A Type-Theoretic Basis for an Object-Oriented Refinement Calculus [1]

Emil Sekerinski

Dept. of Computer Science, Åbo Akademi
Turku, Finland
Emil.Sekerinski@abo.fi [2]

[1] To appear in S. J. Goldsack, S. J. H. Kent (Eds.) *Formal Methods and Object Technology*, Springer-Verlag, 1996.

[2] This paper was partly written while the author was at Forschungszentrum Informatik, Karlsruhe, Germany

Abstract

This paper addresses the issue of giving a formal semantics to an object-oriented programming and specification language. Object-oriented constructs considered are objects with attributes and methods, encapsulation of attributes, subtyping, bounded type parameters, classes, and inheritance. Classes are distinguished from object types. Besides usual imperative statements, specification statements are included. Specification statements allow changes of variables to be described by a predicate. They are abstract in the sense that they are non-executable. Specification statements may appear in method bodies of classes, leading to abstract classes.

The motivation for this approach is that abstract classes can be used for problem-oriented specification in early stages and later refined to efficient implementations. Various refinement calculi provide laws for procedural and data refinement, which can be used here for class refinement. This paper, however, focuses on the semantics of object-oriented programs and specifications and gives some examples of abstract and concrete classes.

The semantics is given by a translation of the constructs into the type system $F_{\leq}$, an extension of the simple typed λ-calculus by subtyping and parametric polymorphism: The state of a program is represented by a record. A state predicate is a Boolean valued function from states. Statements, both abstract and concrete, are represented by predicate transformers, i. e. higher order functions mapping state predicates (postconditions) to state predicates (preconditions). Objects are represented by records of statements (the methods) operating on a record of attributes, where the attributes are hidden by existential quantification. Classes are understood as templates for the creation of objects. Classes are represented by records. Inheritance amounts to record overwriting. Subtyping and parametric polymorphism, e. g. for the parameterization of classes by types, are already present in $F_{\leq}$. The advantage of this semantic by translation is that it builds on the features already provided by $F_{\leq}$ (which are all used). Hence no direct reference to the model underlying $F_{\leq}$ needs to be made; a summary of the syntax and rules of $F_{\leq}$ is given.

1 Introduction

A characteristic feature of object-oriented program development is its uniform way of structuring all stages of the development by *classes*: Classes describe objects found during problem analysis, and classes emerge during design and implementation. As classes serve different purposes during the development, they have to be defined in appropriate notations: in a problem-oriented way during analysis and in an efficiently implementable language for the implementation.

For covering the different stages of the development, we present an *object-oriented design notation*. Besides imperative concepts, it features objects with attributes and methods, encapsulation of attributes, subtype-polymorphism, parametric polymorphism, classes, and inheritance. Furthermore it includes *specification statements* . These are non-executable statements, standing for parts which have to be *refined* to concrete (executable) statements [2, 12, 13] . Statements containing specification statements are called *abstract*. Abstract statements can appear in class definitions, resulting in *abstract classes*.

By this extension of the programming language the same structuring principles can be applied equally to programs and specifications. The benefit of such a *wide-spectrum* notation is that transitions between notations for different stages can be avoided and that development steps can (hopefully) be more easily verified.

The contribution of this paper is a novel way of giving a formal semantics to such an object-oriented design notation, and to show its use. The semantics is given by a definitional extension of the type system $F_{\leq}$. The type system $F_{\leq}$ itself is an extension of the simple typed λ-calculus with parametric polymorphism (functions parameterized by types) and subtype-polymorphism (functions operating on values of a type as well as on values of all subtypes) [6, 5, 8]. It was developed to study flexible but decidable typing in simple, functional setting. Here, it is used for an imperative setting: Statements are defined by predicate transformers in $F_{\leq}$ by viewing predicates as Boolean valued functions and predicate transformers as higher-order functions from predicates to predicates.

A feature of the notation contributing to a flexible and decidable typing is the distinction of object types from classes. The type of an object only determines its syntactic interface. The class of an object determines its behaviour, which is given by its methods and its attributes. Subtyping is a structurally defined relation on types. By contrast, inheritance is a means for constructing classes from existing ones, and is completely independent of subtyping. Classes serve as templates for the creation of objects. Objects of different classes may have the same type.

The structure of the paper is as follows. The next section gives the semantics of statements in a subset of $F_{\leq}$. Section 3 introduces objects with attributes and methods, encapsulation of attributes, subtype-polymorphism, parametric polymorphism, classes and inheritance. Section 4 shows the use of the notation by a couple of examples. Section 5 relates the approach to other approaches and draws some conclusions. The appendix describes the type system $F_{\leq}$.

2 Statements as Predicate Transformers

Following various refinement calculi, concrete (executable) statements are embedded into a much richer space where statements may be demonically nondeterministic, angelically nondeterministic and miraculous. (The exact distinction of abstract and concrete statements is left open, as the calculus does not depend on it.)

Statements are typed such that the type of a statement determines the program variables (and their types) on which the statement operates. The definitions in this section are given in the fragment of the type system $F_{\leq}$ which corresponds to the simple typed λ-calculus (see also appendix).

Values and Types. Each value has a type. Types are simple types like *Int* or *Bool* (with values *true* and *false*), or are composed types like *set of* T, the type of sets with values of type T, or $S \to T$, the type of functions from values of type S to values of type T. Value e has type T is written as $e : T$. For the examples we assume that for the types used the usual operations (and constants) are defined.

Of special relevance are record types. A record is a finite association of labels to values, written $(l_1 \mapsto e_1, \ldots, l_n \mapsto e_n)$. The record type $(l_1 : T_1, \ldots, l_n : T_n)$ is the type of records with fields $l_1, \ldots, l_n$ of types $T_1, \ldots, T_n$. The empty record is written as $()$. It is of the empty record type, also written as $()$. The value of field l is selected from a record r by $r.l$.

For notational convenience, we use additional operations on records and record types. If $r : R$ and $s : S$ are records, then the overwrite $r \oplus s$ adds the fields of s to those of r, replacing fields with same label. The result is of type R overwritten by S, denoted by $R \oplus S$. If record t is of type $R \oplus S$, then the restriction $t \mid S$, removes those fields from r which are not in S. Similarly, if $T = R \oplus S$, then $T \mid S$ removes those fields from T which are not in S.

The substitution $[e/y]f$ stands for value expression f in which every free occurrence of y is replaced by value expression e. The value declaration introduces a shorthand for a parameterized value:

$$(val\ y\ (x : T) = e \cdot f) \quad = \quad [(\lambda\ x : T \cdot e)/y]f$$

Similarly, the substitution $[T/A]f$ stands for value expression f with each free occurrence of type variable A replaced by type expression T. The type declaration introduces a shorthand for a type expression:

$$(type\ A = T \cdot f) \quad = \quad [T/A]f.$$

States and State Predicates. A state assigns values to program variables. A state is given by a record. Program variables correspond to names of record fields. A state space is a record type.

State predicates, or predicates for short, are functions from states to *Bool*. The type *Pred* R is the type of predicates over state space (record type) R:

$$Pred\ R \quad = \quad R \to Bool$$

The Boolean operators $\neg, \wedge, \vee, \Rightarrow, \Leftarrow, \Leftrightarrow$ on state predicates are defined by pointwise extension of the corresponding operators on the Boolean values, e. g. for predicates b, c of type *Pred R* we have:

$$b \wedge c \quad = \quad (\lambda\, r : R \cdot b\, r \wedge c\, r)$$

Universal implication and equivalence (equality) of predicates b, c are denoted by $b \leq c$ and $b = c$, respectively. For predicates b, c of type *Pred R* we define:

$$b = c \quad \text{iff} \quad (\forall\, r : R \cdot b\, r \Leftrightarrow c\, r)$$
$$b \leq c \quad \text{iff} \quad (\forall\, r : R \cdot b\, r \Rightarrow c\, r)$$

State predicates are conveniently written as Boolean expressions. For example, $x + c = y$ is a Boolean expression which can be understood as a predicate. If x and y are the program variables in the expression and c is a constant (logical variable), then the corresponding predicate is

$$(\lambda\, s : S \cdot s.x + c = s.y) \quad : \quad \mathit{Pred}\ S \quad ,$$

where $S = (x : \mathit{Int}, y : \mathit{Int})$. Hence the interpretation of a Boolean expression as a predicate depends on the *context*, which determines the program variables and constants.

Statements. If S and T are state spaces, then a statement p with initial state space S and final state space T is identified with a function (predicate transformer) mapping predicates (postconditions) over T to predicates (preconditions) over S. For a predicate c over T, the application $p\, c$ is the weakest precondition of p such that, operationally, the execution of p terminates and ends in a state satisfying c. In Dijkstra's original notation this is written as $wp(p, c)$ [9]. The type *Tran S T* is the type of statements with initial state space S and final state space T. Note that the typing of statements allows the final state space to differ from the initial state space.

$$\mathit{Tran}\ S\ T \quad = \quad \mathit{Pred}\ T \rightarrow \mathit{Pred}\ S$$

Two statements p and q of type *Tran S T* are equal, written $p = q$, means that their weakest preconditions for establishing a certain postcondition are equal. Statement p is refined by statement q, written $p \sqsubseteq q$ means that the weakest precondition for p to establish a certain postcondition is stronger that the weakest precondition for q to establish the same postcondition.

$$p = q \quad \text{iff} \quad (\forall\, c : \mathit{Pred}\ T \cdot p\, c = q\, c)$$
$$p \sqsubseteq q \quad \text{iff} \quad (\forall\, c : \mathit{Pred}\ T \cdot p\, c \leq q\, c)$$

The refinement relation is reflexive ($p \sqsubseteq p$), transitive ($p \sqsubseteq q$ and $q \sqsubseteq r$ implies $p \sqsubseteq r$), and antisymmetric ($p \sqsubseteq q$ and $q \sqsubseteq p$ implies $p = q$). Transitivity is necessary for the stepwise refinement of statements.

Composing Statements. The sequential composition of two statements p : *Tran R S* and q : *Tran S T* corresponds to their functional composition.

$$p\,;q \;=\; (\lambda\, b : \mathit{Pred}\ T \cdot p\ (q\ b))$$

The sequential composition is associative, i. e. for appropriate statements p, q, and r we have $(p\,;q)\,;r = p\,;(q\,;r)$. Sequential composition is monotonic in both operands, meaning that if $p \sqsubseteq p'$ and $q \sqsubseteq q'$ then $p\,;q \sqsubseteq p'\,;q'$.

The demonic choice $q \sqcap r$ of statements q : *Tran S T* and r : *Tran S T* establishes a postcondition only if both statements do so. For the angelic choice $q \sqcup r$ to establish a postcondition it suffices that either q or r does establish it.

$$\begin{array}{lcl} q \sqcap r & = & (\lambda\, b : \mathit{Pred}\ T \cdot (q\ b) \wedge (r\ b)) \\ q \sqcup r & = & (\lambda\, b : \mathit{Pred}\ T \cdot (q\ b) \vee (r\ b)) \end{array}$$

Both demonic and angelic choice are symmetric and associative and are monotonic with respect to refinement. The binary demonic and angelic choice can be generalized to a choice between an arbitrary number of alternatives. Let I be an arbitrary set and let q_i be statements indexed by $i \in I$.

$$\begin{array}{lcl} (\sqcap\, i \in I \cdot q_i) & = & (\lambda\, b : \mathit{Pred}\ T \cdot (\forall\, i \in I \cdot q_i\ b)) \\ (\sqcup\, i \in I \cdot q_i) & = & (\lambda\, b : \mathit{Pred}\ T \cdot (\exists\, i \in I \cdot q_i\ b)) \end{array}$$

Primitive Statements. For a predicate c : *Pred T*, the guard $[c]$ and assertion $\{c\}$ are of type *Tran T T*.

$$\begin{array}{lcl} [c] & = & (\lambda\, b : \mathit{Pred}\ T \cdot c \Rightarrow b) \\ \{c\} & = & (\lambda\, b : \mathit{Pred}\ T \cdot c \wedge b) \end{array}$$

Let $h : S \to T$ be a function from states to states. The update operator $\langle h \rangle$ lifts the state transformer h to a predicate transformer of type *Tran S T*.

$$\langle h \rangle \;=\; (\lambda\, b : \mathit{Pred}\ T \cdot (\lambda\, s : S \cdot b\ (h\ s)))$$

The multiple assignment $v := e$, with a list of program variables v and a list of expressions e, corresponds to an update statement with identical initial and final state space. For example, if $x+y-c$ is an expression with program variables x and y, then

$$x := x + y - c \;=\; \langle \lambda\, s : S \cdot (x \mapsto s.x + s.y - c, y \mapsto s.y) \rangle$$

where $S = (x : \mathit{Int}, y : \mathit{Int})$. The type of this assignment is *Tran S S*.

The variable introduction *enter* $v : V := e$ extends the state space by program variables v with initial values e. The variable elimination *exit* $v : V$ removes the variables v from the state space. For example, with $S = (x : \mathit{Int})$ we have:

$$\begin{array}{lcl} (\mathit{enter}\ x : \mathit{Int} := 7) & = & \langle \lambda\, s : () \cdot (x \mapsto 7) \rangle \\ (\mathit{exit}\ x : \mathit{Int}) & = & \langle \lambda\, s : S \cdot () \rangle \end{array}$$

The type of *enter* $x : \mathit{Int} := 7$ is *Tran* $()$ S, the type of *exit* $x : \mathit{Int}$ is *Tran S* $()$. We omit the types of the introduced or eliminated variables if they are clear from the context.

Embedding Statements. The initial and final state space of a statement comprise only those program variables on which it operates. Frequently, it is convenient to view a statement as operating on an enlarged state space and leaving the extra program variables unmodified. If q is of type *Tran S T* and the program variables of state space R are disjoint from those of S and T, the embedding $q \uparrow R$ is of type *Tran RS RT*, where $RS = R \oplus S$ and $RT = R \oplus T$.

$$q \uparrow R \;=\; (\lambda\, c : \mathit{Pred}\ RT \cdot (\lambda\, a : RS \cdot (q\ (\lambda\, t : T \cdot c\ ((a \mid R) \oplus t)))\ (a \mid S)))$$

For example, consider the assignment $x := 3$ of type *Tran* $(x : \mathit{Int})$ $(x : \mathit{Int})$. With $R = (y : \mathit{Int})$, we have:

$$(x := 3) \uparrow R \quad = \quad x, y := 3, y$$

Making the embedding implicitly, this allows us to write:

$$x := 3 \quad = \quad x, y := 3, y$$

Without the implicit embedding, this expression would not be well-typed. Implicit embedding is also useful for the composition with *abort*, *miracle*, and *skip*. The statement *abort* establishes no postcondition, the statement *miracle* establishes any postcondition, and the statement *skip* does nothing. All three statements are of type *Tran* $()$ $()$.

$$\begin{array}{lcl} \mathit{abort} & = & (\lambda\, b : \mathit{Pred}\ () \cdot \mathit{false}) \\ \mathit{miracle} & = & (\lambda\, b : \mathit{Pred}\ () \cdot \mathit{true}) \\ \mathit{skip} & = & (\lambda\, b : \mathit{Pred}\ () \cdot b) \end{array}$$

We have that $\mathit{abort} = \{\mathit{false}\}$, $\mathit{miracle} = [\mathit{false}]$, and $\mathit{skip} = \{\mathit{true}\} = [\mathit{true}] = \langle \mathit{id} \rangle$, where *id* is the identity function on the empty record type. (With *Id* as defined in the appendix, we have that $\mathit{id} = \mathit{Id}\ ()$.)

Implicit embedding makes it possible to compose these statements in a natural way. For example, we may state that *skip* is unit of sequential composition, i. e. for statement p : *Tran S T* we have:

$$\mathit{skip}\ ;\ p \quad = \quad p\ ;\ \mathit{skip} \quad = \quad p$$

Making the embedding explicit, this stands for:

$$\mathit{skip} \uparrow S\ ;\ p \quad = \quad p\ ;\ \mathit{skip} \uparrow T \quad = \quad p$$

Using implicit embedding, we have that both *abort* and *miracle* are left zero of sequential composition, i. e. for any statement p : *Tran S S* we have $\mathit{miracle}\ ;\ p = \mathit{miracle}$ and $\mathit{abort}\ ;\ p = \mathit{abort}$. Furthermore, *abort* is the least element of the refinement ordering and *miracle* is the greatest, i. e. $\mathit{abort} \sqsubseteq p$ and $p \sqsubseteq \mathit{miracle}$ for any statement p : *Tran S S*.

Derived Statements. The conditional is defined by a demonic choice with guarded alternatives.

$$\textit{if } b \textit{ then } q \textit{ else } r \textit{ end} \quad = \quad (([b] \,;\, q) \sqcap ([\neg b] \,;\, r))$$

Because sequential composition and demonic choice are monotonic in both operands, so is the above conditional monotonic in both q and r.

Let F be a monotonic function from statements to statements (w. r. t. refinement). Recursion is defined explicitly by a generalized demonic choice.

$$(\mu\, r \cdot F\, r) \quad = \quad (\sqcap\, q \in \{r \mid r = F\, r\} \cdot q)$$

Because F is monotonic, F has by the Knaster-Tarski fixpoint theorem a fixpoint in the lattice of statements, which is indeed $(\mu\, r \cdot F\, r)$. Iteration is defined as a special case of recursion. It is well-defined because both sequential composition and conditional are monotonic with respect to refinement.

$$\textit{while } b \textit{ do } q \textit{ end} \quad = \quad (\mu\, r \cdot \textit{ if } b \textit{ then } q \,;\, r \textit{ else skip end})$$

The multiple variable declaration $(\textit{var } v : V := e \cdot p)$ introduces new variables $v : V$ with initialization e, executes the body p of the declaration, and eliminates the variables v again.

$$(\textit{var } v : V := e \cdot p) \quad = \quad (\textit{enter } v : V := e \,;\, p \,;\, \textit{exit } v : V)$$

The specification statement $v : [v' \cdot b]$ chooses nondeterministically (demonically) values v' such that predicate b holds and assigns them to the variables v. If no such values exist, it behaves as *miracle*.

$$v : [v' \cdot b] \quad = \quad (\sqcap\, v' \in T \cdot [b] \,;\, v := v')$$

3 Objects as Packed Records

In the view of objects as records, attributes and methods of an object correspond to record fields [7]. Subtyping is a relation which is defined on the structure of types. For record types, the subtype may have more fields than the supertype and the types of the common fields are also in a subtype relation.

The view of objects as records leads to a certain cyclic dependency because methods operate on records of which they are part of. This dependency is usually resolved by recursive types. Here, we follow the alternative way of using existentially quantified types instead, as proposed in [16], which avoids the complexity of recursive types, but makes *packing* and *opening* of records necessary. A slight additional complication is introduced because in this encoding scheme an object is represented by a pair of records (one for the attributes and one for the methods), rather than a single record.

Objects and Object Types. Consider points in a two-dimensional plane. For example, the coordinates of a point can be given by a record $\textit{coord}0$ of type $\textit{Coord}0$.

$$\begin{array}{lcl} \textit{coord}0 & = & (x \mapsto 9, y \mapsto 3) \\ \textit{Coord}0 & = & (x : \textit{Int}, y : \textit{Int}) \end{array}$$

A point object encapsulates its coordinates and makes them accessible only through the methods *set* with values parameters $newx : Int, newy : Int$, method *get* with result parameters $curx : Int, cury : Int$, and parameterless method *mirror*. Parameter passing is done by enlarging and reducing the state space. That is, the body of the method *set* can access the variables $newx, newy$ from its initial state space and has to remove them at the end. Dually, the body of the method *get* adds variables $curx, cury$ to the state space. For example, $meth0$ is a record of type $Meth0$ with such method bodies:

$$\begin{array}{lcl} meth0 & = & (set \quad \mapsto x, y := newx, newy \,;\, exit\ newx, newy, \\ & & \ get \quad \mapsto enter\ curx, cury \,;\, curx, cury := x, y, \\ & & \ mirror \mapsto x, y := -x, -y) \end{array}$$

$$\begin{array}{lcl} Meth0 & = & (set \quad : Tran\ (Coord0 \oplus (newx : Int, newy : Int))\ Coord0, \\ & & \ get \quad : Tran\ Coord0\ (Coord0 \oplus (curx : Int, cury : Int)), \\ & & \ mirror : Tran\ Coord0\ Coord0) \end{array}$$

A point object is basically a pair with coordinates and methods, for example represented by the record $(attr \mapsto coord0, meth \mapsto meth0)$. However, the type of a point shall only reveal its method interface, but not its representation by coordinates. Hence, for defining the type *Point* of points, we replace the type $Coord0$ in $Meth0$ above by a new type variable *Attr*. Then, using existential quantification, we express that objects of type *Point* consists of attributes of type *Attr* and methods operating on these attributes, for some type *Attr*:

$$Point \quad = \quad (\Sigma\ Attr \cdot (attr : Attr, meth : PointMeth))$$

where

$$\begin{array}{lcl} PointMeth & = & (set \quad : Tran\ (Attr \oplus (newx : Int, newy : Int))\ Attr, \\ & & \ get \quad : Tran\ Attr\ (Attr \oplus (curx : Int, cury : Int)), \\ & & \ mirror : Tran\ Attr\ Attr) \end{array}$$

Types like *Point* are called object types. For object types we introduce a more convenient notation, which does not mention the type of the private attributes (it is a bound variable which can be given an arbitrary name anyway). For example, an equivalent definition of *Point* is:

$$\begin{array}{lcl} Point & = & object \\ & & \quad set\ (newx : Int, newx : Int), \\ & & \quad get\ ()\ curx : Int, curx : Int, \\ & & \quad mirror\ () \\ & & end \end{array}$$

Now we consider colored points, where the color shall be a public attribute . This is simply achieved by requiring that the private attribute type is a subtype of the public part.

$$\begin{array}{lcl} Col & = & (color : Int) \\ ColPoint & = & (\Sigma\ Attr <: Col \cdot (attr : Attr, meth : PointMeth)) \end{array}$$

In a more convenient notation, an equivalent definition of *ColPoint* is:

$$\begin{array}{lll} \mathit{ColPoint} & = & \mathit{object} \\ & & \quad \mathit{color} : \mathit{Int}, \\ & & \quad \mathit{set}\ (\mathit{newx} : \mathit{Int}, \mathit{newx} : \mathit{Int}), \\ & & \quad \mathit{get}\ ()\ \mathit{curx} : \mathit{Int}, \mathit{curx} : \mathit{Int}, \\ & & \quad \mathit{mirror}\ () \\ & & \mathit{end} \end{array}$$

For the rest of the paper, let *Pub* stand for $(\mathit{pub} : P)$, where *pub* is a list of field names and P is a list of types. Let *Attr* be a type variable. Let *Meth* stand for

$$\begin{array}{l} (m1 : \mathit{Tran}\ (\mathit{Attr} \oplus (v1 : V1))\ (\mathit{Attr} \oplus (r1 : R1)), \\ \ldots, \\ \mathit{mi} : \mathit{Tran}\ (\mathit{Attr} \oplus (\mathit{vi} : \mathit{Vi}))\ (\mathit{Attr} \oplus (\mathit{ri} : \mathit{Ri}))) \quad , \end{array}$$

where $v1, \ldots, \mathit{vi}, r1 \ldots, \mathit{ri}$ are lists of labels and $V1, \ldots, \mathit{Vi}, R1, \ldots, \mathit{Ri}$ are lists of types. Using this convention, the general encoding scheme for object types is:

$$\begin{array}{l} \mathit{object}\ \mathit{pub} : P, m1\ (v1 : V1)\ r1 : R1, \ldots, \mathit{mi}\ (\mathit{vi} : \mathit{Vi})\ \mathit{ri} : \mathit{Ri}\ \mathit{end} \quad = \\ \quad (\Sigma\ \mathit{Attr} <: \mathit{Pub} \cdot\ (\mathit{attr} : \mathit{Attr}, \mathit{meth} : \mathit{Meth})) \end{array}$$

Creating Objects. A value of an existentially quantified type is created by packing. For example, an object with attributes *coord*0 of type *Coord*0 and methods *meth*0 is created by:

$$\mathit{pt}0 \quad = \quad \mathit{pack}\ (\mathit{attr} \mapsto \mathit{coord}0, \mathit{meth} \mapsto \mathit{meth}0)\ \mathit{by}\ \mathit{Coord}0\ \mathit{as}\ \mathit{Point}$$

Here, the record $(\mathit{attr} \mapsto \mathit{coord}0, \mathit{meth} \mapsto \mathit{meth}0)$ is the contents of the object, *Coord*0 is the hidden type, and *Point* is the type of the resulting value. We can introduce a shorthand for this construction, writing simply:

$$\mathit{pt}0 \quad = \quad \langle\!|\mathit{coord}0 \cdot \mathit{meth}0|\!\rangle$$

Here the dot separates the public from the private part. The types of the hidden attributes and the resulting object are determined by the constituents *coord*0 and *meth*0.

An object of type *ColPoint* with a public attribute is created by:

$$\begin{array}{lll} \mathit{pt}1 & = & \mathit{pack}\ (\mathit{attr} \mapsto \mathit{coord}0 \oplus (\mathit{color} \mapsto \mathit{green}), \mathit{meth} \mapsto \mathit{meth}0) \\ & & \mathit{by}\ \mathit{Coord}0 \oplus (\mathit{color} : \mathit{Int})\ \mathit{as}\ \mathit{ColPoint} \end{array}$$

Note that in this example we make use of implicit embedding: the method bodies in *meth*0 have to be embedded into a state space with additional program variable $\mathit{color} : \mathit{Int}$. As a shorthand we write:

$$\mathit{pt}1 \quad = \quad \langle\!|\mathit{coord}0 \cdot (\mathit{color} \mapsto \mathit{green})\ ;\ \mathit{meth}0|\!\rangle$$

Here the semicolon is used for separating the public attributes from the methods.

For an object like $pt0$ above, a private attribute cannot be selected because the type *Point* hides them. Hence the expression $pt0.x$ is not well-typed. However, neither can the methods be selected this way. Although the names of the methods are known by the type *Point*, the type of the methods depends on the type of the attributes, which is hidden in *Point*. Hence an expression like $pt1.mirror$ cannot be assigned a type. The only possible operation with a packed value is opening it, as shown in the next paragraphs.

Attribute Selection. A public attribute like *color* of $pt1$ above can be selected by first opening $pt1$. In the expression

$$\begin{array}{l} \mathit{open}\ pt1\ \mathit{as}\ \mathit{Attr} <: \mathit{Col}\ \mathit{by}\ \mathit{cont} : (\mathit{attr} : \mathit{Attr}, \mathit{meth} : \mathit{PointMeth}) \\ \mathit{in}\ \mathit{cont.attr.color} \end{array}$$

the type variable *Attr* is introduced for the hidden type that can be used in the scope following *by*. The name *cont* is introduced for the contents of $pt1$ that can be used in the scope following *in*. Then the public attribute is selected, yielding *green* of type *Int* as result. The general form of *open* is:

$$\mathit{open}\ e\ \mathit{as}\ A <: S\ \mathit{by}\ x : T\ \mathit{in}\ f$$

Overloading the dot notation, we write the selection of a public attribute p of an object e by $e.p$. If p is in *pub* (see above), the general definition is:

$$\begin{array}{lcl} e.p & = & (\mathit{open}\ e\ \mathit{as}\ \mathit{Attr} <: \mathit{Pub}\ \mathit{by}\ \mathit{cont} : (\mathit{attr} : \mathit{Attr}, \mathit{meth} : \mathit{Meth}) \\ & & \mathit{in}\ \mathit{cont.attr}.p) \end{array}$$

Method Calls. The definition of method calls is complicated because a method call does not yield a simple result, but changes the value of the object and the values of the result parameters. As an intermediate step, we define the construct $\mathit{access}\ v\ \mathit{as}\ A <: S\ \mathit{by}\ x : T\ \mathit{in}\ p$. It selects the value of program variable v from the initial state space and opens it with the scope of statement p. Let p be of type *Tran Q R*.

$$\begin{array}{l} (\mathit{access}\ v\ \mathit{as}\ A <: S\ \mathit{by}\ x : T\ \mathit{in}\ p) \quad = \\ \quad (\lambda\, b : \mathit{Pred}\ R \cdot (\lambda\, s : Q \cdot (\mathit{open}\ s.v\ \mathit{as}\ A <: S\ \mathit{by}\ x : T\ \mathit{in}\ (p\ b)s))) \end{array}$$

For a program variable v of type *Obj*, the parameterless method call $v.m()$ selects the value of v from the initial state space, opens it, extends the state space by its attributes, executes the method m in the enlarged state space, and removes the modified attributes from the state space, packing them into the new value of v:

$$\begin{array}{lcl} v.m() & = & (\mathit{access}\ v\ \mathit{as}\ \mathit{Attr} <: \mathit{Pub}\ \mathit{by}\ \mathit{cont} : (\mathit{attr} : \mathit{Attr}, \mathit{meth} : \mathit{Meth}) \\ & & \mathit{in}\ \mathit{pushattr}\ ;\ \mathit{cont.meth.m}\ ;\ \mathit{popattr}) \end{array}$$

where

$$\begin{array}{lcl} \mathit{pushattr} & = & \langle \lambda\, \mathit{oldobj} : (v : \mathit{Obj}) \cdot \mathit{cont.attr} \rangle \\ \mathit{popattr} & = & \langle \lambda\, \mathit{newattr} : \mathit{Attr} \cdot \\ & & \qquad (v \mapsto\ \mathit{pack}\ (\mathit{attr} \mapsto \mathit{newatt}, \mathit{meth} \mapsto \mathit{cont.meth}) \\ & & \qquad\quad \mathit{by}\ \mathit{Attr}\ \mathit{as}\ \mathit{Obj}) \rangle \end{array}$$

Statement *pushattr* of type *Tran* $(v : Obj)$ *Attr* removes the program variable *v* from and adds the attributes of *v* to the state space. Dually, *popattr* of type *Tran Attr* $(v : Obj)$ removes the attributes from the state space and adds the program variable *v* again.

A parameterized method call is reduced to a parameterless method call by adding the value parameters to the state space before the call and removing the result parameters from the state space after the call. For example, if *pt* is a program variable of type *Point*, then:

$$\begin{array}{lcl} pt.set(6,4) & = & (enter\ newx, newy := 6,4\ ;\ pt.set()) \\ a,b := pt.get() & = & (pt.get()\ ;\ a,b := curx, cury\ ;\ exit\ curx, cury) \end{array}$$

Subtyping of Statements and Objects. The subtyping relation of $F_{\leq}$ (see also appendix) leads to a subtyping of statements and objects. Subtyping of statements is, similarly to subtyping of functions, contravariant in the first argument and covariant in the second argument.

$$\frac{E \vdash S' \leq S \qquad E \vdash T \leq T'}{E \vdash Tran\ S\ T \leq Tran\ S'\ T'}$$

This rule follows from the definition of *Tran* and the repeated application of *sub-function*.

Two object types are in a subtype relation, if (1) all public attributes of the supertype are also public attributes of the subtype, (2) the types of the common attributes are in a subtype relation, (3) all methods of the supertype are methods of the subtype with same parameters, (4) the corresponding value parameters are in supertype relation, and (5) the corresponding result parameters are in subtype relation.

For lists *S* and *T* of types, we write $S \leq T$ if the corresponding elements of the lists are in a subtype relation. Using this convention, the subtyping rule for objects types reads as follows:

$$\frac{\begin{array}{c} E \vdash Q \leq P \qquad E \vdash V1 \leq W1 \quad \ldots \quad E \vdash Vi \leq Wi \\ E \vdash S1 \leq R1 \quad \ldots \quad E \vdash Si \leq Ri \end{array}}{\begin{array}{l} E \vdash object\ pub : Q, pub' : Q', m1\ (v1 : W1)\ r1 : S1, \ldots, \\ \qquad mi\ (vi : Wi)\ ri : Si, \ldots, mj\ (vj : Wj)\ rj : Sj\ end \leq \\ \quad object\ pub : P, m1\ (v1 : V1)\ r1 : R1, \ldots, mi\ (vi : Vi)\ ri : Ri\ end \end{array}}$$

For example, we have *ColPoint* $\leq$ *Point*.

The *subsumption* rule states that every value of a type is also a value of a supertype of that type. Therefore, $pt1$ is a value of type *ColPoint* as well as of type *Point*. This implies that, for a program variable *pt* of type *Point*, the assignment

$$pt := pt1$$

is indeed well-typed. However, selecting the attribute *color* from *pt* after the assignment is a type error, as *pt* does not change its type by this assignment.

Type Parameters. The type abstraction $(\Lambda\ A\ <:\ S \cdot e)$ denotes a function which takes a subtype of S as parameter and binds it to A. The result is a value of a type which may depend on S. The value declaration is extended appropriately:

$$(val\ y[A <: S](x : T) = e \cdot f) \quad = \quad [(\Lambda\ A <: S \cdot (\lambda\ x : T \cdot e))/y]f$$

An example of its use is (omitting the scope of the declaration):

$$\begin{array}{lll} type & Colored & = \ object\ color : Int\ end \\ val & Darker\ [C <: Colored]\ (cl0 : C, cl1 : C) & = \\ & \quad if\ cl0.color \leq cl1.color\ then\ cl0\ else\ cl1\ end \end{array}$$

If $pt2$ is an object of type *ColPoint* such that $pt2.color = black$, then we have that:

$$Darker\ ColPoint\ (pt1, pt2) \quad = \quad pt2$$

Classes. A class describes objects with same behavior, i. e. with same method bodies and attribute types. Classes are used as templates for creating objects and classes can be constructed from existing classes by inheritance. The first role implies that a class determines the initial value of objects created from that class, and the second role implies that a class must not hide the private attributes. Hence a class is given by a record of initial privates attributes, a record of initial public attributes and a record of methods. For example, the class of green points at the origin may be defined by:

$$\begin{array}{lll} GreenPoint & = & (private \mapsto (x \mapsto 0, y \mapsto 0), \\ & & public \ \ \mapsto (color \mapsto green), \\ & & methods \mapsto \\ & & \quad (set \ \ \mapsto x, y := newx, newy\ ;\ exit\ x, y, \\ & & \quad get \ \ \mapsto enter\ curx, cury\ ;\ curx, cury := x, y, \\ & & \quad mirror \mapsto x, y := -x, -y)) \end{array}$$

A more conventional notation for this class is:

$$\begin{array}{lll} GreenPoint & = & class \\ & & \quad private\ x : Int := 0, \\ & & \quad private\ y : Int := 0, \\ & & \quad color : Int := green, \\ & & \quad set\ (newx : Int, newy : Int)\ = \\ & & \quad\quad x, y := newx, newy \\ & & \quad get\ ()\ curx : Int, cury : Int\ = \\ & & \quad\quad curx, cury := x, y, \\ & & \quad mirror\ ()\ = \\ & & \quad\quad x, y := -x, -y \\ & & end \end{array}$$

If $C : (private : CPriv, public : CPub, methods : CMeth)$ is a class and *Obj* is the type of objects of class *C*, the function *create C* creates an object of type *Obj*:

$$\begin{array}{l} create\ C\ = \\ \quad pack\ (attr \mapsto C.private \oplus C.public, methods \mapsto C.methods) \\ \quad by\ CPriv \oplus CPub\ as\ Obj \end{array}$$

When applying *create C*, we assume that the type of the objects of class C can be inferred from the context. For example,

$$gp := \mathit{create}\ \mathit{GreenPoint}$$

assigns an object of type *ColPoint* to *gp*.

Inheritance on classes is, in its simplest form, expressed by record overwriting. If C, M are classes then the result of inheriting from C and modifying by M is

$$\begin{array}{l}(\mathit{private} \mapsto C.\mathit{private} \oplus M.\mathit{private},\\ \mathit{public} \quad \mapsto C.\mathit{public} \oplus M.\mathit{public},\\ \mathit{methods} \mapsto C.\mathit{methods} \oplus M.\mathit{methods})\end{array}$$

In a more conventional notation, an equivalent definition of *GreenPoint* is:

$$\begin{array}{lll}\mathit{ColorlessPoint} & = & \mathit{class}\\ & & \quad \mathit{private}\ x : \mathit{Int} := 0,\\ & & \quad \mathit{private}\ y : \mathit{Int} := 0,\\ & & \quad \mathit{set}\ (\mathit{newx} : \mathit{Int}, \mathit{newy} : \mathit{Int}) \ =\\ & & \qquad x, y := \mathit{newx}, \mathit{newy}\\ & & \quad \mathit{get}\ () \ \mathit{curx} : \mathit{Int}, \mathit{cury} : \mathit{Int} \ =\\ & & \qquad \mathit{curx}, \mathit{cury} := x, y,\\ & & \quad \mathit{mirror}\ () \ =\\ & & \qquad x, y := -x, -y\\ & & \mathit{end}\end{array}$$

$$\begin{array}{lll}\mathit{GreenPoint} & = & \mathit{class}\ (\mathit{ColorlessPoint})\\ & & \quad \mathit{color} : \mathit{Int} := \mathit{green}\\ & & \mathit{end}\end{array}$$

Note that here we make use of implicit embedding: The methods inherited from *ColorlessPoint* operate in an enlarged state space in *GreenPoint*.

4 Using Abstract and Concrete Classes

As classes are values (of a certain structure), classes can be parameterized by both values and types. By contrast, programming languages like Eiffel treat classes as types, allowing only parameterization by types in order to make type-checking decidable. By separating classes and object types, more flexibility is gained. The following examples show the use of type and value parameters in class definitions. They also show the use of abstract statements for method definitions.

We make use of the type *bag*. An empty bag is written as $\prec\succ$, a bag containing only e by $\prec e\succ$. The addition of bags b, c is written as $b+c$, the subtraction as $b-c$. The test e *in* b determines if e is contained in the bag b and $\#b$ determines the number of elements in b.

For a buffer, the basic operations are storing an element in the buffer and taking an element from the buffer. A buffer does not guarantee any order of incoming and outgoing elements. Let *error* be a suitable error handling procedure:

$$
\begin{array}{l}
\mathit{val}\ \mathit{Buffer}[\mathit{Item}] = \\
\quad \mathit{class} \\
\quad\quad \mathit{private}\ q : \mathit{bag\ of}\ \mathit{Item} := \prec\succ, \\
\quad\quad \mathit{put}\ (i : \mathit{Item}) = q := q + \prec i \succ, \\
\quad\quad \mathit{get}\ ()\ i : \mathit{Item} = \\
\quad\quad\quad \mathit{if}\ q \neq \prec\succ\ \ \mathit{then}\ i : [i' \cdot i'\ \mathit{in}\ q]\ ;\ q := q - \prec i \succ\ \ \mathit{else\ error\ end}, \\
\quad\quad \mathit{empty}\ ()\ e : \mathit{Bool} = e := (q = \prec\succ) \\
\quad \mathit{end}
\end{array}
$$

A priority queue is a buffer in which the elements are removed according to their priority. The priority of an element is given by a public attribute of type *Int*.

$$
\begin{array}{l}
\mathit{type}\ \mathit{OrderedItem} = \mathit{object}\ \mathit{key} : \mathit{Int}\ \mathit{end} \\
\\
\\
\mathit{val}\ \mathit{PriorityQueue}\ [\mathit{Item} <: \mathit{OrderedItem}] = \\
\quad \mathit{class}\ (\mathit{Buffer}\ \mathit{Item}) \\
\quad\quad \mathit{get}\ ()\ i : \mathit{Item} = \\
\quad\quad\quad \mathit{if}\ q \neq \prec\succ\ \ \mathit{then} \\
\quad\quad\quad\quad i : [i' \cdot (i'\ \mathit{in}\ q) \wedge (\forall\ x\ \mathit{in}\ q \cdot x.\mathit{key} \leq i'.\mathit{key})]\ ; \\
\quad\quad\quad\quad q := q - \prec i \succ \\
\quad\quad\quad \mathit{else\ error\ end} \\
\quad \mathit{end}
\end{array}
$$

In this example, a public attribute *key* of type *Int* is selected in a Boolean expression. This could not have been expressed if *key* were a method, because method calls are statements and not simple values.

A bounded priority queue is a priority queue which has a limit for the number of elements stored in it. Hence it is parameterized by the type of the elements and their number.

$$
\begin{array}{l}
\mathit{val}\ \mathit{BoundedPriorityQueue}\ [\mathit{Item} <: \mathit{OrderedItem}]\ (\mathit{max} : \mathit{Int}) = \\
\quad \mathit{class}\ (\mathit{PriorityQueue}\ \mathit{Item}) \\
\quad\quad \mathit{put}\ (i : \mathit{Item}) = \{\#q < \mathit{max}\}\ ;\ q := q + \prec i \succ \\
\quad \mathit{end}
\end{array}
$$

A bounded priority queue can be simply implemented by storing the elements

in an array. We omit the initialization of the array because it is irrelevant.

```
val ArrayPriorityQueue [Item <: OrderedItem] (max : Int) =
  class
    private a : array max of Item,
    private n : Int := 0,
    put (i : Item) = a[n] := i ; n := n + 1,
    get () i : Item =
      if n ≠ 0 then
        var p, i := 0, 1 ·
        while i < n do if a[p].key < a[i].key then p := i end end ;
        i := a[p] ; n := n − 1 ; a[p] := a[n]
      else error end,
    empty () e : Bool = e := (n = 0)
  end
```

For a type *Item* $\leq$ *OrderedItem* and integer *max*, objects of classes

- *Buffer Item,*
- *PriorityQueue Item,*
- *BoundedPriorityQueue Item max,*
- *ArrayPriorityQueue Item max*

all have the same type. This is independent of the use of inheritance for defining the classes.

If classes are viewed as abstract data types, the technique of data refinement can be used to establish refinement relationships between classes . In this example, *PriorityQueue Item* refines *Queue Item* and *ArrayPriorityQueue Item max* refines *BoundedPriorityQueue Item max* [17]. This is related to the approaches of [1, 10, 11].

5 Related Work and Conclusions

This work started as an approach to formalize the idea of abstract and concrete classes in an analogy to the amalgamation of abstract and concrete statements in the refinement calculus. The motivation is to overcome the limitations when specifying classes by pre- and postconditions: in this approach, a method call cannot appear in a pre- or postconditions (because method calls in general change program variables). Hence, objects and classes cannot be used for specifying other classes (only for implementing them), which means that the approach does not scale up.

Another approach in the same direction is presented in [18]. Statements are defined by predicate transformers e. g. as in [12], rather than in a typed approach as here. This leads to a simpler semantics without the need for type-theory, but disallows for technical reasons specification statements in methods. Another typed approach is reported in [14], also defining statements by

predicate transformer and subtyping on record types. The semantics is rich enough to model the features of Oberon, but it does not include encapsulation and parametric polymorphism.

The approach taken here has mostly been influenced by the development of the refinement calculus in higher order logic (a variant of the simple typed λ-calculus) in [3]. The difference is that states are encoded here by records rather than tuples. The motivation for this is that inheritance is more naturally defined by records, which are used for modeling the state of objects. This has a number of minor consequences, e. g. a different treatment of variable declarations, of expressions (in guards, assertions and assignments), and of embedding. However, nested declarations of the same program variables cannot be considered in this model.

Our encoding of object types by existential types is inspired by that of [16] in $F^{\omega}_{\leq}$, an extension of $F_{\leq}$ by higher-order polymorphism (functions from types to types). Here, we restrict ourselves to $F_{\leq}$, at the cost of treating some definitions less elegantly.

Correct typing of expressions in $F_{\leq}$ is known to be decidable, with some minor restrictions as reported in [15]. This implies that the correct typing of expressions constructed by operators of this paper is also decidable, although we did not derive (efficient) type-checking algorithms. Such type-checking algorithms could be used as part of a compiler or for type-checking specifications.

The idea of distinguishing classes from object types has already been given a type-theoretic account in [4, 16]. The motivation for this distinction is that decidable properties should be separated from undecidable properties. For example, the equivalence of two types, the subtyping of two types and the containment of a value in a type is decidable. By contrast, it is undecidable whether two classes are equivalent (which would involve comparing the method bodies for equivalence), and it is undecidable whether an object belongs to a class (in the sense that is belongs to the set of reachable values of the class).

A number of object-oriented concepts have not been treated, for example recursive types (like points with method $\mathit{distance}(\mathit{other} : \mathit{Point})$) and object names as values. Also, the treatment of inheritance is simplified by not considering mutually dependent methods. These concepts remain the subject of further work.

Acknowledgments. The comments of Michael Butler on an earlier version were very helpful. Discussions with David Naumann lead to further insight and a number of improvements.

References

[1] Pierre America. A Behavioral Approach to Subtyping in Object-Oriented Programming Languages. In J. W. de Bakker, W. P. de Roever, and G. Rozenberg, editors, *REX School/Workshop on the Foundations of Object-Oriented Languages*, Lecture Notes in Computer Science 489, pages 60–90, Noordwijkerhout, The Netherlands, 1990. Springer-Verlag.

[2] R. J. R. Back and J. von Wright. Refinement Calculus, Part I: Sequential Nondeterministic Programs. In J. W. deBakker, W.-P. deRoever, and G. Rozenberg, editors, *REX Workshop on Stepwise Refinement of Distributed Systems - Models, Formalisms, Correctness*, Lecture Notes in Computer Science 430, pages 42–66, Mook, The Netherlands, 1989. Springer-Verlag.

[3] Ralph J. R. Back. Refinement Calculus, Lattices and Higher Order Logic. In M. Broy, editor, *Program Design Calculi*, NATO ASI Series, pages 53–72. Springer-Verlag, 1993.

[4] Kim B. Bruce and Robert van Gent. TOIL: A new Type-safe Object-oriented Imperative Language. Technical report, Department of Computer Science, Williams College, October 13 1993.

[5] L. Cardelli, J. C. Mitchell, S. Martini, and A. Scedrov. An extension of system F with subtyping. In *Theoretical Aspects of Computer Software*, Lecture Notes in Computer Science 526, Sendai, Japan, 1991. Springer-Verlag.

[6] L. Cardelli and P. Wegner. On Understanding Types, Data Abstraction and Polymorphism. *ACM Computing Surveys*, 17(4):471–522, 1985.

[7] Luca Cardelli. A Semantics of Multiple Inheritance. In G. Kahn, D. MacQueen, and G. Plotkin, editors, *International Symposium on the Semantics of Data Types*, Lecture Notes in Computer Science 173, pages 51–67. Springer-Verlag, 1984.

[8] Pierre-Louis Curien and Giorgio Ghelli. Coherence of Subsumption, Minimum Typing and Type-Checking in $F_{\leq}$. *Mathematical Structures in Computer Science*, 2:55–91, 1992.

[9] E. W. Dijkstra. *A Discipline of Programming*. Prentice-Hall, 1976.

[10] K. Lano and H. Haughton. Reasoning and Refinement in Object-Oriented Specification Languages. In O. Lehrmann Madsen, editor, *European Conference on Object-Oriented Programming '92*, Lecture Notes in Computer Science 615. Springer-Verlag, 1992.

[11] Gary T. Leavens. Modular Specification and Verification of Object-Oriented Programs. *IEEE Software*, 8(4):72–80, 1991.

[12] Carroll C. Morgan. The Specification Statement. *ACM Transactions on Programming Languages and Systems*, 10(3):403–419, 1988.

[13] J. M. Morris. A Theoretical Basis for Stepwise Refinement and the Programming Calculus. *Science of Computer Programming*, 9(3), 1987.

[14] David A. Naumann. Predicate Transformer Semantics of an Oberon-Like Language. In Ernst-R. Olderog, editor, *Programming Concepts, Methods and Calculi*, pages 460–480, San Miniato, Italy, 1994. International Federation for Information Processing.

[15] Benjamin C. Pierce. Bounded Quantification is Undecidable. In Carl A. Gunter and John C. Mitchell, editors, *Theoretical Aspects of Object-Oriented Programming*. MIT Press, Massachusetts, London, 1994.

[16] Benjamin C. Pierce and David N. Turner. Simple Type-Theoretic Foundations For Object-Oriented Programming. *Journal of Functional Programming*, 4(2):207–247, 1994.

[17] Emil Sekerinski. *Verfeinerung in der Objektorientierten Programmkonstruktion*. Doctoral thesis, University of Karlsruhe, 1994.

[18] M. Utting. *An Object-Oriented Refinement Calculus with Modular Reasoning*. Doctoral thesis, University of New South Wales, Kensington, 1992.

Appendix: The Type System $F_{\leq}$

The type system $F_{\leq}$ was developed to study the problems of sound, decidable typing of object-oriented languages in a simple, functional setting. It provides a basis for flexible typing but still guarantees the absence of "message not understood" errors of untyped languages.

$F_{\leq}$ is an extension of the simple typed λ-calculus by type parameters and subtyping. Type parameters allow to express parametric polymorphism, i. e. functions operating on values of different types. Subtyping between two types leads to the idea that a value of a type is also a value of a supertype of that type. This allows to express subtype polymorphism, i. e. functions operating on values of a type and, consequently, on values of all its subtypes.

In this appendix, we present $F_{\leq}$ with record types and existential types, which corresponds to the language *Fun* in [6]. Existential types allow to express encapsulation. Furthermore we use in the examples basic types like *Int* and *Bool* as well products $T \times S$ and sets *set of* T, arrays *array of* T and bags *bag of* T. For these types we assume that suitable operations (and constants) are available. Both record types and existential types, as well as other types, can also be encoded in pure $F_{\leq}$ (see for example [8]). The reader is referred to [6] and [16] for tutorial introductions.

Syntax. The syntactic categories are that of values (e), types (T), variables (x), type variables (A), and labels (l).

T	::=	$\mathit{Bool} \mid \mathit{Int} \mid \ldots$	basic types
	$\mid$	A	type variable
	$\mid$	Top	top type
	$\mid$	$(l_1 : T_1, \ldots, l_n : T_n)$	record type
	$\mid$	$T \to T$	function type
	$\mid$	$(\Pi\, A <: T \cdot T)$	universally quantified type
	$\mid$	$(\Sigma\, A <: T \cdot T)$	existentially quantified type
e	::=	$\mathit{false} \mid \mathit{true} \mid 0 \mid 1 \mid \ldots$	constants

	x	variable
	$(l_1 \mapsto e_1, \ldots, l_n \mapsto e_n)$	record construction
	$e.l$	field selection
	$(\lambda\, x : T \cdot e)$	abstraction
	$e\ e$	application
	$(\Lambda\, A <: T \cdot e)$	type abstraction
	$e\ T$	type application
	pack e by T as T	packing
	(*open e as A* $<:$ *T by x* : *T in e*)	opening

Value e has type T is written as $e : T$. Type S is a subtype of T is written as $S \leq T$. For example, the type *Top* is the supertype of all types, i. e. for any type S, $S \leq$ *Top*. Furthermore we have that any values e is (also) of type *Top*, e : *Top*. When the type bound in an universally quantified type, in an existentially quantified type, in a type abstraction or in an opening is omitted, it is taken to be *Top*:

$$
\begin{array}{lcl}
(\Pi\, A \cdot T) & = & (\Pi\, A <: \mathit{Top} \cdot T) \\
(\Sigma\, A \cdot T) & = & (\Sigma\, A <: \mathit{Top} \cdot T) \\
(\Lambda\, A \cdot e) & = & (\Lambda\, A <: \mathit{Top} \cdot e) \\
(\mathit{open}\ e\ \mathit{as}\ A\ \mathit{by}\ x : T\ \mathit{in}\ e) & = & (\mathit{open}\ e\ \mathit{as}\ A <: \mathit{Top}\ \mathit{by}\ x : T\ \mathit{in}\ e)
\end{array}
$$

Examples of Quantified Types. Universally quantified types are functions with a type parameter and a value result. For example

$$\mathit{Id} \quad = \quad (\Lambda\, A \cdot (\lambda\, x : A \cdot x))$$

is the polymorphic identity function. Its type is $(\Pi\, A \cdot A \to A)$. It can be applied to any type. For example, applying it to *Int* yields the identity function on *Int*:

$$\mathit{Id}\ \mathit{Int} \quad = \quad (\lambda\, x : \mathit{Int} \cdot x)$$

The type of *Id Int* is *Int* $\to$ *Int*.

Existentially quantified types express encapsulation. Intuitively, a value of the type $(\Sigma\, R \cdot S)$ can be thought of as a pair with first component a type R and the second component a value of type S, where S depends on R. For example, the type

$$T \quad = \quad (\Sigma\, R \cdot R \times (R \to \mathit{Bool}))$$

is the type of "objects" which consist of an "attribute" of the hidden type R and a "method" from the attribute type to *Bool*. By

$$o \quad = \quad \mathit{pack}\ (5, (\lambda\, x : \mathit{Int} \cdot \mathit{odd}\ x))\ \mathit{by}\ \mathit{Int}\ \mathit{as}\ T$$

an object of type T is created with attribute 5, method $(\lambda\, x : \mathit{Int} \cdot \mathit{odd}\ x)$ and hidden attribute type *Int*. As the type of the object o is T, its attribute type is not visible, we only know that one exists. By

$$b \quad = \quad (\mathit{open}\ o\ \mathit{as}\ A\ \mathit{by}\ \mathit{cont} : A \times (A \to \mathit{Bool})\ \mathit{in}\ (\mathit{snd}\ \mathit{cont})(\mathit{fst}\ \mathit{cont}))$$

the object is opened, meaning that a name is given to the hidden type and the content. Then the method is extracted from the content and applied to the attribute, resulting in $b = \mathit{true}$.

Subtyping Rules. The inference rules given below allow to decide whether the subtype relation between to types holds. However, this depends on the environment, i. e. the enclosing declarations of the types and values. More formally, an environment is a list whose individual components have the form $x : T$ or $A <: T$, where x is a variable, A is a type variable, and T is a type expression. Each variable and type variable must occur at most once on the left hand side of $x : T$ or $A <: T$. A subtyping judgment of the form $E \vdash S \leq T$ means that from the environment E it follows that S is a subtype of T.

$$E \vdash T \leq T \qquad \text{sub-reflexive}$$

$$\frac{E \vdash S \leq T \qquad E \vdash T \leq U}{E \vdash S \leq U} \qquad \text{sub-transitive}$$

$$E, A <: T, F \vdash A \leq T \qquad \text{sub-var}$$

$$E \vdash T \leq \mathit{Top} \qquad \text{sub-top}$$

$$\frac{E \vdash S_1 \leq T_1 \quad \ldots \quad E \vdash S_m \leq T_m}{E \vdash (l_1 : S_1, \ldots, l_m : S_m, \ldots, l_n : S_n) \leq (l_1 : T_1, \ldots, l_m : T_m)} \qquad \text{sub-record}$$

$$\frac{E \vdash S' \leq S \qquad E \vdash T \leq T'}{E \vdash S \to T \leq S' \to T'} \qquad \text{sub-function}$$

$$\frac{E \vdash S' \leq S \qquad E, A <: S' \vdash T \leq T'}{E \vdash (\Pi\, A <: S \cdot T) \leq (\Pi\, A <: S' \cdot T')} \qquad \text{sub-universal}$$

$$\frac{E \vdash S' \leq S \qquad E, A <: S' \vdash T \leq T'}{E \vdash (\Sigma\, A <: S \cdot T) \leq (\Sigma\, A <: S' \cdot T')} \qquad \text{sub-existential}$$

Typing Rules. The inference rules given next allow to infer the minimal type of a value expression. (Every well-formed value expression is of type *Top*, so inferring an arbitrary type for a value expression is trivial.) A typing judgment of the form $E \vdash e : T$ means that from the environment E the type T for the expression e can be inferred.

$$\frac{E \vdash e : S \quad E \vdash S \leq T}{E \vdash e : T} \qquad \text{subsumption}$$

$$E, x : T, F \vdash x : T \qquad \text{var-type}$$

$$\frac{E \vdash e_1 : T_1, \quad \ldots \quad E \vdash e_n : T_n}{E \vdash (l_1 \mapsto e_1, \ldots, l_n \mapsto e_n) : (l_1 : T_1, \ldots, l_n : T_n)} \qquad \text{record-cons-type}$$

$$\frac{E \vdash e : (l_1 : T_1, \ldots, l_n : T_n)}{E \vdash e.l_i : T_i} \qquad \text{field-selection-type}$$

$$\frac{E, x : S \vdash e : T}{E \vdash (\lambda\, x : S \cdot e) : S \to T} \qquad \text{abstraction-type}$$

$$\frac{E \vdash e : S \to T \qquad E \vdash f : S}{E \vdash e\ f : T} \quad \text{application-type}$$

$$\frac{E, A <: S \vdash e : T}{E \vdash (\Lambda\ A <: S \cdot e) : (\Pi\ A <: S \cdot T)} \quad \text{type-abstraction-type}$$

$$\frac{E \vdash e : (\Pi\ A <: S \cdot T) \qquad E \vdash S' \leq S}{E \vdash e\ S' : [S'/A]T} \quad \text{type-application-type}$$

$$\frac{E \vdash S' <: S \qquad E \vdash e : [S'/A]S}{E \vdash (\mathit{pack}\ e\ \mathit{by}\ S'\ \mathit{as}\ (\Sigma\ A <: S \cdot T)) : (\Sigma\ A <: S \cdot T)} \quad \text{packing-type}$$

$$\frac{E \vdash e : (\Sigma\ A <: S \cdot T) \quad E, A <: S, x : T \vdash f : U}{E \vdash (\mathit{open}\ e\ \mathit{as}\ A <: S\ \mathit{by}\ x : T\ \mathit{in}\ f) : U} \quad \text{opening-type}$$

Index