Appeared in: *Proceedings of the 10th European Conference on Object-Oriented Programming, ECOOP'96*, Linz, Austria, July 1996. Springer Verlag.

Integrating Subtyping, Matching and Type Quantification: A Practical Perspective*

Andreas Gawecki Florian Matthes

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

Abstract. We report on our experience gained in designing, implementing and using a strongly-typed persistent programming language (TooL) which integrates object types, subtyping, type matching, and type quantification. Our work complements recent type-theoretical studies of subtyping and type matching by focusing on the issue of how to integrate both concepts into a practical, orthogonal programming language. We also shed some light on the subtle typing issues which we encountered during the construction of a substantial bulk data library where it was necessary to trade-off subtyping against type matching. Our practical experience suggests that the benefits of an integration of subtyping and type matching are achieved at the expense of a significant increase in modeling complexity.

* 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).

1 Introduction and Motivation

The purpose of this paper is to report on our practical experience gained in designing, implementing, and using **TooL**, a language integrating subtyping and type matching, which should be of value for anyone who plans to incorporate type matching into a fully-fledged programming language.

The TooL project started off with the observation that existing statically-typed polymorphic programming languages with subtyping only (Modula-3 [Nelson 1991], C++ [Ellis and Stroustrup 1990], Tycoon [Matthes and Schmidt 1992] or Fibonacci [Albano *et al.* 1994]) provide application programmers with rich re-usable generic class libraries organized into subtype hierarchies. However, the type system of these languages obstructs programmers who intend to maximize code sharing between library classes through implementation inheritance following the successful library design principles of Smalltalk and Eiffel. As discussed in the literature [Black and Hutchinson 1990; Bruce 1994; Bruce *et al.* 1995b; Abadi and Cardelli 1995] and illustrated in the rest of the paper, a more liberal notion of *type matching* is needed, for example, to support the type-safe inheritance of *binary methods* [Bruce *et al.* 1995a].

The design and implementation of TooL has been heavily influenced by experience gained with the **Tycoon** language [Matthes and Schmidt 1992] developed at the University of Hamburg in the framework of the European basic research project FIDE [Atkinson 1996] where six European database language research groups collaborated towards the goal of *Fully Integrated Data Environments* using state-of-the-art language technology.

The rationale behind the original Tycoon type system is to provide a set of unbiased, orthogonal primitives to support various database programming styles, including 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 1989; Cardelli and Longo 1991] and the type theoretic model of $F^{\omega}_{<:}$ [Pierce and Turner 1993].

Based on our extensive experience using Tycoon for large-scale programming (for example, building and maintaining systems with several hundred modules) our motivation behind the design of TooL was to verify the following hypotheses: (1) A *purely* object-oriented language where objects and classes combine aggregation, encapsulation, recursion, parameterization and inheritance leads to program libraries which are more uniform and easier to understand since programmers do not have the freedom to choose between combinations of modules, records, tuples, functions, recursive declarations, etc. (2) Type matching increases code reuse within complex libraries.

To verify these hypotheses it was not only necessary to design and to implement

TooL, but also to *utilize* it for non-trivial library examples. In a nutshell, this experimental validation was carried out by augmenting the Tycoon type system by a notion of type matching, omitting existential type quantification and all higher-order type concepts like kinds, but otherwise adhering closely to the proven type and language concepts of Tycoon. We then used the functionality of the mature and highly-structured Eiffel collection library [Meyer 1990] as a yardstick for the construction of a type-safe TooL bulk type library.

This paper is organized as follows: In section 2 we provide insight into the rationale behind the TooL language design. Section 3 and section 4 as well as the appendix give an overview of the language and the libraries constructed with the language. Section 5 motivates and explains the TooL subtyping, matching and type quantification rules. The TooL inheritance rules and the non-trivial interaction of subtyping, matching and type quantification are discussed in section 6 and section 7, respectively. The impact of this interaction on the use of the TooL language for library programming is described in section 8. The paper ends with a comparison with related work and a summary of our research contributions.

2 TooL Design Goals

The key aspects of the TooL language design can be summarized as follows:

Purely object-oriented: 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 *purely* 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 1977]. Even low-level operations such as integer arithmetic, variable access, and array indexing are uniformly expressed by sending messages to objects.
It should be noted that modern compiler technology eliminates most of the run-time performance overhead traditionally associated with the purely object-oriented approach [Chambers and Ungar 1991; Hölzle 1994; Gawecki 1992]. For example, TooL uses dynamic optimization across abstraction barriers based on a persistent continuation passing style (CPS) program representation to "compile away" many message sends [Gawecki and Matthes 1996].

Higher-order functions as objects: Contrary to other statically-typed object-oriented languages [Goguen 1990], TooL provides statically-scoped higher-order functions which are viewed as first-class objects that understand messages. Thereby control structures like loops and conditionals 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 instance and pool variables (which unify the concepts of global and class variables in

Smalltalk) are accessed by sending messages [Johnson and Foote 1988]. This unification 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.

Strong and static typing: No operation will ever be invoked on an object which does not support it, i.e. errors like "message not understood" cannot occur at run time. Type rules are defined in a "natural-deduction" style based on the abstract TooL syntax similar to [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.

Structural type checking: 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.

Modular type checking: During type checking of a given class only the interfaces of imported classes and of superclasses have to be accessed. 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 is in conflict with substitutability. Therefore, Eiffel requires some form of global data flow analysis at link-time to ensure type correctness. Such an analysis generally

requires a representation of the source code of all classes and methods which 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 TooL Syntax Overview

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. This semantic simplicity and orthogonality is reflected by the abstract syntax of TooL depicted in figure 1 which provides a starting point for the definition of the static semantics of TooL and which constitutes the canonical internal representation used by the TooL language processors.

The definition of the abstract TooL syntax involves syntactic objects that are denoted by meta variables using the following naming conventions:

$X, Y, Self$	type and class identifiers
x, y	value identifiers
ψ	type relations
S	type and value signatures
D	type and value bindings
T	type expressions
A	named type expressions
v	values
m	method selectors
c	named class definitions
$slots$	instance variable declarations
$methods$	method declarations

Since most aspects of the abstract syntax are similar to other polymorphically-typed languages [Cardelli and Longo 1991; Cardelli 1993], we only highlight some of the productions.

A TooL program is a set of (mutually recursive) named class definitions $c_1, \ldots, c_n$.

A class definition defines the name of the $Self$ type (see section 5.2), the name X of the class, the signatures S of the class type parameters, an ordered sequence of direct superclasses $A_1,..,A_n$, a metaclass declaration A (not treated in this paper), and a set of public and private slots and a suite of method declarations. Each method declaration specifies a method selector m_i, the signatures S_i of the method arguments, the method result type T_i, and a method body D_i given

ψ	$::= <:$	subtyping
	$\mid \lessdot\!\!*:$	type matching
	$\mid =$	type equivalence
S	$::= \oslash$	empty signature
	$\mid S, x : T$	value signature
	$\mid S, X \psi T$	type signature
D	$::= \oslash$	empty binding
	$\mid D, x = v$	value binding
	$\mid D, x : T = v$	constrained value binding
	$\mid D, X = T$	type binding
T	$::=$ **Nil**	bottom type constant
	$\mid A$	type identifier or type instantiation
	$\mid$ **Interface** $X(S; Self)$ $\{m_1(S_1)T_1, .., m_n(S_n)T_n\}$	named parameterized object type
A	$::= X$	type identifier
	$\mid X(D)$	type instantiation (type bindings only)
v	$::= b$	constants (integer, character, ... literals)
	$\mid$ **self** $\mid$ **super**	receiver object
	$\mid x$	value identifier
	$\mid$ **fun**$(S)D$	(polymorphic) function constructor
	$\mid$ **send**(m,v,D)	message send
c	$::= Self\ \psi$ **class** $X(S; A_1, .., A_n; A)$ **public** *slots methods* **private** *slots methods*	named class definition
slots	$::= x_1 : T_1, .., x_n : T_n$	
methods	$::= m_1(S_1)T_1 = D_1, .., m_n(S_n)T_n = D_n$	

Fig. 1. The TooL abstract syntax

by a list of bindings. The return value computed by a method is the value of its last value binding. A class declaration implicitly defines an object type **Interface** $X(S; Self)\{m_1(S_1)T_1, .., m_n(S_n)T_n\}$ with the same name.

A message send operation defines a message m which is sent to an object determined by evaluating the receiver expression v, passing a sequence D of type and value bindings as actual parameters.

Within bindings and signatures, a distinguished *anonymous* identifier '_' can be used to denote omitted identifiers. For example, the expression `x.m(3)` in the concrete syntax is mapped to the binding $_ =$ **send**$(m, x, (\oslash, _ = 3))$.

A function type (denoted as **Fun**(S):T in the concrete syntax of our examples) is represented as an object type with an apply method with signature S and result type T. A function abstraction **fun**$(S)D$ evaluates to an instance of such an object type.

Each slot $x : T$ is mapped to a pair of an access method and an update method

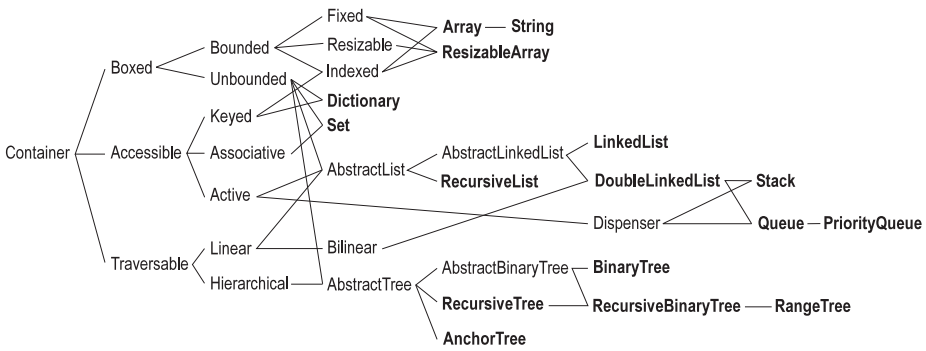

Fig. 2. Overview of the TooL bulk type library

with the signatures $x_{get}(\oslash) : T$ and $x_{set}(\oslash, _ : T) : \mathbf{Void}$, respectively.

TooL supports a separation of classes into a *public* and a *private* part. 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.

Despite the semantic simplicity of TooL, a rich set of syntactic variants exists 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. In fact, the TooL language is based on extensible grammars [Cardelli *et al.* 1994] which give users full control over the concrete syntax of TooL, for example, to support domain-specific abstractions like query language notations or concurrency control schemes. The default concrete syntax used in this paper is similar to C++ and Java.

4 The TooL Bulk Type Library

Figure 2 shows the part of the TooL class library relevant for bulk data manipulation. A bold font indicates concrete (instantiable) classes, whereas a non-bold font indicates abstract classes, i.e., classes where some or all method bodies consist of empty bindings (cf. figure 1).

This class hierarchy is designed to maximize code sharing between library classes through implementation inheritance following the successful library design principles of Smalltalk and Eiffel [Meyer 1990]. For example, the abstract classes Boxed, Accessible, and Traversable provide code fragments which capture certain aspects of containers and which can be inherited, combined and refined in subclasses to implement concrete classes such as priority queues.

In the rest of the paper, we will utilize this library not only to explain some of the more subtle TooL language design decisions, but also to shed some light on the practical issues that arise if subtyping, type matching and type quantification interact.

5 Basic TooL Type Concepts

In this section we motivate and explain the TooL subtyping, matching and type quantification rules. Their impact on the TooL inheritance rules and their non-trivial interaction is discussed in later sections.

5.1 Structural Subtyping

The main motivation for 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 5.3.

As a practical example for substitutability in the TooL libraries, 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 WriteStream of characters is expected, we have to show the subtype relationship File <: WriteStream(Char).

Note that 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 which is bounded by the type Object (cf. section 5.3) and exports a single method put to append an object e of type E to the stream. The (possibly independently defined) unparameterized class File also exports a put method, which only accepts characters, in addition to a get 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. 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.* 1995b]).

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 which 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. At runtime, however, only the methods defined in class Nil and its single superclass Object (e.g. identity testing and printing) may be legally applied, sending any other message to nil triggers an exception. The value nil is used to initialize instance variables and to mark empty slots in hash tables. The type Nil is also used to type expressions which raise an exception.

5.2 Type Matching

As explained in the introduction, a key contribution of TooL is to provide a second relation between object types in addition to subtyping. This relation is called *matching* (denoted by A <*: B) and has been introduced in type-theory [Black and Hutchinson 1990; Bruce 1994] to overcome some well-known problems with subtyping [Canning *et al.* 1989]. In general, matching does not support subsumption (see section 7 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. Similar to MyType in PolyTOIL [Bruce *et al.* 1995b] or like Current in Eiffel, a distinguished type identifier Self is used in TooL to indicate a self reference which is automatically updated during subclassing to refer to the new subclass.

Similar to the subtype relation, the matching relation is defined by structural induction on object types; it is reflexive and transitive and has Void and Nil as its top and bottom elements, respectively. An object type A matches another object

type B (denoted by A <*: B) if they are subtypes (A <: B) under the assumption that the corresponding Self types are equal. In the example below, the relationship Int <*: Equality holds implicitly, again without any explicit declarations of relationships between these two classes.

In the following example, we define a class Equality supporting a single infix equality predicate ("="):

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

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

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

In this example, the method type parameter T can be instantiated by arbitrary types which *match* the type Equality, like the following class Int which 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.

As explained in section 6, the explicit type quantification in the definition of the method "!=" can be 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 6.1.

5.3 Type Quantification

Classes as discussed up to now introduce named type constants. Classes can be turned into *generic classes* by type parameterization. Type parameterization is important for the type-safe definition of generic container classes and polymorphic iteration abstractions. For example, virtually all subclasses of Container (see figure 2) have one or several formal type parameters.

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
  map(F <: Object, f :Fun(:E):F) :Set(F)
  printOn(aStream :WriteStream(Char))
```

This class interface shows that type parameterization is also available in individual method and function signatures, for instance in the higher-order inject method which 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.

In TooL, parameterized classes are not simple templates which can only be type-checked after instantiation as in C++ or in Trellis. Type parameters are *bounded* by a type, permitting local, modular type checking within the scope of the quantifier. For example, the element type of the set returned by the polymorphic map method depends on the argument type of the function f passed as a run-time argument to the map function.

As indicated by 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, thus intSet.inject(0, plus) is equivalent to intSet.inject(:Int, 0, plus). This is particularly useful in the typing of control structures modeled with message passing. We use a simple but incomplete inference algorithm similar to the one described in [Cardelli 1993] which works well in practice.

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

6 Inheritance

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 which can be reused and modified by multiple inheritance. This is one of the main differences between TooL and Tycoon.

Similar to CLOS [Bobrow *et al.* 1988], a TooL class definition may give an ordered specification of its direct superclasses, for example the container class Indexed with an element type E inherits from two superclasses (see figure 2):

```
class Indexed(E <*: Equality)
super Bounded(E), Keyed(Int, E)
```

Possible inheritance conflicts (name clashes between inherited methods) 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]. More elaborated schemes of conflict resolution are possible, for example allocating different roles for objects as in Fibonacci [Albano *et al.* 1994]. We chose to omit such sophisticated features to keep our language simple and to focus on the typing issues discussed in the subsequent sections.

If a method specification m(S):T takes precedence during method lookup over another method specification m(S'):T', the result type T has to be a subtype of T' assuming that the signatures S' are subsignatures of the signatures S. The full type rules for inheritance in TooL are similar to those of PolyTOIL [Bruce *et al.* 1995b] and have to take parameterized classes and Self type specifications (see section 6.1) into account.

Type parameters of superclasses may be refined during inheritance. For example, a subclass PointSet can be defined which further constrains the element type of class Set to be a subtype of Point. This more specific type information enables us to access the coordinates of points stored within the set to compute 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}
```

In this example, a consistency check is performed by the TooL type-checker to verify that the specified actual type parameter conforms to the bound specified for the formal parameter, i.e Point has to be a subtype of Object.

6.1 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. Due to subclassing, this assumption must take *all* possible extensions of c into account since we want to perform modular type-checking. As described in this section, a major contribution of TooL is to give the library programmer explicit control over this particular assumption.

In the most commonly used object-oriented languages (e.g. C++, ObjectPascal, Modula-3, Eiffel), subclassing means subtyping. In these languages, Self has to 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 1994], PolyTOIL [Bruce *et al.* 1995b]), the subtype hierarchy implicitly defined by the subtype relation '<:' does not have to be the same as the class hierarchy explicitly defined by inheritance declarations: class specialization by inheritance can lead to *incompatible* types which are not related by subtyping any more. In these languages, inheritance ensures *matching* of subclasses. This means that the only assumption that can be made during modular type-checking is that Self will always match the current class. This *match-bounded* Self typing provides more flexibility because the programmer is not constrained to produce subtypes during subclassing.

Finally, there are programming situations where library designers would like to express the constraint that all subclasses will have identical types and only differ in their method implementations. For example, the TooL leaf classes (Int, Char, ...) which provide builtin functionality for literal constants (numbers, characters, ...) require this constraint to allow literal constants to appear as return values in methods with return type Self.

Therefore, TooL offers three kinds of Self typing (subtype-bounded, match-bounded and equivalence), leaving the choice to the programmer to use one of the following notations:

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

The notation class Equality ... is a shorthand for Self <*: class Equality .... We now discuss the advantages and disadvantages of theses alternatives in turn.

In general, match-bounded Self typing is used in classes close to the root of the inheritance lattice to support inheritance of methods with contravariant occurrences of Self, e.g. binary methods. A switch to subtype-bounded Self typing can

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

then be performed when going down the inheritance lattice to support subsumption.

For example, 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 class Equality, but we override the default implementation of equality which 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 classes Point and ColoredPoint, we have explicitly specified which assumption the type-checker should make about Self, 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 which expects a Point as an argument, e.g. the dot method of Pen.

6.2 Enforcing the Self Constraint

A Self constraint assumed during modular type checking is enforced when actual subclassing takes place:

1. Inheritance from a match-bounded class Self <*: **class** c is equivalent to copying the method signatures of c into the subclass.
   For example, the match-bounded Self in class Equality above makes it possible to overwrite the equality method with contravariant occurrences of Self in the subclass Point, producing a subclass which is not a subtype of Equality.
2. Inheritance from a subtype-bounded class Self <: **class** c is equivalent to copying the method signatures of c into the subclass and replacing all inherited occurrences of Self by c, turning all explicit self references into implicit ones. Therefore, subclasses do not have to match c any more. For 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).
At 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 specialized automatically in subclasses. However, this approach is unsound if we do not treat positive (covariant) and negative occurrences of Self as different types. From our practical experience we believe that the additional complexity of introducing two distinct Self type variables is not worth the relatively small gain in expressive power.
As a consequence of the subtype constraint, 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 in the following way, trying to take the color attribute into account during the comparison:

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

This code fails to type-check correctly in TooL.[3]
Note that a match-bound Self specification for class Point would enable a further refinement of the equality method in class BadColoredPoint. But then subsumption on self would be lost and the paint method in class Point would fail to type-check.
More general, to support *both* subsumption and refinement in TooL, a dynamic type test in the equality method in BadColoredPoint is required to check whether the argument is colored or not. A restricted form of such a dynamic type test is performed in languages which support multi-methods (see section 9).

3. Inheritance from a class Self = **class** C is equivalent to (1) and requires a check that no additional methods are defined or refined in the subclass.

In all three cases inheritance implies code sharing.

7 Reconciling Subtyping, Matching and Quantification

Up to now, we have discussed the subtyping and matching relation in isolation. Since TooL makes heavy use of parameterized types, a given piece of TooL code

[3] If the paint method would be invoked on such an incorrectly defined colored point, the code in class Point could break since we pass an object (i.e. self) with an interface which does not conform to Point to the pen. The dot method of pen might compare the incorrectly defined colored point with some other ordinary point, which leads to a runtime error since the other point does not support colors.

typically refers to multiple types and type *variables*, some of which are bounded by matching, others by subtyping. It is therefore important to have expressive type rules which establish relationships between elements in the subtyping and matching lattice.

The following TooL type rule states 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.

Intuitively, this rule is sound because matching requires that X must contain at least the methods present in the method suite M, and if ObjectType$(Self)M[X/Self]$ is a subtype of T, X must also contain at least the methods present in T (due to reflexivity of subtyping). This means that, by the definition of matching (see also [Abadi and Cardelli 1995; Bruce *et al.* 1995b]), X can only be not a subtype of T if M contains methods with negative (contravariant) occurrences of Self. But this case is covered since these have been replaced by X.

While it seems possible to map all other TooL type rules (not shown in this paper) in a rather straightforward way onto corresponding type rules of core languages developed for the formal study of type systems incorporating some form of matching, this particular rule indeed looks rather "ad-hoc" and suspicious to a type theoretician. Without this rule, many of TooL class definitions would not type-check successfully. From a type-theoretical point of view it should therefore be interesting to formally prove the soundness of this rule or to come up with an equivalent or a more general rule, which might be easier to prove.

This TooL type rule is utilized, for example, to prove the trivial relationship X <: Void for any type variable X which is known to match some object type. Otherwise, special inference rules involving the top type Void have to be added to the type system (as, for example, in PolyTOIL [Bruce *et al.* 1995b]).

Bounded type parameterization leads to other, more compelling practical examples where the type rule *[Match vs. Subtype]* is needed. For example, we might define a subclass of Set by inheritance which further constrains the set element types to match Equality, overriding the element constraint E <: Object (see section 5.3):

```
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. The type rule *[Match vs. Subtype]* ensures that all types matching Equality will also be subtypes of Object. Without this rule we could not ensure type safety by checking the methods in EqualitySet in isolation, but we would have to type-check Set again in the context of the new subclass (where the element type is bounded differently), violating our design principle of modular type checking.

Unfortunately, the type 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 larger 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.

For practical programming in TooL, the lack of a rule to deduce matching from subtyping is a major problem in library construction, since it is not "safe" to simply replace matching by the "stronger" subtyping constraint wherever it holds between any two classes.

8 Programming Experience

In this section we try to assess the impact of the increased type system expressiveness gained by the introduction of matching on the practical value of the TooL language.

First, subtyping and type matching both interact well with other TooL language concepts such as type quantification and Self type constraints. Moreover, students with some background in strongly-typed higher-order programming languages like Modula-3 or Tycoon grasp these concepts rather fast. However, as already discussed in section 2 and 7, problems arise as soon as programmers wish to combine the advantages of both partial orders on types.

For example, library designers typically prefer match-bounded quantification to maximize code reuse. This may conflict with the goal of library clients which like to exploit the subsumption property of subtyping to absorb later (unforseen) system extensions. As a concrete example, consider a method f in an application class with the following signature:

```
f(d :Dictionary ...) ...
```

This method can be applied uniformly to all instances of classes D derived by subtyping from the class Dictionary. To achieve the same effect for all classes D matching Dictionary, a verbose explicit match-bounded quantification has to be used in all application methods which is unlikely to be carried out in practice:

```
f(D <*: Dictionary d :D ...) ...
```

A similar argument holds for type parameters of classes. For example, the constraint E <*: Equality for the class EqualitySet defined in the previous section works fine only if we refrain from deriving subclasses by subtyping. If we attempt to instantiate EqualitySet with type ColoredPoint, the type checker would fail to verify ColoredPoint <*: Equality since the type Self in class ColoredPoint is replaced by Point during subtype-bounded inheritance.

Again, a more elaborate parameterization would solve this problem:

```
class EqualitySet2(T <*: Equality, E <: T)
super Set(E)
  includes(x :E) :Bool { elements.some(fun(e :E) {e = x}) }
```

More complex parameterizations (such as alternating chains of subtype- and match-bounded type parameters) do not seem to be necessary. Up to now, we have found no compelling example where a subclass of a subtype-bounded class needs to be refined by match-bounded inheritance.

Since such investigations can only be carried out by implementing practical programming languages and by using them for application development, the TooL experiment constitutes a valid complement to ongoing type-theoretical research on the integration of type matching and subtyping.

9 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 1994] also integrates subtyping and matching but lacks type rules which relates one notion to the other (see section 7). PolyTOIL is a recent successor to TOOPLE that adds polymorphism but is restricted to match-bounded quantification [Bruce *et al.* 1995b]. The parameterized classes of TooL could be modeled with type operators in PolyTOIL. However, the interaction between parameterization and inheritance is not addressed in [Bruce *et al.* 1995b].

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

In LOOP [Eifrig *et al.* 1994], no distinction 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 bounded type quantification, 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].

10 Conclusion

The main results of our work presented in this paper can be summarized as follows:

Type matching and subtyping can be integrated orthogonally and cleanly into a fully-fledged, practical programming language based on a rather small set of constructs at the type and at the value level. In particular, type matching interacts well with type quantification.

In a type system with subtyping and matching, modular type checking requires the programmer to specify how the receiver type Self is related to the type of

the enclosing class. TooL offers three possibilities: (1) Self is a subtype of the enclosing class; (2) Self matches the enclosing class; (3) Self is identical to the enclosing class.

Existing type-theoretical models of languages incorporating subtyping and type matching lack type rules to derive relationships between elements in the subtype and the matching lattice which are important for library construction. We define and informally justify such a type rule for TooL. Together with the presentation of the TooL core syntax we thus provide useful input for further type-theoretical work in this area. In this context it is interesting to note that TooL inherits undecidability from $F_{<:}$ [Pierce 1994] since it employs the same powerful contravariant subtyping rule on polymorphic functions (i.e. methods). However, this contravariant rule could be replaced by a more rigid rule which avoids undecidability without invalidating any of the existing TooL library classes.

A comparison of the TooL class library with the Tycoon bulk type library supports both our hypotheses stated in the introduction of this paper: TooL as a purely object-oriented language with type matching leads to more uniform program libraries with an increased code reuse. However, we also observed some practical difficulties encountered by programmers designing large libraries involving both matching and subtyping.

References

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

Albano et al. *1994:* Albano, A., Ghelli, G., and Orsini, R. Fibonacci reference manual: A preliminary version. FIDE Technical Report Series FIDE/94/102, FIDE Project Coordinator, Department of Computing Sciences, University of Glasgow, Glasgow G128QQ, 1994.

Atkinson 1996: Atkinson, M.P. *Fully Integrated Data Environments*. Springer-Verlag (to appear), 1996.

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. Type-checking 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: type-checking Smalltalk in a production environment. In *Proceedings OOPSLA'93*, pages 215–230, October 1993.

Bruce et al. *1995a:* Bruce, K.B., Cardelli, L., Castagna, G., The Hopkins Object Group, Leavens, G.T., and Pierce, B. On binary methods. Technical report, DEC SRC Research Report, 1995.

Bruce et al. *1995b:* Bruce, K.B., Schuett, A., and Gent, R. van. PolyTOIL: a type-safe polymorphic object-oriented language. In *Proceedings ECOOP'95*. Springer-Verlag, 1995.

Bruce 1994: Bruce, Kim B. A paradigmatic object-oriented programming language: Design, static typing and semantics. *Journal of Functional Programming*, 4(2), April 1994.

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 1989: Cardelli, L. Typeful programming. Technical Report 45, Digital Equipment Corporation, Systems Research Center, Palo Alto, California, May 1989.

Cardelli 1993: Cardelli, L. An implementation of $F_{<:}$. Technical Report 97, Digital Equipment Corporation, Systems Research Center, Palo Alto, California, February 1993.

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, pages 1–15, 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.

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.

Gawecki and Matthes 1996: Gawecki, A. and Matthes, F. Exploiting persistent intermediate code representations in open database environments. In *Proceedings of the 5th Conference on Extending Database Technology, EDBT'96*, Avignon, France, March 1996. (to appear).

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 1977: Hewitt, C. Viewing control structures as patterns of passing messages. *Artificial Intelligence*, 8:323–364, 1977.

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.

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.

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.

Meyer 1990: Meyer, B. Lessons from the design of the eiffel libraries. *Communications of the ACM*, 33(9):69–88, September 1990.

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

Nelson 1991: Nelson, G., editor. *Systems programming with Modula-3.* Series in innovative technology. Prentice Hall, Englewood Cliffs, New Jersey, 1991.

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 1994: Pierce, B. C. Bounded quantification is undecidable. *Information and Computation*, 112(1):131–165, July 1994. Also in Carl A. Gunter and John C. Mitchell, editors, *Theoretical Aspects of Object-Oriented Programming: Types, Semantics, and Language Design* (MIT Press, 1994).

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.