Typing in object-oriented languages: Achieving expressiveness and safety *

Kim B. Bruce

Williams College
September 10, 1996

Abstract

While simple static-typing disciplines exist for object-oriented languages like C++, Java, and Object Pascal, they are often so inflexible that programmers are forced to use type casts to get around the restrictions. At the other extreme are languages like Beta and Eiffel, which allow more freedom, but require run-time or link-time checking to pick up the type errors that their type systems are unable to detect at compile time.

This paper presents a collection of sample programs which illustrate problems with existing type systems, and suggests ways of improving the expressiveness of these systems while retaining static type safety. In particular we will discuss the motivations behind introducing "MyType", "matching", and "match-bounded polymorphism" into these type systems.

We also suggest a way of simplifying the resulting type system by replacing subtyping by a type system with a new type construct based on matching. Both systems provide for binary methods, which are often difficult to support properly in statically-typed languages.

The intent is to explain why the problems are interesting via the series of sample programs, rather than getting bogged down with pages of type-checking rules and formal proofs. The technical details (including proofs of type safety) are available elsewhere.

Contents

*This research was partially supported by NSF grant CCR-9424123.

List of Figures

1 Introduction

The object-oriented paradigm has been adopted by an increasing number of programmers and organizations over the last decade because of its clear advantages in organizing and reusing software components. It would clearly be advantageous to be able to provide static type systems for object-oriented languages that are of the same quality as those available for more standard procedural languages. Unfortunately commercially available object-oriented languages fall far short of that goal. The static type systems of object-oriented languages tend to be either insecure or more inflexible than one might desire. In some cases the rigidity of the type system leads programmers to rely on type casts (sometimes checked at run-time, sometimes not) in order to obtain the expressiveness desired. In other cases, the type systems are too flexible, requiring the run-time system to generate link-time or run-time checks to ensure the integrity of the computation. In this paper we explore the type-checking systems of object-oriented programming languages, examining problems and suggesting solutions.

1.1 Why type checking?

Every value generated in a program is associated with a type. In a strongly typed language, the language implementation is required to check the types of operands in order to ensure that nonsensical operations, like dividing the integer 5 by the string "hello", are not performed. In a dynamically typed language most operations are type-checked just before they are performed. In a statically typed language, every expression of the language is assigned a type at compile time. If the type system can ensure that the value of each expression has a type compatible with the type of the expression, then type checking of most operations can be moved to compile time.

There are many advantages to having a statically type-checked language. These include providing earlier (and usually more accurate) information on programmer errors, providing documentation on the interfaces of components (e.g., procedures, functions, and packages or modules), eliminating the need for run-time type checks, which can slow program execution, and providing extra information that can be used in compiler optimizations. One possible disadvantage of static typing is that because static type checkers are necessarily conservative, a static type checker for a programming language may disallow a program that would in fact execute without error. Thus statically typed programming languages may be less expressive than dynamically typed languages.

Procedural languages like Pascal [Wir71], CLU [L^+81], Modula-2 [Wir85], and Ada 83 [US 80], and functional languages like ML [HMM86] and Haskell [HJW92] have reasonably safe static typing systems. While some of these languages have a few minor holes in the type system (*e.g.*, variant records in Pascal), languages like CLU and Ada provide fairly secure type systems. Moreover, support for polymorphism has been very helpful in increasing the expressiveness in statically typed imperative and functional programming languages like CLU, Ada, ML, and Haskell.

In object-oriented programming languages, typing issues are more focussed on whether a message can be sent to a particular object (*i.e.*, whether the receiver has a method which can be executed in response to the message). Nevertheless the basic issues are very similar. However, extra complications arise from the presence of subtyping and the use of a *pseudo-variable* (usually written as `self` or `this`) to stand for the object executing the method. Because of subtyping, actual parameters to methods (or functions or procedures) can be of a type different from that specified in the declaration of the corresponding formal parameters. Because of inheritance, method bodies which are compiled for one class can be reused in subclasses. We must make sure that these features which support reuse do not cause holes in the typing system.

Unfortunately the situation for static type checking in object-oriented languages is not as good as for procedural languages. The following is a list of some properties of type-checking systems of some of the more popular object-oriented languages (or the object-oriented portions of hybrid languages).

- Some show little or no regard for static typing (*e.g.*, Smalltalk [GR83]).
- Some have relatively inflexible static type systems, which require type casts to overcome deficiencies of the type system. These type casts may be unchecked, as in C++ [Str86] and Object Pascal [Tes85], or checked at run-time, as in Java [AG96].
- Some provide mechanisms like "typecase" statements to allow the programmer to instruct the system to check for more refined types than can be determined by the type system (*e.g.*, Modula-3 [CDG^+88], Simula-67 [BDMN73], and Beta [KMMPN87]).
- Some allow "reverse" assignments from superclasses to subclasses, which require run-time checks (*e.g.*, Beta, Eiffel [Mey92]).
- Some require that parameters of methods overridden in subclasses have exactly the same types as in the superclasses (*e.g.*, C++, Java, Object Pascal, and Modula-3), resulting in more inflexibility than would be desirable, while others allow *too* much flexibility in changing the types of parameters or instance variables, requiring extra run-time or link-time checks to catch the remaining type errors (*e.g.*, Eiffel and Beta).

Thus all of these languages either require programmers to program around deficiencies of the type system, require run-time type-checking, or allow run-time type errors to occur. While features like typecase statements and run-time checked casts or reverse assignments may occasionally be necessary to handle difficult problems with heterogeneous data structures, we would prefer to have type systems which allow us to program as naturally as possible, while catching as many type errors as possible at compile time. As we shall see later, many problems arise because of the conflation of type with class and with the mismatch of the inheritance hierarchy with subtyping. Whatever the cause, there appears to be a lot of room for improvement in moving toward a combination of better security and greater expressiveness in the type systems.

1.2 Plan of the paper

In the rest of this paper we discuss the complications that arise in designing static type-checking systems for object-oriented languages, and sketch some ways of avoiding these problems by providing more flexible and expressive type systems. Of course we wish to ensure that the resulting systems are type safe.

We begin in section 2 by reviewing briefly the definitions of types, classes, subtypes, and subclasses, and illustrating their uses in object-oriented languages. In section 3 we discuss relatively simple type-checking systems like those in C++, Object Pascal, and Modula-3, in order to see what problems arise with the most obvious type systems. In section 4 we see that we can easily add more flexibility by allowing programmers to replace methods in subclasses by new ones whose types are subtypes of the original.

In many cases this still does not provide enough expressiveness for the type system to capture the programmer's intentions. Thus in section 5 we introduce the type expression *MyType* which is used to provide a flexible type for *self*, the receiver of a message. In the following section we introduce the very important notion of *matching*, a relation similar to, but distinct from, subtyping. The importance of matching will result from the fact that it is much closer to the inheritance ordering than subtyping is, allowing us to type check methods in such a way that they remain type safe when inherited. We discuss problems that arise with subtyping and so-called binary methods in section 7. We then evaluate what the addition of *MyType* means for the type system in section 8.

In section 9 we introduce a kind of constrained polymorphism called match-bounded polymorphism that allows us to write reusable code that is more flexible in handling objects of different types. In the following section we show how to use this to provide a solution to Eiffel's covariant typing problems. We also briefly discuss Eiffel's link-time system validity check and Bertrand Meyer's recent "no polymorphic catcalls" proposal to deal with these problems.

In section 11 we step back to look at the constructs we have introduced with the view of creating a simpler system. This reexamination will lead us to consider the rather radical step of designing a system that dispenses with the notion of subtyping. We close with a summary and discuss related work.

2 Types and Subtypes, Classes and Subclasses

The notions of type and class are often confounded in object-oriented programming languages. In fact they play quite distinct roles, and can be usefully distinguished from each other. Types provide interface information that determines when certain operations are legal, while classes provide implementation information including the names and initial values of instance variables and the names and bodies of methods. In the first subsection we discuss types and subtypes, while in the following subsection we discuss the notions of class and subclass.

2.1 Types and subtypes

A *type* in a programming language represents a set of values and the operations and relations that are applicable to them. For example the *Integer* type represents the set of whole numbers and the usual integer operations and relations, including $=$, $<$, $>$, $+$, $-$, $*$, *div*, and *mod*. An object type representing a point represents a collection of points and the messages that can be sent to point objects.

Strongly typed languages provide type-checking mechanisms to ensure that nonsensical operations are not applied to values. Such a language is sometimes said to be *type safe*. For example a strongly typed language will ensure that two characters are not added together as though they were integers. While these type checks can be provided at run-time, in this paper we will concentrate on static or compile-time type-checking mechanisms.

In *statically-typed programming languages* like Pascal and C, expressions are assigned types by the type checker. A set of *typing rules* based on the structure of expressions are used to build up a static type for each expression. The type checker, if correct, guarantees that if an expression has static type T, then, when that expression is evaluated at run time, the result will be a value of type T. In particular, then, a static type-checking system determines in which contexts an expression may legitimately occur.

In object-oriented languages, the most important operation is sending a message to an object. In this case one goal of type checking is to ensure that inappropriate messages are not sent to objects. In particular if a message is sent to an object, the type system should ensure that the object has a method with the same name, and that the formal parameter and return types are compatible with those of the call. From a typing point of view, then, a message send to an object is similar to extracting a field from a record in which the field happens to be a function.

Types in imperative or functional programming languages include base types like *Integer*, *Real*, *Character*, etc., as well as operators which can be used to build new types. These operators can be used to build record types, array types, and function and procedure types, among others. Object-oriented languages replace most or all of these complex types with object types. Object types provide information on the names and types of the methods supported by objects of that type.

A type S is a *subtype* of a type T (written $S <: T$) if an expression of type S can be used in any context that expects an element of type T. Another way of putting this is that any expression of type S can masquerade as an expression of type T. This definition is usually expressed in typing rules by adding a *subsumption* rule stating that if $S <: T$ and expression e has type S then e also has type T. Thus in a type system with subsumption, if an expression can be assigned a type T, it can also be assigned any type that is a supertype of T. This is in contrast to languages without subtyping in which most expressions can be assigned a unique type.

The support for subtyping provides added flexibility in constructing legal expressions of a language. For instance, let x be a variable of type T. If e is an expression of type T, then $x := e$ is a legal assignment statement. If S is a subtype of T and e' has type S, then $x := e'$ will also be a legal assignment statement. Similarly an actual parameter of type S may be used in a function or procedure call when the corresponding formal parameter's type is declared to be T. In most pure object-oriented languages, these mechanisms are supported by holding objects as implicit references, and interpreting assignment and passing parameters as binding new names to existing objects, i.e., as ways of creating sharing.

How can we determine when one type is a subtype of another? A technical discussion of this topic would take us far afield from the aims of this paper into the realm of programming language semantics and type theory. Instead we will present intuitive arguments for determining when two types are subtypes. Rest assured that the intuition can be backed up by more formal semantic arguments. Because the typing of message sends has similarities to typing records of functions, we begin with examining the simpler cases of record and function types now, holding off on object types until later. We also include a discussion of references (*i.e.*, types of variables) here, in order to prepare for the later discussion of instance variables in objects. The subtyping rules in the rest of this section are based on those given by Cardelli [Car88a].

2.1.1 Record types

Records associate values to particular labels. Thus a record representing a sandwich might associate the label *bread* with the value *rye* and the label *filling* with *cheese*. We will write such a record as

$$\{bread = rye, filling = cheese\}.$$

The type of a record specifies the type of the value corresponding to each label. In the example, the type associated with *bread* might be *BreadType* and the type associated with *filling* might be *FoodType*. We will write this record type as

$$SandwichType = \{bread\colon BreadType; filling\colon FoodType\}.$$

As a short-hand notation, we write a record with labels l_i of type T_i for $1 \leq i \leq n$ in the form

$$\{l_i\colon T_i\}_{1 \leq i \leq n}.$$

For simplicity, we deal here only with immutable (or "read-only") records of the sort found in functional programming languages like ML. No operations are available that update particular fields of these records. One can only create them as a whole and extract the values of particular fields. We discuss later in this section the impact of allowing updatable fields.

Suppose R and S are record types, with $R <: S$. In order for elements of R to masquerade as elements of S, expressions of type R need to support all of the operations applicable to expressions of type S.

Because the only operation available on a record is to extract a labeled field,[1] for R to masquerade as S, R must contain a corresponding field for each field l of S. Moreover the type of that field in R must be a subtype of the corresponding field in S, since if $r\colon R$ then $r.l$ must be usable in any context in which the same field selection of an element of type S makes sense.

In Figure 1 we show a record $r\colon \{l_1\colon T_1; l_2\colon T_2; l_3\colon T_3; l_4\colon T_4\}$ masquerading as a record of type $\{l_1\colon U_1; l_2\colon U_2; l_3\colon U_3\}$. Notice that the result of extracting the l_i field of r must be able to be treated as being of type U_i. Notice also that type R may have more labelled fields than S (since the extra fields don't get in the way of any of the operations applicable to S).

[1] One might also expect record construction to be an operation on a record, but we distinguish here (as is typical in object-oriented languages) between operations that can be applied to a value of a particular type, and operations that result in a value of that type (usually called constructors).

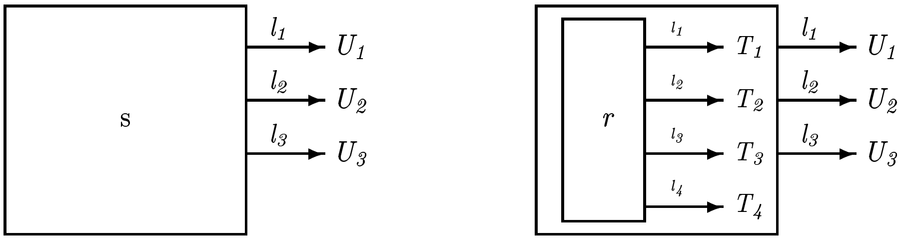

Figure 1: A record s: $\{l_1: U_1; l_2: U_2; l_3: U_3\}$, and r: $\{l_1: T_1; l_2: T_2; l_3: T_3; l_4: T_4\}$ masquerading as an element of type $\{l_1: U_1; l_2: U_2; l_3: U_3\}$.

Thus a subtype of a record may have labeled fields whose types are a subtype of the original, and may also have more fields than the original record type. We write this formally as follows:

$$\{l_j: T_j\}_{1\leq j\leq n} <: \{l_i: U_i\}_{1\leq i\leq k} \text{ if } k \leq n \text{ and for all } 1 \leq i \leq k,\ T_i <: U_i.$$

For example, let

$$\mathit{CheeseSandwichType} = \{\mathit{bread}: \mathit{BreadType}; \mathit{filling}: \mathit{CheeseType}; \mathit{sauce}: \mathit{SauceType}\}.$$

If *CheeseType* $<:$ *FoodType*, then the subtyping rule for records implies *CheeseSandwichType* $<:$ *SandwichType*. A record of type *CheeseSandwichType* can masquerade as a *SandwichType* since it has the *bread* and *filling* fields expected of a *Sandwich* and the results of extracting the *filling* field (a value of type *CheeseType*) can masquerade as a value of type *FoodType*.

2.1.2 Function types

Functions map values of one type to elements of another type. For instance, the function *length* takes strings to integers by returning the length of the string provided as an argument. The type of a function is determined by the type of the arguments that the function may be applied to and the type of the values returned from the function.

We write *Func(S): T* for the type of functions that take a parameter of type S and return a result of type T. Thus the type of the *length* function is *Func(string): integer*.

If $(\mathit{Func}(S)\colon\ T) <: (\mathit{Func}(R)\colon\ U)$, then we should be able to use an element of the first functional type in any context in which an element of the second type would type check.

Suppose we have a function f with type *Func(R): U*. In order to use an element, f', of type $\mathit{Func}(S)\colon\ T$ in place of f, the function f' must be able to accept an argument of type R and return a value of type U. See Figure 2. But f' is defined to accept arguments of type S. Now, as can be seen from the figure, f' can be applied to an argument, r, of type R if $R <: S$. In that case, using subsumption, r can be treated as an element of type S, making $f'(r)$ type-correct. Similarly, if the output of f' has type T then $T <: U$ will guarantee that the output can be treated as an element of type U. Summarizing,

$$(\mathit{Func}(S)\colon\ T) <: (\mathit{Func}(R)\colon\ U) \text{ if } R <: S \text{ and } T <: U.$$

Again assuming that *CheeseSandwichType* $<:$ *SandwichType*, we get that

$$\mathit{Func}(\mathit{Integer})\colon\ \mathit{CheeseSandwichType} <: \mathit{Func}(\mathit{Integer})\colon\ \mathit{SandwichType},$$

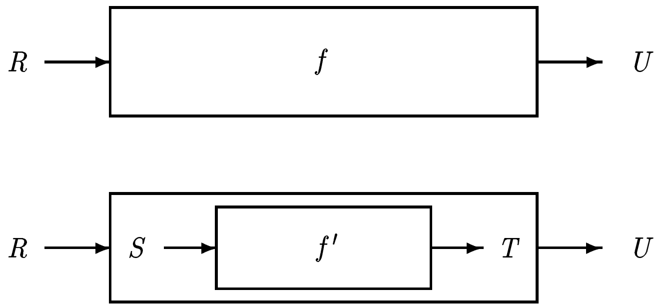

Figure 2: A function f: $Func(R)$: U, and f': $Func(S)$: T masquerading as f.

but

$$Func(SandwichType)\colon\ Integer <\colon Func(CheeseSandwichType)\colon\ Integer.$$

Procedure types, written $Proc(S)$, may be subtyped as though they were degenerate function types that always return a default type *Unit*, which has only a single value. Thus

$$Proc(S) <\colon Proc(R), \text{ if } R <\colon S.$$

Notice that the ordering of parameter types in function and procedure subtyping is the reverse of what might initially have been expected, while the output types of functions are ordered in the expected way. We say that subtyping for parameter types is *contravariant* (*i.e.*, goes the opposite direction of the relation being proved), while the subtyping for result types of functions is *covariant* (*i.e.*, goes in the same direction). The contravariance for parameter types can be initially confusing, because it is permissible to replace an actual parameter by another whose type is a subtype of the original. However the key is that in the subtyping rule for function types it is the function, *not* the actual parameter, which is being replaced.

Let us look at another example to illustrate why contravariance is appropriate for type changes in the parameter position of functions and procedures. The contravariant rule for procedures tells us that it is possible to replace a procedure, p, of type $Proc(CheeseType)$ by a procedure, p', of type $Proc(FoodType)$. The procedure p can be applied to any value, *cheese*, of type *CheeseType*. Because $CheeseType <\colon FoodType$, *cheese* can masquerade as an element of type *FoodType*. As a result, p' can also be applied to the value *cheese*. Thus p', and indeed any procedure of type $Proc(FoodType)$ can masquerade as an element of type $Proc(CheeseType)$.

2.1.3 Types of variables

Variables of type T have different properties from ordinary expressions of type T, because variables may be assigned to. For instance if *fv* is a variable of type *FoodType*, and *apple* is a value of type *FoodType*, then the statement $fv := apple$ is a type-correct statement, whereas $apple := fv$ is clearly not correct.

If *CheeseType* $<\colon$ *FoodType* and *cheddar* is a value of type *CheeseType*, then $fv := cheddar$ is a type-correct statement, because we can always replace a value of a given type by a value of a subtype. On the other hand, if *cv* is a variable of type *CheeseType*, then $cv := apple$ is *not* type-correct. (For instance there may be an operation *melt* which can be applied to cheeses but not general foods, and that would break when applied to *cv* if it held a value which was not of type *CheeseType*.) Thus it is *not* type-correct to replace a variable of a given type by a variable of a subtype. In this section we examine when a variable of one type may be replaced by a variable of another type.

We refer to the types of variables as *reference* types, and say that a variable declared to have type T actually has type *ref* T.[2] This name is reasonable since a variable of type T actually denotes a location (or reference) in which one can store a value of type T. As we saw in the example above, the fact that variables may be assigned to will have a great impact on the subtype properties (or rather the lack of them) of reference types.

Suppose we have a variable x' of type S (*i.e.*, an expression of type *ref* S), that we wish to have masquerade as a variable of type T. See Figure 3. We can think of a variable of type T as having two values: an L-value and an R-value. The L-value is the location corresponding to the variable, while the R-value is the value of type T actually stored there. For example, if n is an integer variable, the first occurrence of n in $n := n + 1$ is in a value-receiving context, while the second occurrence is in a value-supplying context.

The use of variables in value-supplying contexts is represented in the figure by the operation (arrow) labeled "*v*al" coming out of the variable (representing the R-value of the variable). (A few languages – ML is one – actually require an explicit "dereferencer" like *val* in all value-supplying contexts.) Notice that if x is a variable of type T, *val* returns a value of type T.

By the definition of subtype, in order for a variable x' of type S to be able to masquerade as a value of type T in all contexts of this kind, we need $S <: T$. This should be clear from the right-hand diagram in the figure, where in order for x' to provide a compatible value using the *val* operator, $S <: T$.

A value-receiving context is one in which a variable of type T is assigned to, *e.g.*, a statement of the form $x := e$, for e an expression of type T. This is illustrated in the figure by an arrow labeled "$:=$" going into the variable. In this context we will be interpreting the variable as a reference or location (*i.e.*, the L-value) in which to store a value. We have already seen that an assignment like this is type safe if e has a type that is a subtype of the type of the variable x. Thus if we wish to use a variable of type S in all contexts of this form, we must ensure that $T <: S$. Again this should be clear from the right-hand diagram in the figure.

Going back to the example at the beginning of this section, suppose we have an assignment statement, *cv*$:=$ *cheese*, for *cv* a variable of type *CheeseType* and *cheese* a value of type *CheeseType*. If *fv* is a variable of type *FoodType*, then we can insert *fv* in place of *cv* in the assignment statement, obtaining *fv*$:=$ *cheese*. Because *CheeseType* $<:$ *FoodType*, this assignment is legal.

Thus for a variable of type S to masquerade as a variable of type T in value-supplying contexts we must have $S <: T$, while its use in value-receiving contexts require $T <: S$. It follows that there are no non-trivial subtypes of variable (reference) types. Thus,

$$\textit{ref}\ S <: \textit{ref}\ T \text{ iff } S \simeq T,$$

where $S \simeq T$ abbreviates $S <: T$ and $T <: S$. We can think of $\simeq$ as defining an equivalence class of types including such things as pairs of record types that differ only in the order of fields. It is common to ignore the differences between such types and to consider them equivalent.

[2] Thus, in this paper the statement, "x is a variable of type T," has the same meaning as "x has type *ref* T". This is similar to the usage in ML.

Figure 3: A variable x: *ref* T, and x': *ref* S masquerading as x.

Another way of understanding the behavior of reference and function types under subtyping is to consider the different roles played by suppliers and receivers of values. Any slot in a type expression that corresponds to a supplier of values must have subtyping behave covariantly (the same direction as the full type expression), while any slot corresponding to a receiver of values must behave contravariantly (the opposite direction). Thus L-values of variables and parameters of functions, both of which are receivers of argument values, behave contravariantly with respect to subtyping. On the other hand, the R-values of variables and the results of functions, both of which are suppliers of values, behave covariantly. Because variables have both behaviors, any changes in type must be simultaneously contravariant and covariant. Hence subtypes of reference types must actually be equivalent.

We can use this analysis to lead us to subtyping rules for updatable records and arrays. Suppose there exists a record update operation *update* r *with* $l = e$, which returns a new record value whose l field is e, while the other fields have the same values as in r. If r has a record type T that has a field with label l and type T, then the *update* expression above will be type-correct if e has type T. In the presence of such an operation, the record subtyping rule will have to be modified to:

$$\{l_j\colon T_j\}_{1 \le j \le n} <: \{l_i\colon U_i\}_{1 \le i \le k} \text{ if } k \le n \text{ and for all } 1 \le i \le k,\ T_i \simeq U_i.$$

Thus the subtype has at least the fields of the supertype, but corresponding fields must have equivalent types because of the updating operation.

If full record assignments are possible (*i.e.*, $r := r'$, for r and r' both records), then we leave it to the reader to show that record types have no non-trivial subtypes.

Similar arguments can be used to show that if arrays allow individual components to be updated, then $Array(S) <: Array(T)$ only if $S \simeq T$. Thus the Java [AG96] rule that $Array(S) <: Array(T)$ if $S <: T$ is only type-safe if the arrays are read-only (similarly to the record expressions in section 2.1.1). For example, if A has type $Array(S)$, B has type $Array(T)$, and t has type T, then $B[i] := t$ is type-correct, while $A[i] := t$ is generally not. The Java designers compensate for this *static* type error by performing *dynamic* checks when an individual component of an array is assigned to.[3]

2.1.4 Object types

The subtyping relation among object types can be rather subtle. In the simple type system considered in section 3, object types are considered subtypes when the record of method types of the subtype extends that of the supertype. That is, a subtype may contain more methods than the

[3] Apparently, Java includes this faulty rule in order to allow generic sorts (and similar operations) to be statically type checked. Support for parametric polymorphism (see section 10.2) would allow the creation of type-correct generic sorts without the need for this unsafe rule.

```
class Point
   var
      x := 0: Integer;
      y := 0: Integer
   methods
      function getx():integer
        begin
          return x
        end;
      function gety():integer
        begin
          return y
        end;
      procedure move(dx, dy: integer)
        begin
          x := x + dx;
          y := y + dy
        end
end class;
```

Figure 4: A Point class.

supertype, but the corresponding methods must have the same types. This is more restrictive than the definition of subtype for immutable record types given above. We will see in section 4 that a less restrictive notion of subtyping is also correct.

2.2 Classes and Subclasses

Objects consist of both state and operations. The state is typically represented by a collection of *instance variables*, while the operations are usually referred to as *methods*. *Classes* serve as extensible generators of objects, providing the names and initial values for instance variables and the names and bodies for methods. For example, the *Point* class in Figure 4 specifies that point objects that are *instances* of this class will each include x and y instance variables (which are initialized to 0) and will support methods *getx*, *gety*, and *move*. The references to x and y in the methods refer to the instance variables in the point object receiving the message.

In our example language, objects can be created by applying the *new* primitive to a class. Thus evaluation of *new Point* results in the creation of a new point object. Each invocation of *new Point* results in the creation of a distinct point object.

While a class specifies the names, types, and initial values of instance variables and the names, types, and meanings of the methods, the values of instance variables of individual objects can be changed by the execution of that object's methods. On the other hand the methods associated with an object are fixed and may not be modified once the object has been created. In the terminology used in the previous subsection, instance variables of objects are mutable, while methods are immutable. Not surprisingly, this difference between instance variables and methods has an impact on the subtyping rules for objects developed later.

We do not confuse a class with the type of objects generated by the class. A class contains implementation information rather than just describing the observable properties of an object. We discuss this distinction in more detail in the next subsection.

If p is an expression representing a point, then the expression $p.move(1,2)$ results in the look up and execution of p's *move* method (with parameters 1 and 2). This is usually referred to in object-oriented languages as *sending the message* $move(1,2)$ to p.

In Smalltalk and many other object-oriented languages, instance variables are not accessible outside of the methods of an object. In such a language, the use of an expression $p.x$ to access p's instance variable x would result in an error message. While the method call $p.getx()$ might evaluate and return the current value of p's instance variable, x, it is legal because it acts through a method of the class. For simplicity, we adopt Smalltalk's convention that all instance variables are hidden outside of the object, and that all methods are visible.

Subclasses support the ability to create incremental differences in behavior by allowing the programmer to define a new class by *inheriting* the code of an existing class, while possibly replacing or adding instance variables and methods. Thus class SC is a *subclass* of C if either SC is defined by inheritance from C or is defined by inheritance from some subclass of C.

For example, suppose there exists a graphic window class supporting moving, resizing, and redrawing windows. It is possible to define a window with scroll bars by inheriting from the original window class and adding instance variables to represent the scroll bar and methods to support their use. It will also be necessary to modify the redrawing method to draw the scroll bars in the correct positions.

The ability to define a new class by specifying only its differences with an existing class is the source of much of the power and productivity of the object-oriented paradigm. It allows the programmer to reuse old code of a superclass in the definition of a subclass. It is thus important that type-checking rules for object-oriented languages preserve this ability.

We present a simpler example in Figure 5 where we define *Colorpoint* as a subclass of *Point*. We add a new instance variable, *color*, and methods to access and set the color. To make the example more interesting, we also modify the *move* method so that a newly moved colorpoint is set to be red. The expression $super.move(dx, dy)$ in *Colorpoint*'s *move* method indicates that the method body of *move* from the superclass should be executed before the assignment of *red* to the *color* instance variable. The specification in the header of the subclass declaration that *move* is to be modified is not strictly necessary, but is a useful safeguard to ensure that an existing method is not accidentally modified.

Before finishing our discussion of classes and subclasses we need to introduce one more construct. During the execution of a method, it is frequently necessary to refer to the object executing that method, either to pass it as a parameter or to invoke one of its own methods. For example, in a class defining doubly-linked nodes, a method for attaching a new node to the right of an existing node might consist of code for inserting the new node to the right, followed by code sending a message to the new node asking it to attach the existing node to its left. (See the code for the method *attachRight* in the doubly-linked node class in section 5.) In order to be able to express this, most object-oriented languages include a name for the object executing the method. Existing languages use terms like *self* (Object Pascal and Smalltalk), *this* (C++ and Java), and *Current* (Eiffel). We will use the term *self* in this paper.

In most object-oriented languages, unqualified references to a method m inside a method body is treated as an abbreviation for $self.m$. For clarity we will not adopt that convention and will

```
class Colorpoint inherits Point modifying move
   var
      color := blue: ColorType
   methods
      function getColor():ColorType
        begin
          return color
        end;
      procedure setColor(newColor: ColorType)
        begin
          color := newColor
        end;
      procedure move(dx, dy: integer)
        begin
          super.move(dx,dy); color := red
        end
end class;
```

Figure 5: A Colorpoint subclass

write out the message sends in full. However, because instance variables of an object are only accessible inside methods of that object, we will write x rather than $self.x$ for instance variable accesses inside method bodies.

The notation $self.m$ should remind us that, in general, the methods of an object can be mutually interdependent. That is, any method of an object can invoke any other method of the object during execution of its body by sending an appropriate message to *self*. Thus modifying one method of a class in a subclass may have an impact on other methods that use it. This (potential) dependence of each method on the other methods of the class will have an important impact on type checking the methods in a subclass, as any change to the type of one method may affect the types of expressions in the bodies of other methods.

2.3 Differences between subtypes and subclasses

There are important differences between subtypes and subclasses in supporting reuse. Subclasses allow one to reuse the code inside classes – both instance variable declarations and method definitions. Thus they are useful in supporting code reuse *inside* a class. Subtyping on the other hand is useful in supporting reuse externally, giving rise to a form of polymorphism called inclusion polymorphism [CW85]. That is, once a data type is determined to be a subtype of another, any function or procedure that could be applied to elements of the supertype can also be applied to elements of the subtype. Another way of putting this is that subclassing supports reuse of components, while subtyping supports reuse of context.

Notice that the subtype relation depends only on the public interfaces of objects, not their implementations. In particular if one type is a subtype of another, it is not necessary for objects of those types to have arisen from classes that are in the subclass relation. By the same argument, objects of the same type need not have arisen from the same class. They may have been generated by classes with different collections of (hidden) instance variables and method bodies, but whose (visible) methods have the same types.

In languages supporting abstract data types, one may replace the implementation of a data type in a program by another which has the same public interface. It is possible in object-oriented languages for objects with the same public interface (*i.e.*, same type), but different implementations, to be used interchangably and simultaneously in a program.

The papers [BHJ+87] and [CHC90] were two of the first to examine typing problems in statically-typed object-oriented languages. Both papers make the point that subtyping has only to do with interface and not implementation. Thus objects from several classes may be of the same type. Thus there need be no connection between subclass and subtype.

Nevertheless, most existing object-oriented programming languages do identify type and class as well as subtype and subclass. That is, a class is identified with the type of objects it generates, while only subclasses generate subtypes. (It is worth noting that some languages, like C++, support a form of private inheritance which does not result in subtypes.) The new language Java provides more separation of types from classes than is usual by providing interfaces which are distinct from the classes which may conform to them.

We believe that it is unfortunate to identify subtyping and inheritance, as types should only refer to the interface of an object, not its implementation. While the analysis of typing in the rest of this paper can be profitably applied to languages which do identify subclass and subtype, our analysis also shows how object-oriented languages could profit from the separation of these notions.

3 Simple type systems are lacking in flexibility

Languages like Object Pascal, Modula-3, and C++ arose as object-oriented extensions of imperative programming languages. The static type systems of these languages have relatively simple and straightforward type systems whose features are similar to those of the procedural languages from which they were derived. In these simple type systems, the programmer has little flexibility in redefining methods in subclasses, as they require that a redefined method have exactly the same type as the original method in the superclass.[4] We refer to type systems that restrict the types of methods in subclasses to be identical to those in superclasses as invariant type systems.

Interestingly, even in these systems, when a method is inherited or redefined in the subclass, one is often able to deduce more refined types for methods than the language allows to be written. We present examples illustrating this below.

First a note on the terminology used in our examples. We use the convention in this paper of writing *CType* for the type of objects generated by class C. Thus in Figure 7, *NodeType* is the type of objects generated by class *Node*. As mentioned in the previous subsection, we will find it helpful to keep the notions of class and type separate. The type represents only the public interface of the object (in our case the names and types of all of the methods, but not the instance variables), while the class includes names and initial values for instance variables and names and code for methods.

[4]The new C++ draft standard and some current compilers now allow return types of methods to be replaced by subtypes in subclass (derived class) definitions as long as the return type is a pointer type.

3.1 The need to change return types in subclasses

In our first examples we show that it is useful to be able to modify the return type of methods when they are redefined in subclasses (and sometimes even when they are *not*).

As a quick first example, suppose we change the *Point* and *ColorPoint* classes from section 2.2 so that the *move* method actually returns a new point rather than just updating the instance variables of the receiver. Thus the declaration of the *move* method in *Point* becomes:

```
function move(dx, dy: integer): PointType;
    ...
```

When we define *ColorPoint* as a subclass, we now want to have *move* return a colorpoint rather than a point. There is no problem in writing the code to do this. However, the invariant type systems do not allow this more refined type to be written in the declaration of the method in *ColorPointType*. This type system requires the return type to remain as *PointType*, which then would require the programmer to perform a type cast or similar operation in order to convince the type system to actually treat the returned value as a color point.

While this example may seem a bit contrived, it shows up in a very important built-in operation in many object-oriented languages. In most pure object-oriented languages (*e.g.*, Eiffel, Java, and Smalltalk), all objects are represented as references (*i.e.*, implicit pointers). Thus assignment results in sharing, rather than copying. In these languages, it is useful to have an operation which makes a new copy or clone of an object. A common way of supporting this is to provide a *ShallowClone* method in the top-most class of the object hierarchy (a class sometimes called *Object*), so that all other classes automatically inherit it.

The shallow copy is made by copying the values of instance variables and taking the same suite of methods as the original. If the instance variables hold references to other objects, only the references are copied, not the objects being referred to. Thus if this shallow clone method (call it *ShallowClone*) is applied to the head of a linked list, only the head node is copied while the rest of the list is shared between the new and old lists.

What should be the type of *ShallowClone*? When defined in the class *Object*, it seems apparent that it should return a value of type *ObjectType*. However, when this is inherited by our class *Point*, we would like it to return a value of type *PointType*. In the invariant type systems, the return type of *ShallowClone* remains *ObjectType* even though the method actual returns a value which is a point!

Often it is desirable to write a *DeepClone* method which is based on *ShallowClone*. One typically first writes code to send the *ShallowClone* message to *self* to make the shallow copy, and then code to clone all objects held in the instance variables of the original object.

(**** THINK ABOUT ELIMINATING THIS *******) Suppose we have a class C that includes a method *DeepClone*, which returns an object of type *CType*. See Figure 6. Suppose we now define a subclass SC of C which includes new methods as well as a new instance variable holding an object. Then *DeepClone* should be redefined to clone the contents of this new instance variable after all of the code in the original *DeepClone* from C has been executed.

Unfortunately, the rules of the simple type systems require that *DeepClone* for SC to also return a *CType*, just as in C, even though it is obvious that it actually returns an object of type *SCType*. While this is not type-unsafe, it represents an unnecessary loss of information in the type system. If we write $anSC.DeepClone().newMeth()$ for $anSC$ of type *SCType*, the type checker will complain, even though the object resulting from $anSC.DeepClone()$ has a method *newMeth*.

```
class C
   var
     ...
   methods
      ...
      function DeepClone(): CType
        begin ... end
end class;

class SC inherits C modifying DeepClone
   var
     ...
   methods
      ...
      function DeepClone(): SCType    -- illegal type change
        begin ... end;
      procedure newMeth()           -- method not contained in C
        begin ... end
end class;
```

Figure 6: Typing DeepClone methods in subclasses.

In these circumstances, Object Pascal, C++, and Java programmers would normally be forced to perform a type cast to tell the compiler that the cloned object has the type *SCType*. In the case of Object Pascal and C++, the type cast is unchecked. In Java, it would be checked at run-time. Modula-3 programmers would use a typecase statement which also performs a run-time check to get the same effect.

One could attempt working around these deficiencies in the static type system by making up a new name for the revised *DeepClone* method (e.g., *SCDeepClone*). Unfortunately this would mean that inherited methods that included a message send of *DeepClone* would call the old *DeepClone* for *C* rather than the updated method from *SC* actually desired. As a result the value in the new instance variable will not be cloned, possibly causing problems later in the program. Thus the restriction on changing types of methods in subclasses gets in the way of the programmer, even though there should be no real typing problem. Not surprisingly, we have similar problems even writing down the type of the built-in *ShallowClone*. In Object Pascal, *ShallowClone* is simply given a type indicating that it returns an element of the *Top* type. It must then be cast to the appropriate type.

3.2 The need to change parameter and instance variable types in subclasses

Our next example is one in which it would be convenient to change both return types and parameter types of methods. The particular typing problem arises in connection with what are often called *binary methods*. Binary methods are those methods that have a parameter whose type is intended to be the same as the receiver of the message. Functions reflecting orders, such as *eq*, *lt*, and *gt*, or other familiar binary operations or relations are good examples of such methods. These are written

```
class Node
   var
      value = 0: Integer;
      next = nil: NodeType
   methods
      function getValue():Integer
        begin
          return value
        end;
      procedure setValue(newValue: Integer)
        begin
          value := newValue
        end;
      function getNext():NodeType
        begin
          return next
        end;
      procedure setNext(newNext: NodeType)
        begin
          next := newNext
        end;
      procedure attachright(newNext: NodeType)
        begin
          self.setNext(newNext)
        end
end class;
```

Figure 7: Node class

with single parameters in object-oriented languages because the receiver of the message plays the role of the other parameter. Other compelling examples arise in constructing linked structures. Binary methods are often difficult to deal with in statically-typed object-oriented programming languages. See [BCC$^+$96] for an extended discussion of typing problems with binary methods.

Figure 7 is a sample class definition to implement nodes for a singly-linked list. In the class there is one instance variable for the value stored in the node and another to indicate the successor node.[5] There are methods to get and set the values stored in the node, and to get and set the successor of the node.

Notice that function *getNext* returns a value of type *NodeType*, the type of object generated by class *Node*, while the procedures *setNext* and *attachRight* each take a parameter of type *NodeType*. The method *attachRight* currently does exactly the same as *setNext*(*newNext*); its use and value will become more apparent in section 5 when it is modified in a subclass.

[5]We note that all objects will be represented as references (implicit pointers). As a result, there is no problem supporting recursive types. The value *nil* is used as a null reference and is considered to be an element of all object types.

Suppose we now wish to define a subclass of *Node*, *DoubleNode*, which implements doubly-linked nodes, while taking advantage of the code for methods in *Node*. To do this we need to add to *Node* an additional instance variable, *previous*, as well as new methods to retrieve and set the *previous* node. Note that if *DoubleNodeType* is the type of objects generated from the class, then we will want both the *next* and *previous* instance variables to have type *DoubleNodeType*, and the relevant methods to take parameters or return values of type *DoubleNodeType* rather than *NodeType*. This is particularly important because we do not want to attach a singly-linked node to a doubly-linked node.

Unfortunately in the simple type system described here we have no way of changing these types, either automatically or manually, in the subclass. Thus to get the desired typings, we would have to define *DoubleNode* independently of *Node*, even though much of the code is identical. This is clearly undesirable.[6]

3.3 Other typing problems

In both of the examples above, the problem types in methods have been the type of the object being defined. While this is an extremely important special case, there are other examples where a type should be changed in a subclass, but the original and revised types are different from that corresponding to the class being defined. This often arises when we have objects with components. For example, suppose we have a circle class with a *getcenter* method, which returns a point. If we define a subclass that represents a color circle, it would be natural to wish to redefine *getcenter* to return a color point. This would be illegal by the rules on method types in these object-oriented languages. We wish to have a typing system that allows such changes, as long as they are type safe.

In this section we illustrated several problems with invariant type systems. In each case the difficulty arose from a desire to change the types of methods which are modified in subclasses. However no changes to the types of methods are allowed in these type systems. In order to alleviate this rigidity in the simple systems, in the next section we allow more flexibility in the type system by supporting some disciplined changes to types of methods in subclasses, whether or not the methods are modified. This will solve simple problems like those arising in the circle and colored circle classes. However, more extensive changes will be required to take care of examples involving the deep and shallow clone methods and with the *Node* and *DoubleNode* classes.

4 Toward more flexible type systems

As pointed out in the last section, there are many circumstances under which we would like to change the types of methods in subclasses. What keeps us from making arbitrary changes to the types of these methods? The main complication in changing the types of methods in subclasses is that a method in a class may call any other method from the class by sending the appropriate message to *self*. That is, changes to one method may have an impact on both the typing and meaning of another method in the same class. In this section we discuss the changes to method types that are guaranteed *not* to cause typing problems.

[6]It is worth noting here that because it contains a binary method, *DoubleNodeType* will not be a subtype of *NodeType*. See section 5 for a detailed discussion.

```
class C
   methods
      function m(s:S): T
        begin ... end;
      function n(an_s:S): U
        begin
           ... self.m(an_s) ...
        end
end class;

class SC inherits C modifying m
   methods
      function m(s:S'): T'  -- For which S',T' will this change be safe?
         ...
end class;
```

Figure 8: Changing types of methods in subclasses.

4.1 Subtyping of method types in subclasses

Figure 8 shows a well-typed class containing methods m and n, where m has type $Func(S)$: T. The body of method n includes a message send of $m(an\_s)$ to *self*. We presume that the body of method n will be well-typed if m has type $Func(S)$: T. Suppose we override m in a subclass so that it has type $Func(S')$: T'. In general, the only way to be sure that the occurrence of $self.m$ in the body of n is still compatible is to require that $Func(S')$: $T' <: Func(S)$: T (this is, after all, exactly what the subtype relation guarantees).

Thus we can guarantee type safety when forming subclasses if we restrict ourselves to overriding a method with a new one whose type is a subtype of the type of the original.[7]

By a similar analysis of message sending outside of the object's methods, one can easily show that this restriction on changing the types of methods in subclasses is sufficient to guarantee that the resulting object types will be subtypes. Later we will add more powerful constructs to our language that will result in subclasses sometimes failing to generate subtypes. However the possibility of mutually recursive definitions of methods will still require us to restrict changes in types of methods to subtypes.

We can write down the subtype relation on object types more formally. An object type that supports methods m_i of type T_i for $1 \leq i \leq n$ can be written either

```
ObjectType
   m1: T1;
   ...
   mn: Tn
end ObjectType
```

[7] Of course it is technically possible to keep track of which methods are called by other methods and annotate each class with that information. While this would allow more freedom in making changes to some methods, this seems too painful for regular use and may require a data flow analysis of even indirect uses of an object.

or more compactly as $\mathit{ObjectType}\{m_i\colon T_i\}_{1\le i\le n}$. The rule for subtyping object types can be written as

$$\mathit{ObjectType}\{m_j\colon S_j\}_{1\le j\le m} <: \mathit{ObjectType}\{m_i\colon T_i\}_{1\le i\le n} \text{ if } n \le m \text{ and for each } i \le n,\ S_i <: T_i.$$

Notice that this is essentially the same rule that we had for record types. When we add the new construct *MyType* in the next section, the definition of subtyping for object types will become more complex.

Because methods are always functions or procedures, we can indicate more exactly the changes allowed to types of methods in subclasses. Recall from the previous section that subtyping of function types is contravariant in the parameter type and covariant in the result type. Thus, in overriding a method in a subclass, we may replace a result type by a subtype and a parameter type by a supertype. This flexibility in changing result types is clearly very useful. However few compelling examples seem to exist of the value of replacing a parameter type by a supertype. The most likely scenario for thus changing parameter types would be if the original method had a parameter specification that was needlessly constraining and could thus be easily broadened in the method for the subclass.

4.2 Examples using flexible types

A few examples will help illustrate how we can use this extra flexibility, as well as where further flexibility is needed.

In Figure 9 we show parts of a *Circle* class with a method *getCenter* whose type is *Func(): PointType*. That is, it is a parameterless procedure that returns a point representing the center of the circle. Suppose *ColorPointType* is a subtype of *PointType*. Then a *ColorCircle* subclass could redefine *getCenter* to have type *Func(): ColorPointType*, thus returning a colored point for the center of the circle.

On the other hand, if *Circle* class also has a method *changeCenter* with type *Proc*(*PointType*) then we may *not* change its type in *ColorCircle* to *Proc*(*ColorPointType*). Because *PointType* occurs in a contravariant (parameter) position in the type of *changeCenter*, it may only be replaced by a supertype, which would not be of much use here. If, however, we dropped *changeCenter* in favor of the procedure *move*, whose only parameters are integers, then no difficulty arises with the subclass.

While the replacement of method types by subtypes in subclasses solves some typing problems, it does not solve them all.

Unfortunately, we do not have the same flexibility in changing the types of instance variables as we do with methods. Recall from the previous section that a variable whose declared type is T is actually a value of type *ref* T. As discussed in section 2.1, reference types have no subtypes because they can be used in either value-receiving (on the left side of :=) or value-supplying positions. As a result, it is not possible to change the types of instance variables in subclasses.

Returning to our *Circle* example, if *Circle* has an instance variable *center* of type *PointType*, we may not replace its type by *ColorPointType* in the subclass. Instead, we must add a new instance variable *color* with type *ColorType*, so that *ColorCircle* will have *center* and *color* instance variables. The body of the redefined *getCenter* method in *ColorCircle* (which does return an element of type *ColorPointType*) will have to create a color point value from the values in *center* and *color*. This is not as convenient as we might like, but it does allow us to use inheritance where the more rigid simple typing discipline did not.

```
class Circle
   var
      center = OrigPoint: PointType;
      ...
   methods
      function getCenter(): Point
        begin
          return center
        end;
      procedure changeCenter(newCenter: Point)
        begin
          center := newCenter
        end;
      procedure move(dx,dy: Integer)
      ...
end class;

class ColorCircle inherits Circle modifying getCenter, changeCenter
   var
      color = red: ColorType
   methods
      function getColor(): ColorType
        begin
          return color
        end;
      procedure setColor(newColor: ColorType)
        begin
          color := newColor
        end;
      function getCenter(): ColorPoint  -- legal change of type
        begin ... end;
      procedure changeCenter(newCenter: ColorPoint) -- illegal change of type
        begin ... end
end class;
```

Figure 9: Circle and ColorCircle classes with more flexible types.

This ability to replace result types by subtypes helps us out with the clone example as well, since if SC is a subclass of C, we want *DeepClone* in C to have type *Func(): CType*, while it should have type *Func(): SCType* in SC. Because this is a covariant change in the result parameter, this is a legal change in this more flexible system.

However, there still remains a problem here. We would occasionally like to inherit *DeepClone* in a subclass without change (for example, if we add or override methods without adding new instance variables). However in order to get its type to change appropriately, we would have to override *DeepClone* just in order to change the result type. We also cannot simply include a call of *super.DeepClone*() from the corresponding method of SC, as it returns a value of type *CType*, which is a supertype of what is needed. Thus we would have to rewrite *DeepClone* from scratch in such situations. The changes suggested in the next section will allow us to specify that the result type should change automatically in subclasses, even if we do not override the method.

Finally, notice that we still are unable to write the *Node* / *DoubleNode* example from the previous section. The problem is that even if *DoubleNodeType* were a subtype of *NodeType*[8], the method *setNext* has type *Proc*(*NodeType*) in *Node* and type *Proc*(*DoubleNodeType*) in *DoubleNode*. Because the type of the parameter should vary contravariantly, not covariantly, this change of type would be illegal in the subclass. In particular, if the type of the parameter of method *setNext* is changed, while the type of the parameter of method *attachRight* is not, a type error will result. Thus, if we wish to make covariant changes to the parameter type of a method m, we may have to examine the bodies of all other methods of the class (including those which are inherited) to identify which depend on the one being changed. We may then be required to make appropriate changes to those methods in order to preserve type safety. (But don't despair yet. In the next two sections we will see how to handle uniformly some important special cases, including the *Node* / *DoubleNode* example.)

As noted earlier, the new draft standard C++ proposal loosens the rules of C++ to allow covariant changes to the result types of methods that are redefined in subclasses. There is no similar proposal to allow contravariant changes to parameter types, presumably because, though safe, there are few cases in which this would actually be helpful.

Eiffel, on the other hand, does allow covariant changes to both instance variables and parameter types of methods in subclasses. This decision has provoked great controversy. (See [Coo89] for an early proposal to fix the Eiffel Stype system.) The arguments boil down to the following. On the one hand, proponents argue that covariance for function parameter types is often useful, and that actual run-type errors rarely result in practice. The counter-argument is that unrestricted covariance for parameter types and instance variables is unsound. Both sides are correct. One of the major problems in designing static type systems for object oriented programming languages has been to design sound typing rules that support the examples where covariance appears to be necessary.

In the next section we will introduce a new keyword, *MyType*, which can be used inside classes to stand for the type of *self*. The use of this keyword will allow us to provide for type-safe covariant changes to parameter and instance variables in many important cases, including the *DoubleNode* example. The cost of providing this support for covariance is that subclasses will no longer always generate subtypes.

[8]As noted in the previous section, it isn't!

5 Introducing *MyType*

The improved flexibility in allowing changes to types of methods in subclasses introduced in the previous section allowed some improvement in the type-checking capabilities of our object-oriented language, but it did not help with our node or cloning examples. Providing more accurate information on the type of *self* will turn out to be the key to overcoming these difficulties.

We have not yet discussed what type should be assigned to *self* for the purposes of static typing. Suppose we have a class C with a method m, whose body contains one or more occurrences of *self*. Languages with simple type systems like C++, Java, Object Pascal, and Modula-3 presume that *self* has the same type as the objects generated from the class being defined. By our conventions, this would mean we would assume that *self* has the type *CType* in order to type check m.

However, look at what happens when we define a subclass SC of C. If m is inherited without change, it is roughly equivalent to copying the code in the body of m into SC.[9] Thus, the meaning of *self* changes to denote an element of type *SCType*. However, type checking the body of m under the assumption that the type of *self* is *CType* is weaker than assuming its type is *SCType* as long as *SCType* is a subtype of *CType*. That is, if *SCType* is a subtype of *CType* and m type checks under the assumption that *self* has type *CType*, then it is guaranteed to type check under the (stronger) assumption that *self* has type *SCType*. Thus as long as subclasses always generate subtypes (as is the case in the simple type discipline described in section 3), we will not have difficulty with inherited methods becoming type-unsafe in subclasses.

Returning to the *Node* example in Figure 7, suppose we find a way to write *DoubleNode* as a subclass of *Node*. As remarked earlier, *setNext* should be a binary method. That is, its argument should be of the same type as the object it is sent to. Thus while *setNext* has a parameter of type *NodeType* in class *Node*, it ought to have a parameter of type *DoubleNodeType* in subclass *DoubleNode*. But then, because of contravariance, *DoubleNodeType* will not be a subtype of *NodeType*. Nevertheless it makes perfectly good sense to define *DoubleNode* via inheritance from *Node* since most of the code of corresponding methods is identical. Aside from adding the new *getPrev* and *setPrev* methods and the instance variable *prev*, only one of the methods of *Node* has revised code – *attachRight*, the others are exactly the same as in the class *Node*.

Thus if we do manage to increase expressiveness of the language in order to write these subclasses that we were previously barred from writing, then subclasses will no longer generate subtypes. As a result we will have to be more careful in type-checking methods involving *self*. It will no longer be sufficient to type check methods by assuming that the type of *self* has the same type as the objects generated from the class.

In order to be able to handle *self* more accurately in type checking, we introduce the keyword of *MyType* for its type. The key idea here is that *MyType* will play two different but related roles in describing the types of objects. In the typing of a particular object, the type expression *MyType* can simply be seen as another name for the type of the object. However, when we write the code for methods in a class, we will do it with the understanding that the meanings of *self* and *MyType* are variable and will change in tandem within subclasses.

We may think of *self* and *MyType* as abbreviations for recursive definitions of the object and its type. In fact, much of the early work on the semantics and typing of object-oriented languages was done in the context of recursively defined records of methods. Early papers reflecting this approach

[9] This is actually only true if the body of m does not include any mention of *super*, but we can ignore that possibility here.

```
class Node
   var
      value = 0: Integer;
      next = nil: MyType
   methods
      function getValue():Integer
        begin
          return value
        end;
      procedure setValue(newValue: Integer)
        begin
          value := newValue
        end;
      function getNext():MyType
        begin
          return next
        end;
      procedure setNext(newNext: MyType)
        begin
          next := newNext
        end;
      procedure attachRight(newNext: MyType)
        begin
          self.setNext(newNext)
        end
end class;
```

Figure 10: Node class with *MyType*.

include [Car88a, CP89, Red88] for untyped languages and [CHC90, Mit90] for typed languages.

The use of *MyType* provides for the smooth change of method types in subclasses, as is desired for clone operations. For example, we may specify the type of the *ShallowClone* and *DeepClone* methods to be *Func(): MyType*. If we send either of the messages *ShallowClone*() or *DeepClone*() to an object of type T, the result will be of the same type, T, as *MyType* stands for the name of the type of the object executing the method.

The use of *MyType* also solves our typing problems with *Node*. We have rewritten the *Node* example from Figure 7 in Figure 10, replacing all occurrences of *NodeType* by *MyType*. We have also used *MyType* to specify the type of the instance variable *next*.

We now define *DoubleNode* as a subclass of *Node* in Figure 11. The use of *MyType* in the types of instance variables and methods ensures that all occurrences will change uniformly in the subclass.

Language constructs giving the type of *self* an explicit name exist in Trellis/Owl [SCB+86], Eiffel [Mey88, Mey92], and Emerald [BHJ+87]. In Eiffel, the name for the type of *self* is a special instance of a more general construct that allows the programmer to use an expression of the form

```
class DoubleNode inherits Node modifying attachRight
   var
      prev = nil: MyType
   methods
      function getPrev():MyType
        begin
          return prev
        end;
      procedure setPrev(newPrev: MyType)
        begin
          prev := newPrev
        end;
      procedure attachRight(newNext: MyType)
        begin
          self.setNext(newNext);
          newNext.setPrev(self)
        end
end class;
```

Figure 11: Doubly-linked node class.

like x, for x an identifier. This construct evaluates at compile time to the static type of x. Because *self* in Eiffel is written *Current*, *MyType* is written *like Current*.

While we have shown that the use of *MyType* makes it possible to write cloning methods and allows us to write *DoubleNode* as a subclass of *Node*, we have not yet discussed how methods involving *self* and *MyType* can be type checked. While *self* has type *MyType*, what are we allowed to assume about *MyType*, when type checking methods? If we type check under the assumption that *MyType* is the same as the type of the objects generated by the class (*e.g.*, type check the methods of *Node* under the assumption that *MyType* is the same as *NodeType*), how can we be certain that the inherited methods will still be type correct in the subclass, where *MyType* will have a different meaning? In the next section we address this question.

6 The matching relation between types

Our type-checking rules for methods need to accommodate the fact that methods can be inherited in subclasses, where the meanings of *self* and *MyType* are different from those in the original class. The difficulty, then, is that within a particular object, *MyType* has a *fixed* meaning (the type of the object), yet inside a class it will have a more *flexible* meaning, because methods and instance variables from the class might also be inherited in a subclass. In this section we show how to cope with the changing meaning of *self* and *MyType* in classes.

Our goal is to type check methods in a class in such a way that they are guaranteed to be type-safe no matter how they are inherited. We saw in the previous section that we avoid typing problems if we replace the types of methods by subtypes in subclasses. The following definition of *matching* captures exactly this change in method types between object types.

We say that object type T' *matches* object type T, written $T' <\!\# \, T$, iff for each method $m_i\colon\ T_i$ of T there is a corresponding method $m_i\colon\ T_i'$ of T' such that $T_i' <: T_i$. Put slightly differently,

$$ObjectType\{m_j\colon T_j'\}_{1 \leq j \leq m} <\!\# \; ObjectType\{m_i\colon T_i\}_{1 \leq i \leq n} \text{ iff } n \leq m \text{ and for each } i \leq n,\ T_i' <: T_i.$$

It follows that if SC is a subclass of C then $SCType <\!\# \, CType$. We will see below that matching is weaker than subtyping. The difference between this and our earlier definition of subtyping for objects is the presence of the keyword *MyType*.

One important point here: in determining whether two object types match in the above definition, we determine if $T_i' <: T_i$ by treating *MyType* as an unconstrained free variable.[10] As a result the two types will certainly be subtypes in any particular context where *MyType* stands for some fixed type.

6.1 Type checking classes using matching

As discussed earlier, when defining subclasses we will insist that the types of redefined methods be subtypes of their types in the superclass. As a result the types of objects generated by subclasses will always match the type generated by the superclass. Thus we can type check methods of a class C under the assumption that *self*: *MyType* and $MyType <\!\# \, CType$. This will ensure that inherited methods are still type safe since the meaning of *MyType* in any subclass of C will match *CType*. (A proof of this can be found in [Bru94] or [BSvG95].)

We have one more puzzle to solve before we can type check classes. Suppose that there is an object, o, of type S with a method, m, whose type T involves *MyType*. What is the type of $o.m$? The obvious answer is that its type is T, modified so that all free occurrences of *MyType* are replaced by S, the type of the receiver. After all, *MyType* was designed to represent the type of the receiver.

We can make this more precise as follows. Let the notation $T[S/MyType]$ represent the type T in which all free[11] occurrences of *MyType* have been replaced by the type S. Then $o.m$ has type $T[S/MyType]$ if m has type T and o has type S. For example, if function *equal* is given type $Func(MyType)\colon\ Bool$ then $(Func(MyType)\colon\ Bool)[Ctype/MyType] = Func(CType)\colon\ Bool$.

Suppose $SCType <\!\# \, CType$, the type of method m in *CType* is T, and that the type of m in *SCType* is ST. By the definition of matching we know that $ST <: T$. If o is an object of type *SCType* then the type of $o.m$ is $ST[SCType/MyType]$. But when we determined that $ST <: T$, we made no assumptions about *MyType*. As a result, it is possible to show that $ST[SCType/MyType] <: T[SCType/MyType]$, because uniform substitution of a type expression for a type variable does not affect subtyping. (See the subtyping rules in [Bru94], for example.) In particular, then, $o.m$ also has type $T[SCType/MyType]$ by subsumption.

Thus we can type check an expression representing a message send to an object even if we don't know the object's exact type. In particular if $SCType <\!\# \, ObjectType m\colon T$ and o has type *SCType* then $o.m$ has type $T[SCType/MyType]$.[12]

[10] Actually it is safe to assume that $MyType <\!\# \, ObjectType\{m_i\colon T_i\}_{1 \leq i \leq n}$ when determining whether $T_i' <: T_i$, but we will not bother with that subtlety here. See [Bru94, BSvG95] for details.

[11] In case of nested object types, occurrences of *MyType* always refer to the most closely enclosing object type definition. We treat the ObjectType constructor as binding all free occurrences of *MyType* that occur within its definition.

[12] Though as above it might also have some other type which is a subtype.

Let us examine how we would type check the node example in Figure 10. The type correctness of *getValue* and *setValue* are obvious. The methods *getNext* and *setNext* are type correct because *next* has type *MyType*. Only the typing of the body of *attachRight* requires more careful analysis. As stated above, we assume *self* has type *MyType* and $MyType <\!\# NodeType$. Because $MyType <\!\# NodeType$, we know that *self* has a *setNext* method whose type is a subtype of $Proc(MyType)$. We conclude that the type of $self.setNext$ is a subtype of $Proc(MyType)[MyType/MyType] = Proc(MyType)$, since the substitution of *MyType* for itself has no effect. Because *newNext* has type *MyType*, the expression $self.setNext(newNext)$ is well typed.

We now type check the subclass *DoubleNode* in Figure 11. The assumptions we used in type checking *Node* (*i.e.*, that *self* has type *MyType*, and $MyType <\!\# NodeType$) remain true in the subclass. (In fact, in the subclass we are allowed to assume $MyType <\!\# DoubleNodeType$, which implies $MyType <\!\# NodeType$ by transitivity since $DoubleNodeType <\!\# NodeType$.) Hence we have no need to go back and re-examine any inherited methods for type correctness. The type correctness of methods *getPrev* and *setPrev* follow easily from the fact that *prev* has type *MyType*.

Thus we need only concern ourselves with the redefined method *attachRight*. In order to attach *newNext* to the right of *self*, first *newNext* is assigned to the receiver's *next* field via the call to *setNext*. Then the parameter, *newNext*, is asked to set *self* to its *prev* field via the message send of *setPrev* to *newNext* with parameter *self*. The first message send to *self* is type correct for the same reasons as in *Node*. The type correctness of the second message send follows by the same reasoning since *newNext* and *self* both have type *MyType*. Note that the method would not type check if the actual parameter to $newNext.setPrev$ were of type *DoubleNodeType* rather than *MyType*.

In summary, if

1. redefined methods in subclasses are constrained to have types which are subtypes of those given in the superclass, and
2. we type check methods of a class under the assumptions that *self*: *MyType* and *MyType* matches the type of object generated by the class,

then we will be guaranteed that the inherited methods remain type-safe in subclasses.

6.2 Matching is necessary in type checking classes

Our assumption of $MyType <\!\# CType$ which was used in type checking the class C, is relatively weak. More method bodies would type check if we assumed that $MyType = CType$, but we now show that they would not necessarily *remain* type-correct when inherited.

Figure 12 presents an example of a class/subclass pair in which a method (in this case *useEq*) of a class type checks under the assumption that *MyType* is exactly the type generated by the class (in this case *TestType*), but when inherited causes a run-time error due to a typing problem.

The subclass *Subtest* simply adds a new instance variable and method, and redefines the method *eq* (without changing the typing). If s is an object generated by class *Subtest* and p is an element generated by *Test* (and hence having type *TestType*), then the call $s.useEq(p)$ will cause a run-time error while within the body of the call of $s.eq(p)$ made from the body of *useEq*. The problem is that the receiver, s, is of type *SubTestType*, while the argument is of type *TestType*. Hence, when the version of *eq* from *SubTest* is called, it will send the message *getOtherval* to the parameter, p, which does not have a corresponding method to handle the call, causing a run-time error.

```
class Test
   var
      value = 0: Integer
   methods
      function getValue(): Integer {return value}
      function eq(pt:MyType):Bool {return (value = pt.getValue)}
      function useEq(pt: TestType):Bool {return eq(pt)}
end class;

class SubTest subclass of Test modifying eq
   var
      otherval = 0: Integer
   methods
      function getOtherval():Integer: {return otherval}
      function eq(pt:MyType):Bool {return (value = pt.getValue) &
                                    (otherval = pt.getOtherval)}
end class;
```

Figure 12: Node class with *MyType*.

Notice that the problem arises in the method *useEq* which was not changed in *SubTest*. According to our type-checking rules, the type of *s.useEq* should be *Func(SubTestType): Bool* (obtained from (*Func*(*MyType*): *Bool*) by replacing all occurrences of *MyType* by *SubTestType*). As a result, the application to a parameter of type *TestType* is clearly an error.

How would the weaker type-checking assumption adopted in this section keep us from this error? If we type check the methods of *Test* under the assumption that $MyType <\# TestType$, then the body of the method *useEq* will not type check. The problem is that the message send $eq(pt)$ in the body of *useEq* requires a parameter of type *MyType*, while *pt* has type *TestType*. Because we only know that $MyType <\# TestType$, we cannot deduce that *pt* also has type *MyType*. Thus type checking the method body of *useEq* results in a type error, keeping us out of the trouble illustrated above.

The reader may find it annoying to have type-checking rules which make illegal the code in Test, even though no errors will occur in objects generated directly from this class. However it is easy to change the code so that it is legal. Either change the type of the formal parameter of *eq* from *MyType* to *TestType* or change the formal parameter of *useEq* from *TestType* to *MyType*. While either change will enable the revised class to be type-correct, they have different impacts on the functionality of corresponding methods of subclasses.

This subtleness of type checking may make the reader nervous about uses of *MyType*, but the arguments presented earlier in this section should be convincing for its safety. Moreover, formal proofs of the soundness of the type-checking rules exist, and can be found in [Bru94] and [BSvG95], for example.

The appendix includes an example of a method that type checks under the strong assumption that $MyType = CType$, but that fails under the weaker assumption that $MyType <\# CType$. As a result, ignoring the fact that the meaning of *MyType* changes in subclasses can result in broken

```
procedure breakit(n1, n2: NodeType);
begin
   n1.attachRight(n2)
end
```

Figure 13: Procedure illustrating that subclasses need not generate subtypes.

code. Type-checking under the matching assumption guarantees that no type errors result during execution.

7 Binary methods complicate subtyping

While the use of *MyType* provides us with a great deal of flexibility, there is one negative consequence of its use that we hinted at earlier. The problem arises with the binary methods discussed earlier. We can now describe these more precisely as methods that contain a parameter of type *MyType*. The methods *attachRight*, *setNext*, and *setPrev* from the *Node* and *DoubleNode* classes in Section 5 are examples of binary methods. Other familiar examples include methods *equal* and *lessThan* that would require a parameter of the same type as the receiver, and then return Boolean values.

7.1 Subclasses do not generate subtypes

As suggested earlier, binary methods cause problems with subtypes because the interpretation of the *MyType* parameter of the binary method in the subclass is "smaller" than the interpretation in the superclass (varies covariantly), in contrast to the rules for subtyping of functions, which require only contravariant changes to parameters. Thus if a class has a binary method, its subclasses will not generate subtypes (though methods returning *MyType*do not in themselves get in the way of subtyping).

We can make this problem more concrete by showing how our type system would be unsafe if we assumed that *DoubleNodeType* were a subtype of *NodeType*. As expected, the problem arises with a binary method, *attachRight*. Suppose we write the simple procedure *breakit* in Figure 13.

If *n1* is of type *NodeType*, then *n1.attachRight* has type $Proc(MyType)[NodeType/MyType] = Proc(NodeType)$. If *n2* also has type *NodeType*, $n1.attachRight(n2)$ type checks correctly, as does the definition of *breakit*.

Now suppose that we make the call $breakit(dn, n)$ where *dn* is of type *DoubleNodeType* and *n* is of type *NodeType*. This causes the message *attachRight* to be sent to *dn*. However the corresponding method in *dn* expects a parameter of type *DoubleNodeType*. Instead the parameter *n* of type *NodeType* is provided. When the code of *attachRight* from class *DoubleNode* is executed, first the method *setNext* is sent to *self*, causing no problems. However the method *setPrev* is then sent to the actual parameter *n*. Because *n* has no method with this name, the program would encounter a "message not understood" error at this point.

What could have gone wrong? Since the code of the procedure appears to type check correctly, it must be the call which is not correct. Here the problem is the assumption that *DoubleNodeType* is a subtype of *NodeType*! In fact they are not subtypes, and it is illegal to make the call of *breakit* with *dn* as a parameter.

7.2 A new definition of subtyping for object types

Now that we have introduced *MyType*, we must extend our definition of subtyping to consider the different meanings of *MyType* in the type expressions being compared.

We say that a type T is *monotonic* in type identifier S iff $S <: S'$ implies that $T <: T[S'/S]$ for all S' not occurring in T. For example, by our subtyping rules, the function type $Func(S)\colon U$ is monotonic in U, but fails to be monotonic in S.

A safe definition of subtyping for object types is:

$ObjectType\{m_j\colon T'_j\}_{1\leq j\leq m} <: ObjectType\{m_i\colon T_i\}_{1\leq i\leq n}$ if the two types are equivalent *or*

1. $n \leq m$ and for each $i \leq n$, $T'_i <: T_i$, and
2. each method type T_j in T is monotonic in *MyType*.

It follows from this definition that $ObjectType\{m_i\colon T_i\}_{1\leq i\leq n}$ has no proper subtypes if any of the m_i has a parameter of type *MyType*.

Notice that the only difference between matching and subtyping is the addition of the second clause in the definition of subtyping. Thus if two object types are subtypes, they match, but not vice-versa. However, if all of the method types are monotonic in *MyType*, then the two types match iff they are subtypes.

DoubleNodeType is *not* a subtype of *NodeType*, because *attachRight* has a parameter of type *MyType*. However, *DoubleNodeType* $<\#$ *NodeType*. In contrast, there is no difficulty with subtyping in classes containing methods like the *ShallowClone* and *DeepClone* methods discussed earlier, since *MyType* occurs only as the result type.

This new definition of subtypes for object types is based on the definition of subtyping for recursive types in [AC93] and on the definition in [BHJ$^+$87], as type expressions with *MyType* can be interpreted as being implicitly recursively defined. The rules in that paper would actually allow more types to be subtypes than can be determined with the above rule. The reason is that the general rules given there allow arbitrary "unfolding" of recursively defined types. We do not allow that here because of complications that would arise in typing expressions. It is possible to strengthen the above definition by using the assumption *MyType* $<\#$ $ObjectType\{m_i\colon T_i\}_{1\leq i\leq n}$ to prove $T'_i <: T_i$ in the first part of the definition.

While other choices of subtyping rules for object types are possible, we have found that this particular choice of rules is both simple and effective.

8 Evaluating the use of *MyType*

The net gains with the introduction of *MyType* are substantial. We can now write down the types of methods in classes in such a way that they will automatically change as desired in subclasses. We have also increased the expressibility of the language by making it possible to use inheritance to define subclasses (such as *DoubleNode*) which could not have previously been defined by inheritance. When these new subclasses have binary methods, however, they may not give rise to subtypes.

We did not run into this mismatch between subclass and subtype earlier, because we were not even allowed to write down subclasses when we wished to have binary methods. With the use of *MyType*, we now can write such subclasses and make very effective use of them in real programs.

As indicated earlier, the paper [CHC90] examined typing problems in object-oriented languages by looking at a typed semantics of object-oriented languages. In that paper the authors provided examples similar to the one shown here that show that subclasses do not always generate subtypes; it also discussed replacing method types by subtypes in subclasses.

It is possible to restrict the use of *MyType* to only covariant positions (*e.g.*, result types of functions) in order to ensure that subclasses always generate subtypes. Trellis/Owl [SCB+86] imposes essentially this restriction by imposing the requirement that a subclass is legal only if the generated object type is a subtype of that of the superclass. Thus, while allowing subtyping on method types in subclasses, it would not allow the creation of a subclass for a class with binary methods.

However practical experience leads us to consider the support of binary method to be very important. Moreover, since support for binary methods has no impact on the type correctness or subtyping properties of programs or classes that do not contain binary methods, we see no useful reason for banning contravariant occurrences of *MyType*.

Let us summarize where we are now in the development of a static type-checking system for object-oriented languages. We have introduced the type expression, *MyType*, which is used in methods to stand for the type of the object executing the method. The use of *MyType* allows us to define methods whose types change automatically in subclasses in order to provide more accurate type information. When a method in a class is overridden in a subclass, its type may be changed to a subtype of what it was in the superclass.

We introduced the notion of matching in order to capture more accurately the relation between the types of objects generated by superclasses and subclasses. If class SC is a subclass of C then the corresponding types are guaranteed to be in the matching relation. The rules for type checking methods in classes allow us to assume *self* has type *MyType* and that *MyType* matches the object type generated by the class. The relatively weak matching assumption on *MyType* ensures that inherited methods continue to be type correct in subclasses. Assuming that *MyType* is the same as the object type generated by the class can lead to typing problems in inherited methods in subclasses.

Our definition of subtyping is more restricted than the definition of matching. If a class definition contains any contravariant occurrences of *MyType*, in particular, if the class contains one or more binary methods, subclasses will not generate subtypes.

Moreover, once we have determined that an object's type matches another object type, we can determine a type for the result of sending a message to the object by replacing all occurrences of *MyType* in the method's type by the object's type. Thus it turns out that matching, not subtyping, is the most relevant for determining the type of message sending.

The type system of our language is now significantly more expressive than the simple type system with which we started. In particular, we are now able to write the cloning methods which caused us problems in section 3. We can also now define *DoubleNode* as a subclass of *Node*, as desired earlier. However there remain a few subclasses we would like to write that are still out of our reach because of typing restrictions. These include the circle / color circle example from section 3, and other examples which seem to require covariant changes to the types of instance variables or the parameters of methods in subclass definitions. The use of *MyType* took care of these examples when the type being changed was that of the objects being defined in the class. In the next section we introduce bounded type parameters, which will allow us to overcome these other problems as well.

```
List(T) = ObjectType
             function find(T): Boolean;
             procedure insert(T);
             procedure delete(T);
             function isEmpty(): Boolean;
             ...
          end ObjectType;
```

Figure 14: List type function.

9 Combining parametric polymorphism with matching

In this section we illustrate the extra flexibility obtained by supporting a form of parametric polymorphism that is similar to that provided in CLU's parameterized clusters and Ada's generic packages.

9.1 Polymorphism and container classes

Polymorphism is very important for clean programming of a variety of problems, especially those involving data structures called "container classes". Container classes are those classes representing collections of objects, usually of the same or related types.

A good example of a container class is a list of elements. The code for a particular list implementation is essentially the same, no matter what type of element is held in the list. Similarly the interfaces vary only in the type of the element held in the list. As a result we can represent the type of lists with *List*, a function from types to types. (See Figure 14.) We can create type expressions representing lists of various types by simply applying the type function to the intended type of the elements. Thus $List(Int)$ represents the type of a list of integers, while $List(Window)$ represents the type of a list of windows.

9.2 Constraining polymorphism - the failure of subtyping

Some of these container classes, however, require the types of elements contained in the data structure to support certain operations. For instance, a binary search tree will only work with elements whose types support comparisons between elements. As another example, a data structure representing a collection of objects on a computer screen may require each element to have a bounding rectangle. How can we express this dependency in order to ensure that type functions are applied only to the appropriate types?

Let us look at the case of binary search trees in order to get more insight into the problem. In order to insert and find an element in a binary search tree we must be able to compare that element with other elements of the type. In Figure 15 we present an object type, *Comparable*, which supports the necessary comparison operations. Any type of element that can be stored in a binary search tree must support at least these operations. How can we use this to restrict the types to which the type function, *BinarySearchTree*, is applied?

We might expect that any type that is a subtype of *Comparable* would be acceptable. However, because *Comparable* contains method types that include contravariant occurrences of *MyType* (*i.e.*,

```
Comparable = ObjectType
                function equal(MyType): Boolean;
                function greaterThan(MyType): Boolean;
                function lessThan(MyType): Boolean;
             end ObjectType;
```

Figure 15: The type *Comparable*.

binary methods), no non-trivial subtypes of *Comparable* exist.

9.3 Match-bounded polymorphism

Interestingly, matching allows us to express exactly the types that we are interested in. If $T <\!\# \; Comparable$, then T must have *equal*, *greaterThan*, and *lessThan* methods, each with function types that expect an argument of the same type as the receiver, and return Booleans. This is exactly what we want. This mechanism of restricting type parameters using matching is called *match-bounded polymorphism* or *bounded matching.*

The definitions of the type functions *BinSearchTreeType* and *BinTreeNodeType* in Figure 16 provide examples of the use of match-bounded polymorphism in the definition of a function from types to types. The type *Top* which serves as the upper bound for the type parameter in *BinTreeNodeType* is a built-in type that every other type matches. The corresponding class provides methods like *ShallowClone* and *equal* which are implicitly inherited by every other class of the language.

Figure 17 shows part of the definitions of a node class and binary search tree class, both of which are parameterized by the types of elements they contain. (The node class is also parameterized by an initial value for the node value. This can be treated as syntactic sugar for a function returning a class.) The need for the constraint on the type parameter T in *BinSearchTree* is illustrated by the expression $elt.equal(current.getValue())$. Because $T <\!\# \; Comparable$ and *elt* has type T, *elt* is guaranteed to have a method *equal* with the necessary typing.

A more complicated example can be constructed of an ordered linked list, which is parameterized both by the type of value it contains (which is required to match *Comparable*) and by the type of node it is composed of, a type that is required to match *NodeType* (the type of objects generated by the *Node* class from the previous section). For instance, the user can instantiate the ordered list with an object type supporting integers with the usual order and with type *NodeType* in order to obtain a singly-linked list of integers. Alternatively, one can instantiate it with an object type representing strings and with the type *DoubleNodeType* in order to obtain a doubly-linked list of strings. (See [BSvG95] for details of the example.)

9.4 History of matching and match-bounded polymorphism

Bounded polymorphism (using subtyping rather than matching) was introduced in [CW85]. That paper also included a very extensive discussion of types and subtypes in object-oriented languages. The failure of bounded polymorphism using subtyping to capture the constraints needed in examples like our binary search tree example above was first pointed out in [CCH+89] and [Hut87]. In [CCH+89], the authors invented a generalization of bounded polymorphism, called F-bounded quantification, which provided the correct constraints on polymorphism. F-bounded quantification

```
TypeFunction BinSearchTreeType(T <# Comparable) =
   ObjectType
      function find(T): Boolean;
      procedure insert(T);
      function isEmpty(): Boolean;
   end ObjectType;

TypeFunction BinTreeNodeType(T <# Top) =
   ObjectType
      function getValue():T;
      procedure setValue(T);
      function getLeft():MyType;
      procedure setLeft(MyType);
      function getRight():MyType;
      procedure setRight(MyType);
   end ObjectType;
```

Figure 16: Binary search tree type functions.

is a generalization of bounded quantification using subtyping in which the bounded variable is allowed to appear in the bound. Using this in languages supporting recursive types in place of *MyType* provides the same functionality as match-bounded polymorphism.

For example we can define

```
TypeFunction CompFcn(T) = ObjectType
                function equal(T): Boolean;
                function greaterThan(T): Boolean;
                function lessThan(T): Boolean;
             end ObjectType;
```

Then, using F-bounded quantification, we can rewrite the bound on *BinSearchTreeType* as

$$\texttt{TypeFunction BinSearchTreeType}(\texttt{T} <: \texttt{CompFcn}(\texttt{T}))$$

Notice that the bounded variable, T, occurs in the upper bound, $CompFcn(T)$. Unfolding rules of recursive types can then be used to show that the expected (recursive) object types satisfy the constraint.

The notion of matching given here and match-bounded polymorphism were introduced in [Bru94] as a simpler way of expressing the constraints of F-bounded quantification in order to specify the semantics of typed object-oriented languages. That paper also pointed out the usefulness of matching in type checking classes and subclasses. As pointed out in [AC95], a somewhat better encoding for matching can be obtained through the use of higher-order subtyping [PS95].

A concept similar to match-bounded polymorphism seems to have been invented independently by several language designers. The term *matching* was introduced in [BH91] for a definition that is very similar to that of F-bounded quantification. This construct was used in the distributed

```
class BTreeNode(T <# Top; v:T)
   var
      value = v: T;
      left = nil: MyType;
      right = nil: MyType
   methods
      function getValue(): T
        begin
          return value
        end;
      procedure setValue(newValue: T) ...
      function getLeft(): MyType
        begin
          return left
        end;
      procedure setLeft(newLeft: MyType)
        begin
          left := newLeft
        end;
      function getRight(): MyType  ...
      procedure setRight(newRight: MyType)  ...
end class;

class BinSearchTree(T <# Comparable)
   var
      current = nil: BTreeNodeType(T);
      root = nil: BTreeNodeType(T);
   methods
      function find(elt: T): Boolean
        begin
          if root <> nil then
            ... if elt.equal(current.getValue()) then ...
        end;
      procedure insert(newElt: T)  ...
      function isEmpty(): Boolean
        begin
          return (root = nil)
        end
end class;
```

Figure 17: BinSearchTree classes.

language Emerald [BHJL86, BHJ+87, Hut87]. The language School ([RIR93]) also constrains polymorphism with a mechanism similar to F-bounded quantification.

Interestingly, the type restrictions obtained by match-bounded polymorphism are equivalent to those obtained by mechanisms for restricting type parameters in ADT-style languages supporting polymorphism, like CLU and Ada. In Ada, for instance, one would write:

```
generic T with
   function equal(T,T): Boolean;
   function lessThan(T,T): Boolean;
   function greaterThan(T,T): Boolean;
package BinSearchTree ... end BinSearchTree;
```

While the functions are written as functions of two parameters here (rather than the single parameter for the object-oriented style), the restrictions on the admissible types are equivalent. This provides further evidence that matching is a very natural notion in object-oriented languages. In fact the object-oriented language Theta [DGLM95, DGLM94] uses a similar notation to express restrictions on type parameters that are equivalent to match-bounded polymorphism.

10 Solutions to Eiffel's covariant type problems

The object-oriented language Eiffel allows covariant changes to types of instance variables and to both parameter and result types of methods in subclasses. Supporters of Eiffel argue that allowing these covariant changes is essential in order to support flexible program development. However, as we have seen, this permits the introduction of type errors.

10.1 Eiffel's system validity check

The Eiffel language definition [Mey92] attempts to compensate for these problems by mandating a link-time "system validity check" that performs a dataflow analysis of the complete final program at link time in order to identify those expressions that are at risk for unsafe method calls. Unfortunately, systems that depend on a dataflow analysis for type safety are fragile, as potentially problematic code can go undetected for a long while until a "dangerous" method call is made in a new or modified class.

While a detailed algorithm for the system validity check can be found in [Mey92], it has apparently never been implemented, and is not available in any of the existing commercial Eiffel compilers. As a result, it is unknown how conservative (or expensive) this system validity check is.

10.2 Solving covariance problems with match-bounded polymorphism

We prefer systems which catch type errors at compile time, rather than postponing their discovery until link time or run time. Cook, in [Coo89], presented several suggestions on how to fix the problems with Eiffel. An important part of these suggestions is the use of bounded quantification to replace the uses of covariance in instance variable and parameter types, in particular, those occurrences arising from uses of Eiffel's *like* x construct.

In section 5, we provided a solution to the problem of covariant changes to parameter and instance variable types for the cases when the types involved can be represented using *MyType*. In those cases the type represented by *MyType* changes automatically in subclasses. Our assumptions on the meaning of *MyType* in type checking methods ensure that this implicit covariant change to parameter types causes no type problems. In the rest of this section we show how to use match-bounded polymorphism to replace the apparent need for covariant changes when types other than *MyType* are required to change.

We revisit the *Circle* / *ColorCircle* example discussed in section 4, and presented in Figure 9, in order to see how to use match-bounded polymorphism to model this in a type-safe way. Recall that *Circle* has a *getCenter* method that returns a point, while we wished the same method to return a color point in the subclass, *ColorCircle*. Subtyping allowed us to change the return type as desired, but we later ran into difficulty because we were not allowed to change the instance variable, *center*, from type *PointType* to *ColorPointType* in the subclass.

Using match-bounded polymorphism, we can work around this by adding a type parameter that is required to match *PointType*. This type parameter represents the type of the center of the circle. See the class *PCircle* in Figure 18. Now any subclass of *PCircle* can instantiate the type variable by any type that matches *PointType*, solving our problem, though at the cost of requiring the programmer to plan ahead for subsequent changes in subclasses.

A new color circle can be created by evaluating *new PColorCircle*(*ColorPointType*, *cpt*) where *cpt* is a color point that is to be the location of the center of the circle. Thus the addition of match-bounded polymorphism has allowed us to get around the difficulties caused by the restrictions on changing instance variable types. Because the bodies of the methods of *PCircle* are type checked under the assumption that $CenterType <\# PointType$, they are guaranteed to work when the subclass is instantiated with *ColorPointType*.

Our second example, due to Shang [Sha91], is presented in Figure 19. It is typical of those given to illustrate the need for covariant subtyping on parameter types in subclasses. In this example, an *Animal* can eat any kind of food, while an *Herbivore* eats only plants.

Because the parameter of *eat* changes in a covariant way, this is an illegal subclass according to the typing rules with which we have been working. In fact, it is easy to code up an example using these definitions that would type check correctly according to the Eiffel rules, yet be unsafe, if *Herbivore* were allowed to be a subclass of *Animal*. Nevertheless this seems like an obvious change in parameters for the subclass because we clearly wish to place more restrictions on the parameter in the subclass. This covariant change in parameters is characteristic of many desired subclass relationships in object-oriented programming languages.

Because covariant changes to parameters can cause type-checking problems, we would like to avoid this definition of *Herbivore*. To do this we need to analyze more carefully the modeling which is reflected in these classes.

While for each item of type *FoodType* there may be a kind of animal that can eat that food, virtually all kinds of animals have some dietary restrictions. Cows eat grass and hay, but do not eat meat. Humans eat a variety of plants and animals, but do not typically eat either grass or hay. Thus while it is proper to conclude that any item of *FoodType* is potentially edible by some animal, it would be incorrect to conclude that any, let alone all, food items will be edible by all animals. As a result, the *Animal* class definition, particularly with the functionality assigned to *eat*, does not really accurately reflect the situation to be modeled. A more realistic model of the class would allow us to change the parameter type for the *eat* method for each particular kind of animal.

The polymorphic *PAnimal* class in Figure 20 is parameterized over the type of food eaten by the animal. Match-bounded polymorphism has been used to impose a bound of *FoodType* on the

```
class PCircle(CenterType <# PointType, OrigPoint: CenterType)
   var
      center := OrigPoint: CenterType;
      radius := 1: Integer
   methods
      function getCenter():CenterType
        begin
          return center
        end;
      procedure changeCenter(newCenter:CenterType)
        begin
          center := newCenter
        end;
      procedure move(dx,dy: Integer)
        begin
          center.move(dx,dy)
        end;
      ...
end class;

class PColorCircle(CenterType <# ColorPointType, OrigPoint: CenterType)
      inherits PCircle(CenterType, OrigPoint)
   var
      color := red: ColorType
   methods
      function getColor():ColorType
        begin
          return color
        end;
      procedure setColor(newColor:ColorType)
        begin
          color := newColor
        end;
      ...
end class;
```

Figure 18: Polymorphic Circle classes.

```
class Animal
   var
      ...
   methods
      procedure eat(meal: FoodType)
      ...
end class;

class Herbivore inherits Animal modifying eat
   var
      ...
   methods
      procedure eat(meal: PlantType)  -- illegal change of type
      ...
end class;
```

Figure 19: Animal and herbivore classes

type parameter, where *FoodType* might include methods returning the weight and number of calories that would be extracted if the item is consumed. Notice that in the (parameterized) subclass, *PHerbivore*, we have replaced the upper bound of *FoodType* by *PlantType*, where *PlantType* is an object type that must match *FoodType*. The requirement that *PlantType* match *FoodType* comes from the fact that *PHerbivore*(*MyFoodType*) is defined to be a subclass of *PAnimal*(*MyFoodType*). But *PAnimal*(*MyFoodType*) is only well-defined if *MyFoodType* $<\#$ *FoodType*. If *PlantType* $<\#$ *FoodType* and *MyFoodType* $<\#$ *PlantType* then transitivity implies that *MyFoodType* $<\#$ *FoodType*.

As an example, suppose there is an object type *Vegetable* $<\#$ *PlantType*. Then we can create the class *PHerbivore*(*Vegetable*) which generates objects that can eat only vegetables. Note that *PHerbivore*(*Vegetable*) is a legal subclass of *PAnimal*(*Vegetable*), since the corresponding methods have exactly the same types, though the types of the objects they generate are not in the subtype relation. On the other hand, *PHerbivore*(*PlantType*) does *not* inherit from *PAnimal*(*FoodType*) and hence is not its subclass.

The method *eat* of the parameterized *PAnimal* class will be type checked under the assumption that *MyFoodType* $<\#$ *FoodType*, since that is all that is known about the parameter type from its declaration. Notice that this is exactly the same kind of assumption we make about *MyType* when type checking methods in classes. What we have done here with parameterized classes is to hand-code the same kind of flexibility that we get for free with *self* and *MyType*. While this takes more work on the programmer's part, we get the effect of covariance while retaining type safety – clearly something of value!

10.3 Meyer's solution: Banning polymorphic catcalls

Bertrand Meyer, the designer of Eiffel, has rejected this sort of solution to the covariance typing problems in Eiffel, because it requires the programmer to plan ahead in order to accommodate subclasses. That is, the designer of the *Animal* class must know in advance to make the class polymorphic in the food type or the class representing herbivores will not be definable as a subclass.

```
class PAnimal(MyFoodType <# FoodType)
   var
      ...
   methods
      procedure eat(meal: MyFoodType)
      ...
end class;

class PHerbivore(MyFoodType <# PlantType)
      inherits PAnimal(MyFoodType) modifying eat
   var
      ...
   methods
      procedure eat(meal: MyFoodType)  -- Same type as in PAnimal!
                 -- Declaration need not be repeated if no change in body.
      ...
end class;
```

Figure 20: Polymorphic animal and herbivore classes

In the fall of 1995, Meyer [Mey95] proposed a different solution to the covariance typing problems. His solution was to identify and ban what he called *polymorphic catcalls.* While this notion is too complex to be defined here, this proposal was intended to bar particular kinds of message sends to objects whose type was not known exactly.

To illustrate how this proposal would preserve type safety, let us return to our *breakit* procedure in Figure 13 which shows that subclasses need not be subtypes. In that example the message $attachRight(n2)$ was sent to $n1$, where $n1$ was a parameter declared to have type $NodeType$. The problem arose when the object corresponding to $n1$ actually had type $DoubleNodeType$. We avoided this problem by not allowing $DoubleNodeType$ to be a subtype of $NodeType$. It failed to be a subtype because *attachRight* of $NodeType$ was a binary method

Meyer's new proposal would avoid the problem as follows. He would label $n1.attachRight(n2)$ a catcall since the type of the formal parameter of *attachRight* is changed in a covariant way in a subclass of $Node$. The receiver $n1$ would be termed polymorphic since the exact type of its value is unknown. This follows since it is a formal parameter and hence any element of a subclass of *Node* may be used as a parameter. Hence the entire expression is considered a polymorphic catcall and thus is illegal.

An interesting part of the definition is that the original expression would not be considered a catcall until a subclass of *Node* is declared. Thus the procedure *breakit* would be legal in an Eiffel program unless and until a subclass is defined. Meyer argues that an incremental compiler could determine this change in status at no great cost.

Thus in the solution proposed earlier in this paper, the procedure *breakit* is legal, but it may only be applied to parameters of type $NodeType$ (since $NodeType$ has no subtypes). We could also use match-bounded polymorphism to rewrite the procedure as in Figure 21. This is still type safe, but is more flexible in that it would allow calls of the form $nobreakit(SomeNodeType, nd1, nd2)$

```
procedure nobreakit(T <# NodeType; n1, n2: T);
begin
   n1.attachRight(n2)
end
```

Figure 21: A type safe rewrite of *breakit* using match-bounded polymorphism.

as long as *nd1* and *nd2* both have type *SomeNodeType*. In particular it could be called with *DoubleNodeType* and two actual parameters of type *DoubleNodeType*.

In Meyer's proposal, the original *breakit* would be legal only as long as *Node* had no subclasses. When the first subclass was defined, the procedure would become illegal and have to be rewritten or eliminated.

We have several concerns with Meyer's new proposal.

1. The definition of polymorphic catcall (and related restrictions) is very complex, and would be difficult for most programmers to understand in detail.

2. We are not completely confident that this proposal will actually solve all of the type problems of Eiffel. The original proposal did not take care of some problems arising from changes to instance variables in subclasses. While the rules have since been corrected, the complexity of the rules and the lack of a proof of correctness leaves room for doubt.

3. The proposal may be too conservative. A receiver of a message is polymorphic if it can evaluate to an object of more than one class. In most large systems subtyping and inheritance are used a great deal, and this may lead to a large number of polymorphic expressions to which catcalls may not be sent.

One of Meyer's criteria for a good solution to the covariant subtyping problem is that it not require the programmer to plan in advance for future changes in parameters that might be required in subclasses. Thus he would argue against the use of type parameters in the previous subsection in order to write the *Animal* and *Herbivore* classes, because the programmer would have to realize when writing the *Animal* class that the parameter to the *eat* method might need to be changed in the subclass.

However, Meyer's solution suffers from a similar problem in that if a subclass is defined, it will invalidate the existing *Animal* code, forcing it to be rewritten.

Only experience with Meyer's *polymorphic catcall* rules will determine whether or not there is any validity to these concerns. Meanwhile, the type system discussed in this paper provides an alternative solution to the covariant typing problems of Eiffel, albeit one that requires the programmer to plan ahead.

11 Replacing subtyping with matching

> You know you have achieved perfection of design, not when you have nothing more to add, but when you have nothing more to take away. *Antoine de Saint Exupery*

At this point we have an extremely flexible static typing system for object-oriented languages. Method types may be replaced by subtypes in subclasses, *MyType* supports automatic updating of parameter types in important special cases, while a combination of *MyType* and bounded matching provides support for safe uses of covariance in parameter types. While contravariant occurrences of *MyType* do cause a loss of subtyping, the matching relation (which is preserved by subclasses) appears to be more important in type checking message sends and in restricting polymorphic types. Most important of all, we have a type system that is provably type safe (see [BSvG95] for formal details of a type system, an operational semantics, and a proof of type safety for a language similar to what has been described here).

Not surprisingly, this has come at some increased cost in complexity. While a strong case can be made that any language (object-oriented or not) should support some version of bounded polymorphism, the bounds must be specified using the new relation, matching, rather than subtyping. More troubling, the difference between subtyping and matching on object types is very subtle, depending only on the absence of contravariant occurrences of *MyType*.

When faced with such a system, it is reasonable to look for ways to reduce complexity. The existence of two relations as similar as subtyping and matching raises the question of whether both are really necessary. We were forced to introduce matching as a way of capturing the relation between types of objects generated by superclasses and subclasses. This notion proved to be important in type checking method bodies and in defining the set of types to which a parameterized type or class could be applied. A case can be made that matching is more natural than subtyping for object types because it is simpler (no need to worry about contravariant occurrences of *MyType*). An argument could also be made that most object-oriented programmers would, in fact, give the definition of matching if asked to define subtyping for object types in such a system.

In the rest of this section we introduce a new experimental typing system for object-oriented languages that drops subtyping entirely in favor of matching. To accomplish this we introduce one new construct involving matching in order to recapture some of the flexibility of subtyping.

11.1 Simplifying matching

In section 4 we eased the restrictions on changing types of methods in subclasses to replace the types of methods by subtypes. While this provided us with a somewhat more flexible type system, it is interesting to note that none of the examples examined once we introduced *MyType* actually took advantage of this flexibility. The use of *MyType* took care of most of the desired flexibility, and the other examples where covariance in parameters or instance variables seemed to be needed were not helped by allowing subtyping of method types. Instead, they required the introduction of bounded matching to attain the flexibility needed. Thus, it is reasonable to wonder whether it was really necessary to allow methods in subclasses to have types that are subtypes of those in the superclass.

As expected, the notion of matching will play a key role in replacing subtyping. In section 5, matching was defined in terms of subtyping of corresponding method types. Since we now wish matching to be primitive, we give a new more restrictive definition of matching. We define

$$\mathit{ObjectType}\{m_i\colon T_i\}_{1\leq i\leq k} \mathrel{<\#} ObjectType\{m_i\colon T_i\}_{1\leq i\leq n} \text{ iff } n \leq k.$$

Thus one object type matches another if the first can be obtained by adding more methods to the second. No method types may be changed in object types that match. (Of course, as before, occurrences of *MyType* mark places where types automatically change meaning.)

11.2 Replacing subtyping by hash types

All previous uses of subtyping in parameters that are of object type could be replaced by polymorphic functions using bounded matching. Thus a function f to be used with objects of any type that matches type *Window* could be written:

$$\mathit{function}\ f(W <\!\!\# \ \mathit{Window},\ w\colon\ W)\ \ldots$$

Unfortunately, this notation is a bit heavy, and programmers are not likely to enjoy passing around type parameters everywhere that they formerly used subtyping.

To remedy this we introduce a new type constructor. If T is an object type, then $\# T$ will also be a type. An element will have type $\# T$ if it has any type that matches T. That is, if a has type S and $S <\!\!\# \ T$ then a also has type $\# T$. We also have a form of subsumption with #-types. If $S <\!\!\# \ T$ and a has type $\# S$ then a also has type $\# T$. (A similar kind of notation occurs in [Car88b], where power types are introduced to stand for the collection of types which are subtypes of a given type. Ada 95 [Int95] also includes a similar notation, written $T'\mathit{Class}$, to designate the collection of types which are subclasses of T.)

We can now rewrite the declaration of the function f above as follows:

$$\mathit{function}\ f(w\colon\ \#\mathit{Window})\ \ldots$$

Thus f can be applied to any value with a type that matches *Window*. Note that this offers at least the flexibility of subtyping. In fact, if *Window* contains any binary methods, it will have no non-trivial subtypes, while there are many types that would match it. While only object types can be annotated with # in order to get the effect of subtyping, in most object-oriented languages, only object types and simple types are available to be passed as parameters. (If the language does not interpret record types as degenerate object types, one could define a notion of matching for record types in which matching is simply record extension.)

We have emulated subtyping of function parameters with #-types, but what about the other important use of subtyping, making assignments to variables? This case is more difficult in that this use cannot be easily expressed with bounded matching. However, we can take our #-types seriously, and use them to provide types for variables that are intended to hold values of multiple types. If $x\colon\ \# T$ is a variable declaration and $e\colon\ S$ for $S <\!\!\# \ T$, then $x := e$ will be type-correct.

We emphasize that this is no longer an abbreviation for a use of bounded matching. It is an entirely new type construct.[13] The introduction of #-types makes it easier for a programmer to express more precisely his/her intentions.

For example, suppose that we define the type function *List* as in Figure 14. In a language with subtyping, if we applied *List* to a type S, the list could also hold elements of any subtype of S. Of course, if S has binary methods, it will not have non-trivial subtypes, and we could not have a heterogeneous list of these elements. The point is that the choice of whether the list is homogeneous or heterogeneous is determined not by the programmer, but depends only on whether or not S has a binary method.

In the language we propose here without subtyping, applying *List* to type S would provide the interface for a homogeneous list: all of the elements would have exactly type S. We can also provide a list definition using #-types that will give us a heterogeneous list. Thus a language with #-types

[13] For those familiar with existential types, we note that $\# T$ can be encoded as $\exists t <\!\!\# \ T.\ t$.

provides the programmer with much greater expressiveness by providing the capability for either homogeneous or heterogeneous data structures depending on whether the programmer chooses to use regular types or hash types.

In Appendix A we provide an extended example of the definition of a heterogeneous list in a language with #-types. As in the example, one would normally wish to restrict the elements of heterogeneous data structures to have types which match a fixed type (in this case, the type *rect_type*). This is generally necessary to determine the collection of methods that can be applied to all elements of the data structure.

11.3 Hash types are not compatible with binary methods

This extra power and flexibility does not come entirely for free. In fact, there is one extra complication in sending messages that correspond to binary methods. If all we know about the type of an expression is that it *matches* a particular type, then we cannot send it a message which corresponds to a binary method. This is a necessary consequence of the need in a binary method to have a parameter with exactly the same type as the receiver. If one doesn't know the exact type of the receiver, you certainly won't know what type of value to provide as the parameter.

For example, recall that in our discussion of typing in section 6, we determined that if o: S and $S <\!\#\ \mathit{ObjectType}\ \{m\!: T\}$ then the type of $o.m$ would be $T[S/\mathit{MyType}]$. It was important for that rule that we knew the exact type of o (at least up to subtype). Now what happens if o has type $\#S$? Since we only know o's type up to matching, we cannot just blindly apply the usual type-checking rule. However it can be shown that if T has no contravariant occurrences of *MyType*, $o.m$ can be given type $T[\#S/\mathit{MyType}]$. If T does have contravariant occurrences of *MyType* (*e.g.*, it is a binary method), then we cannot statically determine the type of $o.m$.

This failure to determine a type for the message send of a binary method should not be a surprise. We illustrate with the *Node* example from section 6. If we only know that *n1* has type *#NodeType* and that *n2* has type *DbleNodeType*, then we cannot determine whether or not *n1.setPrev(n2)* is legal. It will only be legal if the type of the value of *n1* is exactly *DbleNodeType*. Any other type matching *NodeType* would not result in a well-typed expression.

One practical consequence is that we can generally only send non-binary messages to elements of a heterogeneous data structure (*i.e.*, one whose value fields are #-types) unless the language provides a type-case or similar statement to help discriminate the types at run-time. Note however that the existence of binary methods in a type S does not prevent the sending of non-binary messages to objects of type $\#S$.

While forbidding binary messages to be sent to #-types might seem to be a major handicap, we have found that it proves not to be in practice. The first reason is that, unlike the case with homogeneous data structures, one rarely wants binary methods when dealing with heterogeneous data structures. In the heterogeneous list example in Appendix A, for example, the *gt* and *eq* methods both take parameters of type *#rect_type*. It would not make sense for them to take parameters of type *MyType* or *#MyType* since then you might not be able to compare two elements of the list with different types, even though both match *rect_type*.

A second reason for not worrying about this limitation is that if the base type had binary methods which were used in the data structure, we could not obtain heterogeneous data structures at all in the original language with subtyping. A good example of this is an ordered list, which depends on the base type have binary methods for the ordering. As a result we still have more flexibility in this new system. We just have to be more careful with binary methods than with those with no contravariant occurrences of *MyType*.

Finally, in the case where binary methods are needed, programmers will need to use explicit bounded matching rather than #-types. This way the programmer can ensure that the parameters are of the same type as the receiver. While this is a little less compact than the # types, it still gives the same flavor.

11.4 Evaluation

A possible disadvantage of the system described above is that we do not allow the types of methods to be explicitly changed in subclasses. (As before, they implicitly change because of occurrences of *MyType*.) While we believe that the constructs included in this language do provide sufficient functionality for most purposes, it is possible to generalize the definition of matching in order to support the change of methods in subclasses. In order to do this it is necessary to define a matching relation on function types, since the types of methods are functions.

This can be done in a way that provides more flexibility in defining subclasses, but we remain unconvinced that this provides enough benefit to be worth the resulting complexity. As with the examples provided earlier in this paper, most examples we have looked at can be handled easily with a combination of the simple matching presented in this section, *MyType*, and match-bounded polymorphism.

Another possible disadvantage of this language design is that it forces the programmer to mark those parameters and variables in a program that may later need to take values whose types only match the given types. This requirement to plan ahead is indeed a negative point in this design. However, it is worth pointing out that, despite the hype to the contrary, it already takes great care to design classes that can be successfully inherited from. This helps explain the common wisdom that good class libraries are more helpful in code reuse than libraries in traditional imperative languages, *but* good class libraries are much harder to design than libraries in traditional languages. Thus we would argue that good object-oriented design currently takes significant planning ahead for change on the part of the programmer. As a final justification, we note that the designers of Ada 95 made a similar decision to require the programmer to mark those variables and parameters which are allowed to hold values whose types are subtypes of a given type.

In summation, the language described in this section is considerably simpler than the language presented in the earlier sections of this paper. In spite of the elimination of subtyping from the language, the use of #-types and bounded matching seems to provide a language significantly more expressive than the object-oriented languages in common use today. In particular, all of the programs discussed earlier in this paper can be statically typed in this new language. *None* of the final programs presented took advantage of subtyping in any way!

Leaf Petersen and the author have designed and implemented a language, LOOM, which corresponds to the design given in this section. A technical report is currently being written which will describe the language in more detail. The paper [Pet96] presents an overview of the language and describes a module system that provides greater support for programming in the large. Our use of LOOM in working out many examples leads us to believe that it combines an appealing simplicity with sufficient expressive power to model most situations likely to arise in practice.

12 Conclusions and related work

In this paper we sketched several useful modifications to the simple type-checking systems of object-oriented languages like C++, Java, and Object Pascal. These included allowing method types to be replaced by subtypes in subclasses, introducing a name, *MyType*, for the type of *self* – supporting automatic updating of parameter types in important special cases, and introducing match-bounded polymorphism as a means of expressing the desired restrictions on type parameters. These mechanisms provided us with a more expressive type system that allowed us to write a large number of subclasses that, while intuitively correct, were ruled out by the simple type system. Moreover, one can actually prove that this system is type safe, meaning that any program which type checks is guaranteed not to have run-time type errors. A more formal description of such a language is given in [BSvG95].

We believe that these constructs should become increasingly common in new or revised versions of statically-typed object-oriented programming languages. For instance, we have formulated a proposal to add *MyType* and match-bounded polymorphism to Java.

In the last section of this paper, we introduced a more recently developed type system that uses matching to replace subtyping. In this system subtyping (and the subsumption rule) were dropped, and the original version of matching was replaced by a much simpler relation in which methods could only be added in matching types, with no (explicit) changes allowed to the existing method types. (Of course the method bodies themselves could be overridden in subclasses; it is only the types which are frozen.) While very simple, this type system, which also included match-bounded polymorphism, allowed us to easily express all of the difficult examples included in this paper that caused problems in the simple type system.

Our current personal choice for a type system for statically typing object-oriented languages is this pure matching language. We believe that it combines simplicity with the expressiveness to solve many of the most difficult typing problems that arise in object-oriented languages. The #-types in this language also allow the programmer to make an explicit choice between homogeneous and heterogeneous data structures, something generally not available in languages in which subtyping cannot be turned off.

The only negatives that arise in this language's use are the result of tradeoffs that may be worth taking on other grounds. While it is necessary to mark those formal parameters and variables that are capable of holding values of matching types, this provides information to the compiler that may be useful in producing optimized code. In particular, this may result in paying the extra implementation costs of subtyping only in those places where it is actually needed. It does, however, require the programmer to plan ahead as to where this kind of flexibility is desired. A restriction is that binary messages cannot be sent to objects whose only known types are #-types.

The contents of this paper are based on the work of researchers working in type theory and semantics as well as language designers. In the body of this paper we attempted to provide fairly complete references to the ideas discussed here, but there were many important contributions that we did not have room to discuss. The paper [DT88] presents an interesting survey of the state of type theory in object-oriented languages as of 1987-88. The collection [GM94] contains many of the important early contributions to the theory of object-oriented programming languages. The forthcoming book [AC96] provides a more unified approach to the semantics and type theory of object-oriented programming languages. Other important sources of information in this area are the proceedings of the annual ECOOP (European Conference on Object-Oriented Programming) and OOPSLA (Object-Oriented Programming: Systems, Languages, and Applications) conferences. The Journal of Function Programming published a special issue in 1994 on the theory of object-oriented languages which contains the papers [Aba94, Bru94, PT94], while the journal Theory and Practice of Object-oriented Systems (TAPOS) will publish a special issue on types in object-oriented languages in late 1996.

Acknowledgements: Thanks to Luca Cardelli, Peter Wegner, Tony Simons and the anonymous referees for very helpful comments and suggestions on earlier drafts of this paper. Andrew Black made detailed comments on successive drafts which greatly improved the exposition. Special thanks to Robert van Gent, Angela Schuett, and Leaf Petersen, whose collaborative efforts in language design led to a better understanding of the issues of static typing in object-oriented languages. This paper was written in part while the author was in residence at the Newton Institute for Mathematical Science at the University of Cambridge. I thank the institute and its staff for their hospitality and support during my stay.

References

[Aba94] Martin Abadi. Baby Modula-3 and a theory of objects. *Journal of functional programming*, 4:249–283, 1994.

[AC93] Roberto Amadio and Luca Cardelli. Subtyping recursive types. *ACM Transactions on Programming Languages and Systems*, 15(4):575–631, 1993.

[AC95] Martin Abadi and Luca Cardelli. On subtyping and matching. In *Proceedings ECOOP '95*, pages 145–167, 1995.

[AC96] Martin Abadi and Luca Cardelli. *A Theory of Objects*. Springer-Verlag, 1996. to appear.

[AG96] Ken Arnold and James Gosling. *Java*. Addison Wesley, 1996.

[BCC+96] Kim B. Bruce, Luca Cardelli, Giuseppe Castagna, The Hopkins Objects Group, Gary T. Leavens, and Benjamin Pierce. On binary methods. *Theory and Practice of Object-Oriented Systems*, 1996. to appear.

[BDMN73] G.M. Birtwistle, O.-J. Dahl, B. Myhrhaug, and K. Nygaard. *SIMULA Begin*. Aurbach, 1973.

[BH91] A. Black and N. Hutchinson. Typechecking polymorphism in Emerald. Technical Report CRL 91/1 (Revised), DEC Cambridge Research Lab, 1991.

[BHJ+87] A. P. Black, N. Hutchinson, E. Jul, H. M. Levy, and L. Carter. Distribution and abstract types in Emerald. *IEEE Transactions on Software Engineering*, SE-13(1):65–76, 1987.

[BHJL86] A. Black, N. Hutchinson, E. Jul, and H. Levy. Object structure in the Emerald system. In *Proc. ACM Symp. on Object-Oriented Programming: Systems, Languages, and Applications*, pages 78–86, October 1986.

[BP96] Kim B. Bruce and Leaf Petersen. Subtyping is not a good "match" for object-oriented languages. Technical report, Williams College, 1996. to appear.

[Bru94] K. Bruce. A paradigmatic object-oriented programming language: design, static typing and semantics. *Journal of Functional Programming*, 4(2):127–206, 1994. An earlier version of this paper appeared in the 1993 POPL Proceedings.

[BSvG95] Kim B. Bruce, Angela Schuett, and Robert van Gent. PolyTOIL: A type-safe polymorphic object-oriented language, extended abstract. In *ECOOP '95*, pages 27–51. LNCS 952, Springer-Verlag, 1995. A complete version of this paper with full proofs is available via http://www.cs.williams.edu/~kim/.

[Car88a] L. Cardelli. A semantics of multiple inheritance. *Information and Computation*, 76:138–164, 1988. Special issue devoted to *Symp. on Semantics of Data Types,* Sophia-Antipolis (France), 1984.

[Car88b] L. Cardelli. Structural subtyping and the notion of powertype. In *Proc 15th ACM Symp. Principles of Programming Languages*, pages 70–79, 1988.

[CCH+89] P. Canning, W. Cook, W. Hill, J. Mitchell, and W. Olthoff. F-bounded quantification for object-oriented programming. In *Functional Prog. and Computer Architecture*, pages 273–280, 1989.

[CDG+88] L. Cardelli, J. Donahue, L. Galssman, M. Jordan, B. Kalsow, and G. Nelson. Modula-3 report. Technical Report SRC-31, DEC systems Research Center, 1988.

[CHC90] William R. Cook, Walter L. Hill, and Peter S. Canning. Inheritance is not subtyping. In *Proc. 17th ACM Symp. on Principles of Programming Languages*, pages 125–135, January 1990.

[Coo89] W.R. Cook. A proposal for making Eiffel type-safe. In *European Conf. on Object-Oriented Programming*, pages 57–72, 1989.

[CP89] W. Cook and J. Palsberg. A denotational semantics of inheritance and its correctness. In *Proc. ACM Conf. on Object-Oriented Programming: Systems, Languages and Applications*, pages 433–444, 1989.

[CW85] L. Cardelli and P. Wegner. On understanding types, data abstraction, and polymorphism. *Computing Surveys*, 17(4):471–522, 1985.

[DGLM94] Mark Day, Robert Gruber, Barbara Liskov, and Andrew C. Meyers. Abstraction mechanisms in Theta. Technical report, MIT Laboratory for Computer Science, 1994.

[DGLM95] Mark Day, Robert Gruber, Barbara Liskov, and Andrew C. Meyers. Subtypes vs. where clauses: Constraining parametric polymorphism. In *Proc. ACM Symp. on Object-Oriented Programming: Systems, Languages, and Applications*, pages 156–168, 1995.

[DT88] S. Danforth and C. Tomlinson. Type theories and object-oriented programming. *ACM Computing Surveys*, 20(1):29–72, 1988.

[GM94] Carl A. Gunter and John C. Mitchell. *Theoretical Aspects of Object-Oriented Programming.* MIT Press, Cambridge, MA, 1994.

[GR83] A. Goldberg and D. Robson. *Smalltalk–80: The language and its implementation.* Addison Wesley, 1983.

[HJW92] Paul Hudak, S. Peyton Jones, and Philip Wadler. Report on the programming language Haskell, a non-strict purely functional language (version 1.2). *SIGPLAN Notices*, 27(5), May 1992.

[HMM86] R. Harper, D.B. MacQueen, and R. Milner. Standard ML. Technical Report ECS–LFCS–86–2, Lab. for Foundations of Computer Science, University of Edinburgh, March 1986.

[Hut87] N. Hutchinson. *Emerald: An object-oriented language for distributed programming.* PhD thesis, University of Washington, 1987.

[Int95] Intermetrics. *Ada 95 Reference Manual, version 6.0.* 1995.

[KMMPN87] Bent Bruun Kristensen, Ole Lehrmann Madsen, Birger Moller-Pedersen, and Kristen Nygaard. The Beta programming language. In Bruce Shriver and Peter Wegner, editors, *Research Directions in Object-Oriented Programming*, pages 7–48. M.I.T. Press, Cambridge, MA, 1987.

[L+81] B. Liskov et al. *CLU Reference Manual*, volume 114 of *Lecture Notes in Computer Science.* Springer-Verlag, 1981.

[Mey88] B. Meyer. *Object-Oriented Software Construction.* Prentice-Hall, 1988.

[Mey92] B. Meyer. *Eiffel: the language.* Prentice-Hall, 1992.

[Mey95] Bertrand Meyer. Static typing. *OOPS Messenger*, 6(4):20–29, 1995. Text of an OOPSLA '95 address.

[Mit90] J.C. Mitchell. Toward a typed foundation for method specialization and inheritance. In *Proc. 17th ACM Symp. on Principles of Programming Languages*, pages 109–124, January 1990.

[Pet96] Leaf Petersen. *A module system for LOOM.* Williams College Senior Honors Thesis, 1996.

[PS95] Benjamin Pierce and Martin Steffen. Higher-order subtyping. *Theoretical Computer Science*, 1995. To appear. A preliminary version appeared in IFIP Working Conference on Programming Concepts, Methods and Calculi (PROCOMET), June 1994, and as University of Edinburgh technical report ECS-LFCS-94-280 and Universität Erlangen-Nürnberg Interner Bericht IMMD7-01/94, January 1994.

[PT94] Benjamin C. Pierce and David N. Turner. Simple type-theoretic foundations for object-oriented programming. *Journal of functional programming*, 4:207–247, 1994. An earlier version appeared in Proc. of POPL '93, pp. 299-312.

[Red88] U.S. Reddy. Objects as closures: Abstract semantics of object-oriented languages. In *Proc. ACM Symp. Lisp and Functional Programming Languages*, pages 289–297, July 1988.

[RIR93] N. Rodriguez, R. Ierusalimschy, and J. L. Rangel. Types in School. *SIGPLAN Notices*, 28(8), 1993.

[SCB+86] C. Schaffert, T. Cooper, B. Bullis, M. Kilian, and C. Wilpolt. An introduction to Trellis/Owl. In *OOPSLA '86 Proceedings*, pages 9–16. ACM SIGPLAN Notices,21(11), November 1986.

[Sha91] David Shang. Type-safe reuse of prototype software. In *Proc. 3rd Software Engineering and Knowledge Engineering International Conference*, 1991.

[Str86] B. Stroustrup. *The C^{++} Programming Language*. Addison-Wesley, 1986.

[Tes85] L. Tesler. Object Pascal report. Technical Report 1, Apple Computer, 1985.

[US 80] US Dept. of Defense. *Reference Manual for the Ada Programming Language*. GPO 008-000-00354-8, 1980.

[Wir71] Niklaus Wirth. The programming language Pascal. *Acta Informatica*, 1:35–63, 1971.

[Wir85] Niklaus Wirth. *Programming in Modula-2, 3rd edition*. Springer-Verlag, 1985.

A Example linked list program using matching

The following is a sample program which uses a static type system similar to that presented in section 11. The program defines and uses heterogeous linked lists of elements, each element of which must have a type matching *rect_type*. Because *OrdList_class* is polymorphic in the type of the nodes in the list, we can easily instantiate the lists to be either singly or doubly-linked.

We have left out inessential portions of the code the program in order to conserve space. The keyword *include* used in the definition of *DbleNode_type* simply indicates that all methods declared in *Node_type* are also included in *DbleNode_type*.

```
program heterogeneous_linked_lists;
-- Program developing heterogeneous singly- and doubly-linked lists.
-- Elements of the linked list must be of type which matches rect_type.

type

  rect_type = ObjectType
       gt: func(#rect_type):bool;
       eq: func(#rect_type):bool;
```

```
      get_height: func():integer;
      draw: proc();
    end ObjectType;

  Node_type = ObjectType
       get_next: func():mytype;
       get_val: func():#rect_type;
       set_next: proc(mytype);
       set_val: proc(#rect_type);
       attach_right: proc(mytype)
    end ObjectType;

  DbleNode_type = ObjectType include Node_type
       get_prev:func():mytype;
       set_prev:proc(mytype);
    end ObjectType;

  TypeFunction OrdList_type(U <# Node_type) = ObjectType
       find: func(#rect_type):bool;
       add: proc(U);
       drawall: proc();
    end ObjectType;

classes

  class gen_rect_class(tp,lft,bot,rght,newz: integer)
    var
      top = tp: integer;
      left = lft: integer;
      bottom = bot: integer;
      right = rght: integer;
      z = newz: integer;
    methods
      function gt(other: #rect_type): bool
        begin
          return (z > other.get_height())
        end;
      function eq(other: #rect_type): bool
        ...
      function get_height(): integer
        ...
      procedure draw
        ...
    end class;
```

```
class Node_class(v: #rect_type)
  var
     val = v: #rect_type;
     next = nil: mytype;
  methods
     function get_next(): mytype
       begin
          return next
       end;
     function get_val(): #rect_type
       begin
          return val
       end;
     procedure set_next(nxt:mytype)
       begin
          next := nxt
       end;
     procedure set_val(vl: #rect_type)
       begin
          val := vl
       end;
     procedure attach_right = procedure(newNext: mytype)
       begin
          self.set_next(newNext)
       end;
  end class;

class DbleNode_class(v: #rect_type)
         inherits Node_class(v) modifying attach_right
  var
     prev = nil: MyType
  methods
     function getPrev():MyType
       ...
     procedure setPrev(newPrev: MyType)
       ...
     procedure attachRight(newNext: MyType)
       begin
          self.setNext(newNext);
          newNext.setPrev(self)
       end
end class;

class OrdList_class(U <# Node_type)
  var
```

```
   head = nil: U;
methods
   function find(match:#rect_type): bool
     var
       current: U;
     begin
       current := head;
       while (current != nil) & match.gt(current.get_val()) do
         current := current.get_next()
       end;
       if (current != nil) & (current.get_val()).eq(match)) then
         return true
       else
         return false
     end;

   procedure add(new_node:U)
    var
      prev: U;
      current: U;
    begin
      if head = nil then
        head := new_node;
        new_node.set_next(nil);
      else if head.get_val().gt(new_node.get_val()) then
              new_node.attach_right(head);
              head := new_node;
            else
              prev := head;
              current := head.get_next();
              while (current != nil) &
                  current.get_val().gt(new_node.get_val()) do
                prev := current;
                current := current.get_next();
              end;
              if current = nil then
                prev.attach_right(new_node);
                new_node.set_next(nil);
              else
                new_node.attach_right(current);
                prev.attach_right(new_node);
              end;
            end;
       end;
     end;
```

```
        drawall = procedure()
          var
            current: U;
            cur_val: #rect_type;
          begin
            current := head;
            while (current != nil) do
              cur_val := current.get_val();
              cur_val.draw();
              current := current.get_next();
            end;
          end;
        end;
    end class;

var
    temp_rect: rect_type;
    shape: #rect_type;
    lnode: Node_type;
    dnode: DbleNode_type
    some_node: #Node_type;
    slist: #OrdList_type[Node_type];
    dlist: OrdList_type[DbleNode_type];

begin -- main program
    temp_rect := new(gen_rect_class(1,1,4,4,5));
    lnode := new(Node_class(temp_rect));
    slist := new(OrdList_class(Node_type));
    dnode := new(DbleNode_class(temp_rect));
    dlist := new(OrdList_class(DbleNode_type));
    slist.add(lnode.clone());
 -- illegal: slist.add(dnode.clone());
 --     Can't add a doubly-linked node to a singly-linked list.
 -- illegal: slist := dlist
 --     OrdList_type[DbleNode_type] does not match OrdList_type[Node_type].
    temp_rect := new(gen_rect_class (2,2,3,3,2));
    lnode.set_val(temp_rect);
    slist.add(lnode);
    some_node := dnode.get_next();
    lnode := lnode.get_next();
    shape := lnode.get_val();
 -- illegal: temp_rect := lnode.get_val();
 --     Result of get_val() has type #rect_type.
    lnode.setval(shape)
```

```
   PrintNum(shape.get_height());
end.
```

The two new type-checking rules for #-types introduced in section 11 are necessary to type check this program. The first states that if an expression e has type T, then e also has type $\#T$. The second states that if e has type $\#T$ and $T <\# U$, then e also has type $\#U$. The first of these rules allows the assignment to *slist* and the use of *temp_rect* as a parameter in the message send *lnode.set_val*(*temp_rect*). Both of these rules are used in type checking the assignment to *some_node*.

The program is written with a syntax and style similar to that of the language LOOM [BP96]. LOOM differs from the above in a few syntactic details. It also supports the use of classes as first-class values (*e.g.*, classes can be returned as values from functions), provides finer control over the visibility of methods, and includes a module system for programming in the large. A description of the module facilities can be found in [Pet96].