TooL: A Persistent Language Integrating Subtyping, Matching and Type Quantification*

Andreas Gawecki Florian Matthes

Universität Hamburg
Vogt-Kölln Straße 30
D-22527 Hamburg, Germany
{gawecki,matthes}@informatik.uni-hamburg.de

Abstract

This paper presents the type system of the TooL persistent programming language that captures much of the flavor of Smalltalk within a safe static typing discipline. Following the spirit of Smalltalk providing a highly flexible and extensible programming environment based on a small set of expressive language primitives, TooL provides only few built-in type concepts with rich semantics that achieve power through systematic use and orthogonality. The paper focuses on the language design issues that arise in the course of integrating type concepts that are well-understood in isolation like object types, subtyping, type matching, and type quantification into a practical database programming language. We illustrate these issues by code samples taken from the TooL generic bulk data library.

*This research is supported by ESPRIT Basic Research, Project FIDE, #6309 and by a grant from the German Israeli Foundation for Research and Development (*bulk data classification*, I-183 060).

Contents

1 Introduction

The design and implementation of the persistent Tycoon object-oriented language TooL is heavily influenced by language design experience gained in the Tycoon project carried out at Hamburg University. The Tycoon[1] project follows an *add-on* approach to generic database programming that emphasizes type-safe scalability and extensibility [Matthes and Schmidt 1993; Matthes 1993].

At the system level, Tycoon offers orthogonal persistence and support for distribution and migration of data, code and threads between persistent address spaces in local and wide area networks [Matthes and Schmidt 1994; Mathiske *et al.* 1995]. Moreover, bidirectional gateways to external languages can be realized systematically. These gateways are captured at the language level as polymorphically typed generic libraries [Matthes *et al.* 1994].

The rationale behind the original Tycoon type system is to provide a set of unbiased, orthogonal primitives to support multiple database programming methodologies like functional, imperative and different flavors of object-oriented modeling. Tycoon is based on function types, record types, and recursive types in a full higher-order type system where subtyping and unrestricted existential and universal quantification is provided over types, type operators and higher-order type operators, including a limited form of dependent types. Tycoon is therefore similar to Quest [Cardelli 1990; Cardelli and Longo 1991] and the type theoretic model of $F^{\omega}_{<:}$ [Pierce and Turner 1993] but it avoids the notion of kinds (types of types) by generalizing subtyping to higher-order types.

Unfortunately, the expressiveness of Tycoon's type system does not suffice to adequately capture the notion of type *matching* [Black and Hutchinson 1990; Bruce 1993; Bruce *et al.* 1993b; Abadi and Cardelli 1995], even by complex higher-order encodings. Matching supports type-safe inheritance of binary methods as it is required for pure object-oriented modeling and for the construction of reusable bulk data libraries [Matthes and Schmidt 1991].

In this paper, we describe the design rationale and type system of TooL that has a strong bias towards a pure object-oriented programming methodology in the spirit of Smalltalk but otherwise adopts most of the Tycoon principles outlined above. TooL aims for expressive power by the systematic use and orthogonal combination of few semantically rich primitives: Object types combine aggregation, encapsulation, recursion, parameterization and inheritance. At the type level there are two inductively defined, structural orderings on types (subtyping and matching) and universal type quantification (parametric polymorphism) that can be combined in interesting ways that we discuss by examples from the TooL bulk type library in this paper.

This paper is structured as follows. After an overview of the language and a description of its underlying design principles in section 2, we present TooL's fundamental typing concepts in section 3. The following section illustrates how to exploit these polymorphic typing concepts for the construction of pure object-oriented class libraries using inheritance. Again, the interaction between subtyping and matching is of particular interest. The remaining sections report on the TooL system implementation, provide a comparison with related research, and point out future work.

[1]TYped Communicating Objects in Open eNvironments

2 TooL Design Rationale and Language Overview

Since the focus of the Tycoon project is not to verify type-theoretic concepts but to develop a fully-fledged persistent programming environment, we highlight in this section the design principles behind the TooL language.

TooL minimizes built-in language functionality in favor of flexible system add-ons, both at the level of values and at the level of types.

TooL supports the classical object model where objects are viewed as abstract data types encapsulating both state and behavior. Similar to Smalltalk [Goldberg and Robson 1983] and Self [Ungar and Smith 1987], TooL is a *pure* object-oriented language in the sense that *every* language entity is viewed as an object and *all* kinds of computations are expressed uniformly as (typed) patterns of passing messages [Hewitt 1987]. Even low-level operations like integer arithmetic, variable access, and array indexing are uniformly expressed by sending messages to objects.

Contrary to other statically-typed object-oriented languages and specification languages [Goguen 1990], TooL provides statically-scoped higher-order functions that are also viewed as first-class objects that understand messages. Thereby control structures like loops, conditionals, function calls and exception handling also do not have to be built into the language, but can be defined as add-ons using objects and dynamic binding.

To improve code reusability, even the access to instance and pool variables (that unify the concept of global and class variables in Smalltalk) is done by sending messages [Johnson and Foote 1988]. This obviates the need for special scoping and typing rules concerning instance and pool variables.

TooL's simplicity at the value level leads to a significant complexity reduction at the type level where it is only necessary to define type and scoping rules for class signatures, message sends and inheritance clauses which we explore in the rest of the paper.

Despite the semantic simplicity of TooL, there is a rich set of syntactic variants to write down message sends. This syntactic sugar helps to define compact yet easy to read message patterns, for example to capture control structures or arithmetic computations. For "power-users" developing domain-specific abstractions (like query language notations or concurrency control schemes), the TooL language is based on extensible grammars [Cardelli *et al.* 1994] to enable dynamic syntax extensions.

Finally, it should be noted that modern compiler technology eliminates most of the run-time performance overhead traditionally associated with the pure object-oriented approach [Chambers and Ungar 1991; Hölzle 1994; Gawecki 1992]. For example, TooL uses dynamic optimizations across abstraction barriers based on persistent CPS representations to "compile away" many message sends [Gawecki and Matthes 1995].

3 TooL's Basic Typing Concepts

The TooL language is strongly and statically typed in the sense that no operation will ever be invoked on an object that does not support it, i.e. errors like "message not understood" cannot occur at run time. In this section we motivate and describe TooL's basic typing concepts, namely modular and structural type-checking, as well as subtyping and matching. To formalize these concepts, we have defined a full set of natural deduction-style type rules on the abstract TooL syntax [Gawecki *et al.* 1995] in the spirit of [Milner *et al.* 1990] and [Matthes and Schmidt 1992]. In the remainder of the paper we restrict ourselves to an informal discussion of the finer points of these type rules.

3.1 Type-Checking in Persistent and Distributed Environments

TooL programs are executed in a persistent and distributed enviroment where it is necessary that software components can be developed and maintained independently (over time and across space) which leads to the following requirements.

Structural Subtyping: Several conventional object models couple the implementation of an object with its type by identifying types with class names (e.g. C++, ObjectPascal, Eiffel). In these models, an object of a class named A can only be used in a context where an object of class A or one of its statically declared superclasses is expected. This implies that type compatibility is based on a single inheritance lattice which is difficult to be maintained in a persistent and distributed scenario.

Therefore, TooL has adopted a more expressive notion of type compatibility based on *structural subtyping* (called *conformance* in [Hutchinson 1987]). Intuitively, an object type A is a subtype of another object type B when it supports at least the operations supported by B. That is, TooL views types as (unordered) sets of method signatures, abstracting from class or type names during the structural subtype test. The additional flexibility of structural subtyping is especially useful if A and B have been defined independently, without reference to each other. Such situations occur in the integration of pre-existing external services, in the communication between sites in distributed systems [Birell *et al.* 1993], and on access to persistent data [Abadi *et al.* 1989].

Modular Type-Checking of a class only requires access to the interfaces of imported classes and of superclasses. In particular, it should be possible to type-check new subclasses without having to re-check method implementation code in superclasses again. Modular type checking speeds up the type-checking process significantly, thus supporting rapid prototyping within an incremental programming environment. It also has the advantage that class libraries – developed independently by different vendors – can be delivered in binary form without (a representation of) their source code with the option of type-safe subclassing at the customer side.

Modular type-checking requires the *contravariant* method specialization rule for soundness, which means that the types of method arguments are only permitted to be generalized when object types are specialized. The contravariant rule has been criticized of being counter-intuitive [Meyer 1989]. Accordingly, Eiffel has adopted a *covariant* method specialization rule which requires some form of global data flow analysis at link-time to ensure type correctness. Such an analysis (besides being time-consuming) generally requires the source code (or a close representation of it) of all classes and methods that constitute the whole program to be available to the type-checker at link-time which is not acceptable in our setting. TooL provides a partial solution to the covariance/contravariance problem without giving up modular type-checking by adopting the notion of *type matching* which allows the covariant specialization of method arguments in the important special case where the argument type is equal to the receiver type.

3.2 Structural Subtyping

The most important application of subtyping is *subsumption* or *substitutability*: we may use an object of a subtype in situations where an object of some of its supertypes is expected. Another application of subtyping in TooL is bounded type quantification as discussed in section 3.3 and 4.2.

As an example for substitutability, consider the printOn method defined in class Object to print any TooL object onto a stream of characters:

```
class Object
  printOn(aStream :WriteStream(Char))
```

For example, we may send a string literal object the message printOn to print itself onto the object stdout (standard output) which is an instance of the class **File**.

```
"a string".printOn(stdout)
```

In order to substitute a file in a context where a write stream of characters is expected, we have to show the subtype relationship File <: WriteStream(Char).

In TooL, it is not required that an explicit inheritance relationship between these two classes exists. The subtype relationship holds implicitly and can be deduced from the structure of the following class interfaces:

```
class WriteStream(E <: Object)
  put(e :E) :Void

class File
  get :Char
  put(ch :Char) :Void
  close :Void
```

The generic class WriteStream is parameterized with an element type E that is bounded by the type Object (cf. section 4.2) and exports a single method put to append an object e of type E to the stream. The independently defined, unparameterized class File also exports a put method that takes only characters in addition to a put and close method.

Subtyping is transitive and reflexive and the TooL subtype lattice contains all closed object types which correspond to non-parameterized class interfaces. For convenience, TooL class definitions implicitly define a corresponding object type and there is no extra syntax in TooL to define object types (e.g. as in PolyTOIL [Bruce *et al.* 1993b]).

The top element of the subtype lattice is called Void, which is an object type with an empty method suite. Currently, all TooL classes are descendants of a more specialized class Object that already provides some core methods (e.g. testing for object identity and printing).

The bottom element of our type lattice is the type constant Nil. The only subtype of Nil is Nil itself (due to reflexivity). The special object nil (the *undefined object*) is the single instance of this type. Conceptually, the type Nil consists of an infinite method suite, supporting any operation with any signature. However, sending any message to nil (other than identity testing and printing) triggers a runtime exception. The value nil is used to initialize instance variables and to mark *not yet filled* slots in hash tables. The type Nil is also used to type expressions that raise an exception.

3.3 Type Parameterization

Classes (as discussed up to now) introduce type constants. Classes can be turned into *generic classes* by type parameterization. Parameterization is particularly useful when defining generic container classes. As a database group, our current research focuses on the design of a highly reusable container class library with sophisticated iteration and query facilities, utilizing the advanced typing capabilities of TooL.

In TooL, parameterized classes are not simple templates that can only be type-checked after instantiation like in C++ or in Trellis. Type parameters are *bounded* by a type that permits local, modular type checking within the scope of the quantifier.

In the following example, the element type of sets is parameterized, but constrained to be a subtype of Object in order to allow some basic messages to elements (e.g. comparisons for object identity and printing):

```
class Set(E <: Object)
  add(e :E) :Void
  includes(e :E) :Bool
  inject(F <: Object, unit :F, f :Fun(:F, :E):F) :F
  printOn(aStream :WriteStream(Char))
```

This class interface shows that type parameterization is also available in individual method and function signatures, like in the higher-order inject method that iterates over all elements of type E within the set, accumulating the values computed by a binary user-specified function f on arguments of type F and E, given an initial value unit of type F.

As can be seen from the example above, TooL incorporates the full power of bounded parametric polymorphism as found in $F_{<:}$ [Cardelli *et al.* 1991].

TooL provides type argument synthesis in message sends and function applications which makes it possible to write, for example, intSet.inject(0, plus) instead of intSet.inject(:Int, 0, plus). This is particularly useful in the typing of control structures modeled with message passing.

Functions do not introduce additional complexity at the type level since they are treated as objects supporting an apply method. This also scales to polymorphic and higher-order functions.

3.4 Type Matching

In addition to subtyping, TooL provides a second relation between (usually recursive) object types, called *matching* (denoted by A <*: B) which has been proposed [Black and Hutchinson 1990; Bruce 1993] to overcome some well-known problems with subtyping [Canning *et al.* 1989]. In general, matching does not support subsumption (see section 3.5 for a relaxation of this statement), but it supports the inheritance and specialization of methods with negative (contravariant) occurrences of the recursion variable (with *binary methods* as a special case).

Intuitively, the matching relation captures certain forms of self-referential similarity of object types. In the following example, we define a class Equality supporting a single infix equality predicate ("="):

```
class Equality
  "="(x :Self) :Bool
```

The keyword Self (similar to MyType in PolyTOIL [Bruce 1993]) is used in TooL to indicate the self reference of object types explicitly.

Exploiting the notion of matching in TooL we can write a generic method "!=" that compares two objects for equality and returns the negated result. All that is required to know is that both objects are of some type T that matches Equality:

```
"!="(T <*: Equality, x :T, y :T)
  { !(x = y) }
```

In this example, the method type parameter T can be instantiated by arbitrary types that *match* the type Equality, like the following class Int, that exports two additional binary infix operators:

```
class Int
  "="(x :Self) :Bool
  "+"(x :Self) :Self
  "-"(x :Self) :Self
```

Note that the *contravariant* occurrence of the Self type in the method signature of "=" prevents a subtype relationship between Int and Equality. Therefore, we cannot use subtyping instead of matching in the signature of "!=" (e.g. T <: Equality) if we want to apply this polymorphic method to objects of type Int, instantiating T with Int.

Similar to the subtype relation, the TooL matching relation is defined by structural induction on object types; it is reflexive and transitive and has Void and Nil at its top and bottom element, respectively. An object type A matches another object type B (denoted by A <*: B) iff they are subtypes (A <: B) under the assumption that the corresponding Self types are equal. In the example above, the relationship Int <*: Equality holds implicitly, again without any explicit declarations of relationships between these two classes.

As explained in section 4, the explicit type quantification in the definition of the method "!=" is usually avoided by defining "!=" in class Equality utilizing the quantification of the Self type within classes during inheritance.

In class definitions, the TooL programmer has the choice between implicit recursion by class name (to promote subtyping) and explicit recursion with the keyword Self (to enable matching). For example, one could write

```
class Equality
  "="(x :Equality) :Bool
```

to decouple the receiver type of the infix message from its argument type in future subclasses. Similar design issues are discussed in section 4.1.

3.5 Interaction between Subtyping, Matching and Parameterization

Up to now, we have only discussed the subtyping and matching relation in isolation. Since TooL makes heavy use of parameterized types, a given piece of TooL code typically refers to multiple types and type *variables*, some of which are bounded by matching, others by subtyping. It is therefore crucial to have expressive type rules that refer elements in the subtyping and matching lattice.

The following type rule states that we can assume that, within a static context S, a type variable X is a subtype of a given type T, if we know that, within the same static context, X matches an object type with method suite M, *and* we are able to prove that this object type is a subtype of T, whereby all occurrences of Self within the method suite have been replaced by X:

[Match vs. Subtype]

$$\frac{S \vdash X <\!\!*\!: ObjectType(Self)M \quad S, X <: T \vdash ObjectType(Self)M[X/Self] <: T}{S \vdash X <: T}$$

From the languages incorporating matching, only TooL and Emerald [Black and Hutchinson 1990] provide such a rule which is generalized to parameterized types in TooL. The rule can be viewed as a safe, conservative approximation of the proof steps taken by the type-checker if the exact type structure of X was known.

The above rule can be utilized, for example, to prove the trivial relationship X <: Void for any type variable X that is known to match some object type. Otherwise, special inference rules involving the top type Void would be necessary (as, for example, in PolyTOIL [Bruce *et al.* 1993b]).

Unfortunately, the inference rule above is one-way only and there is no symmetric rule to prove matching of type variables from known subtype relationships. In particular, we cannot know whether two types match (say, A <*: C) if the only thing we know about them is that the smaller one (A) is a subtype of some other type (say, A <: B). Even if this other type (B) is itself a subtype of the bigger type (i.e. B <: C), the matching relation between these two types is unknown since the type Self might have been replaced with the name of the class without affecting subtyping. In our design work towards TooL, this deficiency turned out to be the main cause of problems when integrating subtyping and matching into a single language. As we will see in the next section, these problems can be solved by adequate type parameterization.

4 Reconciling Inheritance and Polymorphism

In this section, we describe how to exploit the basic polymorphic typing concepts introduced in the previous section for the construction of object-oriented class libraries using inheritance. Again, the interaction between subtyping and matching is of particular interest.

In the preceding discussion, classes were introduced mainly as a mechanism to describe object types. In TooL, classes also serve as repositories of type and behavior specifications that can be reused and modified by multiple inheritance.

Similar to CLOS [Bobrow *et al.* 1988], a TooL class definition may give an ordered specification of its direct superclasses. Possible inheritance conflicts (name clashes) are resolved by a linearization of the inheritance tree (i.e. a class precedence list) performed by a topological sort on the superclass lattice. Inheriting from the same class more than once has no effect: TooL has no repeated inheritance as in Eiffel [Meyer 1988] or C++ [Ellis and Stroustrup 1990]. Even more elaborated schemes of conflict resolution are possible, for example allocating different roles for objects as in Fibonacci [Albano *et al.* 1993]. We chose to omit such sophisticated features to keep our language simple and to focus on the typing issues discussed in the subsequent sections.

4.1 Typing Self

During the type-checking of a given TooL class C, the method bodies have to be checked with a certain assumption about the type Self of the receiver object denoted by self. This assumption must take into account *all* possible extensions of C due to subclassing since we want to perform modular type-checking.

In most commonly used object-oriented languages (e.g. C++, ObjectPascal, Modula-3, Eiffel), subclassing means subtyping. In these languages, Self is known to be a subtype of the current class[2]. We say that Self is *subtype-bounded* by the type of the current class.

In some newer languages (e.g. TOOPLE [Bruce 1993], PolyTOIL [Bruce *et al.* 1993b]), the subtype hierarchy (implicitly defined by the subtype relation '$<:$') does not have to be equal to the class hierarchy (explicitly defined by inheritance declarations): class *specialization* (by inheritance) can lead to *incompatible* types that are not related by subtyping any more. This was motivated by research results on subtyping and inheritance [Cook *et al.* 1990].

But the inheritance rules do ensure *matching* of subclasses in these languages. This means that the only assumption that can be made during (modular) type-checking a certain class is that any subclass, and, therefore, the type Self, will always match the current class. From this point of view, the matching relation reflects the restrictions on constructing subclasses by extension or modification of methods of the superclass.

[2]If the notion of a type Self is part of the language at all, which is not the case in C++, for example.

This *match-bounded*[3] Self typing provides more flexibility because we are not constrained to produce subtypes during subclassing. There are, however, situations in which it would be better to know for sure that subclasses will indeed be subtypes.

Fortunately, it is possible to integrate both kinds of Self typing (subtype-bounded and match-bounded) into a single language, leaving the choice to the programmer. In general, it is useful to use match-bounded Self typing in higher classes (supporting inheritance of methods with contravariant occurrences of Self, e.g. binary methods), and switching to subtype-bounded self typing when going down the inheritance lattice (supporting subsumption). To provide this flexibility in TooL, the constraint on the Self type that is assumed by the type-checker can be specified explicitly using the following notation:

```
class Equality ...
Self <*: class Equality ...
Self <: class Equality ...
Self = class Equality ...
```

where class Equality ... is equivalent to Self <*: class Equality .... We now discuss the advantages and disadvantages of theses alternatives in turn.

Suppose we have modeled points in the usual way, with coordinates as slots and an equality operation. We would like to inherit the default implementation of inequality from our class Equality, but we override the default implementation there (that uses simple object identity):

```
Self <*: class Equality
  super Object
  "="(x :Self) :Bool { self == x }
  "!="(x :Self) :Bool { !(self = x) }

Self <: class Point
  super Equality
  x :Int
  y :Int
  "="(aPoint :Self) :Bool
    { x = aPoint.x & y = aPoint.y }
  paint(aPen :Pen) :Void
    { aPen.dot(self) }

Self <: class ColoredPoint
  super Point
  color :Color
  ...
```

In the class Point (and also in ColoredPoint), we have explicitly specified which assumption on Self the type-checker should make, i.e. that subclasses will always be subtypes of Point. This specification allows us to exploit subsumption with the receiver object (denoted with the pseudo-variable self) by passing it as a parameter to an operation that expects a Point as an argument, e.g. the dot method of Pen.

[3] We avoid the term *F-bounded* here to prevent confusion about our understanding of matching as higher-order subtyping (cf. section 3.4)

The specification that subclasses will be subtypes is, of course, verified when actual subclassing takes place: no further specializations of methods with contravariant occurrences of Self are allowed. For example, we may not overwrite the equality method in the subclass ColoredPoint, even though we might be tempted to take the color attribute into account during the comparison:

```
Self <: BadColoredPoint
  super Point
  color :Color
  "="(aColoredPoint :Self) :Bool
    { super."="(aColoredPoint) & color = aColoredPoint.color }
```

This code fails to type-check in TooL. If the paint method would be invoked on such a (incorrectly defined) colored point, the code in class Point may break since we pass to the pen an object (i.e. self) with an interface that does not conform to Point. The dot method of pen might compare the (incorrectly defined) colored point with some other (ordinary) point, forcing a runtime error since the other point does not support colors.

The effect of the above Self constraint specification is that, during type-checking subclasses (e.g ColoredPoint) of the given class, all occurrences of Self in superclasses are replaced by the corresponding superclass name, turning explicit self references into implicit ones[4]. This means in particular that a subclass need not match a superclass any more in TooL. In the above example, ColoredPoint does not match Equality since the Self argument type of the equality method (defined in the superclass) has been replaced by Point (the name of the superclass) during subclassing.

Another constraint on the type Self that can be specified in TooL is that it will always match the current class. This constraint is the default in TooL since it is the most flexible one with respect to subtyping. We specified this constraint on Self in the class Equality above, which permitted us to overwrite the equality method with contravariant occurrences of Self in the subclass Point, producing a subclasses that is not a subtype of Equality. Note that we could make the same design decision with class Point, allowing further refinement of the equality method in class BadColoredPoint. But then we loose subsumption, the paint method in class Point will fail to type-check. If we want to support both subsumption and refinement, the only possible solution is to perform a dynamic type test in the equality method in BadColoredPoint to check whether the other given point is colored or not. A restricted form of such a dynamic type test is performed in languages that support multi-methods (see section 6).

The third (and last) possible constraint on the type Self that can be specified is that it will be *exactly* the same as the type of the current class. This type of Self constraint is required in some leaf classes of TooL (e.g. Int, Char) that provide builtin functionality for literal constants (numbers, characters). It allows literal constants to be used as return values in

[4]At a first glance, it seems to suffice to replace the negative (contravariant) occurrences of Self only, as proposed in [Eifrig *et al.* 1994]. This would provide more accurate type information in subclasses for the positive occurrences of Self since these would be automatically specialized in subclasses. However, this approach is unsound if we do not tread positive (covariant) and negative occurrences of Self as different types. We are currently investigating whether the benefits of such an approach will outweight the additional complexity of the type system.

methods that are declared to return a value of type Self. Such declarations are usually given in some superclass.

4.2 Parameterized Classes

Referring to the example class Set(E) defined in section 3.3, we may define a subclass PointSet that further constrains the element type to be a subtype of Point (cf. section 4.1). This more specific type information enables us to access the coordinates of points stored within the set, which can be used to implement a method that computes the average value of all X coordinates:

```
class PointSet(E <: Point)
  super Set(E)
  averageX :Int
    self.inject(0, fun(total :Int, e :E) total+e.x) / size
```

As we can see in this example, type parameters of superclasses must be instantiated explicitly in the subclass. This is the place where a consistency check is made by the TooL type-checker: the specified actual type parameter must conform to the bound specified for the formal parameter, i.e Point must be a subtype of Object.

In TooL, type parameters to classes can also be *match-bounded* by some object type. For example, we might define a subclass of Set that constrains the element types to match Equality:

```
class EqualitySet(E <*: Equality)
  super Set(E)
  includes(x :E) :Bool
    elements.some(fun(e :E) e = x)
```

Here, the more specific constraint on the element type E enables us to use the equality test (rather than object identity) for the set membership test.

Note that our inference rule to obtain subtyping from matching (section 3.5) is required here in order to ensure that all types matching Equality will also be subtypes of Object. Otherwise, we cannot ensure type safety by checking the methods in EqualitySet in isolation, but we have to type-check Set again in the context of the new subclass (where the element type is bounded differently), losing the modular property of our type checker.

Although the above example class would also type-check if we had used subtyping for the bound specification (e.g. class EqualitySet(E <: Equality)), we would not be allowed to instantiate this class with an element type Point (cf. section 4.1) since Point is not a subtype of Equality. Unfortunately, a similar argument prevents us from instantiating our EqualitySet with an element type ColoredPoint (as defined in section 4.1), since ColoredPoint does not match Equality[5]! This situation is clearly unsatisfactory. We would like to instantiate our container class with any type that supports a sufficient equality operation, which is obviously the case for ColoredPoints.

[5]Recall that, according to the inheritance rules of TooL, the type Self is replaced in superclasses by the corresponding superclass name iff a subtype-bounded Self typing is used.

The solution to this problem is a somewhat more complex parameterization of our container class that combines both subtyping and matching:

```
class EqualitySet(T <*: Equality, E <: T)
  super Set(E)
  includes(x :E) :Bool
    { elements.some(fun(e :E) e = x) }
```

At first glance, this parameterization appears to be unnecessarily complex, but it intimately reflects our modeling within the inheritance hierarchy: there is a type T (Point) that matches Equality, and the element type E (ColoredPoint) is a subtype thereof.

Currently, the above parameterization evolves as a standard design pattern within the TooL class library. More complex parameterizations (such as alternating chains of subtype- and match-bounded type parameters) do not seem to be necessary. This observation gives rise to hope that programmers will be able to understand and use the parameterization with both subtyping and matching, an issue that must be taken into account when designing a practical programming language.

4.3 Typing Metaclasses

Since classes are objects (like any TooL language entity) and every TooL object is required to be an instance of some class, the need for *metaclasses* arises naturally. While in Smalltalk-80, the metaclass hierarchy is (implicitly) constrained to parallel (mirror) the normal class hierarchy, in TooL these two hierarchies are decoupled. This raises opportunities for powerful reflective meta-programming [Kiczales *et al.* 1991; Briot and Cointe 1989; Ferber 1989; Danforth and Forman 1994]. Not only traditional constructs that are usually hard-wired into the language (like abstract or virtual classes, for example), but also more elaborate database functionality like class extension management and query facilities can be provided gracefully as a system add-on by defining new metaclasses.

The major problem with metaclass typing is how to type the new message that may be sent to an instance of a metaclass (which is a class), and that returns an instance of the instance of the metaclass. In Strongtalk, the special keyword Instance has been introduced for this purpose. But it has turned out in TooL that the class parameterization mechanism (section 4.2) is already sufficiently powerful. The idea is to parameterize each metaclass with the instance type. For example, the metaclass Class (which is the default metaclass in TooL) is defined as follows:

```
class Class(Instance <: Object)
  super Behavior
  new :Instance
    { <builtin "Class new"> }   ; implementation: a builtin method
```

Instances of this metaclass (i.e., ordinary classes) are then typed by instantiating the Instance type parameter with the class itself.

4.4 Privacy and Encapsulation

TooL supports the clean separation of the *interface specification* of an object (the set of messages an external client can send to the object), and the implementation of an object (i.e. the corresponding method bodies to these messages). These two parts of an object are specified in the *public* and *private* parts of a class description, respectively. The only object which can legally apply private methods is the object itself. The expressions where these methods can be used are restricted to (statically determinable) messages to the pseudo-variables `self` and `super`.

Currently, no attempt is made in TooL to provide some extra typing of the *specialization interface* [Lamping 1993] that is visible to subclasses of a given class. In TooL, all private methods and slots are visible within subclasses, i.e. the *private* protection level corresponds to the *protected* level of C++.

5 TooL Implementation Status

TooL is fully implemented within the Tycoon system infrastructure developed by our group at Hamburg University during the last four years. The implementation of TooL itself required six months of work of the first author (an experienced Tycoon programmer) who developed a TooL-specific type-checker, a dependency manager for separately compiled classes, and extensions to the Tycoon runtime system for dynamic method dispatch. The efficiency of the current implementation is sufficient for small to medium-sized programs. Work is in progress to adapt the Tycoon static and dynamic optimizer to the specific needs of TooL, incorporating customized compilation [Chambers and Ungar 1991; Hölzle 1994; Gawecki 1992].

The newly developed fully-fledged polymorphic type-checker implements structural conformance tests between recursive types using a cache of already verified conformance relations to detect cycles as described in [Amadio and Cardelli 1993]. Furthermore, it utilizes the technique of explicit substitutions [Abadi *et al.* 1990] for lazy polymorphic type variable instantiation.

The TooL parser was built using the Tycoon extensible grammar package that is based on work described in [Cardelli *et al.* 1994]. TooL also reuses the Tycoon intermediate code representation *TML* (Tycoon Machine Language), an untyped lambda calculus extended with imperative constructs that serves as an optimizable, portable program representation in distributed heterogeneous environments. TML is based on persistent CPS terms described in [Gawecki and Matthes 1994; Gawecki and Matthes 1995],

The TooL runtime system utilizes the Tycoon Store Protocol which provides a data-model-independent object store protocol based on the notion of a *persistent heap* that shields TML evaluators (and TooL programmers) from operational aspects of the underlying persistent store like access optimization, storage reclamation, concurrency or recovery. A key contribution of the TSP to the overall Tycoon system functionality is support for *orthogonal persistence* [Atkinson and Bunemann 1987]: objects and classes (including code and threads) can exist as long or as short as required by the application. Programmers do not need to write explicit code to move data between persistent and volatile store.

Currently, there exist three different TSP implementations that can be combined freely with the existing TML evaluators: a compact, paged main memory store implementation with a simple file-based persistence mechanism [Matthes *et al.* 1992]; an interface to the Napier persistent object store [Brown *et al.* 1991] that provides an efficient single-user stable store with recovery mechanisms based on shadow copying exploiting low-level memory management hardware support; and a gateway to the object-oriented database system ObjectStore [Lamb *et al.* 1992] supporting recoverable multi-user access to shared databases in client-server architectures. All store implementations feature automatic garbage collection and portable data import and export to files.

The Tycoon runtime system is written in C and available on several hardware and software platforms including SunOS, Solaris, Macintosh, Linux and Windows.

6 Related Work

The common formal interpretation of matching is as a form of F-bounded subtyping [Canning *et al.* 1989]. Abadi and Cardelli [Abadi and Cardelli 1995] propose to interpret matching as higher-order subtyping, arguing that this interpretation leads to better properties of the matching relation, e.g. reflexivity and transitivity. The implementation of the matching relation in TooL conforms to this interpretation. An equivalent interpretation has been given in [Black and Hutchinson 1990], using a somewhat different terminology: Black and Hutchinson use the terms *namemaps* (object types) and *namemap generators* (type operators).

To our knowledge, Emerald [Black and Hutchinson 1990] was the first language incorporating both subtyping and matching, but it does not support classes and inheritance. No distinction between ordinary recursion and self-reference was made.

The languages TOOPLE [Bruce *et al.* 1993a] also integrates subtyping and matching but lacks type rules that relate both notions (see section 3.5). PolyTOIL is a recent successor to TOOPLE that adds polymorphism but is restricted to match-bounded quantification [Bruce *et al.* 1993b]. TooL's parameterized classes could be modeled with type operators in PolyTOIL. However, the interaction between parameterization and inheritance is not addressed in [Bruce *et al.* 1993b].

Strongtalk [Bracha and Griswold 1993] aims like TooL to support strong typing in a pure object-oriented language. Universal type quantification is provided, but no form of (match or subtype-) bounded quantification is available. As discussed in section 4.3, contrary to TooL, Strongtalk relies on extra typing machinery to type-check metaclasses.

In LOOP [Eifrig *et al.* 1994], no destinction between subtyping and matching is made, attempting to merge the two relations into one, which seems to be the least common denominator of the two relations. The subtyping rules of LOOP are not as powerful as those of TooL and PolyTOIL. While LOOP does not provide any form of polymorphism, correctness and decidability have been proved formally.

Multi-methods have been proposed as a solution to the binary method problem (covariance vs. contravariance) by several authors [Ghelli 1991; Castagna 1994; Chambers 1993]. Multi-methods circumvent the problem by choosing an appropriate method implementation on behalf of the dynamic class or type of *every* argument of a message, not merely the receiver alone. However, multi-methods expose other problems of which the lack of encapsulation is the most serious one. Moreover, most current multi-method approaches identify classes with types again and, therefore, identify inheritance and subtyping. Even in Cecil [Chambers and Leavens 1994], where these problems are addressed (subtype and inheritance graphs are allowed to differ), an explicit declaration of subtyping is required. Therefore, all these models do not scale well into distributed, open environments where some form of structural subtyping (or matching) is needed [Black and Hutchinson 1990].

7 Conclusion

The main contribution of this work is the integration of subtyping, type matching and type parameterization in an orthogonal language design that minimizes built-in language functionality in favor of flexible system add-ons, both at the level of values and at the level of types. The TooL persistent language is fully implemented and we have reported on our initial experience in using TooL for pure object-oriented library construction.

More work remains to be done, like a formal investigation of the soundness, completeness and expressive power of our type system. In this context it is interesting to note that TooL inherits undecidability from $F_{<:}$ [Pierce 1992] since it employs the same powerful contravariant subtyping rule on polymorphic functions and methods. However, during several years of use of the Tycoon language TL we never ran into problems with (ill-typed) programs that caused our type-checker to loop endlessly, which we regard as a strong evidence that this particular form of undecidability is of little practical consequence.

References

Abadi and Cardelli 1995: Abadi, M. and Cardelli, L. On Subtyping and Matching. In *Proceedings ECOOP'95*. Springer-Verlag, 1995. (To appear).

Abadi et al. *1989:* Abadi, M., Cardelli, L., Pierce, B. C., and Plotkin, G.D. Dynamic typing in a statically typed language. Technical Report 47, Digital Equipment Corporation, Systems Research Center, Palo-Alto, California, June 1989.

Abadi et al. *1990:* Abadi, M., Cardelli, L., Curien, P.-L., and Lévy, J.-J. Explicit substitutions. Technical Report 54, Digital Equipment Corporation, Systems Research Center, Palo-Alto, California, February 1990.

Albano et al. *1993:* Albano, A., Bergamini, R., Ghelli, G., and Orsini, R. An introduction to the database programming language Fibonacci. FIDE Technical Report Series FIDE/92/64, FIDE Project Coordinator, Department of Computing Sciences, University of Glasgow, Glasgow G128QQ, 1993.

Amadio and Cardelli 1993: Amadio, R.M. and Cardelli, L. Subtyping recursive types. *ACM Transactions on Programming Languages and Systems*, 15(4), September 1993.

Atkinson and Bunemann 1987: Atkinson, M.P. and Bunemann, P. Types and persistence in database programming languages. *ACM Computing Surveys*, 19(2), June 1987.

Birell et al. *1993:* Birell, A., Nelson, G., Owicki, S., and Wobber, E. Network objects. In *14th ACM Symposium on Operating System Principles*, pages 217–230, June 1993.

Black and Hutchinson 1990: Black, Andrew P. and Hutchinson, Norman C. Typechecking polymorphism in Emerald. Technical Report TR 90-34, Dept. of Computer Science, University of Arizona, December 1990.

Bobrow et al. *1988:* Bobrow, D.G., De Michiel, L.G., Gabriel, R.P., Keene, S.E., Kiczales, G., and Moon, D.A. Common lisp object system specification. *ACM SIGPLAN Notices*, 23, September 1988.

Bracha and Griswold 1993: Bracha, Gilad and Griswold, David. Strongtalk: typechecking Smalltalk in a production environment. In *Proceedings OOPSLA'93*, pages 215–230, October 1993.

Briot and Cointe 1989: Briot, J-P. and Cointe, P. Programming with explicit metaclasses in Smalltalk-80. In *Proceedings of the Object-Oriented Programming Systems, Languages and Applications Conference, New Orleans, Louisiana*, pages 419 – 431, 1989.

Brown et al. *1991:* Brown, A.L., Mainetto, G., Matthes, F., Müller, R., and McNally, D.J. An open system architecture for a persistent object store. Persistent Programming Research Report CS/91/9, University of St. Andrews, Department of Computing Science, September 1991.

Bruce et al. *1993a:* Bruce, K.B., Crabtree, J., Murtagh, T.P., Gent, Robert van, Dimock, Allyn, and Muller, Robert. Safe and decidable type checking in an object-oriented language. In *Proceedings OOPSLA'93*, pages 29–46, October 1993.

Bruce et al. *1993b:* Bruce, K.B., Schuett, A., and Gent, R. van. PolyTOIL: a type-safe polymorphic object-oriented language. In *Proceedings ECOOP'95*. Springer-Verlag, 1993. (To appear).

Bruce 1993: Bruce, K.B. Safe type checking in a statically typed object-oriented language. In *Proceedings of the Twentieth ACM Symposium on Principles of Programming Languages*, 1993.

Canning et al. *1989:* Canning, P.S., Cook, W.R., Hill, W.L., and Olthoff, W. F-bounded polymorphism for object-oriented programming. In *Proceedings of Conference on Functional Proramming Languages and Computer Architecture (FPCA'89), Imperial College, London*, pages 273–280, September 1989.

Cardelli and Longo 1991: Cardelli, L. and Longo, G. A semantic basis for Quest. *Journal of Functional Programming*, 1(4):417–458, October 1991.

Cardelli et al. *1991:* Cardelli, L., Martini, S., Mitchell, J.C., and Scedrov, A. An extension of system F with subtyping. In Ito, T. and Meyer, A.R., editors, *Theoretical Aspects of Computer Software, TACS'91*, Lecture Notes in Computer Science, pages 750–770. Springer-Verlag, 1991.

Cardelli et al. *1994:* Cardelli, L., Matthes, F., and Abadi, M. Extensible grammars for language specialization. In Beeri, C., Ohori, A., and Shasha, D.E., editors, *Proceedings of the Fourth International Workshop on Database Programming Languages, Manhatten, New York*, Workshops in Computing, pages 11–31. Springer-Verlag, February 1994.

Cardelli 1990: Cardelli, L. The Quest language and system (tracking draft). Technical report, Digital Equipment Corporation, Systems Research Center, Palo-Alto, California, 1990. (shipped as part of the Quest V.12 system distribution).

Castagna 1994: Castagna, G. Covariance and contravariance: conflict without a cause. Technical Report liens-94-18, LIENS, October 1994.

Chambers and Leavens 1994: Chambers, Craig and Leavens, Gary T. Typechecking and modules for multi-methods. In *Proceedings OOPSLA '94*, volume 29 of *ACM SIGPLAN Notices*, pages 1–15. Association for Computing Machinery, October 1994.

Chambers and Ungar 1991: Chambers, C. and Ungar, D. Making pure object-oriented languages practical. In *Proceedings of the Object-Oriented Programming Systems, Languages and Applications Conference, Phoenix, Arizona*, pages 1–15, October 1991.

Chambers 1993: Chambers, C. Object-oriented multi-methods in Cecil. In *Proceedings of the ECOOP'92 Conference, Uetrecht, the Netherlands*, pages 33–56. Springer-Verlag, July 1993.

Cook et al. *1990:* Cook, W.R., Hill, W.L., and Canning, P.S. Inheritance is not subtyping. In *Proceedings of the Seventeenth ACM Symposium on Principles of Programming Languages*, pages 125–135, 1990.

Danforth and Forman 1994: Danforth, S. and Forman, I. R. Reflections on metaclass programming in SOM. In *Proceedings OOPSLA'94*, pages 440–452, October 1994.

Eifrig et al. *1994:* Eifrig, J., Smith, S., Trifonov, V., and Zwarico, A. Application of OOP type theory: State, decidability, integration. In *Proceedings OOPSLA'94*, pages 16–30, October 1994.

Ellis and Stroustrup 1990: Ellis, M.A. and Stroustrup, B. *The Annotated C++ Reference Manual.* Addison-Wesley Publishing Company, 1990.

Ferber 1989: Ferber, J. Computational reflection in class based object oriented languages. In *Proceedings of the Object-Oriented Programming Systems, Languages and Applications Conference, New Orleans, Louisiana*, pages 317 – 326, 1989.

Gawecki and Matthes 1994: Gawecki, A. and Matthes, F. The Tycoon Machine Language TML - an optimizable persistent program representation. FIDE Technical Report FIDE/94/100, Fachbereich Informatik, Universität Hamburg, Germany, July 1994.

Gawecki and Matthes 1995: Gawecki, A. and Matthes, F. Integrating query and program optimization using persistent CPS representations. (submitted for publication), March 1995.

Gawecki et al. *1995:* Gawecki, A., Matthes, F., and Schmidt, J.W. Definition of the Tycoon object-oriented Language TooL. (In preparation), May 1995.

Gawecki 1992: Gawecki, A. An optimizing compiler for Smalltalk. Bericht FBI-HH-B-152/92, Fachbereich Informatik, Universität Hamburg, Germany, September 1992. In German.

Ghelli 1991: Ghelli, G. A static type system for message passing. In *Proceedings of the Object-Oriented Programming Systems, Languages and Applications Conference, Phoenix, Arizona*, pages 129–145, 1991.

Goguen 1990: Goguen, J.A. Higher-order functions considered unnecessary for higher-order programming. In Turner, D., editor, *Research Topics in Functional Programming*, pages 309–351. Addison-Wesley Publishing Company, 1990.

Goldberg and Robson 1983: Goldberg, Adele and Robson, David. *Smalltalk 80: the Language and its Implementation.* Addison-Wesley, May 1983.

Hewitt 1987: Hewitt, C. Viewing control structures as patterns of passing messages. *Artificial Intelligence*, (8):323–364, 1987.

Hölzle 1994: Hölzle, U. *Adaptive Optimization for Self: Reconciling high performance with Exploratory Programming.* PhD thesis, Stanford University, August 1994.

Hutchinson 1987: Hutchinson, Norman C. *Emerald: An Object-Based Language for Distributed Programming.* PhD thesis, University of Washington, September 1987.

Johnson and Foote 1988: Johnson, Ralph E. and Foote, Brian. Designing reusable classes. *Journal of Object-Oriented Programming*, 1(2), 1988.

Kiczales et al. *1991:* Kiczales, G., Rivieres, J., and Bobrow, D.G. *The Art of the Metaobject Protocol.* MIT Press, 1991.

Lamb et al. *1992:* Lamb, C., Landis, G., Orenstein, J., and Weinreb, D. The ObjectStore database system. *Communications of the ACM*, 34(10):50–64, 1992.

Lamping 1993: Lamping, John. Typing the specialization interface. In *Proceedings OOPSLA '93*, pages 201–214, October 1993.

Mathiske et al. *1995:* Mathiske, B., Matthes, F., and Schmidt, J.W. On migrating threads. In *Proceedings of the Second International Workshop on Next Generation Information Technologies and Systems, Naharia, Israel*, June 1995. (to appear).

Matthes and Schmidt 1991: Matthes, F. and Schmidt, J.W. Bulk types: Built-in or add-on? In *Proceedings of the Third International Workshop on Database Programming Languages, Nafplion, Greece.* Morgan Kaufmann Publishers, September 1991.

Matthes and Schmidt 1992: Matthes, F. and Schmidt, J.W. Definition of the Tycoon Language TL – a preliminary report. Informatik Fachbericht FBI-HH-B-160/92, Fachbereich Informatik, Universität Hamburg, Germany, November 1992.

Matthes and Schmidt 1993: Matthes, F. and Schmidt, J.W. System construction in the Tycoon environment: Architectures, interfaces and gateways. In Spies, P.P., editor, *Proceedings of Euro-Arch'93 Congress*, pages 301–317. Springer-Verlag, October 1993.

Matthes and Schmidt 1994: Matthes, F. and Schmidt, J.W. Persistent threads. In *Proceedings of the Twentieth International Conference on Very Large Data Bases, VLDB*, pages 403–414, Santiago, Chile, September 1994.

Matthes et al. *1992:* Matthes, F., Müller, R., and Schmidt, J.W. Object stores as servers in persistent programming environments – the p-quest experience. FIDE Technical Report Series FIDE/92/48, FIDE Project Coordinator, Department of Computing Sciences, University of Glasgow, Glasgow G128QQ, July 1992.

Matthes et al. *1994:* Matthes, F., Müßig, S., and Schmidt, J.W. Persistent polymorphic programming in Tycoon: An introduction. FIDE Technical Report FIDE/94/106, Fachbereich Informatik, Universität Hamburg, Germany, August 1994.

Matthes 1993: Matthes, F. *Persistente Objektsysteme: Integrierte Datenbankentwicklung und Programmerstellung.* Springer-Verlag, 1993. (In German.).

Meyer 1988: Meyer, B. *Object-oriented Software Construction.* International Series in Computer Science. Prentice Hall, Englewood Cliffs, New Jersey, 1988.

Meyer 1989: Meyer, B. Static typing for Eiffel. (Technical report distributed with Eiffel Release 2), July 1989.

Milner et al. *1990:* Milner, R., Tofte, M., and Harper, R. *The Definition of Standard ML.* MIT Press, Cambridge, Massachusetts, 1990.

Pierce and Turner 1993: Pierce, B.C. and Turner, D.N. Statically typed friendly functions via partially abstract types. Rapport de Recherche 1899, INRIA, Domaine de Voluceau Rocquencourt 78153 Le Chesnay Cedex - France, May 1993.

Pierce 1992: Pierce, B. C. Bounded quantification is undecidable. In *Proceedings of the Nineteenth ACM Symposium on Principles of Programming Languages*, pages 305–315, January 1992.

Ungar and Smith 1987: Ungar, D. and Smith, R.B. Self: The power of simplicity. In *Proceedings of the Object-Oriented Programming Systems, Languages and Applications Conference, Orlando, Florida*, pages 227–242, 1987.