CS 430/530
Formal Semantics

Zhong Shao

Yale University
Department of Computer Science

Polymorphism and Abstract Types

March 27, 2025

System F of Polymorphic Types

The language **F** is a variant of **T** in which we eliminate the type of natural numbers, but add, in compensation, polymorphic types:[1]

Typ	τ	$::=$	t	t	variable
			$\mathtt{arr}(\tau_1;\tau_2)$	$\tau_1 \rightarrow \tau_2$	function
			$\mathtt{all}(t.\tau)$	$\forall(t.\tau)$	polymorphic
Exp	e	$::=$	x	x	
			$\mathtt{lam}\{\tau\}(x.e)$	$\lambda\,(x:\tau)\,e$	abstraction
			$\mathtt{ap}(e_1;e_2)$	$e_1(e_2)$	application
			$\mathtt{Lam}(t.e)$	$\Lambda(t)\,e$	type abstraction
			$\mathtt{App}\{\tau\}(e)$	$e[\tau]$	type application

The statics of **F** consists of two judgment forms, the *type formation* judgment,

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

and the *typing judgment*,

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

System F Statics

The rules defining the type formation judgment are as follows:

$$\frac{}{\Delta, t\ \mathsf{type} \vdash t\ \mathsf{type}} \tag{16.1a}$$

$$\frac{\Delta \vdash \tau_1\ \mathsf{type} \quad \Delta \vdash \tau_2\ \mathsf{type}}{\Delta \vdash \mathtt{arr}(\tau_1; \tau_2)\ \mathsf{type}} \tag{16.1b}$$

$$\frac{\Delta, t\ \mathsf{type} \vdash \tau\ \mathsf{type}}{\Delta \vdash \mathtt{all}(t.\tau)\ \mathsf{type}} \tag{16.1c}$$

The rules defining the typing judgment are as follows:

$$\frac{}{\Delta\ \Gamma, x : \tau \vdash x : \tau} \tag{16.2a}$$

$$\frac{\Delta \vdash \tau_1\ \mathsf{type} \quad \Delta\ \Gamma, x : \tau_1 \vdash e : \tau_2}{\Delta\ \Gamma \vdash \mathtt{lam}\{\tau_1\}(x.e) : \mathtt{arr}(\tau_1; \tau_2)} \tag{16.2b}$$

$$\frac{\Delta\ \Gamma \vdash e_1 : \mathtt{arr}(\tau_2; \tau) \quad \Delta\ \Gamma \vdash e_2 : \tau_2}{\Delta\ \Gamma \vdash \mathtt{ap}(e_1; e_2) : \tau} \tag{16.2c}$$

$$\frac{\Delta, t\ \mathsf{type}\ \Gamma \vdash e : \tau}{\Delta\ \Gamma \vdash \mathtt{Lam}(t.e) : \mathtt{all}(t.\tau)} \tag{16.2d}$$

$$\frac{\Delta\ \Gamma \vdash e : \mathtt{all}(t.\tau') \quad \Delta \vdash \tau\ \mathsf{type}}{\Delta\ \Gamma \vdash \mathtt{App}\{\tau\}(e) : [\tau/t]\tau'} \tag{16.2e}$$

System F Statics

Lemma 16.1 (Regularity)**.** *If* $\Delta\ \Gamma \vdash e : \tau$*, and if* $\Delta \vdash \tau_i\ \mathsf{type}$ *for each assumption* $x_i : \tau_i$ *in* Γ*, then* $\Delta \vdash \tau\ \mathsf{type}$*.*

Proof By induction on rules (16.2). □

The statics admits the structural rules for a general hypothetical judgment. In particular, we have the following critical substitution property for type formation and expression typing.

Lemma 16.2 (Substitution)**.** *1. If* $\Delta, t\ \mathsf{type} \vdash \tau'\ \mathsf{type}$ *and* $\Delta \vdash \tau\ \mathsf{type}$*, then* $\Delta \vdash [\tau/t]\tau'\ \mathsf{type}$*.*

2. If $\Delta, t\ \mathsf{type}\ \Gamma \vdash e' : \tau'$ *and* $\Delta \vdash \tau\ \mathsf{type}$*, then* $\Delta\ [\tau/t]\Gamma \vdash [\tau/t]e' : [\tau/t]\tau'$*.*

3. If $\Delta\ \Gamma, x : \tau \vdash e' : \tau'$ *and* $\Delta\ \Gamma \vdash e : \tau$*, then* $\Delta\ \Gamma \vdash [e/x]e' : \tau'$*.*

System F Examples

Returning to the motivating examples from the introduction, the polymorphic identity function, I, is written

$$\Lambda(t)\,\lambda\,(x : t)\,x;$$

it has the polymorphic type

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

Instances of the polymorphic identity are written $I[\tau]$, where τ is some type, and have the type $\tau \to \tau$.

Similarly, the polymorphic composition function, C, is written

$$\Lambda(t_1)\,\Lambda(t_2)\,\Lambda(t_3)\,\lambda\,(f : t_2 \to t_3)\,\lambda\,(g : t_1 \to t_2)\,\lambda\,(x : t_1)\,f(g(x)).$$

The function C has the polymorphic type

$$\forall(t_1.\forall(t_2.\forall(t_3.(t_2 \to t_3) \to (t_1 \to t_2) \to (t_1 \to t_3)))).$$

Instances of C are obtained by applying it to a triple of types, written $C[\tau_1][\tau_2][\tau_3]$. Each such instance has the type

$$(\tau_2 \to \tau_3) \to (\tau_1 \to \tau_2) \to (\tau_1 \to \tau_3).$$

System F Dynamics

$$\frac{}{\mathtt{lam}\{\tau\}(x.e)\ \mathsf{val}} \tag{16.3a}$$

$$\frac{}{\mathtt{Lam}(t.e)\ \mathsf{val}} \tag{16.3b}$$

$$\frac{[e_2\ \mathsf{val}]}{\mathtt{ap}(\mathtt{lam}\{\tau_1\}(x.e);e_2) \longmapsto [e_2/x]e} \tag{16.3c}$$

$$\frac{e_1 \longmapsto e_1'}{\mathtt{ap}(e_1;e_2) \longmapsto \mathtt{ap}(e_1';e_2)} \tag{16.3d}$$

$$\left[\frac{e_1\ \mathsf{val} \quad e_2 \longmapsto e_2'}{\mathtt{ap}(e_1;e_2) \longmapsto \mathtt{ap}(e_1;e_2')}\right] \tag{16.3e}$$

$$\frac{}{\mathtt{App}\{\tau\}(\mathtt{Lam}(t.e)) \longmapsto [\tau/t]e} \tag{16.3f}$$

$$\frac{e \longmapsto e'}{\mathtt{App}\{\tau\}(e) \longmapsto \mathtt{App}\{\tau\}(e')} \tag{16.3g}$$

System F Type Safety

Lemma 16.3 (Canonical Forms)**.** *Suppose that* $e : \tau$ *and* e val*, then*

1. *If* $\tau = \mathtt{arr}(\tau_1; \tau_2)$*, then* $e = \mathtt{lam}\{\tau_1\}(x.e_2)$ *with* $x : \tau_1 \vdash e_2 : \tau_2$.
2. *If* $\tau = \mathtt{all}(t.\tau')$*, then* $e = \mathtt{Lam}(t.e')$ *with* t $\mathsf{type} \vdash e' : \tau'$.

Proof By rule induction on the statics. □

Theorem 16.4 (Preservation)**.** *If* $e : \tau$ *and* $e \longmapsto e'$*, then* $e' : \tau$.

Proof By rule induction on the dynamics. □

Theorem 16.5 (Progress)**.** *If* $e : \tau$*, then either* e val *or there exists* e' *such that* $e \longmapsto e'$.

Proof By rule induction on the statics. □

System F: Polymorphic Definability

The language **F** is astonishingly expressive. Not only are all (lazy) finite products and sums definable in the language, but so are all (lazy) inductive and coinductive types. Their definability is most naturally expressed using definitional equality, which is the least congruence containing the following two axioms:

$$\frac{\Delta\ \Gamma, x : \tau_1 \vdash e_2 : \tau_2 \quad \Delta\ \Gamma \vdash e_1 : \tau_1}{\Delta\ \Gamma \vdash (\lambda\,(x : \tau)\,e_2)(e_1) \equiv [e_1/x]e_2 : \tau_2} \tag{16.4a}$$

$$\frac{\Delta, t\ \mathsf{type}\ \Gamma \vdash e : \tau \quad \Delta \vdash \rho\ \mathsf{type}}{\Delta\ \Gamma \vdash \Lambda(t)\,e[\rho] \equiv [\rho/t]e : [\rho/t]\tau} \tag{16.4b}$$

In addition, there are rules omitted here specifying that definitional equality is a congruence relation (that is, an equivalence relation respected by all expression-forming operations).

System F Definability: Products

The nullary product, or unit, type is definable in **F** as follows:

$$\mathtt{unit} \triangleq \forall(r.r \to r)$$

$$\langle\rangle \triangleq \Lambda(r)\,\lambda\,(x : r)\,x$$

The identity function plays the role of the null tuple, because it is the only closed value of this type.

Binary products are definable in **F** by using encoding tricks similar to those described in Chapter 21 for the untyped λ-calculus:

$$\tau_1 \times \tau_2 \triangleq \forall(r.(\tau_1 \to \tau_2 \to r) \to r)$$

$$\langle e_1, e_2\rangle \triangleq \Lambda(r)\,\lambda\,(x : \tau_1 \to \tau_2 \to r)\,x(e_1)(e_2)$$

$$e \cdot \mathtt{l} \triangleq e[\tau_1](\lambda\,(x : \tau_1)\,\lambda\,(y : \tau_2)\,x)$$

$$e \cdot \mathtt{r} \triangleq e[\tau_2](\lambda\,(x : \tau_1)\,\lambda\,(y : \tau_2)\,y)$$

The statics given in Chapter 10 is derivable according to these definitions. Moreover, the following definitional equalities are derivable in **F** from these definitions:

$$\langle e_1, e_2\rangle \cdot \mathtt{l} \equiv e_1 : \tau_1$$

and

$$\langle e_1, e_2\rangle \cdot \mathtt{r} \equiv e_2 : \tau_2.$$

System F Definability: Sums

The nullary sum, or void, type is definable in **F**:

$$\texttt{void} \triangleq \forall(r.r)$$

$$\texttt{abort}\{\rho\}(e) \triangleq e[\rho]$$

Binary sums are also definable in **F**:

$$\tau_1 + \tau_2 \triangleq \forall(r.(\tau_1 \to r) \to (\tau_2 \to r) \to r)$$

$$\texttt{l} \cdot e \triangleq \Lambda(r)\,\lambda\,(x : \tau_1 \to r)\,\lambda\,(y : \tau_2 \to r)\,x(e)$$

$$\texttt{r} \cdot e \triangleq \Lambda(r)\,\lambda\,(x : \tau_1 \to r)\,\lambda\,(y : \tau_2 \to r)\,y(e)$$

$$\texttt{case}\, e\, \{\texttt{l} \cdot x_1 \hookrightarrow e_1 \mid \texttt{r} \cdot x_2 \hookrightarrow e_2\} \triangleq$$

$$e[\rho](\lambda\,(x_1 : \tau_1)\,e_1)(\lambda\,(x_2 : \tau_2)\,e_2)$$

provided that the types make sense. It is easy to check that the following equivalences are derivable in **F**:

$$\texttt{case}\, \texttt{l} \cdot d_1\, \{\texttt{l} \cdot x_1 \hookrightarrow e_1 \mid \texttt{r} \cdot x_2 \hookrightarrow e_2\} \equiv [d_1/x_1]e_1 : \rho$$

and

$$\texttt{case}\, \texttt{r} \cdot d_2\, \{\texttt{l} \cdot x_1 \hookrightarrow e_1 \mid \texttt{r} \cdot x_2 \hookrightarrow e_2\} \equiv [d_2/x_2]e_2 : \rho.$$

System F Definability: Natural Numbers

Because the *only* operation we can perform on a natural number is to iterate up to it, we may simply *identify* a natural number, n, with the polymorphic iterate-up-to-n function just described. Thus, we may define the type of natural numbers in **F** by the following equations:

$$\begin{aligned}
\mathtt{nat} &\triangleq \forall(t.t \to (t \to t) \to t) \\
\mathtt{z} &\triangleq \Lambda(t)\,\lambda\,(z:t)\,\lambda\,(s:t \to t)\,z \\
\mathtt{s}(e) &\triangleq \Lambda(t)\,\lambda\,(z:t)\,\lambda\,(s:t \to t)\,s(e[t](z)(s)) \\
\mathtt{iter}\{e_1; x.e_2\}(e_0) &\triangleq e_0[\tau](e_1)(\lambda\,(x:\tau)\,e_2)
\end{aligned}$$

The encodability of the natural numbers shows that **F** is *at least as expressive* as **T**. But is it *more* expressive? Yes! It is possible to show that the evaluation function for **T** is definable in **F**, even though it is not definable in **T** itself. However, the same diagonal argument given in Chapter 9 applies here, showing that the evaluation function for **F** is not definable in **F**. We may enrich **F** a bit more to define the evaluator for **F**, but as long as all programs in the enriched language terminate, we will once again have an undefinable function, the evaluation function for that extension.

System F: Parametricity Properties

A remarkable property of **F** is that polymorphic types severely constrain the behavior of their elements. We may prove useful theorems about an expression knowing *only* its type—that is, without ever looking at the code. For example, if i is any expression of type $\forall(t.t \to t)$, then it is the identity function. Informally, when i is applied to a type, τ, and an argument of type τ, it returns a value of type τ. But because τ is not specified until i is called, the function has no choice but to return its argument, which is to say that it is essentially the identity function. Similarly, if b is any expression of type $\forall(t.t \to t \to t)$, then b is equivalent to either $\Lambda(t)\,\lambda\,(x:t)\,\lambda\,(y:t)\,x$ or $\Lambda(t)\,\lambda\,(x:t)\,\lambda\,(y:t)\,y$. Intuitively, when b is applied to two arguments of a given type, the only value it can return is one of the givens.

Parametricity: *properties of a program in system F can be proved from only its types --- theorems for free*

System F: Parametricity Properties

Any function $i : \forall(t.t \to t)$ in **F** enjoys the following property:

For any type τ and any property $\mathcal{P}$ of the type τ, then if $\mathcal{P}$ holds of $x : \tau$, then $\mathcal{P}$ holds of $i[\tau](x)$.

To show that for any type τ, and any x of type τ, the expression $i[\tau](x)$ is equivalent to x, it suffices to fix $x_0 : \tau$, and consider the property $\mathcal{P}_{x_0}$ that holds of $y : \tau$ iff y is equivalent to x_0. Obviously, $\mathcal{P}$ holds of x_0 itself, and hence by the above-displayed property of i, it sends any argument satisfying $\mathcal{P}_{x_0}$ to a result satisfying $\mathcal{P}_{x_0}$, which is to say that it sends x_0 to x_0. Because x_0 is an arbitrary element of τ, it follows that $i[\tau]$ is the identity function, $\lambda\,(x : \tau)\,x$, on the type τ, and because τ is itself arbitrary, i is the polymorphic identity function, $\Lambda(t)\,\lambda\,(x : t)\,x$.

System F: Parametricity Properties

A similar argument suffices to show that the function b, defined above, is either $\Lambda(t)\,\lambda\,(x : t)\,\lambda\,(y : t)\,x$ or $\Lambda(t)\,\lambda\,(x : t)\,\lambda\,(y : t)\,y$. By virtue of its type, the function b enjoys the parametricity property

> *For any type τ and any property $\mathcal{P}$ of τ, if $\mathcal{P}$ holds of $x : \tau$ and of $y : \tau$, then $\mathcal{P}$ holds of $b[\tau](x)(y)$.*

Choose an arbitrary type τ and two arbitrary elements x_0 and y_0 of type τ. Define $\mathcal{Q}_{x_0,y_0}$ to hold of $z : \tau$ iff either z is equivalent to x_0 or z is equivalent to y_0. Clearly $\mathcal{Q}_{x_0,y_0}$ holds of both x_0 and y_0 themselves, so by the quoted parametricity property of b, it follows that $\mathcal{Q}_{x_0,y_0}$ holds of $b[\tau](x_0)(y_0)$, which is to say that it is equivalent to either x_0 or y_0. Since τ, x_0, and y_0 are arbitrary, it follows that b is equivalent to either $\Lambda(t)\,\lambda\,(x : t)\,\lambda\,(y : t)\,x$ or $\Lambda(t)\,\lambda\,(x : t)\,\lambda\,(y : t)\,y$.

Data Abstraction

Data abstraction introduces an *interface* that serves as a contract between the *client* and the *implementor* of an abstract type.

- The interface specifies what the client may rely on for its own work, and, what the implementor must provide to satisfy the contract.
- The interface isolates the client from the implementor so that each may be developed in isolation from the other.

Representation Independence:

one implementation can be replaced by another without affecting the behavior of the client, provided that the two implementations meet the same interface and that each simulates the other with respect to the operations of the interface.

Abstract Types

Data abstraction = System F + *existential types*.

- Interfaces are existential types that provide a collection of operations acting on an unspecified, or abstract, type.

- Implementations are packages, the introduction form for existential types, and clients are uses of the corresponding elimination form.

Existential types are closely connected with universal types

Representation independence is an application of the **parametricity properties** of polymorphic functions**.**

System FE = F + Existential Types

The syntax of **FE** extends **F** with the following constructs:

Typ	τ	$::=$	$\texttt{some}(t.\tau)$	$\exists(t.\tau)$	interface
Exp	e	$::=$	$\texttt{pack}\{t.\tau\}\{\rho\}(e)$	$\texttt{pack}\,\rho\,\texttt{with}\,e\,\texttt{as}\,\exists(t.\tau)$	implementation
			$\texttt{open}\{t.\tau\}\{\rho\}(e_1; t, x.e_2)$	$\texttt{open}\,e_1\,\texttt{as}\,t\,\texttt{with}\,x{:}\tau\,\texttt{in}\,e_2$	client

The statics of **FE** is given by these rules:

$$\frac{\Delta, t\ \mathsf{type} \vdash \tau\ \mathsf{type}}{\Delta \vdash \texttt{some}(t.\tau)\ \mathsf{type}} \tag{17.1a}$$

$$\frac{\Delta \vdash \rho\ \mathsf{type} \quad \Delta, t\ \mathsf{type} \vdash \tau\ \mathsf{type} \quad \Delta\ \Gamma \vdash e : [\rho/t]\tau}{\Delta\ \Gamma \vdash \texttt{pack}\{t.\tau\}\{\rho\}(e) : \texttt{some}(t.\tau)} \tag{17.1b}$$

$$\frac{\Delta\ \Gamma \vdash e_1 : \texttt{some}(t.\tau) \quad \Delta, t\ \mathsf{type}\ \Gamma, x : \tau \vdash e_2 : \tau_2 \quad \Delta \vdash \tau_2\ \mathsf{type}}{\Delta\ \Gamma \vdash \texttt{open}\{t.\tau\}\{\tau_2\}(e_1; t, x.e_2) : \tau_2} \tag{17.1c}$$

System FE Dynamics

$$\frac{[e \text{ val}]}{\texttt{pack}\{t.\tau\}\{\rho\}(e) \text{ val}} \tag{17.2a}$$

$$\left[\frac{e \longmapsto e'}{\texttt{pack}\{t.\tau\}\{\rho\}(e) \longmapsto \texttt{pack}\{t.\tau\}\{\rho\}(e')}\right] \tag{17.2b}$$

$$\frac{e_1 \longmapsto e_1'}{\texttt{open}\{t.\tau\}\{\tau_2\}(e_1; t, x.e_2) \longmapsto \texttt{open}\{t.\tau\}\{\tau_2\}(e_1'; t, x.e_2)} \tag{17.2c}$$

$$\frac{[e \text{ val}]}{\texttt{open}\{t.\tau\}\{\tau_2\}(\texttt{pack}\{t.\tau\}\{\rho\}(e); t, x.e_2) \longmapsto [\rho, e/t, x]e_2} \tag{17.2d}$$

System FE Type Safety

Lemma 17.1 (Regularity)**.** *Suppose that* $\Delta\ \Gamma \vdash e : \tau$*. If* $\Delta \vdash \tau_i\ \mathit{type}$ *for each* $x_i : \tau_i$ *in* Γ*, then* $\Delta \vdash \tau\ \mathit{type}$*.*

Proof By induction on rules (17.1), using substitution for expressions and types. □

Theorem 17.2 (Preservation)**.** *If* $e : \tau$ *and* $e \longmapsto e'$*, then* $e' : \tau$*.*

Proof By rule induction on $e \longmapsto e'$, using substitution for both expression- and type variables. □

Lemma 17.3 (Canonical Forms)**.** *If* $e : \mathtt{some}(t.\tau)$ *and* $e\ \mathsf{val}$*, then* $e = \mathtt{pack}\{t.\tau\}\{\rho\}(e')$ *for some type* ρ *and some* e' *such that* $e' : [\rho/t]\tau$*.*

Proof By rule induction on the statics, using the definition of closed values. □

Theorem 17.4 (Progress)**.** *If* $e : \tau$*, then either* $e\ \mathsf{val}$ *or there exists* e' *such that* $e \longmapsto e'$*.*

Proof By rule induction on $e : \tau$, using the canonical forms lemma. □

System FE: Data Abstraction Example

To illustrate the use of **FE**, we consider an abstract type of queues of natural numbers supporting three operations:

1. Forming the empty queue.
2. Inserting an element at the tail of the queue.
3. Removing the head of the queue, which is assumed non-empty.

The crucial property of this description is that nowhere do we specify what queues actually *are*, only what we can *do* with them. The behavior of a queue is expressed by the existential type $\exists(t.\tau)$, which serves as the interface of the queue abstraction:

$$\exists(t.\langle \texttt{emp} \hookrightarrow t, \texttt{ins} \hookrightarrow \texttt{nat} \times t \to t, \texttt{rem} \hookrightarrow t \to (\texttt{nat} \times t)\,\texttt{opt}\rangle).$$

System FE: Data Abstraction Example

An implementation of queues consists of a package specifying the representation type, together with the implementation of the associated operations in terms of that representation. Internally to the implementation, the representation of queues is known and relied upon by the operations. Here is a very simple implementation e_l in which queues are represented as lists:

$$\texttt{pack}\ \texttt{natlist}\ \texttt{with}\ \langle \texttt{emp} \hookrightarrow \texttt{nil}, \texttt{ins} \hookrightarrow e_i, \texttt{rem} \hookrightarrow e_r \rangle\ \texttt{as}\ \exists(t.\tau),$$

where

$$e_i : \texttt{nat} \times \texttt{natlist} \to \texttt{natlist} = \lambda\,(x : \texttt{nat} \times \texttt{natlist}) \ldots,$$

and

$$e_r : \texttt{natlist} \to \textbf{(nat x natlist) opt} \quad = \lambda\,(x : \texttt{natlist}) \ldots.$$

System FE: Data Abstraction Example

It is also possible to give another implementation e_p of the same interface $\exists(t.\tau)$, but in which queues are represented as pairs of lists, consisting of the "back half" of the queue paired with the reversal of the "front half." This two-part representation avoids the need for reversals on each call and, as a result, achieves amortized constant-time behavior:

$$\texttt{pack}\ \texttt{natlist} \times \texttt{natlist}\ \texttt{with}\ \langle \texttt{emp} \hookrightarrow \langle \texttt{nil}, \texttt{nil} \rangle, \texttt{ins} \hookrightarrow e_i, \texttt{rem} \hookrightarrow e_r \rangle\ \texttt{as}\ \exists(t.\tau).$$

In this case, e_i has type

$$\texttt{nat} \times (\texttt{natlist} \times \texttt{natlist}) \to (\texttt{natlist} \times \texttt{natlist}),$$

and e_r has type

$$(\texttt{natlist} \times \texttt{natlist}) \to \textbf{(nat x (natlist x natlist)) opt}$$

These operations "know" that queues are represented as values of type $\texttt{natlist} \times \texttt{natlist}$ and are implemented accordingly.

System FE Definable in System F

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

$$\texttt{pack}\,\rho\,\texttt{with}\,e\,\texttt{as}\,\exists(t.\tau) \triangleq \Lambda(u)\,\lambda\,(x : \forall(t.\tau \to u))\,x[\rho](e)$$

$$\texttt{open}\,e_1\,\texttt{as}\,t\,\texttt{with}\,x{:}\tau\,\texttt{in}\,e_2 \triangleq e_1[\tau_2](\Lambda(t)\,\lambda\,(x : \tau)\,e_2)$$

An existential is encoded as a polymorphic function taking the overall result type u as argument, followed by a polymorphic function representing the client with result type u, and yielding a value of type u as overall result. Consequently, the `open` construct simply packages the client as such a polymorphic function, instantiates the existential at the result type, τ_2, and applies it to the polymorphic client. (The translation therefore depends on knowing the overall result type τ_2 of the `open` construct.) Finally, a package consisting of a representation type ρ and an implementation e is a polymorphic function that, when given the result type u and the client x, instantiates x with ρ and passes to it the implementation e.

Representation Independence

An important consequence of parametricity is that it ensures that clients are insensitive to the representations of abstract types

Bisimilarity relates two implementations of an abstract type such that the behavior of a client is unaffected by swapping one implementation by another that is bisimilar to it.

if R is a bisimulation relation between any two implementations of the abstract type, then the client behaves identically on them.

A client c of an abstract type $\exists(t.\tau)$ has type $\forall(t.\tau \rightarrow \tau 2)$, where t does not occur free in $\tau 2$ (but may, of course, occur in τ).

The fact that t does not occur in the result type ensures:

- the behavior of the client is independent of the choice of relation between the implementations, provided that this relation R is preserved by the operations that implement it.

Representation Independence

Explaining what is a bisimulation is best done by example. Consider the existential type $\exists(t.\tau)$, where τ is the labeled tuple type

$$\langle \mathtt{emp} \hookrightarrow t, \mathtt{ins} \hookrightarrow \mathtt{nat} \times t \to t, \mathtt{rem} \hookrightarrow t \to (\mathtt{nat} \times t)\,\mathtt{opt} \rangle.$$

Theorem 48.12 ensures that if ρ and ρ' are any two closed types, and if R is a relation between expressions of these two types, then if the implementations $e : [\rho/x]\tau$ and $e' : [\rho'/x]\tau$ respect R, then $c[\rho]e$ behaves the same as $c[\rho']e'$. It remains to define when two implementations respect the relation R. Let

$$e \triangleq \langle \mathtt{emp} \hookrightarrow e_{\mathsf{m}}, \mathtt{ins} \hookrightarrow e_{\mathsf{i}}, \mathtt{rem} \hookrightarrow e_{\mathsf{r}} \rangle$$

and

$$e' \triangleq \langle \mathtt{emp} \hookrightarrow e'_{\mathsf{m}}, \mathtt{ins} \hookrightarrow e'_{\mathsf{i}}, \mathtt{rem} \hookrightarrow e'_{\mathsf{r}} \rangle.$$

Representation Independence

For these implementations to respect R means that the following three conditions hold:

1. The empty queues are related: $R(e_{\mathsf{m}}, e'_{\mathsf{m}})$.
2. Inserting the same element on each of two related queues yields related queues: if $d : \tau$ and $R(q, q')$, then $R(e_{\mathsf{i}}(d)(q), e'_{\mathsf{i}}(d)(q'))$.
3. If two queues are related, then either they are both empty, or their front elements are the same and their back elements are related: if $R(q, q')$, then either
   (a) $e_{\mathsf{r}}(q) \cong \texttt{null} \cong e'_{\mathsf{r}}(q')$, or
   (b) $e_{\mathsf{r}}(q) \cong \texttt{just}(\langle d, r \rangle)$ and $e'_{\mathsf{r}}(q') \cong \texttt{just}(\langle d', r' \rangle)$, with $d \cong d'$ and $R(r, r')$.

Representation Independence

To see how this works in practice, let us consider informally two implementations of the abstract type of queues defined earlier. For the reference implementation, we choose ρ to be the type `natlist`, and define the empty queue to be the empty list, define insert to add the given element to the head of the list, and define remove to remove the last element of the list. The code is as follows:

$$t \triangleq \texttt{natlist}$$

$$\texttt{emp} \triangleq \texttt{nil}$$

$$\texttt{ins} \triangleq \lambda\,(x : \texttt{nat})\,\lambda\,(q : t)\,\texttt{cons}(x; q)$$

$$\texttt{rem} \triangleq \lambda\,(q : t)\,\texttt{case}\,\texttt{rev}(q)\,\{\texttt{nil} \hookrightarrow \texttt{null} \mid \texttt{cons}(f; qr) \hookrightarrow \texttt{just}(\langle f, \texttt{rev}(qr)\rangle)\}.$$

Removing an element takes time linear in the length of the list, because of the reversal.

Representation Independence

For the candidate implementation, we choose ρ' to be the type $\texttt{natlist} \times \texttt{natlist}$ of pairs of lists $\langle b, f \rangle$ in which b is the "back half" of the queue, and f is the *reversal* of the "front half" of the queue. For this representation, we define the empty queue to be a pair of empty lists, define insert to extend the back with that element at the head, and define remove based on whether the front is empty. If it is non-empty, the head element is removed from it and returned along with the pair consisting of the back and the tail of the front. If it is empty, and the back is not, then we reverse the back, remove the head element, and return the pair consisting of the empty list and the tail of the now-reversed back. The code is as follows:

$$t \triangleq \texttt{natlist} \times \texttt{natlist}$$

$$\texttt{emp} \triangleq \langle \texttt{nil}, \texttt{nil} \rangle$$

$$\texttt{ins} \triangleq \lambda\,(x : \texttt{nat})\,\lambda\,(\langle bs,\ fs \rangle : t)\,\langle \texttt{cons}(x; bs),\ fs \rangle$$

$$\texttt{rem} \triangleq \lambda\,(\langle bs,\ fs \rangle : t)\,\texttt{case}\ fs\ \{\texttt{nil} \hookrightarrow e \mid \texttt{cons}(f; fs') \hookrightarrow \langle bs,\ fs' \rangle\},\ \text{where}$$

$$e \triangleq \texttt{case}\ \texttt{rev}(bs)\ \{\texttt{nil} \hookrightarrow \texttt{null} \mid \texttt{cons}(b; bs') \hookrightarrow \texttt{just}(\langle b, \langle \texttt{nil}, bs' \rangle \rangle)\}.$$

Representation Independence

To show these two implementations are bisimilar, we specify a relation **R** between the types natlist and natlist × natlist such that the two implementations satisfy the three simulation conditions given earlier.

R(l, ⟨b, f ⟩) iff **l = app(b)(rev(f))**

where app is the list append function.

Higher Kinds

The concept of type quantification naturally leads to the consideration of quantification over *type constructors*, such as `list`, which are functions mapping types to types. For example, the abstract type of queues of natural numbers considered in Section 17.4 could be generalized to an abstract *type constructor* of queues that does not fix the element type. In the notation that we shall develop in this chapter, such an abstraction is expressed by the existential type $\exists\, q :: \mathtt{T} \to \mathtt{T}.\sigma$, where σ is the labeled tuple type

$$\langle \mathtt{emp} \hookrightarrow \forall\, t :: \mathtt{T}.t,\ \mathtt{ins} \hookrightarrow \forall\, t :: \mathtt{T}.t \times q[t] \to q[t],\ \mathtt{rem} \hookrightarrow \forall\, t :: \mathtt{T}.q[t] \to (t \times q[t])\ \mathtt{opt} \rangle.$$

Language Fomega (or Fw)

The syntax of kinds of $\mathbf{F}_\omega$ is given by the following grammar:

Kind	κ	$::=$	`Type`	$\mathtt{T}$	types
			`Unit`	1	nullary product
			`Prod`$(\kappa_1;\kappa_2)$	$\kappa_1 \times \kappa_2$	binary product
			`Arr`$(\kappa_1;\kappa_2)$	$\kappa_1 \to \kappa_2$	function

The syntax of constructors of $\mathbf{F}_\omega$ is defined by this grammar:

Con	c	$::=$	u	u	variable
			`arr`	$\to$	function constructor
			`all`$\{\kappa\}$	$\forall_\kappa$	universal quantifier
			`some`$\{\kappa\}$	$\exists_\kappa$	existential quantifier
			`proj[l]`(c)	$c \cdot \mathtt{l}$	first projection
			`proj[r]`(c)	$c \cdot \mathtt{r}$	second projection
			`app`$(c_1;c_2)$	$c_1[c_2]$	application
			`unit`	$\langle\rangle$	null tuple
			`pair`$(c_1;c_2)$	$\langle c_1, c_2\rangle$	pair
			`lam`$(u.c)$	$\lambda\,(u)\,c$	abstraction

Fw Statics: Constructors & Kinds

The statics of constructors and kinds of $\mathbf{F}_\omega$ is specified by the judgment

$$\Delta \vdash c :: \kappa$$

which states that the constructor c is well-formed with kind κ. The hypotheses Δ consist of a finite set of assumptions

$$u_1 :: \kappa_1, \ldots, u_n :: \kappa_n,$$

where $n \geq 0$, specifying the kinds of the active constructor variables.

The statics of constructors is defined by the following rules:

$$\frac{}{\Delta, u :: \kappa \vdash u :: \kappa} \tag{18.1a}$$

$$\frac{}{\Delta \vdash \to :: \mathrm{T} \to \mathrm{T} \to \mathrm{T}} \tag{18.1b}$$

$$\frac{}{\Delta \vdash \forall_\kappa :: (\kappa \to \mathrm{T}) \to \mathrm{T}} \tag{18.1c}$$

$$\frac{}{\Delta \vdash \exists_\kappa :: (\kappa \to \mathrm{T}) \to \mathrm{T}} \tag{18.1d}$$

Fw Statics: Constructors & Kinds

$$\frac{\Delta \vdash c :: \kappa_1 \times \kappa_2}{\Delta \vdash c \cdot \mathtt{1} :: \kappa_1} \tag{18.1e}$$

$$\frac{\Delta \vdash c :: \kappa_1 \times \kappa_2}{\Delta \vdash c \cdot \mathtt{r} :: \kappa_2} \tag{18.1f}$$

$$\frac{\Delta \vdash c_1 :: \kappa_2 \to \kappa \quad \Delta \vdash c_2 :: \kappa_2}{\Delta \vdash c_1[c_2] :: \kappa} \tag{18.1g}$$

$$\frac{}{\Delta \vdash \langle\rangle :: 1} \tag{18.1h}$$

$$\frac{\Delta \vdash c_1 :: \kappa_1 \quad \Delta \vdash c_2 :: \kappa_2}{\Delta \vdash \langle c_1, c_2 \rangle :: \kappa_1 \times \kappa_2} \tag{18.1i}$$

$$\frac{\Delta, u :: \kappa_1 \vdash c_2 :: \kappa_2}{\Delta \vdash \lambda\,(u)\, c_2 :: \kappa_1 \to \kappa_2} \tag{18.1j}$$

Fw Statics: Constructor Equality

$$\frac{\Delta \vdash c :: \kappa}{\Delta \vdash c \equiv c :: \kappa} \tag{18.2a}$$

$$\frac{\Delta \vdash c \equiv c' :: \kappa}{\Delta \vdash c' \equiv c :: \kappa} \tag{18.2b}$$

$$\frac{\Delta \vdash c \equiv c' :: \kappa \quad \Delta \vdash c' \equiv c'' :: \kappa}{\Delta \vdash c \equiv c'' :: \kappa} \tag{18.2c}$$

$$\frac{\Delta \vdash c \equiv c' :: \kappa_1 \times \kappa_2}{\Delta \vdash c \cdot \mathtt{l} \equiv c' \cdot \mathtt{l} :: \kappa_1} \tag{18.2d}$$

$$\frac{\Delta \vdash c \equiv c' :: \kappa_1 \times \kappa_2}{\Delta \vdash c \cdot \mathtt{r} \equiv c' \cdot \mathtt{r} :: \kappa_2} \tag{18.2e}$$

$$\frac{\Delta \vdash c_1 \equiv c_1' :: \kappa_1 \quad \Delta \vdash c_2 \equiv c_2' :: \kappa_2}{\Delta \vdash \langle c_1, c_2 \rangle \equiv \langle c_1', c_2' \rangle :: \kappa_1 \times \kappa_2} \tag{18.2f}$$

Fw Statics: Constructor Equality

$$\frac{\Delta \vdash c_1 :: \kappa_1 \quad \Delta \vdash c_2 :: \kappa_2}{\Delta \vdash \langle c_1, c_2 \rangle \cdot \mathtt{l} \equiv c_1 :: \kappa_1} \tag{18.2g}$$

$$\frac{\Delta \vdash c_1 :: \kappa_1 \quad \Delta \vdash c_2 :: \kappa_2}{\Delta \vdash \langle c_1, c_2 \rangle \cdot \mathtt{r} \equiv c_2 :: \kappa_2} \tag{18.2h}$$

$$\frac{\Delta \vdash c_1 \equiv c_1' :: \kappa_2 \to \kappa \quad \Delta \vdash c_2 \equiv c_2' :: \kappa_2}{\Delta \vdash c_1[c_2] \equiv c_1'[c_2'] :: \kappa} \tag{18.2i}$$

$$\frac{\Delta, u :: \kappa \vdash c_2 \equiv c_2' :: \kappa_2}{\Delta \vdash \lambda\,(u :: \kappa)\, c_2 \equiv \lambda\,(u :: \kappa)\, c_2' :: \kappa \to \kappa_2} \tag{18.2j}$$

$$\frac{\Delta, u :: \kappa_1 \vdash c_2 :: \kappa_2 \quad \Delta \vdash c_1 :: \kappa_1}{\Delta \vdash (\lambda\,(u :: \kappa)\, c_2)[c_1] \equiv [c_1/u]c_2 :: \kappa_2} \tag{18.2k}$$

Fw Statics: Expressions & Types

The statics of expressions of $\mathbf{F}_\omega$ is defined using two judgment forms:

$\Delta \vdash \tau \ \mathsf{type}$	type formation
$\Delta\ \Gamma \vdash e : \tau$	expression formation

Here, as before, Γ is a finite set of hypotheses of the form

$$x_1 : \tau_1, \ldots, x_k : \tau_k$$

such that $\Delta \vdash \tau_i \ \mathsf{type}$ for each $1 \leq i \leq k$.

The types of $\mathbf{F}_\omega$ are the constructors of kind $\mathtt{T}$:

$$\frac{\Delta \vdash \tau :: \mathtt{T}}{\Delta \vdash \tau \ \mathsf{type}}. \tag{18.3}$$

This being the only rule for introducing types, the *only* types are the constructors of kind $\mathtt{T}$.

Definitionally equal types classify the same expressions:

$$\frac{\Delta\ \Gamma \vdash e : \tau_1 \quad \Delta \vdash \tau_1 \equiv \tau_2 :: \mathtt{T}}{\Gamma \vdash e : \tau_2}. \tag{18.4}$$

Fw Statics: Expressions & Types

The language $\mathbf{F}_\omega$ extends $\mathbf{F}$ to permit universal quantification over arbitrary kinds; the language $\mathbf{FE}_\omega$ extends $\mathbf{F}_\omega$ with existential quantification over arbitrary kinds. The statics of the quantifiers in $\mathbf{FE}_\omega$ is defined by the following rules:

$$\frac{\Delta, u :: \kappa \; \Gamma \vdash e : \tau}{\Delta \; \Gamma \vdash \Lambda(u :: \kappa)\, e : \forall\, u :: \kappa.\tau} \tag{18.5a}$$

$$\frac{\Delta \; \Gamma \vdash e : \forall\, u :: \kappa.\tau \quad \Delta \vdash c :: \kappa}{\Delta \; \Gamma \vdash e[c] : [c/u]\tau} \tag{18.5b}$$

$$\frac{\Delta \vdash c :: \kappa \quad \Delta, u :: \kappa \vdash \tau \text{ type} \quad \Delta \; \Gamma \vdash e : [c/u]\tau}{\Delta \; \Gamma \vdash \texttt{pack}\, c\, \texttt{with}\, e\, \texttt{as} \exists\, u :: \kappa.\tau : \exists\, u :: \kappa.\tau} \tag{18.5c}$$

$$\frac{\Delta \; \Gamma \vdash e_1 : \exists\, u :: \kappa.\tau \quad \Delta, u :: \kappa \; \Gamma, x : \tau \vdash e_2 : \tau_2 \quad \Delta \vdash \tau_2 \text{ type}}{\Delta \; \Gamma \vdash \texttt{open}\, e_1\, \texttt{as}\, u :: \kappa\, \texttt{with}\, x : \tau\, \texttt{in}\, e_2 : \tau_2} \tag{18.5d}$$