation terms we may think of the correspondence "pointer chasing" — every list is a pointer to a tagged value indicating whether or not the list is empty and, if not, a pair consisting of its head and tail. (Formally, there is also a value associated with the empty list, namely the sole value of `unit` type. Since its value is predictable from the type, we can safely ignore it.) This interpretation of values of recursive type as pointers is consistent with the typical low-level implementation strategy for data structures such as lists, namely as pointers to cells allocated on the heap. However, by sticking to the more abstract viewpoint we are not committed to this representation, however suggestive it may be, but can choose from a variety of programming tricks for the sake of efficiency.

Exercise 55
Consider the type of binary trees with integers at the nodes. To what sum type would such a type be isomorphic?

This motivates the following general definition of recursive types. The first-order abstract syntax is given by the following grammar:

$$\begin{array}{lrcl} \textit{Types} & \tau & ::= & t \mid \texttt{rec}\,t\,\texttt{is}\,\tau \\ \textit{Expressions} & e & ::= & \texttt{roll(}e\texttt{)} \mid \texttt{unroll(}e\texttt{)} \\ \textit{Values} & v & ::= & \texttt{roll(}v\texttt{)} \end{array}$$

Here t ranges over a set of *type variables*, which are used to stand for the recursive type itself, in much the same way that we give a name to recursive functions to stand for the function itself. For the present we will insist that type variables are used only for this purpose; they may occur only inside of a recursive type, where they are bound by the recursive type constructor itself.

For example, the type $\tau = \texttt{rec}\,t\,\texttt{is}\,\texttt{unit+(int*}t\texttt{)}$ is the recursive type of lists of integers. It is isomorphic to its *unrolling*, the type

$$\texttt{unit+(int*}\tau\texttt{)}.$$

This is the isomorphism described informally above.

The abstract "pointers" witnessing the isomorphism are written `roll(`e`)`, which "allocates" a pointer to (the value of) e, and `unroll(`e`)`, which "chases" the pointer given by (the value of) e to recover its underlying value. This interpretation will become clearer once we have given the static and dynamic semantics of these constructs.

The static semantics of these constructs is given by the following rules:

$$\frac{\Gamma \vdash e : \{\texttt{rec}\, t\, \texttt{is}\, \tau/t\}\tau}{\Gamma \vdash \texttt{roll(}e\texttt{)} : \texttt{rec}\, t\, \texttt{is}\, \tau} \tag{19.18}$$

$$\frac{\Gamma \vdash e : \texttt{rec}\, t\, \texttt{is}\, \tau}{\Gamma \vdash \texttt{unroll(}e\texttt{)} : \{\texttt{rec}\, t\, \texttt{is}\, \tau/t\}\tau} \tag{19.19}$$

These primitive operations move back and forth between a recursive type and its unrolling.

The dynamic semantics is given by the following rules:

$$\frac{}{\texttt{unroll(roll(}v\texttt{))} \mapsto v} \tag{19.20}$$

$$\frac{e \mapsto e'}{\texttt{unroll(}e\texttt{)} \mapsto \texttt{unroll(}e'\texttt{)}} \tag{19.21}$$

$$\frac{e \mapsto e'}{\texttt{roll(}e\texttt{)} \mapsto \texttt{roll(}e'\texttt{)}} \tag{19.22}$$

Exercise 56
State and prove the soundness of this extension of MinML.

Exercise 57
Consider the definition of the type `ilist` *as a recursive type given above. Give definitions of* `nil`*,* `cons`*, and* `listcase` *in terms of the operations on recursive types, sums, and products.*

Chapter 20

Polymorphism

MinML is an *explicitly typed* language. The abstract syntax is defined to have sufficient type information to ensure that all expressions have a unique type. In particular the types of the parameters of a function must be chosen when the function is defined.

While this is not itself a serious problem, it does expose a significant weakness in the MinML type system. For example, there is no way to define a generic procedure for composing two functions whose domain and range match up appropriately. Instead we must define a separate composition operation for each choice of types for the functions being composed. Here is one composition function

```
fun _ (f:string->int):(char->string)->(string->int) is
  fun _ (g:char->string):string->int is
    fun _ (x:string):int is apply(f, apply(g, x)),
```

and here is another

```
fun _ (f:float->double):(int->float)->(int->double) is
  fun _ (g:int->float):int->double is
    fun _ (x:int):double is apply(f, apply(g, x)).
```

The annoying thing is that both versions of function composition execute the same way; they differ only in the choice of types of the functions being composed. This is rather irksome, and very quickly gets out of hand in practice. Statically typed languages have long been criticized for precisely this reason. Fortunately this inflexibility is not an inherent limitation of statically typed languages, but rather a limitation of the particular

type system we have given to MinML. A rather straightforward extension is sufficient to provide the kind of flexibility that is essential for a practical language. This extension is called *polymorphism*.

While ML has had such a type system from its inception (circa 1978), few other languages have followed suit. Notably the Java language suffers from this limitation (but the difficulty is mitigated somewhat in the presence of subtyping). Plans are in the works, however, for adding polymorphism (called *generics*) to the Java language. A compiler for this extension, called Generic Java, is already available.

20.1 Polymorphic MinML

Polymorphic MinML, or PolyMinML, is an extension of MinML with the ability to define *polymorphic functions*. Informally, a polymorphic function is a function that takes a *type* as argument and yields a *value* as result. The type parameter to a polymorphic function represents an *unknown*, or *generic*, type, which can be instantiated by applying the function to a specific type. The types of polymorphic functions are called *polymorphic types*, or *polytypes*.

A significant design decision is whether to regard polymorphic types as "first-class" types, or whether they are, instead, "second-class" citizens. Polymorphic functions in ML are second-class — they cannot be passed as arguments, returned as results, or stored in data structures. The only thing we may do with polymorphic values is to bind them to identifiers with a `val` or `fun` binding. Uses of such identifiers are automatically instantiated by an implicit polymorphic instantiation. The alternative is to treat polymorphic functions as first-class values, which can be used like any other value in the language. Here there are no restrictions on how they can be used, but you should be warned that doing so precludes using type inference to perform polymorphic abstraction and instantiation automatically.

We'll set things up for second-class polymorphism by explicitly distinguishing polymorphic types from monomorphic types. The first-class case can then be recovered by simply conflating polytypes and monotypes.

Abstract Syntax

The abstract syntax of PolyMinML is defined by the following extension to the MinML grammar:

Polytypes	σ	$::=$	$\tau \mid \forall t(\sigma)$
Monotypes	τ	$::=$	$\ldots \mid t$
Expressions	e	$::=$	$\ldots \mid \texttt{Fun}\, t\, \texttt{in}\, e\, \texttt{end} \mid \texttt{inst(}e\texttt{,}\tau\texttt{)}$
Values	v	$::=$	$\ldots \mid \texttt{Fun}\, t\, \texttt{in}\, e\, \texttt{end}$

The variable t ranges over a set of *type variables*, which are written ML-style `'a`, `'b`, and so on in examples. In the polytype $\forall t(\sigma)$ the type variable t is bound in σ; we do not distinguish between polytypes that differ only in the names of bound variables. Since the quantifier can occur only at the outermost level, in ML it is left implicit. An expression of the form $\texttt{Fun}\, t\, \texttt{in}\, e\, \texttt{end}$ is a *polymorphic function* with parameter t and body e. The variable t is bound within e. An expression of the form $\texttt{inst(}e\texttt{,}\tau\texttt{)}$ is a *polymorphic instantiation* of the polymorphic function e at *monotype* τ. Notice that we may *only* instantiate polymorphic functions with monotypes. In examples we write $f\texttt{[}\tau\texttt{]}$ for polymorphic instantiation, rather than the more verbose $\texttt{inst(}f\texttt{,}\tau\texttt{)}$.

We write $\mathrm{FTV}(\tau)$ (respectively, $\mathrm{FTV}(\sigma)$, $\mathrm{FTV}(e)$) for the set of free type variables occurring in τ (respectively, σ, e). Capture-avoiding substitution of a monotype τ for free occurrences of a type variable t in a polytype σ (resp., monotype τ', expression e) is written $\{\tau/t\}\sigma$ (resp., $\{\tau/t\}\tau'$, $\{\tau/t\}e$).

Static Semantics

The static semantics of PolyMinML is a straightforward extension to that of MinML. One significant change, however, is that we must now keep track of the scopes of type variables, as well as ordinary variables. In the static semantics of MinML a typing judgement had the form $\Gamma \vdash e : \tau$, where Γ is a context assigning types to ordinary variables. Only those variables in $\mathrm{dom}\,\Gamma$ may legally occur in e. For PolyMinML we must introduce an additional context, Δ, which is a set of type variables, those that may legally occur in the types and expression of the judgement.

The static semantics consists of rules for deriving the following two judgements:

$\Delta \vdash \tau\ \mathsf{ok}$	*τ is a well-formed type in Δ*
$\Gamma \vdash_\Delta e : \sigma$	*e is a well-formed expression of type σ in Γ and Δ*

The rules for validity of types are as follows:

$$\frac{t \in \Delta}{\Delta \vdash t \ \mathsf{ok}} \tag{20.1}$$

$$\frac{}{\Delta \vdash \mathtt{int} \ \mathsf{ok}} \tag{20.2}$$

$$\frac{}{\Delta \vdash \mathtt{bool} \ \mathsf{ok}} \tag{20.3}$$

$$\frac{\Delta \vdash \tau_1 \ \mathsf{ok} \quad \Delta \vdash \tau_2 \ \mathsf{ok}}{\Delta \vdash \tau_1 \rightarrow \tau_2 \ \mathsf{ok}} \tag{20.4}$$

$$\frac{\Delta \cup \{ t \} \vdash \sigma \ \mathsf{ok} \quad t \notin \Delta}{\Delta \vdash \forall t(\sigma) \ \mathsf{ok}} \tag{20.5}$$

The auxiliary judgement $\Delta \vdash \Gamma$ is defined by the following rule:

$$\frac{\Delta \vdash \Gamma(x) \ \mathsf{ok} \ (\forall x \in \mathrm{dom}(\Gamma))}{\Delta \vdash \Gamma \ \mathsf{ok}}. \tag{20.6}$$

The rules for deriving typing judgements $\Gamma \vdash_\Delta e : \sigma$ are as follows. We assume that $\Delta \vdash \Gamma \ \mathsf{ok}$, $\Delta \vdash \sigma \ \mathsf{ok}$, $\mathrm{FV}(e) \subseteq \mathrm{dom}(\Gamma)$, and $\mathrm{FTV}(e) \subseteq \Delta$. We give only the rules specific to PolyMinML; the remaining rules are those of MinML, augmented with a set Δ of type variables.

$$\frac{\Gamma \vdash_{\Delta \cup \{ t \}} e : \sigma \quad t \notin \Delta}{\Gamma \vdash_\Delta \mathtt{Fun}\, t \,\mathtt{in}\, e \,\mathtt{end} : \forall t(\sigma)} \tag{20.7}$$

$$\frac{\Gamma \vdash_\Delta e : \forall t(\sigma) \quad \Delta \vdash \tau \ \mathsf{ok}}{\Gamma \vdash_\Delta \mathtt{inst(}e\mathtt{,}\tau\mathtt{)} : \{\tau/t\}\sigma} \tag{20.8}$$

For example, here is the polymorphic composition function in PolyMinML:

```
Fun t in
  Fun u in
    Fun v in
      fun _(f:u->v):(t->u)->(t->v) is
        fun _(g:t->u):t->v is
          fun _(x:t):v is apply(f, apply(g, x))
```

It is easy to check that it has type

$$\forall t(\forall u(\forall v((u{\rightarrow}v){\rightarrow}(t{\rightarrow}u){\rightarrow}(t{\rightarrow}v)))).$$

We will need the following technical lemma stating that typing is preserved under instantiation:

Lemma 58 (Instantiation)
If $\Gamma \vdash_{\Delta\cup\{t\}} e : \sigma$*, where* $t \notin \Delta$*, and* $\Delta \vdash \tau$ ok*, then* $\{\tau/t\}\Gamma \vdash_{\Delta} \{\tau/t\}e : \{\tau/t\}\sigma$*.*

The proof is by induction on typing, and involves no new ideas beyond what we have already seen.

We will also have need of the following canonical forms lemma:

Lemma 59 (Canonical Forms)
If $v : \forall t(\sigma)$*, then* $v =$ `Fun` t `in` e `end` *for some* t *and* e *such that* $\emptyset \vdash_{\{t\}} e : \sigma$*.*

This is proved by a straightforward analysis of the typing rules.

Dynamic Semantics

The dynamic semantics of PolyMinML is a simple extension of that of MinML. We need only add the following two SOS rules:

$$\frac{}{\texttt{inst(Fun}\,t\,\texttt{in}\,e\,\texttt{end,}\tau\texttt{)} \mapsto \{\tau/t\}e} \tag{20.9}$$

$$\frac{e \mapsto e'}{\texttt{inst(}e\texttt{,}\tau\texttt{)} \mapsto \texttt{inst(}e'\texttt{,}\tau\texttt{)}} \tag{20.10}$$

It is then a simple matter to prove safety for this language.

Theorem 60 (Preservation)
If $e : \sigma$ *and* $e \mapsto e'$*, then* $e' : \sigma$*.*

The proof is by induction on evaluation.

Theorem 61 (Progress)
If $e : \sigma$*, then either* e *is a value or there exists* e' *such that* $e \mapsto e'$*.*

As before, this is proved by induction on evaluation.

First-Class Polymorphism

The syntax given above describes an ML-like treatment of polymorphism, *albeit* one in which polymorphic abstraction and instantiation is explicit, rather than implicit, as it is in ML. To obtain the first-class variant of PolyMinML, we simply ignore the distinction between poly- and mono-types, regarding them all as simply types. Everything else remains unchanged, including the proofs of progress and preservation.

With first-class polymorphism we may consider types such as

$$\forall t(t \to t) \to \forall t(t \to t),$$

which cannot be expressed in the ML-like fragment. This is the type of functions that accept a polymorphic function as argument and yield a polymorphic function (of the same type) as result. If f has the above type, then f`(Fun` t `in fun _(`x`:`t`):`t `is` x `end end)` is well-formed. However, the application f`(fun _(`x`:int):int is` $+(x, 1)$ `end)` is ill-formed, because the successor function does not have type $\forall t(t \to t)$. The requirement that the argument be polymorphic is a significant restriction on how f may be used!

Contrast this with the following type (which does lie within the ML-like fragment):

$$\forall t((t \to t) \to (t \to t)).$$

This is the type of polymorphic functions that, for each type t, accept a function on t and yield another function on t. If g has this type, the expression `inst(`g`,int)(succ)` is well-formed, since we first instantiate g at `int`, then apply it to the successor function.

The situation gets more interesting in the presence of data structures such as lists and reference cells. It is a worthwhile exercise to consider the difference between the types $\forall t(\sigma)$ `list` and $\forall t(\sigma$ `list`$)$ for various choices of σ. Note once again that the former type cannot be expressed in ML, whereas the latter can.

Recall the following counterexample to type soundness for the early version of ML without the so-called value restriction:

```
let
    val r : ('a -> 'a) ref = ref (fn x:'a => x)
 in
    r := (fn x:int => x+1) ; (!r)(true)
end
```

A simple check of the polymorphic typing rules reveals that this is a well-formed expression, provided that the value restriction is suspended. Of course, it "gets stuck" during evaluation by attempting to add `1` to `true`.

Using the framework of explicit polymorphism, I will argue that the superficial plausibility of this example (which led to the unsoundness in the language) stems from a failure to distinguish between these two types:

1. The type $\forall t(t \rightarrow t\ \texttt{ref})$ of polymorphic functions yielding reference cells containing a function from a type to itself.

2. The type $\forall t(t \rightarrow t)\ \texttt{ref}$ of reference cells containing polymorphic functions yielding a function from a type to itself.

(Notice the similarity to the distinctions discussed above.) For this example to be well-formed, we rely on an inconsistent reading of the example. At the point of the `val` binding we are treating `r` as a value of the latter type, namely a reference cell containing a polymorphic function. But in the body of the `let` we are treating it as a value of the former type, a polymorphic function yielding a reference cell. We cannot have it both ways at once!

To sort out the error let us make the polymorphic instantiation and abstraction explicit. Here's one rendering:

```
let
    val r : All 'a (('a -> 'a) ref) =
      Fun 'a in ref (fn x:'a => x) end
 in
    r[int] := (fn x:int => x+1) ; (!(r[bool]))(true)
end
```

Notice that we have made the polymorphic abstraction explicit, and inserted corresponding polymorphic instantiations. This example is type correct, and hence (by the proof of safety above) sound. But notice that it allocates *two* reference cells, not *one*! Recall that polymporphic functions are values, and the binding of `r` is just such a value. Each of the two instances of `r` executes the body of this function separately, each time allocating a new reference cell. Hence the unsoundness goes away!

Here's another rendering that is, in fact, ill-typed (and should be, since it "gets stuck"!).

```
let
    val r : (All 'a ('a -> 'a)) ref =
      ref (Fun 'a in fn x:'a => x end)
 in
    r := (fn x:int => x+1) ; (!r)[bool](true)
end
```

The assignment to `r` is ill-typed because the successor is not sufficiently polymorphic. The retrieval and subsequent instantiation and application is type-correct, however. If we change the program to

```
let
    val r : (All 'a ('a -> 'a)) ref =
      ref (Fun 'a in fn x:'a => x end)
 in
    r := (Fun 'a in fn x:'a => x end) ; (!r)[bool](true)
end
```

then the expression is well-typed, and behaves sanely, precisely because we have assigned to `r` a sufficiently polymorphic function.

20.2 ML-style Type Inference

ML-style type inference may be viewed as a translation from the implicitly typed syntax of ML to the explicitly-typed syntax of PolyMinML. Specifically, the type inference mechanism performs the following tasks:

- Attaching type labels to function arguments and results.
- Inserting polymorphic abstractions for declarations of polymorphic type.
- Inserting polymorphic instantiations whenever a polymorphic declared variable is used.

Thus in ML we may write

```
val I : 'a -> 'a = fn x => x
val n : int = I(I)(3)
```

This stands for the PolyMinML declarations[1]

```
val I : ∀t(t→t) = Fun t in fun _(x:t):t is x end end
val n : int = inst(I,int→int)(inst(I,int))(3)
```

Here we apply the polymorphic identity function to itself, then apply the result to `3`. The identity function is explicitly abstracted on the type of its argument and result, and its domain and range types are made explicit on the function itself. The two occurrences of `I` in the ML code are replaced by instantiations of `I` in the PolyMinML code, first at type `int→int`, the second at type `int`.

With this in mind we can now explain the "value restriction" on polymorphism in ML. Referring to the example of the previous section, the type inference mechanism of ML generates the first rendering of the example give above in which the type of the reference cell is $\forall t((t \to t)\ \texttt{ref})$. As we've seen, when viewed in this way, the example is not problematic, *provided that* polymorphic abstractions are seen as values. For in this case the two instances of `r` generate two distinct reference cells, and no difficulties arise. Unfortunately, ML does not treat polymorphic abstractions as values! Only one reference cell is allocated, which, in the absence of the value restriction, would lead to unsoundness.

Why does the value restriction save the day? In the case that the polymorphic expression is not a value (in the ML sense) the polymorphic abstraction that is inserted by the type inference mechanism *changes a non-value into a value*! This changes the semantics of the expression (as we've seen, from allocating one cell, to allocating two different cells), which violates the semantics of ML itself.[2] However, if we limit ourselves to values in the first place, then the polymorphic abstraction is only ever wrapped around a value, and no change of semantics occurs. Therefore[3], the insertion of polymorphic abstraction doesn't change the semantics, and everything is safe. The example above involving reference cells is ruled out, because the expression `ref (fn x => x)` is not a value, but such is the nature of the value restriction.

[1] We've not equipped PolyMinML with a declaration construct, but you can see from the example how this might be done.

[2] One could argue that the ML semantics is incorrect, which leads to a different language.

[3] This would need to be proved, of course.

20.3 Parametricity

Our original motivation for introducing polymorphism was to enable more programs to be written — those that are "generic" in one or more types, such as the composition function give above. The idea is that if the behavior of a function *does not* depend on a choice of types, then it is useful to be able to define such "type oblivious" functions in the language. Once we have such a mechanism in hand, it can also be used to ensure that a particular piece of code *can not* depend on a choice of types by insisting that it be polymorphic in those types. In this sense polymorphism may be used to impose restrictions on a program, as well as to allow more programs to be written.

The restrictions imposed by requiring a program to be polymorphic underlie the often-observed experience when programming in ML that if the types are correct, then the program is correct. Roughly speaking, since the ML type system is polymorphic, if a function type checks with a polymorphic type, then the strictures of polymorphism vastly cut down the set of well-typed programs with that type. Since the intended program is one these (by the hypothesis that its type is "right"), you're much more likely to have written it if the set of possibilities is smaller.

The technical foundation for these remarks is called *parametricity*. The goal of this section is to give an account of parametricity for PolyMinML. To keep the technical details under control, we will restrict attention to the ML-like (prenex) fragment of PolyMinML. It is possibly to generalize to first-class polymorphism, but at the expense of considerable technical complexity. Nevertheless we will find it necessary to gloss over some technical details, but wherever a "pedagogic fiction" is required, I will point it out. To start with, it should be stressed that the following *does not apply* to languages with mutable references!

20.3.1 Informal Discussion

We will begin with an informal discussion of parametricity based on a "seat of the pants" understanding of the set of well-formed programs of a type.

Suppose that a function value f has the type $\forall t(t \to t)$. What function could it be?

1. It could diverge when instantiated — $f\ [\tau]$ goes into an infinite loop. Since f is polymorphic, its behavior cannot depend on the choice of τ, so in fact $f\ [\tau']$ diverges for all τ' if it diverges for τ.

2. It could converge when instantiated at τ to a function g of type $\tau \to \tau$ that loops when applied to an argument v of type τ — *i.e.*, $g(v)$ runs forever. Since f is polymorphic, g must diverge on *every* argument v of type τ if it diverges on *some* argument of type τ.

3. It could converge when instantiated at τ to a function g of type $\tau \to \tau$ that, when applied to a value v of type τ returns a value v' of type τ. Since f is polymorphic, g cannot depend on the choice of v, so v' must in fact be v.

Let us call cases (1) and (2) *uninteresting*. The foregoing discussion suggests that the only *interesting* function f of type $\forall t(t \to t)$ is the polymorphic identity function.

Suppose that f is an interesting function of type $\forall t(t)$. What function could it be? A moment's thought reveals that it cannot be interesting! That is, every function f of this type must diverge when instantiated, and hence is uninteresting. In other words, there are no interesting values of this type — it is essentially an "empty" type.

For a final example, suppose that f is an interesting function of type $\forall t(t\ \texttt{list} \to t\ \texttt{list})$. What function could it be?

1. The identity function that simply returns its argument.

2. The constantly-`nil` function that always returns the empty list.

3. A function that drops some elements from the list according to a predetermined (data-independent) algorithm — *e.g.*, always drops the first three elements of its argument.

4. A permutation function that reorganizes the elements of its argument.

The characteristic that these functions have in common is that their behavior is entirely determined by the *spine* of the list, and is independent of the *elements* of the list. For example, f cannot be the function that drops all "even" elements of the list — the elements might not be numbers! The point is that the type of f is polymorphic in the element type, but reveals that the argument is a list of unspecified elements. Therefore it can only depend on the "list-ness" of its argument, and never on its contents.

In general if a polymorphic function behaves the same at every type instance, we say that it is *parametric* in that type. In PolyMinML all polymorphic functions are parametric. In Standard ML most functions are, except

those that involve *equality types*. The equality function is *not* parametric because the equality test depends on the type instance — testing equality of integers is different than testing equality of floating point numbers, and we cannot test equality of functions. Such "pseudo-polymorphic" operations are said to be *ad hoc*, to contrast them from *parametric*.

How can parametricity be exploited? As we will see later, parametricity is the foundation for data abstraction in a programming language. To get a sense of the relationship, let us consider a classical example of exploiting parametricity, the *polymorphic Church numerals*. Let N be the type $\forall t(t{\to}(t{\to}t){\to}t)$. What are the interesting functions of the type N? Given any type τ, and values $z : \tau$ and $s : \tau{\to}\tau$, the expression

$$f\,\texttt{[}\tau\texttt{]}\texttt{(}z\texttt{)}\texttt{(}s\texttt{)}$$

must yield a value of type τ. Moreover, it must behave uniformly with respect to the choice of τ. What values could it yield? The only way to build a value of type τ is by using the element z and the function s passed to it. A moment's thought reveals that the application must amount to the n-fold composition

$$s\texttt{(}s\texttt{(}\ldots s\texttt{(}z\texttt{)}\ldots\texttt{)}\texttt{)}.$$

That is, the elements of N are in 1-to-1 correspondence with the natural numbers.

Let us write $\overline{n}$ for the polymorphic function of type N representing the natural number n, namely the function

```
Fun t in
    fn z:t in
       fn s:t->t in
          s(s(... s)...))
       end
    end
end
```

where there are n occurrences of s in the expression. Observe that if we instantiate $\overline{n}$ at the built-in type `int` and apply the result to 0 and `succ`, it evaluates to the number n. In general we may think of performing an "experiment" on a value of type N by instantiating it at a type whose values will constitute the observations, the applying it to operations z and s for performing the experiment, and observing the result.

Using this we can calculate with Church numerals. Let us consider how to define the addition function on N. Given $\overline{m}$ and $\overline{n}$ of type N, we wish to compute their sum $\overline{m+n}$, also of type N. That is, the addition function must look as follows:

```
fn m:N in
   fn n:N in
      Fun t in
          fn z:t in
             fn s:t->t in
                ...
             end
          end
      end
   end
end
```

The question is: how to fill in the missing code? Think in terms of experiments. Given m and n of type N, we are to yield a value that when "probed" by supplying a type t, an element z of that type, and a function s on that type, must yield the $(m+n)$-fold composition of s with z. One way to do this is to "run" m on t, z, and s, yielding the m-fold composition of s with z, then "running" n on this value and s again to obtain the n-fold composition of s with the n-fold composition of s with z — the desired answer. Here's the code:

```
fn m:N in
   fn n:N in
      Fun t in
          fn z:t in
             fn s:t->t in
                n[t](m[t](z)(s))(s)
             end
          end
      end
   end
end
```

To see that it works, instantiate the result at τ, apply it to z and s, and observe the result.

20.3.2 Relational Parametricity (Optional)

In this section we give a more precise formulation of parametricity. The main idea is that polymorphism implies that certain equations between expressions must hold. For example, if $f : \forall t(t \to t)$, then f must be *equal to* the identity function, and if $f : N$, then f must be *equal to* some Church numeral $\overline{n}$. To make the informal idea of parametricity precise, we must clarify what we mean by equality of expressions.

The main idea is to define equality in terms of "experiments" that we carry out on expressions to "test" whether they are equal. The valid experiments on an expression are determined solely by its type. In general we say that two closed *expressions* of a type τ are equal iff either they both diverge, or they both converge to equal *values* of that type. Equality of closed values is then defined based on their type. For integers and booleans, equality is straightforward: two values are equal iff they are identical. The intuition here is that equality of numbers and booleans is directly observable. Since functions are "infinite" objects (when thought of in terms of their input/output behavior), we define equality in terms of their behavior when applied. Specifically, two functions f and g of type $\tau_1 \to \tau_2$ are equal iff whenever they are applied to equal arguments of type τ_1, they yield equal results of type τ_2.

More formally, we make the following definitions. First, we define equality of closed expressions of type τ as follows:

$$e \cong_{exp} e' : \tau \quad \textit{iff} \quad e \mapsto^* v \Leftrightarrow e' \mapsto^* v.$$

Notice that if e and e' both diverge, then they are equal expressions in this sense. For closed values, we define equality by induction on the structure of monotypes:

$$\begin{array}{rll} v \cong_{val} v' : \texttt{bool} & \textit{iff} & v = v' = \texttt{true} \textit{ or } v = v' = \texttt{false} \\ v \cong_{val} v' : \texttt{int} & \textit{iff} & v = v' = n \textit{ for some } n \geq 0 \\ v \cong_{val} v' : \tau_1 \to \tau_2 & \textit{iff} & v_1 \cong_{val} v_1' : \tau_1 \textit{ implies } v\texttt{(}v_1\texttt{)} \cong_{exp} v'\texttt{(}v_1'\texttt{)} : \tau_2 \end{array}$$

The following lemma states two important properties of this notion of equality.

Lemma 62

1. *Expression and value equivalence are reflexive, symmetric, and transitive.*

2. *Expression equivalence is a* congruence*: we may replace any sub-expression of an expression e by an equivalent sub-expression to obtain an equivalent expression.*

So far we've considered only equality of closed expressions of monomorphic type. The definition is made so that it readily generalizes to the polymorphic case. The idea is that when we quantify over a type, we are not able to say *a priori* what we mean by equality at that type, precisely because it is "unknown". Therefore we *also* quantify over all possible notions of equality to cover all possible interpretations of that type. Let us write $R : \tau \leftrightarrow \tau'$ to indicate that R is a binary relation between valeus of type τ and τ'.

Here is the definition of equality of polymorphic values:

$$v \cong_{val} v' : \forall t(\sigma) \quad \textit{iff} \quad \textit{for all } \tau \textit{ and } \tau', \textit{ and all } R : \tau \leftrightarrow \tau',\ v\,[\tau] \cong_{exp} v'\,[\tau'] : \sigma$$

where we take equality at the type variable t to be the relation R (*i.e.*, $v \cong_{val} v' : t$ iff $v\,R\,v'$).

There is one important *proviso*: when quantifying over relations, we must restrict attention to what are called *admissible* relations, a sub-class of relations that, in a suitable sense, respects computation. Most natural choices of relation are admissible, but it is possible to contrive examples that are not. The rough-and-ready rule is this: a relation is admissible iff it is closed under "partial computation". Evaluation of an expression e to a value proceeds through a series of intermediate expressions $e \mapsto e_1 \mapsto e_2 \mapsto \cdots e_n$. The expressions e_i may be thought of as "partial computations" of e, stopping points along the way to the value of e. If a relation relates corresponding partial computations of e and e', then, to be admissible, it must also relate e and e' — it cannot relate all partial computations, and then refuse to relate the complete expressions. We will not develop this idea any further, since to do so would require the formalization of partial computation. I hope that this informal discussion suffices to give the idea.

The following is Reynolds' Parametricity Theorem:

Theorem 63 (Parametricity)
If $e : \sigma$ is a closed expression, then $e \cong_{exp} e : \sigma$.

This may seem obvious, until you consider that the notion of equality between expressions of polymorphic type is very strong, requiring equivalence under *all possible* relational interpretations of the quantified type.

Using the Parametricity Theorem we may prove a result we stated informally above.

Theorem 64
If $f : \forall t(t{\rightarrow}t)$ is an interesting value, then $f \cong_{val} id : \forall t(t{\rightarrow}t)$, where id is the polymorphic identity function.

Proof: Suppose that τ and τ' are monotypes, and that $R : \tau \leftrightarrow \tau'$. We wish to show that

$$f\ [\tau] \cong_{exp} id\ [\tau'] : t{\rightarrow}t,$$

where equality at type t is taken to be the relation R.

Since f (and id) are interesting, there exists values f_τ and $id_{\tau'}$ such that

$$f\ [\tau] \mapsto^* f_\tau$$

and

$$id\ [\tau'] \mapsto^* id_{\tau'}.$$

We wish to show that

$$f_\tau \cong_{val} id_{\tau'} : t{\rightarrow}t.$$

Suppose that $v_1 \cong_{val} v_1' : t$, which is to say $v_1\, R\, v_1'$ since equality at type t is taken to be the relation R. We are to show that

$$f_\tau(\,v_1\,) \cong_{exp} id_{\tau'}(\,v_1'\,) : t$$

By the assumption that f is interesting (and the fact that id is interesting), there exists values v_2 and v_2' such that

$$f_\tau(\,v_1\,) \mapsto^* v_2$$

and

$$id_{\tau'}(\,v_1'\,) \mapsto^* v_2'.$$

By the definition of id, it follows that $v_2' = v_1'$ (it's the identity function!). We must show that $v_2\, R\, v_1'$ to complete the proof.

Now define the relation $R' : \tau \leftrightarrow \tau$ to be the set $\{\,(v, v) \mid v\, R\, v_1'\,\}$. Since $f : \forall t(t{\rightarrow}t)$, we have by the Parametricity Theorem that $f \cong_{val} f : \forall t(t{\rightarrow}t)$, where equality at type t is taken to be the relation R'. Since $v_1\, R\, v_1'$, we have by definition $v_1\, R'\, v_1$. Using the definition of equality of polymorphic type, it follows that

$$f_\tau(\,v_1\,) \cong_{exp} id_{\tau'}(\,v_1\,) : t.$$

Hence $v_2\, R\, v_1'$, as required. ■

You might reasonably wonder, at this point, what the relationship $f \cong_{val} id : \forall t(t{\rightarrow}t)$ has to do with f's execution behavior. It is a general fact, which

we will not attempt to prove, that equivalence as we've defined it yields results about execution behavior. For example, if $f : \forall t(t \to t)$, we can show that for every τ and every $v : \tau$, $f\,[\tau]\,(v)$ evaluates to v. By the preceding theorem $f \cong_{val} id : \forall t(t \to t)$. Suppose that τ is some monotype and $v : \tau$ is some closed value. Define the relation $R : \tau \leftrightarrow \tau$ by

$$v_1 \, R \, v_2 \textit{ iff } v_1 = v_2 = v.$$

Then we have by the definition of equality for polymorphic values

$$f\,[\tau]\,(v) \cong_{exp} id\,[\tau]\,(v) : t,$$

where equality at t is taken to be the relation R. Since the right-hand side terminates, so must the left-hand side, and both must yield values related by R, which is to say that both sides must evaluate to v.

Chapter 21

Data Abstraction

Data abstraction is perhaps the most fundamental technique for structuring programs to ensure their robustness over time and to facilitate team development. The fundamental idea of data abstraction is the separation of the *client* from the *implementor* of the abstraction by an *interface*. The interface is a form of "contract" between the client and implementor. It specifies the operations that may be performed on values of the abstract type by the client and, at the same time, imposes the obligation on the implementor to provide these operations with the specified functionality. By limiting the client's view of the abstract type to a specified set of operations, the interface protects the client from depending on the details of the implementation of the abstraction, most especially its representation in terms of well-known constructs of the programming language. Doing so ensures that the implementor is free to change the representation (and, correspondingly, the implementation of the operations) of the abstract type without affecting the behavior of a client of the abstraction.

The purpose of this note is to develop a rigorous account of data abstraction in an extension of PolyMinML with *existential types*. Existential types provide the fundamental linguistic mechanisms for defining interfaces, implementing them, and using the implementation in client code. Using this extension of PolyMinML we will then develop a formal treatment of representation independence based on Reynolds's Parametricity Theorem for PolyMinML. The representation independence theorem will then serve as the basis for proving the correctness of abstract type implementations using bisimulation relations.

21.1 Existential Types

21.1.1 Abstract Syntax

The syntax of PolyMinML is extended with the following constructs:

$$
\begin{array}{llcl}
\textit{Polytypes} & \sigma & ::= & \ldots \\
 & & \mid & \exists t(\sigma) \\
\textit{Expressions} & e & ::= & \ldots \\
 & & \mid & \texttt{pack}\,\tau\,\texttt{with}\,e\,\texttt{as}\,\sigma\,\texttt{end} \\
 & & \mid & \texttt{open}\,e_1\,\texttt{as}\,t\,\texttt{with}\,x\texttt{:}\sigma\,\texttt{in}\,e_2\,\texttt{end} \\
\textit{Values} & v & ::= & \ldots \\
 & & \mid & \texttt{pack}\,\tau\,\texttt{with}\,v\,\texttt{as}\,\sigma\,\texttt{end}
\end{array}
$$

The polytype $\exists t(\sigma)$ is called an *existential* type. An existential type is the *interface* of an abstract type. An *implementation* of the existential type $\exists t(\sigma)$ is a *package value* of the form $\texttt{pack}\,\tau\,\texttt{with}\,v\,\texttt{as}\,\exists t(\sigma)\,\texttt{end}$ consisting of a monotype τ together with a value v of type $\{\tau/t\}\sigma$. The monotype τ is the *representation type* of the implementation; the value v is the implementation of the operations of the abstract type. A client makes use of an implementation by *opening* it within a scope, written $\texttt{open}\,e_i\,\texttt{as}\,t\,\texttt{with}\,x\texttt{:}\sigma\,\texttt{in}\,e_c\,\texttt{end}$, where e_i is an implementation of the interface $\exists t(\sigma)$, and e_c is the client code defined in terms of an unknown type t (standing for the representation type) and an unknown value x of type σ (standing for the unknown operations).

In an existential type $\exists t(\sigma)$ the type variable t is bound in σ, and may be renamed at will to satisfy uniqueness requirements. In an expression of the form $\texttt{open}\,e_i\,\texttt{as}\,t\,\texttt{with}\,x\texttt{:}\sigma\,\texttt{in}\,e_c\,\texttt{end}$ the type variable t and the ordinary variable x are bound in e_c, and may also be renamed at will to satisfy non-occurrence requirements. As we will see below, renaming of bound variables is crucial for ensuring that an abstract type is "new" in the sense of being distinct from any other type whenever it is opened for use in a scope. This is sometimes called *generativity* of abstract types, since each occurrence of `open` "generates" a "new" type for use within the body of the client. In reality this informal notion of generativity comes down to renaming of bound variables to ensure their uniqueness in a context.

21.1.2 Correspondence With ML

To fix ideas, it is worthwhile to draw analogies between the present formalism and (some aspects of) the Standard ML module system. We have the following correspondences:

PolyMinML + Existentials	*Standard ML*
Existential type	Signature
Package	Structure, with opaque ascription
Opening a package	`open` declaration

Here is an example of these correspondences in action. In the sequel we will use ML-like notation with the understanding that it is to be interpreted in PolyMinML in the following fashion.

Here is an ML signature for a persistent representation of queues:

```
signature QUEUE =
  sig
    type queue
    val empty : queue
    val insert : int * queue -> queue
    val remove : queue -> int * queue
  end
```

This signature is deliberately stripped down to simplify the development. In particular we leave undefined the meaning of `remove` on an empty queue.

The corresponding existential type is $\sigma_q := \exists q(\tau_q)$, where

$$\tau_q := q\texttt{*}((\texttt{int*}q)\rightarrow q)\texttt{*}(q\rightarrow(\texttt{int*}q))$$

That is, the operations of the abstraction consist of a three-tuple of values, one for the empty queue, one for the insert function, and one for the remove function.

Here is a straightforward implementation of the `QUEUE` interface in ML:

```
structure QL :> QUEUE =
  struct
    type queue = int list
    val empty = nil
    fun insert (x, xs) = x::xs
    fun remove xs =
        let val (x,xs') = rev xs in (x, rev xs') end
  end
```

A queue is a list in reverse enqueue order — the last element to be enqueued is at the head of the list. Notice that we use *opaque* signature ascription to ensure that the type `queue` is hidden from the client!

The corresponding package is $e_q := \texttt{pack}\ \texttt{int list}\ \texttt{with}\ v_q\ \texttt{as}\ \sigma_q\ \texttt{end}$, where

$$v_q := \texttt{(nil,(}v_i\texttt{,}v_r\texttt{))}$$

where v_i and v_r are the obvious function abstractions corresponding to the ML code given above.

Finally, a client of an abstraction in ML might typically open it within a scope:

```
local
  open QL
in
  ...
end
```

This corresponds to writing

```
open QL as q with <n,i,r> : τq in ... end
```

in the existential type formalism, renaming variables for convenience.

21.1.3 Static Semantics

The static semantics is an extension of that of PolyMinML with rules governing the new constructs. The rule of formation for existential types is as follows:

$$\frac{\Delta \cup \{\, t \,\} \vdash \sigma\ \mathsf{ok} \quad t \notin \Delta}{\Delta \vdash \exists t(\sigma)\ \mathsf{ok}} \tag{21.1}$$

The requirement $t \notin \Delta$ may always be met by renaming the bound variable.

The typing rule for packages is as follows:

$$\frac{\Delta \vdash \tau\ \mathsf{ok} \quad \Delta \vdash \exists t(\sigma)\ \mathsf{ok} \quad \Gamma \vdash_\Delta e : \{\tau/t\}\sigma}{\Gamma \vdash_\Delta \texttt{pack}\,\tau\,\texttt{with}\,e\,\texttt{as}\,\exists t(\sigma)\,\texttt{end}} \tag{21.2}$$

The implementation, e, of the operations "knows" the representation type, τ, of the ADT.

The typing rule for opening a package is as follows:

$$\frac{\Delta \vdash \tau' \,\mathsf{ok} \quad \Gamma, x{:}\sigma \vdash_{\Delta \cup \{t\}} e_c : \tau_c \quad \Gamma \vdash_\Delta e_i : \exists t(\sigma) \quad t \notin \Delta}{\Gamma \vdash_\Delta \mathtt{open}\, e_i \,\mathtt{as}\, t \,\mathtt{with}\, x\mathtt{:}\sigma \,\mathtt{in}\, e_c \,\mathtt{end} : \tau_c} \tag{21.3}$$

This is a complex rule, so study it carefully! Two things to note:

1. The type of the client, τ_c, must not involve the abstract type t. This prevents the client from attempting to export a value of the abstract type outside of the scope of its definition.

2. The body of the client, e_c, is type checked without knowledge of the representation type, t. The client is, in effect, polymorphic in t.

As usual, the condition $t \notin \Delta$ can always be met by renaming the bound variable t of the `open` expression to ensure that it is distinct from all other active types Δ. It is in this sense that abstract types are "new"! Whenever a client opens a package, it introduces a local name for the representation type, which is bound within the body of the client. By our general conventions on bound variables, this local name may be chosen to ensure that it is distinct from any other such local name that may be in scope, which ensures that the "new" type is different from any other type currently in scope. At an informal level this ensures that the representation type is "held abstract"; we will make this intuition more precise in Section 21.2 below.

21.1.4 Dynamic Semantics

We will use structured operational semantics (SOS) to specify the dynamic semantics of existential types. Here is the rule for evaluating package expressions:

$$\frac{e \mapsto e'}{\mathtt{pack}\, \tau \,\mathtt{with}\, e \,\mathtt{as}\, \sigma \,\mathtt{end} \mapsto \mathtt{pack}\, \tau \,\mathtt{with}\, e' \,\mathtt{as}\, \sigma \,\mathtt{end}} \tag{21.4}$$

Opening a package begins by evaluating the package expressions:

$$\frac{e_i \mapsto e_i'}{\mathtt{open}\, e_i \,\mathtt{as}\, t \,\mathtt{with}\, x\mathtt{:}\sigma \,\mathtt{in}\, e_c \,\mathtt{end} \mapsto \mathtt{open}\, e_i' \,\mathtt{as}\, t \,\mathtt{with}\, x\mathtt{:}\sigma \,\mathtt{in}\, e_c \,\mathtt{end}} \tag{21.5}$$

Once the package is fully evaluated, we bind t to the representation type and x to the implementation of the operations within the client code:

$$\texttt{open}\,\texttt{pack}\,\tau\,\texttt{with}\,v\,\texttt{as}\,\sigma\,\texttt{end}\,\texttt{as}\,t\,\texttt{with}\,x\texttt{:}\sigma\,\texttt{in}\,e_c\,\texttt{end} \mapsto \{\tau, v/t, x\}e_c \tag{21.6}$$

Observe that *there are no abstract types at run time*! During execution of the client, the representation type is *fully exposed*. It is held abstract *only* during type checking to ensure that the client does not (accidentally or maliciously) depend on the implementation details of the abstraction. Once the program type checks there is no longer any need to enforce abstraction. The dynamic semantics reflects this intuition directly.

21.1.5 Safety

The safety of the extension is stated and proved as usual. The argument is a simple extension of that used for PolyMinML to the new constructs.

Theorem 65 (Preservation)
If $e : \tau$ and $e \mapsto e'$, then $e' : \tau$.

Lemma 66 (Canonical Forms)
If $v : \exists t(\sigma)$ is a value, then $v = \texttt{pack}\,\tau\,\texttt{with}\,v'\,\texttt{as}\,\exists t(\sigma)\,\texttt{end}$ for some monotype τ and some value $v' : \{\tau/t\}\sigma$.

Theorem 67 (Progress)
If $e : \tau$ then either e value or there exists e' such that $e \mapsto e'$.

21.2 Representation Independence

Parametricity is the essence of representation independence. The typing rules for `open` given above ensure that the client of an abstract type is polymorphic in the representation type. According to our informal understanding of parametricity this means that the client's behavior is in some sense "independent" of the representation type.

More formally, we say that an (admissible) relation $R : \tau_1 \leftrightarrow \tau_2$ is a *bisimulation* between the packages

$$\texttt{pack}\,\tau_1\,\texttt{with}\,v_1\,\texttt{as}\,\exists t(\sigma)\,\texttt{end}$$

and

$$\texttt{pack}\,\tau_2\,\texttt{with}\,v_2\,\texttt{as}\,\exists t(\sigma)\,\texttt{end}$$

of type $\exists t(\sigma)$ iff $v_1 \cong_{val} v_2 : \sigma$, taking equality at type t to be the relation R. The reason for calling such a relation R a bisimulation will become apparent shortly. Two packages are said to be *bisimilar* whenever there is a bisimulation between them.

Since the client e_c of a data abstraction of type $\exists t(\sigma)$ is essentially a polymorphic function of type $\forall t(\sigma \rightarrow \tau_c)$, where $t \notin \mathrm{FTV}(\tau_c)$, it follows from the Parametricity Theorem that

$$\{\tau_1, v_1/t, x\}e_c \cong_{exp} \{\tau_2, v_2/t, x\}e_c : \tau_c$$

whenever R is such a bisimulation. Consequently,

$$\texttt{open}\, e_1\, \texttt{as}\, t\, \texttt{with}\, x\texttt{:}\sigma\, \texttt{in}\, e_c\, \texttt{end} \cong_{exp} \texttt{open}\, e_2\, \texttt{as}\, t\, \texttt{with}\, x\texttt{:}\sigma\, \texttt{in}\, e_c\, \texttt{end} : \tau_c.$$

That is, the two implementations are indistinguishable by any client of the abstraction, and hence may be regarded as equivalent. This is called *Representation Independence*; it is merely a restatement of the Parametricity Theorem in the context of existential types.

This observation licenses the following technique for proving the correctness of an ADT implementation. Suppose that we have an implementation of an abstract type $\exists t(\sigma)$ that is "clever" in some way. We wish to show that it is a correct implementation of the abstraction. Let us therefore call it a *candidate* implementation. The Representation Theorem suggests a technique for proving the candidate correct. First, we define a *reference* implementation of the same abstract type that is "obviously correct". Then we establish that the reference implementation and the candidate implementation are bisimilar. Consequently, they are equivalent, which is to say that the candidate is "equally correct as" the reference implementation.

Returning to the queues example, let us take as a reference implementation the package determined by representing queues as lists. As a candidate implementation we take the package corresponding to the following ML code:

```
structure QFB :> QUEUE =
  struct
    type queue = int list * int list
    val empty = (nil, nil)
    fun insert (x, (bs, fs)) = (x::bs, fs)
    fun remove (bs, nil) = remove (nil, rev bs)
      | remove (bs, f::fs) = (f, (bs, fs))
  end
```

We will show that `QL` and `QFB` are bisimilar, and therefore indistinguishable by any client.

Define the relation R : `int list` $\leftrightarrow$ `int list*int list` as follows:

$$R = \{\, (l, (b, f)) \mid l \cong_{val} b\texttt{@rev}(f) \,\}$$

We will show that R is a bisimulation by showing that implementations of `empty`, `insert`, and `remove` determined by the structures `QL` and `QFB` are equivalent relative to R.

To do so, we will establish the following facts:

1. `QL.empty` R `QFB.empty`.
2. Assuming that $m \cong_{val} n$: `int` and $l\, R$ `(`b`,` f`)`, show that

   `QL.insert((`m`,`l`))` R `QFB.insert((`n`,(`b`,`f`)))`.

3. Assuming that $l\, R$ `(`b`,` f`)`, show that

   `QL.remove(`l`)` $\cong_{exp}$ `QFB.remove((`b`,`f`))` : `int*`t,

   taking t equality to be the relation R.

Observe that the latter two statements amount to the assertion that the operations *preserve* the relation R — they map related input queues to related output queues. It is in this sense that we say that R is a bisimulation, for we are showing that the operations from `QL` simulate, and are simulated by, the operations from `QFB`, up to the relationship R between their representations.

The proofs of these facts are relatively straightforward, given some relatively obvious lemmas about expression equivalence.

1. To show that `QL.empty` R `QFB.empty`, it suffices to show that

   `nil@rev(nil)` $\cong_{exp}$ `nil` : `int list`,

   which is obvious from the definitions of append and reverse.

2. For `insert`, we assume that $m \cong_{val} n$: `int` and $l\, R$ `(`b`,` f`)`, and prove that

   `QL.insert(`m`,`l`)` R `QFB.insert(`n`,(`b`,`f`))`.

By the definition of `QL.insert`, the left-hand side is equivalent to m`::`l, and by the definition of `QR.insert`, the right-hand side is equivalent to `(`n`::`b`,`f`)`. It suffices to show that

$$m\texttt{::}l \cong_{exp} (n\texttt{::}b)\texttt{@rev(}f\texttt{)} : \texttt{int list}.$$

Calculating, we obtain

$$\begin{array}{lll} (n\texttt{::}b)\texttt{@rev(}f\texttt{)} & \cong_{exp} & n\texttt{::}(b\texttt{@rev(}f\texttt{)}) \\ & \cong_{exp} & n\texttt{::}l \end{array}$$

since $l \cong_{exp} b$`@rev(`f`)`. Since $m \cong_{val} n$: `int`, it follows that $m = n$, which completes the proof.

3. For `remove`, we assume that l is related by R to `(`b`,`f`)`, which is to say that $l \cong_{exp} b$`@rev(`f`)`. We are to show

$$\texttt{QL.remove(}l\texttt{)} \cong_{exp} \texttt{QFB.remove((}b\texttt{,}f\texttt{))} : \texttt{int*}t,$$

taking t equality to be the relation R. Assuming that the queue is non-empty, so that the `remove` is defined, we have $l \cong_{exp} l'$`@[`m`]` for some l' and m. We proceed by cases according to whether or not f is empty. If f is non-empty, then $f \cong_{exp} n\texttt{::}f'$ for some n and f'. Then by the definition of `QFB.remove`,

$$\texttt{QFB.remove((}b\texttt{,}f\texttt{))} \cong_{exp} \texttt{(}n\texttt{,(}b\texttt{,}f'\texttt{))} : \texttt{int*}t,$$

relative to R. We must show that

$$\texttt{(}m\texttt{,}l'\texttt{)} \cong_{exp} \texttt{(}n\texttt{,(}b\texttt{,}f'\texttt{))} : \texttt{int*}t,$$

relative to R. This means that we must show that $m = n$ and $l' \cong_{exp} b$`@rev(`f'`)` : `int list`.

Calculating from our assumptions,

$$\begin{array}{rcl} l & = & l'\texttt{@[}m\texttt{]} \\ & = & b\texttt{@rev(}f\texttt{)} \\ & = & b\texttt{@rev(}n\texttt{::}f'\texttt{)} \\ & = & b\texttt{@(rev(}f'\texttt{)@[}n\texttt{])} \\ & = & (b\texttt{@rev(}f'\texttt{))@[}n\texttt{]} \end{array}$$

From this the result follows. Finally, if f is empty, then $b \cong_{exp} b'$`@[`n`]` for some b' and n. But then `rev(`b`)` $\cong_{exp}$ n`::rev(`b'`)`, which reduces to the case for f non-empty.

This completes the proof — by Representation Independence the reference and candidate implementations are equivalent.