CS522 - Programming Language Semantics

Polymorphism

Grigore Roşu
Department of Computer Science
University of Illinois at Urbana-Champaign

Types of Polymorphism

The word *polymorphism* comes from the Greek language, where it means "having many forms". In the study of programming languages, people use polymorphism to state that the same expression or operation or segment of code can be used with different types. Generally, there are three kinds of polymorphism in programming languages:

Overloading or coercion. For example, we can use "+" on related entities, such as integers, natural numbers, and/or real numbers, as well as on unrelated ones, such as strings and/or booleans. However, typically the underlying algorithms are entirely different; think, for example, of addition on integers versus addition on float number. Many algebraic specification languages, including Maude, support operator overloading. In combination with subsorting, as most of you are already aware, operator overloading can lead to quite non-trivial theoretical and practical issues. We do not intend to discuss this kind of polymorphism in this class.

Parametric/universal polymorphism. This kind of polymorphism, also encountered under the name of "generics" in the literature, will be discussed in depth shortly. The idea underlying parametric, or universal, polymorphism is that types need not be concrete, but rather have parameters which can be *instantiated* by need. While parametric types can be quite handy in many situations, they may lead to difficult technical problems, especially in the context of type inference.

Subtype polymorphism. This is mostly found and considered indispensable in object-oriented languages, because of the intrinsic need of these languages to support inheritance. We will also discuss subtype polymorphism in this course. As it is customary in discussions on subtyping, we will set up a relatively simple formal framework, extending the simply typed λ-calculus with records.

Parametric/Universal Polymorphism

To capture the essence of parametric polymorphism in a general and uniform way, we next consider an extension of typed λ-calculus with *universal types*. This extension is known in the literature under the names *polymorphic λ-calculus*, *second-order λ-calculus*, or *system F*, and represents the foundation for many works in type theory.

Interestingly, this calculus was invented independently by two famous scientists, the logician Jean-Yves Girard and the computer scientist John Reynolds.

The extra ingredients in this calculus are *type variables*, which can be universally quantified, and *type instantiations*.

The BNF syntax for types and expressions is extended as follows (as usual, we use the color *red* for the new features):

$$\mathit{TypeVar} ::= s \mid t \mid \ldots$$

$$\mathit{Type} ::= S \mid \mathit{Type} \to \mathit{Type} \mid \mathit{TypeVar} \mid (\forall\, \mathit{TypeVar})\, \mathit{Type}$$

(where S is some set of basic (constant) types)

$$\mathit{Var} ::= x \mid y \mid \ldots$$

$$\mathit{Exp} ::= \mathit{Var} \mid \mathit{Exp}\mathit{Exp} \mid \lambda\, \mathit{Var}\colon \mathit{Type}.\mathit{Exp} \mid \lambda\, \mathit{TypeVar}.\mathit{Exp} \mid \mathit{Exp}[\mathit{Type}]$$

Type variables s, t, etc., will be used essentially as parameters for universal types. These should not be confused with the basic types S (such as `bool`, `int`, etc.). A universal type is "quantified" universally by a type variable, with the intuition that it represents a collection of types, one for each particular instance of the parameter. For example, $(\forall t)\; t \to t$ represents the universal type of the (polymorphic) identity function: it can be regarded as a collection of functions, one for each instance of the type t.

Besides the usual λ-expressions, we now allow type-parametric expressions together with a means to instantiate them. Precisely, a λ-expression $\lambda t.E$ represents the λ-expression E parameterized by the type t; that means that E can freely use the type t just like any other type constant (those in S), but, however, when required by an instantiation, say $(\lambda t.E)[T]$ where T is any type, one should be able to replace each free occurrence of t in E by T. The meaning of parametric types will be formally given as usual with equations.

In this enriched context, type assignments need to be extended appropriately to consist of not only typed variables of the form $x : T$, but also of type variables of the form t:

$$\begin{aligned} \mathit{TypeAssignment} ::= \emptyset \mid \mathit{Var}\colon \mathit{Type}, \mathit{TypeAssignment} \mid \\ \mathit{TypeVar}, \mathit{TypeAssignment} \end{aligned}$$

In what follows, we shall always let $E, E', \ldots$ denote expressions, $T, T', \ldots$ types and $X, X', X_1, X_2 \ldots$ type assignments.

Typing Rules

For deriving the well-formed expressions $X \triangleright E\!:\!t$, we consider all the previous typing rules, plus:

$X, x\!:\!T \triangleright x\!:\!T$ if all the free type variables in T are in X

$$\frac{X, t \triangleright E\!:\!T}{X \triangleright \lambda t.E\!:\!(\forall t)T}$$

$$\frac{X \triangleright E\!:\!(\forall t)T}{X \triangleright E[T']\!:\!T[t \leftarrow T']}$$ if all the free type variables in T' are in X

It is intuitively clear what the “free type variables in a type” are - those that are not bound by any universal quantifier; also $T'[t \leftarrow T]$ is the type obtained from T' by replacing each free occurrence of t in T' with T. Note that, like in the case of substitution in λ-calculus, some renamings of bound variables might be necessary in order to avoid type variable captures. All these can be formally expressed:

Exercise 1 *Define formally $\mathit{Free}(T)$ and $T[t \leftarrow T']$.*

Exercise 2 *Define a type checker for System F in Maude. Your type checker should take a closed term E as input and return a type T if and only if $\emptyset \triangleright E : T$ (otherwise it can return anything).*

Equational Rules

We consider all the previous equational rules that we defined for λ-calculus, plus the following three rules giving the expected meaning to the new features. The first two rules are congruence- or (ξ)-like rules for the new syntax, while the third gives meaning to type instantiations:

$$\frac{(\forall X, t)\ E =_T E'}{(\forall X)\ \lambda\, t.E =_{(\forall t)T} \lambda\, t.E'}$$

$$\frac{(\forall X)\ E =_{(\forall t)T} E'}{(\forall X)\ E[T'] =_{T[t\leftarrow T']} E'[T']} \quad \text{if } \mathit{Free}(T') \subseteq X$$

$$(\forall X)\ (\lambda\, t.E)[T'] =_{T[t\leftarrow T']} E[t \leftarrow T'] \quad \text{if } X, t\ \triangleright\ E : T \text{ and } \mathit{Free}(T') \subseteq X$$

All the equations that can be derived using the rules above are well-typed:

Proposition 1 *If* $(\forall X)\ E =_T E'$ *is derivable with the rules above then* $X\ \triangleright\ E : T$ *and* $X\ \triangleright\ E' : T$*.*

Carrying and checking the type of equalities can be quite inconvenient in efforts to efficiently automate the applications of the equational rules above (this would be more or less equivalent to dynamic type checking). A common practice in formal definitions of typed languages is, whenever possible, to drop the subscript types of equalities and to derive instead “untyped” equations. The three rules above would then be:

$$\frac{(\forall X, t)\ E = E'}{(\forall X)\ \lambda\, t.E = \lambda\, t.E'}$$

$$\frac{(\forall X)\ E = E'}{(\forall X)\ E[T'] = E'[T']} \quad \text{if } \mathit{Free}(T') \subseteq X$$

$$(\forall X)\ (\lambda\, t.E)[T'] = E[t \leftarrow T'] \quad \text{if } \mathit{Free}(T') \subseteq X$$

When type subscripts are dropped from equations, a natural question arises: are the equations consistent with the types? While the execution of the rules above becomes much simplified, the problem with dropping the types is that one could be able to derive equalities containing expressions which are not well-typed:

Exercise 3 *Give two examples of such meaningless equalities that can be derived with the untyped equational rules above. The two examples should reflect two different problems of the resulting deduction system (more precisely, of the last two rules above).*

Fortunately, the untyped equations *preserve* the well-typed-ness, which is the main result supporting and justifying type-checking:

Proposition 2 ***(Type preservation)*** *If* $(\forall X)\ E = E'$ *is derivable with the untyped equational rules and* $X\ \triangleright\ E\!:\!T$ *for some type* T*, then* $X\ \triangleright\ E'\!:\!T$*.*

In practical implementations of programming languages, the result above says that if one wants to correctly "execute" a program E, all one needs to do is to type-check E before execution and then ignore the types during the execution as far as the equational rules above are not violated. This allows more efficient implementations, which is precisely why most compilers have built-in type-checkers as front-ends.

Different transitional semantics can be now given by orienting and restricting the applications of the equations above accordingly. Since the obtained transition relations are included in the equational derivation relation, that is, $(\forall X)\ E \to E'$ implies $(\forall X)\ E = E'$, the type preservation property also holds for the various transitional semantics.

Notice that polymorphic λ-calculus is, in some sense, an untyped (w.r.t. type variables) λ-calculus over a typed (w.r.t. usual variables) λ-calculus. For that reason, polymorphic λ-calculus is also often called *second-order typed λ-calculus*. Instead of going through the theoretical intricacies of this calculus, we shall just give some examples showing how it can be used in parameterized programming.

Some Examples

Polymorphic conditional.

Recall that in simply typed λ-calculus we had a "conditional" constant for any type t, namely $\mathrm{cond}_t : \mathrm{bool} \to t \to t \to t$. In polymorphic λ-calculus we can instead define just one constant of polymorphic type, namely $\mathrm{cond} : (\forall t)\ \mathrm{bool} \to t \to t \to t$. Given a type T, it follows by the typing rules that $\mathrm{cond}[T]$ has the type $\mathrm{bool} \to T \to T \to T$. To capture the expected meaning of the conditional, two equational rules need to be added (we only consider the untyped equations here):

$$(\forall X)\ \mathrm{cond}[T]\ \mathrm{true}\ E\ E' = E \quad \text{if } \mathit{Free}(T) \subseteq X$$

$$(\forall X)\ \mathrm{cond}[T]\ \mathrm{false}\ E\ E' = E' \quad \text{if } \mathit{Free}(T) \subseteq X$$

Exercise 4 *Does the type preservation property still hold when the polymorphic conditional is added to the language? If yes, prove it.*

If no, give a counter-example.

The drawback of polymorphic conditionals in particular, and of expressions of polymorphic type in general, is that one needs to instantiate them explicitly whenever one wants to apply them.

It is interesting to note that the conditional in Maude is also polymorphic (type "`show module TRUTH .`"), but that one does not need to instantiate it explicitly.

Exercise 5 *Would it be possible to change the definition of polymorphic λ-calculus so that one does not need to instantiate polymorphic expressions explicitly, that is, so that polymorphic expressions are instantiated automatically depending on the context in which they are used? Comment on the advantages and the disadvantages of such a language.*

Identity function.

Recall the FUN example that we used to motivate the concept of *let-polymorphism*:

```
let i = fun x -> x
in if (i true) then (i 7) else 0
```

Without let-polymorphism the program above will not type, that is, there is no way to find a type - by type inference or otherwise - to i, making useless the “polymorphic” declarations of functions.

However, in polymorphic λ-calculus, one can define the identity function explicitly polymorphic (or universal) as the λ-expression $\lambda\, t.\lambda\, x : t.x$ of universal type $(\forall t)\, t \to t$. Then the FUN program above can be given as the following λ-expression which is well-typed (has the type nat) in polymorphic λ-calculus:

$$(\lambda\, i : (\forall t)\, t \to t\, .\, \mathrm{cond}[\mathrm{nat}]\ (i[\mathrm{bool}]\ \mathrm{true})\ (i[\mathrm{nat}]\ 7)\ 0)\ (\lambda\, t.\lambda\, x : t.x)$$

Function composition.

Function composition is a standard operation in functional programming languages. One would like to generically compose two functions $f : t_1 \to t_2$ and $g : t_2 \to t_3$ to obtain a function $t_1 \to t_3$, for *any* types t_1, t_2 and t_3. The corresponding λ-expression in polymorphic λ-calculus is

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

and will type to $(\forall t_1)(\forall t_2)(\forall t_3)(t_1 \to t_2) \to (t_2 \to t_3) \to t_1 \to t_3$.

Exercise 6 *Derive the type of the function composition expression formally, using the typing rules.*

Exercise 7 *Would it make any sense to introduce the parameter types "by need", that is, to define function composition as*

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

Type this expression and comment on its advantages and disadvantages in contrast to the standard polymorphic function composition above.

On Recursion

Recall that in simply-typed λ-calculus there was no way to type an expression of the form $\lambda\, x\!:\!T.xx$, for any type T. Let us try to represent this expression within System F.

First, note that $\lambda\, x\!:\!(\forall t)\ t \to t.xx$ is not a good choice, because it would not type. Indeed, trying to type it, we need to find some type T such that $x\!:\!(\forall t)\ t \to t\ \triangleright\ xx : T$; then the type of the entire expression would be $((\forall t)\ t \to t) \to T$. Trying to type the expression xx, one gets the type constraint $((\forall t)\ t \to t) \to T = (\forall t)\ t \to t$, which, unfortunately, does not admit a solution even in the enriched context of polymorphic types.

The solution is to use a type instantiation in order to change the type of the first x to one which can be applied on the second x: the expression $\lambda x : (\forall t)\ t \to t\ .\ x[(\forall t)\ t \to t]\ x$, say E, will type to $((\forall t)\ t \to t) \to ((\forall t)\ t \to t)$.

Note, however, that we cannot type anything similar to the untyped ω combinator, namely $(\lambda x.xx)(\lambda x.xx)$, or to the untyped fixed-point operators. As a matter of fact, we cannot simulate recursion in System F without μ.

Exercise 8 *Try, and fail, to simulate recursion in system F. You should see that, despite the fact that one may use instantiation to type expressions previously untypable in simply typed λ-calculus, one actually cannot use polymorphism for dealing with recursion.*

For that reason, like we did in simply typed λ-calculus, to deal with recursion we extend our calculus with the usual μ-abstraction. Assume the same usual typing/equational rules for μ.

More Examples

Polymorphic lists.

Recall that one of the interesting aspects of typing FUN programs in CS422 was the fact that lists were polymorphic. Because of that reason, we were able to regard, for example, the empty list as both a list of integers and as a list of booleans.

Polymorphic λ-calculus supports polymorphic lists very naturally. In what follows we add the list type construct, give the signature of the usual list operators, and show how polymorphic lists can be used in practice. Here we are not concerned with how lists are represented, nor with how the list operators are defined: we just assume them given. We will later discuss Church lists, showing that all the list operators can be defined from more basic principles.

Let us first add the type construct for lists:

$$\mathit{Type} = \ldots \mid \mathsf{list}\ \ \mathit{Type}$$

The usual operators on lists can be now given as just constants of universal types:

$$\mathrm{nil} : (\forall t)\ \mathsf{list}\ t$$

$$\mathrm{cons} : (\forall t)\ t \rightarrow \mathsf{list}\ t \rightarrow \mathsf{list}\ t$$

$$\mathrm{nil?} : (\forall t)\ \mathsf{list}\ t \rightarrow \mathrm{bool}$$

$$\mathrm{car} : (\forall t)\ \mathsf{list}\ t \rightarrow t$$

$$\mathrm{cdr} : (\forall t)\ \mathsf{list}\ t \rightarrow \mathsf{list}\ t$$

Supposing these list operations already defined, let us define a polymorphic “map” operator, which takes as arguments a list and a function and applies that function on each element of the list:

$$
\begin{array}{l}
\lambda\, t_1 \;.\; \lambda\, t_2 \;.\; \lambda\, f \;:\; t_1 \to t_2 . \\
\qquad \mu\, m \;:\; \mathrm{list}\ t_1 \to \mathrm{list}\ t_2 \;. \\
\qquad\qquad \lambda\, l : \mathrm{list}\ t_1 . \\
\qquad\qquad\qquad \mathrm{cond}[\mathrm{list}\ t_2]\ (\mathrm{nil?}[t_1]\ l) \\
\qquad\qquad\qquad\qquad (\mathrm{nil}[t_2]) \\
\qquad\qquad\qquad\qquad (\mathrm{cons}[t_2]\ (f\ (\mathrm{car}[t_1]\ l))\ (m\ (\mathrm{cdr}[t_1]\ l)))
\end{array}
$$

Exercise 9 *Derive the type of the “map” expression above using the formal type inference rules.*

Exercise 10 *Define and type a “reverse” on polymorphic lists.*

Church booleans.

Recall the Church booleans from untyped λ-calculus. They already had a "polymorphic" intuitive meaning: true and false were the first and the second projection functions, respectively, expecting two arguments of the same "type". With the machinery of polymorphic λ-calculus, we can take the "boolean" type to be the following universal type:

$$\mathrm{bool}_\lambda := (\forall t)\ t \to t \to t$$

and the two boolean constants to be:

$$\mathrm{true} := \lambda\, t.\lambda\, x\!:\!t.\lambda\, y\!:\!t.x$$
$$\mathrm{false} := \lambda\, t.\lambda\, x\!:\!t.\lambda\, y\!:\!t.y$$

Note that the two expressions above have indeed the type bool_λ . The logical operators can now be naturally defined. For instance, "not" can be defined as $\lambda\, b\!:\!\mathrm{bool}_\lambda.\lambda\, t.\lambda\, x\!:\!t.\lambda\, y\!:\!t.b[t]\, y\, x$.

Exercise 11 *Define and type the other Church boolean operators.*

Church numerals.

Recall that, in untyped λ-calculus, Church numerals were characterized by their latent application to a "successor function" s and a "zero" element z, which would result in applying s to z a certain number of times. Therefore, the expected type of s would be $t \to t$, where t is the type of z. Thus we can define the polymorphic type of Church numerals as

$$\mathrm{nat}_\lambda := (\forall t)\ (t \to t) \to t \to t$$

Number n can then be encoded as the expression of type nat_λ

$$n_\lambda\ := \lambda t\ .\ \lambda s{:}t \to t\ .\ \lambda z{:}t\ .\ s(s ... (s\ z) ...),$$

with n applications of s. All the usual arithmetical operations on numbers can now be defined. For instance,

$$\begin{aligned}&\mathrm{succ}_{\ \lambda}\ := \lambda n{:}\mathrm{nat}_\lambda\ .\ \lambda t\ .\ \lambda s{:}t \to t\ .\ \lambda z{:}t\ .\ s\ (n[t]\ s\ z)\\ &+_\lambda\ := \lambda n{:}\mathrm{nat}_\lambda\ .\ \lambda m{:}\mathrm{nat}_\lambda\ .\ \lambda t\ .\ \lambda s{:}t \to t\ .\ \lambda z{:}t\ .\ n[t]\ s\ (m[t]\ s\ z).\end{aligned}$$

Exercise 12 *Define the polymorphic multiplication and power.*

Church lists.

Following a similar idea as for Church numerals, one can define Church lists. A list is regarded through its latent application to a "binary operation" f and an "initial value" v, which would result in iteratively applying the binary operation as follows: first to the last element of the list and v, then to the last but one element and the previous result, and so on, until the list is exhausted. For instance, the list $[a, b, c]$ is encoded as $\lambda f \, . \, \lambda v \, . \, f \; a \; (f \; b \; (f \; c \; v))$.

For each type t, let us define the polymorphic type

$$\text{list } t := (\forall p) \; (t \to p \to p) \to p \to p$$

Note, however, that we want the list operations to be themselves polymorphic. For example, we want the type of cons to be $(\forall t) \; t \to \text{list } t \to \text{list } t$. We can now define the usual list operators quite naturally:

$\text{nil} := \lambda\, t \,.\, \lambda\, p \,.\, \lambda\, f \!:\! t \to p \to p \,.\, \lambda\, v \!:\! p \,.\, v$

(the type of nil is $(\forall t)$ list t),

$$\begin{aligned}\text{cons} := \lambda\, t \,.\, \lambda\, \text{head} \!:\! t \,.\, \lambda\, \text{tail} \!:\! \text{list } t \,. \\ \lambda\, p \,.\, \lambda\, f \!:\! t \to p \to p \,.\, \lambda\, v \!:\! p \,. \\ f \text{ head } (\text{tail}[p]\ f\ v)\end{aligned}$$

(the type of cons is $(\forall t)\ t \to$ list $t \to$ list t),

$\text{nil?} := \lambda\, t \,.\, \lambda\, l \!:\! \text{list } t \,.\, l[\text{bool}_\lambda\,]\ (\lambda\, x \!:\! t \,.\, \lambda\, y \!:\! \text{bool}_\lambda\ .\, \text{false})\ \text{true}$

(the type of nil? is $(\forall t)$ list $t \to$ bool_λ),

$\text{car} := \lambda\, t \,.\, \lambda\, l \!:\! \text{list } t \,.\, l[t]\ (\lambda\, x \!:\! t \,.\, \lambda\, y \!:\! t \,.\, x)\ (\text{error}[t])$

(the type of car is $(\forall t)$ list $t \to t$)

Note that car is intended to be a *partial* function, undefined on the empty list; hence we introduced a polymorphic constant error, of type $(\forall t)t$. This constant is also useful for any other partial function definitions. Some prefer to define “error for type t” as an infinite recursion, for example $\mu x : t.x$; in other words, they either replace each instance error$[t]$ by $\mu x : t\ .\ x$, or otherwise add an equation error $= \lambda t \,.\, \mu x\!:\!t \,.\, x$. The drawback of this adhoc convention is that the evaluation of expressions applying car *will not terminate* under call-by-value evaluation strategy.

Exercise 13 *Define and formally type the* cdr *operator.*

Giving System F an Executable Semantics

Let us now focus on the original formulation of System F, namely the one without recursion (μ). Let us orient all the equations left-to-right, thus obtaining a transitional semantics ($\rightarrow$) of System F. One of the most famous results of System F is the following:

> ***Theorem (Girard).*** *(very hard) In System F without μ, the transition relation $\rightarrow$ is confluent and terminates.*

Exercise 14 *Define System F equationally in Maude. Your definition should take a polymorphic λ-expression and evaluate it to its unique normal form.*
Hint. *You can use either the provided generic substitution, or, alternatively, can use the de Bruijn technique to properly avoid variable captures; note that variable captures can now appear both at the λ-expression level and at the type level.*

Type inference/reconstruction

The problem of *type inference* can be stated in the polymorphic λ-calculus framework as follows.

> *Given any untyped λ-expression E, is it the case that E is typable in System F?*

In other words, is there any polymorphic λ-expression E_F in System F such that $\mathsf{erase}(E_F) = E$? Here, the operator erase just loses any type information and can be formally defined as follows:

$$\begin{aligned}
&\mathsf{erase}(x) = x,\\
&\mathsf{erase}(\lambda\, x\!:\!T.E) = \lambda\, x.\mathsf{erase}(E),\\
&\mathsf{erase}(E_1E_2) = \mathsf{erase}(E_1)\mathsf{erase}(E_2),\\
&\mathsf{erase}(\lambda\, t.E) = \mathsf{erase}(E),\\
&\mathsf{erase}(E[T]) = \mathsf{erase}(E)
\end{aligned}$$

This problem, open for more than 20 years, was finally proven to be *undecidable*. Algorithms were developed to partially solve this problem, some of them based on Huet's partial algorithms for *higher-order unification* (another undecidable problem).

Subtype Polymorphism

We next focus on the other major approach to polymorphism, namely *subtype polymorphism*, which is mostly encountered in the context of modern object-oriented programming languages.

To stay focused on the major aspects of subtype polymorphism, we here introduce a very simply λ-calculus language extended with various features that reflect most of the interesting issues related to subtype polymorphism encountered in other languages or formalisms.

Simply Typed λ-Calculus with Records

We extend simply typed λ-Calculus by adding *records*. Let *Field* be a countably infinite set, disjoint from *Var*, and let us extend types and expressions as follows:

$$\mathit{Type} ::= S \mid \mathit{Type} \to \mathit{Type} \mid \{\mathit{Field}\colon \mathit{Type}, \ldots, \mathit{Field}\colon \mathit{Type}\}.$$

$$\mathit{Exp} ::= \ \mathit{Var} \mid \mathit{Exp}\ \mathit{Exp} \mid \lambda \mathit{Var}\colon \mathit{Type}.\mathit{Exp} \mid$$
$$\{\mathit{Field} = \mathit{Exp}, \ldots, \mathit{Field} = \mathit{Exp}\} \mid \mathit{Exp}.\mathit{Field}$$

Thus a record type is a set of typed attributes, each represented as a $(\mathit{Field}, \mathit{Type})$ pair. For example, we can declare a record type, say $person$, as $\{name : string, age : nat, height : nat, \ldots\}$. Two kinds of expressions are introduced for dealing with records:

- *record expressions* (written like sets of equalities, assigning expressions to the fields of a record), and

- *field accessing* expressions.

The first one creates a record by assigning a value to every attribute, while the second fetches the value of an attribute of a record. For instance, the expression

$$\{name = \text{‘}John\ Smith\text{’},\ age = 27,\ height = 180,\ \ldots\ \}$$

defines a record variable, say *john*. We can get the values of its fields with expressions like *john.name*, *john.age*, etc.

Records inherently bring the issue of *subtyping*. Intuitively, if a context requires a record R of a certain kind, it should be the case that a record R' having *more information* than needed be still suitable for that context. Thus we would like to be able to write

$$(\lambda\, p : \{age : nat\}\ .\ (p.age))$$
$$\{name{=}\text{‘}John\ Smith\text{’},\ age{=}27,\ height{=}180,\ \ldots\}$$

However, with the previous typing system, this is not allowed since the argument type is not *exactly the same* as the parameter type. To avoid this kind of unnecessary constraints, but still maintain a rigorous typing discipline, we introduce the following important notion of subtyping, first intuitively and then rigorously:

> *Intuitive definition of subtyping:* Given two types t_1 and t_2, we say that t_1 *is a subtype of* t_2, written $t_1 \leq t_2$, iff t_1 *has at least the same information as* t_2, or, in other words, a value of type t_1 can be used wherever a value of type t_2 is expected; one also may say that t_1 *is more concrete than* t_2, or that t_2 *is more general than* t_1.

Subtyping Rules

Now we set up the rules that will allow us to formally derive subtype relationships of the form $t_1 \leq t_2$, which will be further needed to define the typing system:

(≤-reflexivity) $\quad t \leq t$

(≤-transitivity) $$\frac{t_1 \leq t \qquad t \leq t_2}{t_1 \leq t_2}$$

(≤-arrow) $$\frac{t_2 \leq t_1 \qquad t_1' \leq t_2'}{t_1 \to t_1' \leq t_2 \to t_2'}$$

(≤-record) $$\frac{t_1 \leq t_1' \qquad \ldots \qquad t_m \leq t_m'}{\{l_1 : t_1, \ldots, l_n : t_n\} \leq \{l_1 : t_1', \ldots, l_m : t_m'\}} \quad \text{when } m \leq n$$

The first two rules are clear. To understand the (≤-arrow) rule, let us suppose a context which expects a value of type $t_2 \to t_2'$ but actually receives a value V of type $t_1 \to t_1'$. In a presumptive later use, V might be applied to an argument W of type t_2. To assure that V can handle W, one needs that every inquiry that V makes to its argument (expected to be of type t_1) be answered by W, i.e., that W provides at least as much information as a value of type t_1; thus $t_2 \leq t_1$. Furthermore, the result of applying V to W is a value of type t_1', while a value of type t_2' would be expected; thus $t_1' \leq t_2'$.

The rule (≤-record) says two important things: first, that the subtype record must include the fields of the supertype record, and second, that the types of those fields in the subtype record are subtypes of the types of those fields in the supertype record. Both these facts are needed in order for a value of subtype record to be used in contexts where values of supertype record are expected.

The next two properties of the subtype relation say that one can only derive meaningful subtypings. These properties may be used later to prove important properties of type systems supporting subtyping.

Exercise 15 *If* $t \leq t_2 \to t'_2$ *then* t *has the form* $t_1 \to t'_1$ *such that* $t_2 \leq t_1$ *and* $t'_1 \leq t'_2$*.*

Exercise 16 *If* $t \leq \{l_1 : t'_1, \ldots, l_m : t'_m\}$ *then* t *has the form* $\{l_1 : t_1, \ldots, l_n : t_n\}$ *with* $m \leq n$ *and* $t_i \leq t'_i$*,* $i \in \{1, \ldots, m\}$*.*

Type System

Building upon the subtype relation formally defined above, we can now give subtype-flexible rules for deriving types:

(subsumption) $$\frac{X \triangleright E : t_1}{X \triangleright E : t_2} \qquad \text{when } t_1 \leq t_2$$

(record) $$\frac{X \triangleright E_1 : t_1 \quad \ldots \quad X \triangleright E_n : t_n}{X \triangleright \{l_1 = E_1, \ldots, l_n = E_n\} : \{l_1 : t_1, \ldots, l_n : t_n\}}$$

(field access) $$\frac{X \triangleright E : \{l_1 : t_1, \ldots, l_n : t_n\}}{X \triangleright E.l_j : t_j} \qquad \text{when } j \in \{1, \ldots, n\}$$

(subsumption) allows us to "lift" the type of an expression to any supertype, thus formally justifying our informal claim that "expressions of type t_1 can be used in any context where expressions of supertype t_2 are expected".

(record) allows us to derive a type of a record from the names and the types of its fields, while (field access) allows us to derive the type of a field once the type of the entire record is known.

Proposition 3 *If* $X \triangleright (\lambda x\!:\!t.E)\!:\!t_1 \rightarrow t_2$ *then* $t_1 \leq t$ *and* $X, x\!:\!t \triangleright E\!:\!t_2$*.*

Proposition 4 ***(Substitution.)*** *If* $X, x\!:\!s \triangleright E\!:\!t$ *and* $X \triangleright F\!:\!s$ *then* $X \triangleright E[x \leftarrow F]\!:\!t$*.*

Equational Rules

The following natural equational rules are added to those of simply-typed λ-calculus. We here assume the untyped variants of equational rules, which, as usual, will rise the question of type preservation:

$$(\forall X)\ \{l_1 = E_1, \ldots, l_n = E_n\}.l_i = E_i \text{ for all } i \in \overline{1,n}$$

$$\frac{(\forall X)\ E = E'}{(\forall X)\ E.l = E'.l} \text{ for any field } l$$

$$\frac{(\forall X)\ E_n = E'_n}{(\forall X)\ \{(l_i = E_i)_{i=\overline{1,n-1}}, l_n = E_n\} = \{(l_i = E_i)_{i=\overline{1,n-1}}, l_n = E'_n\}}$$

Exercise 17 *Using the rules for typing and for equation derivation above, show formally that the expression*

$$(\lambda\, p\!:\!\{age\!:\!\mathrm{nat}\}\ .\ (p.age))$$
$$\{name\ =\ \text{‘John Smith’},\ age\ =\ 27,\ height\ =\ 180...\}$$

types to nat *and is equal to 27.*

Proposition 5 ***(Type preservation.)*** *If* $(\forall X)\ E = E'$ *is derivable and* $X\ \triangleright\ E\!:\!t$ *then* $X\ \triangleright\ E'\!:\!t$.

As an operational consequence of the property above, we obtain the so-called “subject reduction” property for the transition relation $\rightarrow$ obtained by orienting the equations:

Corollary 1 *If* $X\ \triangleright\ E\!:\!t$ *and* $E \rightarrow E'$ *then* $X\ \triangleright\ E'\!:\!t$.

Subtyping and Other Language Features

Subtyping occurs many places in computer science. Therefore, it is important to understand how subtyping interacts with different other features that also appear often in practice.

We next investiage the relationship between subtyping and several other interesting and important programming language features, including built-in types, lists, references, arrays, type casting and a bit with objects. The list of features can continue. In fact, it is customary that programming language designers analyze the effectiveness of new conceptual developments by studying their interaction with subtyping.

Built-in Types

Basic subtyping may be given by many other features besides records. As you know, programming languages usually provide *built-in*, also called *basic*, types, such as bool, int, real, etc.

Programmers find it very convenient to assume subtyping on some basic types, e.g., bool $\leq$ int. In fact, in many languages, the constant $true$ is represented as integer 1 and $false$ as 0. This way, one can use a bool expression whenever an integer expression is expected. For example, with a boolean variable b, one can write an expression: $scale * b$, which evaluating to wither 0 or $scale$, depending on whether b is false or true. To support this feature, we need that bool $\leq$ int. Other common subtype relationships are nat $\leq$ int, or int $\leq$ real.

Lists

We can also introduce list types into our type system:

$$Type ::= \ldots \mid list\ Type$$

The subtyping rule for lists is

$$\frac{t_1 \leq t_2}{list\ t_1 \leq list\ t_2}$$

Although this rule seems straightforward and intuitive, it only works when we build and use lists without applying any change on them. As we will next see in the discussion of references and arrays, some restrictions are raised by changing values at given locations in the list.

References

Most languages allow assignments of new values to existing, i.e., already declared, names. In many languages this is supported by introducing the notion of *reference* (also called *location* or *cell index*). We next extend our simply typed λ-calculus to support references and assignments:

$$\mathit{Type} ::= \ldots \mid \mathit{Ref}\ \mathit{Type} \mid \mathit{Unit}$$

$$\mathit{Exp} ::= \ldots \mid \mathit{ref}\ \mathit{Exp} \mid !\mathit{Exp} \mid \mathit{Exp} := \mathit{Exp}$$

Therefore, we allow explicit types for references. For example, $\mathit{Ref}\ (\mathrm{nat} \to \mathrm{nat})$ is a type for references to, or locations storing, functions $\mathrm{nat} \to \mathrm{nat}$. Values of reference types are just like any other values in the languages, that is, they can be passed to and returned as results of functions.

Together with refereces and assignments, *side effects* are unavoidable. *Unit* is the type of expressions, such as assignments, that are intended to be used just for their side effects; these are not supposed to evaluate to any particular value. In other words, one may read *Unit* as “no type”.

Three new expression constructors are introduced.

ref E evaluates to a new location, say L, where the expression E is also stored; this is equivalent to the “new” construct in object oriented languages, or to “malloc” in C. Depending upon the particular evaluation strategy desired in one’s language, the expression E stored at L may be already evaluated to some value. The language construct *ref* is also called *referencing*.

$!E$, which expects E to evaluate to a location, say L, returns the expression/value stored at L. The language construct ! is also called *dereferencing*.

$E := F$ first evaluates E to a location, say L, and then stores F at location L. Again, depending upon the desired evaluation strategy, one may first evaluate F to some value V and then store V at L.

To give a full equational semantics of our λ-calculus language extended with references, we would need to follow the same approach as in `FUN`, that is, to introduce infrastructure for *stores*, to evaluate λ-abstractions to closures, etc. We do not do this here, because it would essentially duplicate what we've already done in the definition of `FUN`. Instead, we just focus on aspects related to typing in the context of references.

Typing references.

The typing rules for references are straighforward:

$$\frac{X \ \triangleright \ E : t}{X \ \triangleright \ ref \ E : Ref \ t}$$

$$\frac{X \ \triangleright \ E : Ref \ t}{X \triangleright !E : t}$$

$$\frac{X \ \triangleright \ E : Ref \ t \qquad X \ \triangleright \ E' : t}{X \ \triangleright \ E := E' : Unit}$$

The subtleties of typing in the context of references come from their interaction with subtyping.

Subtyping and references.

The question to be asked here is how, and under what conditions, can one derive subtyping relations of the form $\mathit{Ref}\ t_1 \leq \mathit{Ref}\ t_2$? In other words, when can a reference to an expression/value of type t_1 be safely used in a context where a reference to an expression/value of type t_2 is expected?

There are two concerns regarding the use of a reference expression/value R of type $\mathit{Ref}\ t_1$ when a reference expression/value of type $\mathit{Ref}\ t_2$ is expected:

1. If R is dereferenced (read) in a context, such as, for example, in $3+!R$, the expression/value stored at R should be safely usable in that context, where an expression/value of type t_2 is expected; therefore, $t_1 \leq t_2$.

2. If R is assigned (written) in a contex, for example using an assignment of the form $R := E$ with E of type t_2, then since

> there can be other places in the program “expecting” (by dereferencing R) the expression/value at location R to have the declared type t_1, one deduces that $t_2 \leq t_1$.

Therefore, the only safe possibility to have $Ref\ t_1 \leq Ref\ t_2$ is that $t_1 = t_2$, which obviously implies $Ref\ t_1 = Ref\ t_2$. We conclude from here that reference types admit no proper subtypes.

While this is an “elegant” conclusion in what regards the implementation of a type system, because one basically needs to do *absolutely nothing* to support it, it is very important to understand the deep motivations underlying it.

> *Don’t speak unless you can improve on the silence*
>
> Spanish Proverb

While this may seem rather straightforward, when references and types live together in a language almost nothing is simple enough to avoid misunderstandings or subtle problems.

Arrays

Arrays can be very easilly added to a language, at least in what regards their typing. For example, in our language we can extend both the types and the expressions as follows:

$$\mathit{Type} ::= \ldots \mid \mathrm{Array}\ \mathit{Type}$$

$$\mathit{Exp} ::= \ldots \mid \mathit{Exp}[\mathit{Exp}] \mid \mathit{Exp}[\mathit{Exp}] := \mathit{Exp}$$

Essentially, arrays are similar to references. In fact, in many languages, arrays are nothing but references pointing to the first location of a block of memory. Consequently, by an analysis of subtyping similar to that for references, we can infer that the only reasonable way to have $\mathrm{Array}\ t_1 \leq \mathrm{Array}\ t_2$ is that $t_1 = t_2$, maning that array types have no proper subtypes. This conclusion can similarly be applied to lists when one is allowed to write at specific locations in lists (with statements of the form car $E := E'$, car $(cdr\ E) := E'$, etc.).

Some programming language designers, while still adopting type systems for their languages, find some of the above (static) subtyping restrictions too strong, arguing that they limit the use of references, arrays, lists, etc. In fact, designers of programming languages tend to easily become "religious"; for example, those designing untyped or dynamically typed languages think of static typing as a serious impediment the programmers have to deal with in one way or another.

Some languages split the task of type checking into a static component and a dynamic one. For example, Java only takes the deferencing (reading) into account during its static type checking, and checks every write at runtime to maintain the type safety. But this is also considered by some researchers as a design flaw ...

Type Casting

Type casting allows one to assign to terms types that type checkers may not be able to find statically. One can regard casting as a "type annotation" which helps the type checker analyze the program. At some extent, unless a language admits some form of dynamic type checking that can lead to different executions when expressions are evaluated to values of different types (such as "instance of" checks), one can regard all the type declarations as just "annotations" to help a particular "static program analysis" tool, the type checker, analyze the program.

The corresponding syntax for type casting in our language can be:

$$Exp := ...|\langle Type\rangle \; Exp$$

We do not discuss the formal (equational) semantics of casting here, but, intuitively, $\langle t\rangle E$ is simply equal to E when E can be shown of type t. In some cases one may be able to show statically that E has type t or that E cannot have type t, in which case the type casting may be simply dropped or a static error reported, but in general the problem is undecidable. In practice, a dynamic checker, or a monitor, is inserted to ensure that the type of E is indeed as claimed; if not, an error or an exception is generated. The benefit of this dynamic check is that the static type checker can then assume that E has indeed the claimed type t and can therefore continue unperturbed the type checking process on the rest of the program. The typing rule of casting is then simply as follows:

$$\frac{X \ \triangleright\ E : t_2}{X \ \triangleright\ \langle t_1 \rangle E : t_1}$$

Therefore, as far as an expression type checks to any type, it can also be cast to any other type, but a dynamic check still needs to be performed. For example, $(\langle\{age : nat\}\rangle x).age$ is assumed of type

nat for static type checking purposes, but it may cause a runtime error if the type t of x is not a record type containing a field age of type nat.

Exercise 18 *Give a simple program containing the expression* $(\langle\{age : nat\}\rangle x).age$ *which types and executes correctly, but which would not type if one replaced the expression above by just* $x.age$*.*

To facilitate casting, many languages have a top type, like the *Object* class in Java.

Syntax-Directed Subtyping

The subtyping rules, together with the typing rules in the context of subtypes, gave us a logical means to entail type judgements of the form $X \triangleright E : t$. However, note that the type system is bound to the fact that all three components of the type judgement, namely X, E, and t, need to be available.

For some typed frameworks, for example simply typed λ-calculus, as we know it is relatively trivial to translate the typing rules into a *typing algorithm*, calculating for a given expression E in a given type environment X a type t, if the expression is indeed well-typed, and reporting an error otherwise.

In the context of subtyping, one needs some additional work to obtain an effective typing algorithm.

Let us first understand what are the complications that appear when one tries to type a program within a language with subtyping.

Unlike typing in the context of our previous, non-subtyped languages, in the context of subtypes the application of the typing rules is *not deterministic*, i.e., *not syntax-directed*. Consider for example the λ-expression, say $mkYounger$,

$$\lambda x : \{age : nat, height : nat\} \; . \qquad \{age = x.age - 20, \;\; height = x.height + 3\}$$

and suppose that we would like to derive that it can have the type $\{age : nat, height : nat\} \rightarrow \{age : nat\}$, say t. There are two different ways to do it:

1) One can first apply typing rules for as long as possible, and then "ask for the help" the subtyping rules:

$$(1) \qquad \frac{\ldots}{\emptyset \triangleright mkYounger{:}\{age{:}nat,height{:}nat\}{\rightarrow}\{age{:}nat,height{:}nat\}}$$

$$(2) \qquad \frac{\ldots}{\{age{:}nat,height{:}nat\}{\rightarrow}\{age{:}nat,height{:}nat\}{\leq}\{age{:}nat,height{:}nat\}{\rightarrow}\{age{:}nat\}}$$

$$(3) \qquad \frac{(1) \quad (2)}{\emptyset \triangleright mkYounger{:}t}$$

2) Alternatively, a subtyping rule can be applied earlier, before typing the outermost expression:

(1) $$\frac{\ldots}{x{:}\{age{:}nat,height{:}nat\} \triangleright \{age{=}x.age{-}1,height{=}x.height{-}10\}{:}\{age{:}nat,height{:}nat\}}$$

(2) $$\frac{}{\{age{:}nat,height{:}nat\}{\leq}\{age{:}nat\}}$$

(3) $$\frac{(1) \quad (2)}{x{:}\{age{:}nat,height{:}nat\} \triangleright \{age{=}x.age{-}1,height{=}x.height{-}10\}{:}\{age{:}nat\}}$$

(4) $$\frac{(3)}{\emptyset \ \triangleright \ mkYounger : t}$$

In order for the entailment system to immediately provide a typing algorithm, one would like to have a well-defined, deterministic way to apply the typing rules by just examining the syntax of the program without any “search”. This is also called “syntax-directed” typing.

Of course, one may argue that, in the above example, chosing one of the two possible computations is not problematic, thanks to their confluence; hence one could pick, for instance, the lexicographically lower computation. But there are some rules in our context of subtyping that are really non-syntax-directed, in the sense that one has no clue where to find the catalyzers needed to continue computation.

Undeterministic subtyping rules.

Consider for instance the rules:

(≤-transitivity) $$\frac{t_1 \leq t \quad t \leq t_2}{t_1 \leq t_2}$$ How to pick a t when applying this rule?

(subsumption) $$\frac{X \ \triangleright\ E : t_1 \quad t_1 \leq t_2}{X \ \triangleright\ E : t_2}$$ How to pick a t_1? Structure/syntax of E is *not* taken into account.

Can one change the rules so that typing becomes syntax-directed and mechanical? The answer is yes.

Eleminating the bad rules.

A first interesting observation is that the (≤-transitivity) rule is *not needed*. The intuitive reason underlying this claim is that the subtyping relation derived using the other subtyping rules, excluding (≤-transitivity), *is already transitive*.

Exercise 19 *Prove formally the claim above.*

A second important observation is that we can also eliminate the other problematic rule, (subsumption), by carefully inspecting its usage. This rule really only needs to be used to type check function applications: $(\lambda\, x : t\ .\ E)E'$ requires that the type of E' is a subtype of t. Then one can simply eliminate the problematic (subsumption) rule and instead modify the λ-abstraction application rule into the following rule. To reflect the fact that the newly obtained type system is different from the original one, we use a different but closely related syntax, namely $X \ \trianglerighteq\ E : t$, to denote the type judgements derived with the latter type system:

$$(\leq\text{-application})\qquad \frac{X \trianglerighteq E_1 : t_1 \rightarrow t'_1,\ X \trianglerighteq E_2 : t_2,\ t_2 \leq t_1}{X \trianglerighteq E_1 E_2 : t'_1}$$

Note that the ($\leq$-application) rule above is syntax-driven, in the sense that the syntax of the goal type judgement (an "application"), tells us precisely what to do next: calculate the type of the two expressions involved and then derive the corresponding subtype relation.

Proposition 6 *Prove that the resulting typing system above has the following properties:*

- *For each expression E and type assignment X, one can derive $X \trianglerighteq E : t$ for a at most one type t;*
- ***(Soundness)*** *$X \trianglerighteq E : t$ implies $X \triangleright E : t$;*
- ***(Completeness)*** *$X \triangleright E : t$ implies $X \trianglerighteq E : t'$, where $t' \leq t$.*

Therefore, in terms of the original type system, the new type system derives *the most concrete type* of an expression.

Syntax-directed subtyping algorithm.

The important proposition above immediately provides an algorithm to decide whether, under a type assignment X, an expression E has a type t:

> *First derive* $X \trianglerighteq E : t'$ *and then check whether* $t' \leq t$*; if any of the two fails than* E *cannot have the type* t *in the type environment* X*.*

Exercise 20 *Define the syntax-directed typing algorithm above in Maude. You need to define both the subtyping relation and the typing rules equationally. Your definition should be executable.*

Typing Conditionals

In simply typed λ-calculus we had to define one conditional constant $cond_t : t \to t \to \mathrm{bool}$ for each type t. With subtyping, like with universal polymorphism, we only need to define one generic conditional. Precisely, we can add a Top constant type defined to be the *most general* type together with a subtyping rule

$$\textsf{(top)} \qquad \frac{-}{t \leq \mathrm{Top}}$$

and then define just one constant conditional expression, $\mathrm{cond} : \mathrm{bool} \to \mathrm{Top} \to \mathrm{Top} \to \mathrm{Top}$. (This actually follows a general technique to simluate universal polymorphism with subtyping).

Previously, the rule for typing conditionals was the following:

$$\frac{X \ \triangleright\ C : \mathrm{bool}, \qquad X \ \triangleright\ E_1 : t, \qquad X \ \triangleright\ E_2 : t}{X \ \triangleright\ \mathrm{cond}\ C\ E_1\ E_2 : t}$$

This rule still works in the context of subtyping, but note, however, that one is expected to use subsumption to lift the possibly more concrete types of E_1 and E_2 to some *common* supertype. In fact, the conditional expression can be derived any type that is a supertype of the most concrete types of E_1 and E_2. For example, consider the following expression ("keep one's height if one is older than 10, otherwise keep one's weight"):

$$\begin{array}{l} \mathrm{cond}\ (x.age > 10) \\ \qquad\qquad \{name = x.name,\ age = x.age,\ height = x.height\} \\ \qquad\qquad \{name = x.name,\ age = x.age,\ weight = x.weight\} \end{array}$$

One cannot apply directly the typing rule of the conditional, because the two branches have different types. But by subsumption one can first calculate some common type to both branches, such as $\{name : String,\ age : nat\}$, or $\{name : String\}$, or $\{age : nat\}$, or even $\{\}$, and then apply the typing rule for conditional.

The limitations of the typing rule for conditional above becomes clear in the context of syntax-directed typing, where one calculates the most concrete types of the two branches and then one wants to calculate a type for the conditional.

First, note that under syntax-directed typing, the typing rule for conditional above is close to useless, because the most concrete types of the branches may be different.

Second, what should the type of the conditional actualy be, knowing the types t_1 and t_2 of its branches? Since we want to calculate the type of the conditional *statically*, unless using sophisticated theorem provers (which typically do not to scale), we cannot know which of the two branches would be taken during an actual execution.

One possibility would be two consider both branches separately and ensure that the program would type regardless of which branch is taken. Unfortunately, the number of possibilities to analyze doubles with each conditional in the program. Therefore, despite its precision in analysis, this exhaustive approach would hardly have any practical use.

An extreme possibility would be to say that the type of the conditional is Top, because we do not know which of the two branches is taken. The problem with this aggressive approach is that all the type information about the two branches is lost, so one may need casting to explicitely "concretize" the type of the conditional to the expectations of a particular context.

The practical solution here is to accept loosing some precision but not all of it. Any common supertype of t_1 and t_2 is clearly better than Top. This suggests that we should actually pick the *least common supertype* of t_1 and t_2, written $t_1 \vee t_2$, as the type of the conditional; since the subtype relation is a partial order, the least common supertype of t_1 and t_2, also called their *join* type, is nothing but the least upper bound of t_1 and t_2 with respect to the subtyping relation. With this, the syntax-driven typing rule of the conditional is the following:

$$\frac{X \ \trianglerighteq \ C : \mathrm{bool} \qquad X \ \trianglerighteq \ E_1 : t_1, \qquad X \ \trianglerighteq \ E_2 : t_2}{X \ \trianglerighteq \ \mathrm{cond}\ C\ E_1\ E_2 : t_1 \vee t_2}$$

> *G: something is not right here, as the rule above destroys the syntax-drivenness. Indeed, we can either apply the application typing rule (in various ways) or the above.*

Thus our sample conditional expression discussed above types to $\{name : String,\ age : nat\}$.

Exercise 21 *(continuation of Exercise 20) Define (also in Maude) the join operation on types and add the "universal" conditional to the language together with its syntax-driven typing.*

Subtypes and Objects

Objects and the object-oriented (OO) paradigm form an important player in today's theory and practice of programming languages. Objects are considered by many software engineers crucial in the process of software development, because they improve the modulality and reusability of code.

Essentially, an object *encapsulates* a state and provides the outside world an *interface* to partially manipulate its state, like for example to access or modify parts of it. We next show how objects and some OO concepts can be supported, without any additional machinery, by typed λ-calculus with subtyping and references.

That should not mean that we are actually claiming that typed high-order languages should replace OO languages. Similarly, the fact that all programming language paradigms can be supported by rewriting does not mean that rewriting can replace all these.

Syntax

The syntax is left almost unchanged:

$$\mathit{Type} ::= \ldots \mid \mathit{Ref}\ \mathit{Type} \mid \mathit{Unit}$$

$$\mathit{Exp} ::= \ldots \mid \mathit{ref}\ \mathit{Exp} \mid !\ \mathit{Exp} \mid \mathit{Exp} := \mathit{Exp} \mid ()$$

The only increment here is () that is used for calling functions without paramenters. This feature is not required, since we can pass a dummy argument that is ignored in the function, but it is more natural and concise for programming. The type of () is *Unit*, as expected:

$$X \ \triangleright\ () : \mathit{Unit}$$

Representing objects

Let us consider a very simple example, that of a counter object which contains a “private” integer value and provides two functions to the outside world, namely $get()$ to get the value of the integer and $inc()$ to increment it. Once such a counter object is created, say c, one would like to be be able to write expressions like $c.get()$ and $c.inc()$, with the obvious meaning.

In our framework, it is then natural to represent objects as records, containing a field for each operation allowed to be visible, or part of the interface, to the outside world. In our particular case, an already created counter object can be represented as a record $\{get = ..., inc = ...\}$. The concrete integer value is not intended to be accessed directly from the outside, so it does not have a corresponding field in the record; neverthless, one still wants to access is indirectly, using the field get.

For the time being, let us assume that the state of an object has already been created. In our case of the simple counter, since the state consists of just one integer, let us assume that is has already been created in an outer scope and that a reference x to it is available. Then the representation of our already created object is:

$$\{get = \lambda\_ : Unit\ .\ !x,\quad inc = \lambda\_ : Unit\ .\ x := (!x + 1)\}$$

We will later see that the state of an object can be actually kept in a record (of references), which is made visible to the functions. As we already know by now, the evaluation strategies play a much more important role in the design of a language in the presence of side effects. Since OO programming is all about side effects, we impose a call-by-value evaluation strategy in our language. Also, function bodies are evaluated only when functions are called; othewise, the body of the function *inc* can be evaluated indefinitely and therefore the counter incremented in an uncontrolled manner.

Creating objects

One needs to be able to create an object before one can use it. Intuitively, creation of an object comes to intializing the state of the object. Considering again our example, we need to create an integer reference with some value and “give” it to the object. To achieve this, we can use a macro, say *createCounter*, defined as follows:

$$\begin{aligned}&(\lambda\, x : \mathit{Ref}\, \mathrm{nat}\ .\\ &\qquad \{\mathit{get} = \lambda\_ : \mathit{Unit}\ .\ !x,\ \ \mathit{inc} = \lambda\_ : \mathit{Unit}\ .\ x := \mathit{succ}(!x)\})\\ &(\mathit{ref}\ 1)\end{aligned}$$

Because of the assumed call-by-value evaluation strategy, each time the macro above is evaluated the following happen (recall also the definition of `FUN`): (1) a location is created and the integer value is stored at that location; (2) that location is bound to x in an environment, say Env, in which the record is evaluated; (3) the two functions in the record are evaluated to corresponding *closures*, each freezing the environment Env (further calls of these functions will therefore see the same location of x).

In short, this macro creates a counter object with the initial value of 1, which types as expected:

$$\emptyset \;\triangleright\; createCounter : \{get : Unit \rightarrow \mathrm{nat}, \;\; inc : Unit \rightarrow Unit\}$$

One would like to have the possibility to create many objects with a similar functionality, in our case many counters, without having to write the creation code over an over again. In a similar fashion to the new construct in object oriented programming, one can define a $newCounter$ macro as the λ-abstraction $\lambda\_ : Unit\ .\ createCounter$. One can now create counter objects in any context, by simply invoking $newCounter()$. Note that a new state is indeed created with each invocation of $newCounter$.

Exercise 22 *Define another macro for creating counter objects, namely one which takes as argument an intenger and creates a counter object initialized to that integer. Therefore, newCounter(7) should create a counter object initialized to 7.*

Subtyping objects

We know from OO that objects that are instances of subclasses can also be regarded as instances of superclasses. Without yet discussing classes and sub/superclasses, let us first see how an object having "more information" than another one, is actually typed to a subtype of the later object in our framework.

Consider an enriched counter, which, besides the usual get and inc methods, has a method $reset$ which resets the counter to 0. The new enriched counter objects have therefore the type:

$$\{get : Unit \to \mathrm{nat}, \quad inc : Unit \to Unit, \quad reset : Unit \to Unit\}$$

Therefore, the type of the enriched counters is a subtype of the simpler counters, which is consistent with the OO intuitions for objects and their types/classes. One can easily write a λ-abstraction for creating reset counter objects, say *newResetCounter*:

$$
\begin{array}{l}
(\lambda \_ : \mathit{Unit}\ . \\
\quad (\lambda x : \mathit{Ref}\ \mathrm{nat}\ . \\
\qquad \{\mathit{get} = \lambda \_ : \mathit{Unit}\ .\ !x, \\
\qquad\ \mathit{inc} = \lambda \_ : \mathit{Unit}\ .\ x := (!x + 1), \\
\qquad\ \mathit{reset} = \lambda \_ : \mathit{Unit}\ .\ x := 0\} \\
\quad)\ (\mathit{ref}\ 1) \\
)
\end{array}
$$

Let us next understand how classes can be encoded in our typed high-order framework.

Classes

In OO programming languages, classes typically are regarded as the types of objects, while objects are regarded as instances of classes. As it turns out, these words can mean almost everything, depending upon context, who says them and who listens to them.

It is customary though to consider that classes contain a certain kind of functionality and structure, while an object of a class contains, or encapsulates, a concrete “state” together with a handle to the class that it is an instances of. By a state, one typically means a type-consistent assignment of values to fields.

With this model in mind, we can refine our definition of an object as follows.

1. First introduce a record type containing all the intended fields of a class; since the values of these fields may be changed during the life-time of an object, the types of the fields must be

reference types. In the case of our counter we have only one field, so let us introduce the record type $\{x : \mathit{Ref}\,\mathrm{nat}\}$ and call it $\mathit{CounterRef}$. Whenever an object is created as an "instance" of a class, one should first create a record allocating concrete values to the fields;

2. Next define the class itself as a λ-abstraction taking a value record type as above and adding methods to it to taste. In our case, we can define a "class" $\mathit{CounterClass}$ as follows:

$$\begin{aligned} &\lambda\, r : \mathit{CounterRef}\ . \\ &\qquad \{\mathit{get} = \lambda\,\_ : \mathit{Unit}\ .\ (r.x), \\ &\qquad\ \ \mathit{inc} = \lambda\,\_ : \mathit{Unit}\ .\ (r.x) := (r.x + 1)\} \end{aligned}$$

Classes therefore evaluate to closures and type to function types taking field record types to method record type. For example, the class $\mathit{CounterClass}$ types to

$$\mathit{CounterRef} \rightarrow \{\mathit{get} : \mathit{Unit} \rightarrow \mathrm{nat},\quad \mathit{inc} : \mathit{Unit} \rightarrow \mathit{Unit}\}$$

3. Create objects by passing desired field records to classes. In our case, a counter with integer value 1 is created by simply evaluating $\mathit{Counter}(\{x = \mathit{ref}\ 1\})$. Also, one can define $\mathit{newCounter}$ as $\lambda\_ : \mathit{Unit}\ .\ \mathit{Counter}(\{x = \mathit{ref}\ 1\})$, and so on.

This way, we therefore have a means to device "classes" and then to create "objects" as instances of them. The type system of our calculus maintains a certain discipline in how objects are created and used, but it may still allow one to develop programs which one would rather want to reject.

Subclasses/Inheritance

Inheritance, or the process of extending existing classes with functionality, is an important, if not the most important, feature of the OO paradigm. The extended classes, despite the fact that they add functionality to the extended classes, are actually called *subclasses*. This terminology is in full synch with our terminology for “subtypes”. Let us next see, again via an example, how we can simulate inheritance within our framework in a consistent way, in the sense that the type system will rank “subclasses” as subtypes.

Let us define a class $ResetCounterClass$ which extends $CounterClass$ by adding a reset “method”:

$$
\begin{array}{l}
\lambda\, r : CounterRef\ . \\
\quad (\lambda\, super : ?\ . \\
\qquad \{get = super.get, \\
\qquad\ \ inc = super.inc, \\
\qquad\ \ reset = \lambda\, \_ : Unit\ .\ (r.x = 0)\}) \\
\quad (CounterClass\ r)
\end{array}
$$

Exercise 23 *What is the type of* $super$ *above (the red question mark)? Type the* $ResetCounterClass$ *above and show that it is a subtype of* $CounterClass$*.*

The use of *super* above is not required, but it helps to highlight the relationship between the subclass and the superclass.

However, although we would like to see a one-to-one relationship between the notions of “subclass” and corresponding “subtype” in our framework, the simulation of the OO paradigm above is so “flexible” that it actually allows one to also remove methods from the superclass, thus breaking the subtyping relationship, disallowing the use of the subclass where the superclass is expected, etc. To prevent that, we may need to provide more powerful mechanisms or primitives in the language, e.g., extends/inherits, together with appropriate typing rules/policies.

Exercise 24 *Comment on the encoding of OO programming in high-order typed languages presented above. Tell straight your opinion and think of desirable OO features that could not be supported this way. Also, show how self-reference can also be supported by allowing recursion (μ) in the language. How about dynamic versus static method dispatch?*