15 *Adding Bounded Polymorphism to $\mathcal{SOOL}$*

Adding F-bounded parametric polymorphism of the sort discussed in Section 4.1 to $\mathcal{SOOL}$ is relatively straightforward, as it is not that different from the addition of parametric polymorphism to the lambda calculus. In this chapter we provide the details of an extension, $\mathcal{PSOOL}$, of $\mathcal{SOOL}$. The reader is invited to go back and review Section 4.1 for the motivation to add F-bounded polymorphism. $\mathcal{PSOOL}$

15.1 Introducing $\mathcal{PSOOL}$

We begin by introducing the syntax of type and value expressions in $\mathcal{PSOOL}$ and then providing type-checking rules. We follow this with a few simple examples of the use of F-bounded polymorphism.

We first add a kind system like that for the polymorphic lambda calculus in Chapter 9. However, for simplicity we restrict the kinds to those that take types as arguments rather than elements of higher kinds. We represent the kind of all types by `*`.

$$\kappa \in \texttt{Kind} ::= \texttt{*} \mid \texttt{*} \Rightarrow \kappa$$

Type constructors are either types or functions that take a type as a parameter and return a constructor. We write `U::`κ to indicate that `U` is a type constructor with kind κ. Let $\mathcal{TC}$ be a set of type constants containing the type constant `TopObject`. Let $\mathcal{L}$ be a set of record labels, and $\mathcal{TI}$ a set of type identifiers. Constructor expressions of $\mathcal{PSOOL}$, $\mathcal{CONSR}_{\mathcal{PSOOL}}(\mathcal{TC}, \mathcal{L}, \mathcal{TI})$, (with their associated kinds) are defined as follows:

1. If `c` $\in \mathcal{TC}$, then `c`:: `*`.
2. If `t` $\in \mathcal{TI}$, then `t`:: `*`.

3. If T:: * and U:: κ, then `TpFunc(t).U`:: * $\Rightarrow \kappa$.
4. If T:: * and F:: * $\Rightarrow \kappa$, then F(T):: κ.
5. If T:: * and U:: *, then `ForAll(t` $<:$ `T). U`:: *.
6. ...

The elided elements in the last case consist of all of the type definitions from $\mathcal{SOOL}$, all given with kind `*`. We omit them here because they would overwhelm the new constructor expressions.

If `T` has kind `*`, then we say that `T` is a type expression of $\mathcal{PSOOL}$ (written `T` $\in \mathcal{TYPE}_{\mathcal{PSOOL}}(\mathcal{TC}, \mathcal{L}, \mathcal{TI})$). Notice that the constructor identifiers and constants are only of kind `*` (and hence will be referred to as type identifiers and constants).

Constructor expressions of the form `TpFunc(t).U` represent functions taking a type `T` to type `[T/t]U`. The expression `F(T)` represents the application of type function `F` to argument `T`.

If `T` and `U` are types, then `ForAll(t` $<:$ `T). U` is the type of polymorphic functions taking subtypes of `T` to values of type `U`.

While constructors may only take types as arguments, it is possible to obtain type functions taking several type parameters by writing them in a curried fashion, that is, defining a type function that returns another type function. An example is `TpFunc(t).TpFunc(u). t` $\rightarrow$ `u`.

As in $\Lambda^{P}_{<:}$, we include in Figure 15.1 a congruence rule, *FuncAppCong*, for simplifying constructor applications. In combination with the corresponding subtyping and type-checking rules, *Cong* $_{<:}$ and *Congruence*, this will allow us to simplify constructor applications appearing in expressions to be type-checked. (While we don't bother to write the corresponding rules, the congruence relation is reflexive, symmetric, and transitive.)

It is important not to confuse polymorphic types – those types with the form `ForAll(t` $<:$ `T). U`, and functions from types to types – those functions with kind `*`$\Rightarrow$`*`and typically written in the form `TpFunc(t).U`. Functions from types to types can be applied to a type and return a type, while a polymorphic type is just a type – the type of an expression that takes a type as a parameter and returns an element of the return type.

We now introduce the expressions of $\mathcal{PSOOL}$. New expressions include bounded polymorphic functions and their applications.

Definition 15.1.1 *The set of pre-expressions, $\mathcal{PEXP}_{\mathcal{PSOOL}}(\mathcal{EC}, \mathcal{L}, \mathcal{EI}, \mathcal{TI})$ of $\mathcal{PSOOL}$ over a set $\mathcal{EC}$ of expression constants, a set $\mathcal{L}$ of labels, a set $\mathcal{EI}$ of expres-*

FuncAppCong $$\frac{\mathtt{T::\ *}}{\mathcal{C} \vdash (\mathtt{TpFunc(t).\,U})(\mathtt{T'}) \cong [\mathtt{T'}/\mathtt{t}]\mathtt{U}}$$

Congruence $$\frac{\mathcal{C},\mathcal{E} \vdash \mathtt{M}\colon \mathtt{T} \qquad \mathcal{C} \vdash \mathtt{T} \cong \mathtt{T'}}{\mathcal{C},\mathcal{E} \vdash \mathtt{M}\colon \mathtt{T'}}$$

Cong $_{<:}$ $$\frac{\mathtt{S} \cong \mathtt{S'}, \qquad \mathtt{T'} \cong \mathtt{T}, \qquad \mathcal{C} \vdash \mathtt{S} <: \mathtt{T}}{\mathcal{C} \vdash \mathtt{S'} <: \mathtt{T'}}$$

Figure 15.1 Congruence rules for $\mathcal{PSOOL}$.

sion identifiers, and a set $\mathcal{TI}$ of type identifiers, is given by the following context-free grammar (where we assume $\mathtt{t} \in \mathcal{TI}$ *and* $\mathtt{T}, \mathtt{U} \in \mathcal{TYPE}_{\mathcal{PSOOL}}(\mathcal{TC}, \mathcal{L}, \mathcal{TI})$ *):*

$\mathtt{E} \in \mathtt{Exp}$ ::= $\ldots \mid$ `polyFunc(t <: T): U is Block` $\mid$ `E[T]`

As above, the elided expressions represent the usual expressions of $\mathcal{SOOL}$, which are omitted here.

An expression of the form `polyFunc(t <: T): U is Block` represents a polymorphic function that takes a type parameter and returns the element of type `U` determined by evaluating `Block`. The second new expression represents the application of a polymorphic function to a type parameter. Notice that `F(T)` represents the application of a type function to a type, while `E[T]` represents the application of a polymorphic function to a type.

We present a detailed example of these constructs later in this chapter. However simple examples of a type function and a polymorphic function follow.

```
PFcn = TpFunc(t).t → Integer;

polyx = polyFunc(t <: Point): t → Integer is
              function(p: t): Integer is p ⇐ getx();
```

`PFcn` is a type function that send a type `T` to the type `T → Integer`. On the other hand, `polyx` is a polymorphic function that takes a subtype, `T`, of `Point`, and a value `p` of type `T`, and returns the integer obtained by sending a `getx` message to `p`. The type of `polyx` is `ForAll(t <: Point).PFcn(t)`.

The subtyping rules are the same as those for $\mathcal{SOOL}$ except for the addition of rules for constructor functions, constructor applications, and bounded

Constructor $_{<:}$ $$\frac{\mathcal{C} \vdash \texttt{U}' <: \texttt{U}}{\mathcal{C} \vdash \texttt{TpFunc(t).U}' <: \texttt{TpFunc(t).U}}$$

ConstructorApp $_{<:}$ $$\frac{\mathcal{C} \vdash \texttt{F}::\ \texttt{*} \Rightarrow \kappa, \qquad \texttt{T}::\ \texttt{*}, \qquad \mathcal{C} \vdash \texttt{F}' <: \texttt{F}}{\mathcal{C} \vdash \texttt{F}'\texttt{(T)} <: \texttt{F(T)}}$$

BdPoly $_{<:}$ $$\frac{\mathcal{C} \cup \{\texttt{t} <: \texttt{T}\} \vdash \texttt{U}' <: \texttt{U}}{\mathcal{C} \vdash \texttt{ForAll(t} <: \texttt{T).U}' <: \texttt{ForAll(t} <: \texttt{T).U}}$$

Figure 15.2 New subtyping rules for $\mathcal{PSOOL}$.

PolyFcn $$\frac{\mathcal{C} \cup \{\texttt{t} <: \texttt{T}\}, \mathcal{E} \vdash \texttt{Block: U}}{\mathcal{C}, \mathcal{E} \vdash \texttt{polyFunc(t} <: \texttt{T): U is Block: ForAll(t} <: \texttt{T).U}}$$

where T may involve t.

PolyFcnApp $$\frac{\begin{array}{c}\mathcal{C}, \mathcal{E} \vdash \texttt{E: ForAll(t} <: \texttt{T).U} \\ \mathcal{C}, \mathcal{E} \vdash \texttt{T}' <: [\texttt{T}'/\texttt{t}]\,\texttt{T}\end{array}}{\mathcal{C}, \mathcal{E} \vdash \texttt{E[T}'\texttt{]}: [\texttt{T}'/\texttt{t}]\,\texttt{U}}$$

Figure 15.3 Typing rules for new expressions of $\mathcal{PSOOL}$.

polymorphic types. The rules, which are similar to the corresponding rules for $\Lambda^P_{<:}$, are given in Figure 15.2.

The definition of static type environments, $\mathcal{E}$, does not need to be modified from Definition 10.2.3, nor does the definition of type constraint system, $\mathcal{C}$, from Definition 12.1.1, which already included bounded types.

Type-checking rules for the new expressions are given in Figure 15.3. Because the definition of type constraints allows the constraint T on a type identifier t to involve that identifier, we may type check instances of F-bounded polymorphism.

Recall the following example of GJ code from Section 4.1.

```
interface OrderableF<T> {
   public boolean equal(T other);
   public boolean greaterThan(T other);
   public boolean lessThan(T other);
}
```

In 𝒫𝒮𝒪𝒪ℒ we model `OrderableF` as a function from types to types:

```
OrderableF = TpFunc(t).ObjectType {
     equal:t → Boolean;
     greaterThan:t → Boolean;
     lessThan:t → Boolean
 }
```

Thus `OrderableF` takes an object type parameter, `T <: TopObject`, and returns an object type with `equal`, `greaterThan`, and `lessThan` methods that take parameters of type `T`.

`OrderableF` can then be used in a class definition as follows:

```
BPOrderedList:ForAll(Elt <: OrderableF(Elt)).
                                   BPClassTp(Elt) =
   polyFunc(Elt <: OrderableF(Elt)):BPClassTp(Elt) is {
      return class({| ... |}, {| ... |})
   }
```

if `class({|...|}, {|...|})` has type `BPClassTp(Elt)`.

As with 𝒮𝒪𝒪ℒ, we write programs in a language supporting convenient abbreviations. Parametric functions returning classes will be written as parameterized classes. Thus the above examples can be written in the following more readable style:

```
OrderableF(t <: TopObject) = ObjectType {
     equal:t → Boolean;
     greaterThan:t → Boolean;
     lessThan:t → Boolean
}

class BPOrderedList(Elt <: OrderableF(Elt)) {
     return class { ... };
```

If we also have

```
SomeOrdType = ObjectType {
     equal:SomeOrdType → Boolean;
     greaterThan:SomeOrdType → Boolean;
     lessThan:SomeOrdType → Boolean;
     ...
}
```

then `new BPOrderedList(SomeOrdType)` will create a new ordered list with elements of type `SomeOrdType` if we can show that `SomeOrdType` $<:$ `BPOrderedList(SomeOrdType)`.

Interestingly, we have no way of showing directly that

$$\mathcal{C} \vdash \texttt{SomeOrdType} <: \texttt{OrderableF(SomeOrdType)}.$$

However, `OrderableF(SomeOrdType)` is congruent to

```
SmallOrdType = ObjectType {
    equal: SomeOrdType → Boolean;
    greaterThan: SomeOrdType → Boolean;
    lessThan: SomeOrdType → Boolean
}
```

and $\emptyset \vdash$ `SomeOrdType` $<:$ `SmallOrdType`. Thus by $Cong_{<:}$ it is type safe to apply `BPOrderedList` to `SomeOrdType`.

15.2 Translational semantics of $\mathcal{PSOOL}$

The translation of $\mathcal{PSOOL}$ to $\Lambda^{P}_{<:}$ is straightforward, as $\Lambda^{P}_{<:}$ has constructs corresponding to all new constructs added in $\mathcal{PSOOL}$. In order to simplify the notation we assume that restrictions have been made as suggested in Section 13.2 to ensure that class types have unique types. Thus we can define the translation by induction on the expressions rather than on their typings. As a result we will write the translation in the more notationally compact form of $\mathcal{T}_{\mathcal{C}}[\![\texttt{E}]\!]$ rather than $\mathcal{T}_{\mathcal{C}}[\![\mathcal{C}, \mathcal{E} \vdash \texttt{E}\colon \texttt{T}]\!]$.

The translation of types and type functions is given in Figure 15.4, while the translation of expressions is given in Figure 15.5. We do not bother to repeat the translation of expressions of $\mathcal{SOOL}$ as they are unchanged.

We interpret `TopObject` as an object type with no methods, but it could easily be changed to include methods of the sort contained in Java's `Object` class, though all other object type definitions would have to implicitly contain those method names.

The proof of type safety is a straightforward extension of that for $\mathcal{SOOL}$, and is left as an exercise for the reader. The proof consists of adding cases for bounded polymorphic functions and polymorphic function application. Type expressions involving function applications may be reduced using the congruence rules.

$$\begin{aligned}
\mathcal{T_C}[\![\texttt{TpFunc(t).U}]\!] &\triangleq \lambda(t <: \mathcal{T_C}[\![\texttt{T}]\!]).\,\mathcal{T_C}[\![\texttt{U}]\!]\\
\mathcal{T_C}[\![\texttt{F(T)}]\!] &\triangleq \mathcal{T_C}[\![\texttt{F}]\!](\mathcal{T_C}[\![\texttt{T}]\!])\\
\mathcal{T_C}[\![\texttt{TopObject}]\!] &\triangleq \mathcal{T_C}[\![\texttt{ObjectType}\ \{\!|\ |\!\}]\!]\\
\mathcal{T_C}[\![\texttt{t}]\!] &\triangleq t\\
\mathcal{T_C}[\![\texttt{ForAll(t <: T).U}]\!] &\triangleq \forall(t <: \mathcal{T_C}[\![\texttt{T}]\!]).\,\mathcal{T_C}[\![\texttt{U}]\!]
\end{aligned}$$

Figure 15.4 Translation of type constructors and the new types of $\mathcal{PSOOL}$ to corresponding type constructors and types in $\Lambda^{P}_{<:}$.

$$\mathcal{T_C}[\![\mathcal{C},\mathcal{E} \vdash \texttt{E[T']}\texttt{: [T'/t]U}]\!] \triangleq \mathcal{T}^X_{\mathcal{C}}[\![\mathcal{C},\mathcal{E} \vdash \texttt{E: ForAll(t <: T).U}]\!]\,[\mathcal{T}^X_{\mathcal{C}}[\![\texttt{T'}]\!]]$$

$$\begin{aligned}
&\mathcal{T_C}[\![\mathcal{C},\mathcal{E} \vdash \texttt{polyFunc(t <: T): U is Block: ForAll(t <: T).U}]\!] \triangleq\\
&\qquad\qquad \Lambda(t <: \mathcal{T_C}[\![\texttt{T}]\!]).\,\mathcal{T_C}[\![\mathcal{C}',\mathcal{E} \vdash \texttt{Block: U}]\!]
\end{aligned}$$

Figure 15.5 Translation of selected expressions of $\mathcal{PSOOL}$ to expressions in $\Lambda^{P}_{<:}$.

15.3 Summary

In this chapter we showed that the addition of F-bounded polymorphism to $\mathcal{SOOL}$ to form $\mathcal{PSOOL}$ is straightforward and raises no new issues in the translational semantics. This allows us to model object-oriented languages like GJ.

Next chapter we push the boundaries of statically typed object-oriented languages to begin investigating adding extra expressiveness similar to that found in languages like Eiffel and Beta. However, we will accomplish this while retaining a statically type-safe language.