Inheritance Is Not Subtyping

William R. Cook Walter L. Hill Peter S. Canning
Hewlett-Packard Laboratories
P.O. Box 10490 Palo Alto CA 94303-0969

Abstract

In typed object-oriented languages the subtype relation is typically based on the inheritance hierarchy. This approach, however, leads either to insecure type-systems or to restrictions on inheritance that make it less flexible than untyped Smalltalk inheritance. We present a new typed model of inheritance that allows more of the flexibility of Smalltalk inheritance within a statically-typed system. Significant features of our analysis are the introduction of polymorphism into the typing of inheritance and the uniform application of inheritance to objects, classes and types. The resulting notion of *type inheritance* allows us to show that the type of an inherited object is an inherited type but not always a subtype.

1 Introduction

In strongly-typed object-oriented languages like Simula [1], C++ [28], Trellis [25], Eiffel [19], and Modula-3 [9], the inheritance hierarchy determines the conformance (subtype) relation. In most such languages, inheritance is restricted to satisfy the requirements of subtyping. Eiffel, on the other hand, has a more expressive type system that allows more of the flexibility of Smalltalk inheritance [14], but suffers from type insecurities because its inheritance construct is not a sound basis for a subtype relation [12].

In this paper we present a new typed model of inheritance that supports more of the flexibility of Smalltalk inheritance while allowing static type-checking. The typing is based on an extended polymorphic lambda-calculus and a denotational model of inheritance. The model contradicts the conventional wisdom that inheritance must always make subtypes. In other words, we show that incremental change, by implementation inheritance, can produce objects that are not subtype compatible with the original objects. We introduce the notion of *type inheritance* and show that an inherited object has an inherited type. Type inheritance is the basis for a new form of polymorphism for object-oriented programming.

Much of the work presented here is connected with the use of self-reference, or recursion, in object-oriented languages [3, 4, 5]. Our model of inheritance is intimately tied to recursion in that it is a mechanism for incremental extension of recursive structures [11, 13, 22]. In object-oriented languages, recursion is used at three levels: objects, classes, and types. We apply inheritance uniformly to each of these forms of recursion while ensuring that each form interacts properly with the others. Since our terminology is based on this uniform development, it is sometimes at odds with the numerous technical terms used in the object-oriented paradigm. Our notion of object inheritance subsumes both delegation and the traditional notion of class inheritance, while our notion of class inheritance is related to Smalltalk metaclasses.

Object inheritance is used to construct objects incrementally. We show that when a recursive object definition is inherited to define a new object, a corresponding change is often required in the type of the object. To achieve this effect, polymorphism is introduced into recursive object definitions by abstracting the type of *self*. Inheritance is defined to specialize the inherited definition to match the type of the new object being defined. A form of polymorphism developed for this purpose, called F-bounded polymorphism [3], is used to characterize the extended types that may be created by inheritors.

Class inheritance supports the incremental definition of classes, which are parameterized object definitions. A class is recursive if its instances use the class to create new instances. When a class is inherited to define a new class, the inherited creation operations are updated to create instances of the new class. Since class recursion is also related to recursion in the object types, the polymorphic typing of inheritance is extended to cover class recursion. We also introduce a generalization of class inheritance that allows modification of instantiation parameters.

A final application of inheritance is to the definition of recursive types. Type inheritance extends a recursive record type to make a new type with similar recursive structure but more fields. Because of an interaction between function subtyping and recursion, an inherited type is not necessarily a subtype of the type from which it was derived. This is a second sense in which inheritance is not subtyping. Type inheritance is useful for constructing the types of objects produced by object inheritance. In addition, F-bounded polymorphic functions can be applied to the types of objects that inherit from a given object definition and then to objects of that type; objects with different inherited types are prevented from being mixed together.

The typed model of inheritance is directly relevant to the analysis and design of programming languages. It indicates how object-oriented languages could be extended to support more flexible forms of inheritance while retaining static typing. Several other theories of typed inheritance have also been proposed [15, 20, 30]. A preliminary comparison reveals a similarity of approach. However, the models are based on a wide range of theoretical foundations, and more research is required to resolve the differences.

The next section surveys terminology and background on objects, types, and inheritance for our model of object-oriented programming. The following three sections are organized to address the use of inheritance for the three kinds of recursion found in object-oriented programs. Section 3 examines the relationship between object recursion and type recursion, and introduces a polymorphic typing of inheritance to allow more flexibility in the presence of recursive types. Section 4 introduces recursive classes with instantiation parameters and extends inheritance to allow modification of parameters by subclasses. Section 5 defines a notion of inheritance for types and demonstrates its connection to object inheritance. Section 6 illustrates these features with a practical programming example. Section 7 applies the model to the analysis of object-oriented languages and compares it to other models of typed inheritance.

2 Background

2.1 A Typed Record Calculus

A typed polymorphic lambda-calculus with records is used to describe the typing of inheritance. The language is functional and explicitly typed; no provision is made for imperative constructs or type inference. Imperative constructs, which support mutable object state, are not included because they do not affect the analysis of inheritance. An untyped version of the calculus is used to introduce new constructs before giving a typed presentation.

2.2 Records

A record is a finite mapping of labels to values. A record with fields $x_1, \ldots, x_n$ and associated values $v_1, \ldots, v_n$ is written $\{x_1 = v_1, \ldots, x_n = v_n\}$. If the values have type $\sigma_1, \ldots, \sigma_n$ then the record has type $\{x_1 : \sigma_1, \ldots, x_n : \sigma_n\}$. Selecting the a component of a record r is given by $r.a$.

Cardelli [6, 8] identified record subtyping as an important form of polymorphism in object-oriented programming. The main idea is that if a record r has fields $x_1 : \sigma_1, \ldots, x_k : \sigma_k$ and also $x_{k+1} : \sigma_{k+1}, \ldots, x_\ell : \sigma_\ell$, then a record r' with fields $x_1 : \sigma_1, \ldots, x_k : \sigma_k$ can be constructed from r by omitting fields. Therefore, any record with type $\{x_1 : \sigma_1, \ldots, x_k : \sigma_k, \ldots, x_\ell : \sigma_\ell\}$ can be coerced into a record of type $\{x_1 : \sigma_1, \ldots, x_k : \sigma_k\}$. The general form of this coercion allows the field values to be coerced as well:

$$\frac{\sigma_1 \leq \rho_1, \ldots, \sigma_k \leq \rho_k,}{\{x_1 : \sigma_1, \ldots, x_k : \sigma_k, \ldots, x_\ell : \sigma_\ell\} \leq \{x_1 : \rho_1, \ldots, x_k : \rho_k\}}$$

In our model, record types indicate exactly what fields a record contains. This differs from Cardelli, who uses a *subsumption* model in which a record type represents all records that have at least the specified fields. We do not use subsumption because it complicates the problem of record combination.

2.3 Record Combination

The language supports a simple record combination operator, **with**, that joins two records. The typing of **with** is defined by a typed introduction rule.

$$\frac{\begin{array}{c} e_1 : \{x_1 : \sigma_1, \ldots, x_j : \sigma_j, x_{j+1} : \tau_1, \ldots, x_k : \tau_{k-j}\} \\ e_2 : \{x_{j+1} : \sigma_{j+1}, \ldots, x_n : \sigma_n\} \qquad (k \leq n) \end{array}}{e_1 \textbf{ with } e_2 : \{x_1 : \sigma_1, \ldots, x_n : \sigma_n\}}$$

If there are common fields, $x_{j+1}, \ldots, x_k$, they may have different types in the two records. The conflict is resolved by taking the value from e_2. An analogous operator, $+$, is defined on record types. The evaluation rule performs the corresponding operation on record values. The operator **with** is well-behaved because our types are exact specifications of the fields in a record. If subsumption were allowed, the actual value of e_2 could have more fields than mentioned in its type. According to the evaluation rule, these fields would take precedence over the fields in e_1 resulting in a unsound typing.

Our record combination operator is simpler than the ones proposed by Rémy [23] and Wand [29, 30]. The simply typed version of **with** is sufficient for the analysis in this paper.

2.4 Recursive Types

The notation for a recursive type defined by $T = F[T]$ is $\mu t.F[t]$. A recursive type is equal to its infinite expansion. One step in this expansion is given by the *unrolling* rule:

$$\mu t.F[t] = F[\mu t.F[t]]$$

One recursive subtype is a subtype of another if their infinite expansions are in a subtype relation. An induction rule is used to specify the subtype relation [7].

$$\frac{\Gamma,\ s \leq t\ \vdash\ \sigma[s] \leq \tau[t]}{\Gamma\ \vdash\ \mu s.\ \sigma[s] \leq \mu t.\ \tau[t]}$$

To illustrate, the types T_1 and T_2 are subtypes of T. These subtypes do not have the same pattern of recursion as T. T_3 is not a subtype of T, even though it has the same recursive structure, because of the contravariance [6] of the function type in the b field.

$$\begin{aligned}
T &= \mu t.\{a : int, c : t, b : t \rightarrow t\} \\
T_1 &= \{a : int, c : T, b : T \rightarrow T, d : bool\} \\
T_2 &= \mu t.\{a : int, c : t, b : T \rightarrow t, d : bool\} \\
T_3 &= \mu t.\{a : int, c : t, b : t \rightarrow t, d : bool\}
\end{aligned}$$

2.5 Polymorphism

Subtype-bounded polymorphism [10] allows functions to be written that operate uniformly over all subtypes of a given type. A bounded polymorphic function is defined by the expression $\Lambda\, t \leq \sigma\, .\, e$. If assuming $t \leq \sigma$ gives e type τ, then the polymorphic function has type $\forall t \leq \sigma.\tau$. For recursive types, however, there are forms of structural similarity not captured by subtyping, as illustrated above. The types that have the *recursive structure* of $T = \mu t.F[t]$ are those that satisfy the constraint $t \leq F[t]$. In the example above, $F[t] = \{a : int, c : t, b : t \rightarrow t\}$. Of types T_1, T_2 and T_3, only T_3 satisfies the constraint $T_3 \leq F[T_3]$. *F-bounded polymorphism* [3], written $\forall t \leq F[t].\sigma$, supports parametric quantification over the recursive types that share the recursive structure specified by F. Examples demonstrating the use of F-bounded polymorphism in typing functions involving recursive types are given in [3].

2.6 Objects

As in [5, 11, 22, 31], we represent objects as records whose fields contain methods. The methods of an object may refer to each other, so objects are naturally viewed as mutually recursive definitions. The traditional interpretation of mutual recursion from denotational semantics is as a fixed point of a function on environments of identifiers. For example, an object handling messages $m_1, \ldots, m_j$ by methods $e_1, \ldots, e_j$ is a fixed point of the function P.

$$P = \lambda(self).\ \{m_1 = e_1,\ \ldots,\ m_j = e_j\}. \tag{1}$$

The expressions $e_1, \ldots, e_j$ may contain references to *self*; for example, to call the m method with argument 3 one would write *self.m*(3). The function P is a definition of the object $\mathbf{Y}(P)$. The type of an object is a record type, and is often a recursive record type [2, 3, 4, 5, 24]. Recursion in object types is associated, for example, with a method that simply returns the pseudovariable *self*.

While our language does not support mutable object state, lexically bound variables can be used to parameterize objects (see Section 4.1). If mutable variables were supported, any pattern of shared state among different objects could be defined, including those characteristic of delegation systems [18]. To simplify the presentation, some methods are simple values instead of functions; if state were introduced then lambda-abstractions would be required on all methods to delay evaluation.

2.7 Inheritance

An untyped, compositional model of inheritance based on the fixed-point function, $\mathbf{Y}$, was developed independently by Cook [11, 13] and Reddy [22]. Given a definition, P, of a *parent* recursive value $\mathbf{Y}(P)$ and a self-referential modifier M, one may construct a *child* value $\mathbf{Y}(\lambda(s).\,M(s)(P(s)))$. This example illustrates inheritance as an operation on recursive definitions.

Inheritance allows a new object definition to be derived from an existing one where self-reference in the inherited object definition is unified with self-reference in new methods.

$$\begin{aligned} C = \lambda(self).\ P(self)\ &\mathbf{with} \\ \{m'_1 = e'_1,\ \ldots,\ &m'_k = e'_k\} \end{aligned} \tag{2}$$

The use of the pseudovariable *super* to refer directly to parent methods is also supported in the model. This generalization is illustrated in Section 6.

A simple typing of inheritance is easily derived by adding type-constraints to the basic inheritance model [11]. If P has type $\sigma \rightarrow \sigma$ for some σ, then the object $\mathbf{Y}(P)$ has type σ. If the modifier has type $\tau \rightarrow \sigma \rightarrow \tau$ to produce an object of type τ from one of type σ, then $\lambda(s).\,M(s)(P(s))$ has type $\tau \rightarrow \tau$ and its fixed point is an object of type τ. The significant constraint introduced by inheritance is that τ must be a subtype of σ for the application $P(s)$ to be type-correct.

3 Object Inheritance

3.1 Problems in the Simple Typing of Inheritance

In this section we illustrate some problems in using the simple typing of inheritance to define objects with recursive types. One problem arises because the simple typing of inheritance does not always provide the most precise type possible. Consider a simple object definition with a method *i* returning the value 5 and a method *id* that returns the object itself.

$$P = \lambda(\mathit{self}).\ \{i = 5, \mathit{id} = \mathit{self}\}$$

The object $\mathbf{Y}(P)$ has type $\sigma = \mu t.\{i : \mathit{int}, \mathit{id} : t\}$, thus the simple typing of inheritance gives P the type $\sigma \rightarrow \sigma$. A child C is defined by inheriting P and adding a single boolean field b.

$$C = \lambda(\mathit{self}).\, P(\mathit{self})\ \mathbf{with}\ \{b = \mathit{true}\}$$

In the simple typing of inheritance, $\mathbf{Y}(C)$ has type $\sigma + \{b : \mathit{bool}\}$, or $\tau_1 = \{i : \mathit{int}, \mathit{id} : \sigma, b : \mathit{bool}\}$. Note that τ_1 is not directly recursive in the type of the *id* method.

Expanding C to eliminate $P(\mathit{self})$ and combine records gives an equivalent expression.

$$C_1 = \lambda(\mathit{self}).\ \{i = 5, \mathit{id} = \mathit{self}, b = \mathit{true}\}$$

$\mathbf{Y}(C_1)$ has type $\tau_2 = \mu t.\{i : \mathit{int}, \mathit{id} : t, b : \mathit{bool}\}$. Since $\tau_2 < \tau_1$ the simple typing has resulted in less precise type for the inherited object than is possible.

A more serious problem occurs when attempting to use contravariant [6] recursive types. Consider a new object definition with an equality method instead of an identity method.

$$P' = \lambda(\mathit{self}).\ \{i = 5, \mathit{eq} = \lambda(o).(o.i = \mathit{self}.i)\}$$

Although several typings for $\mathbf{Y}(P')$ are possible, the one that expresses *eq* as a binary method is recursive in the type of the *eq* method.

$$\sigma' = \mu t.\{i : \mathit{int}, \mathit{eq} : t \rightarrow \mathit{bool}\}$$

Now C' can be defined, by inheriting P' while adding a b field and redefining *eq*.

$$\begin{aligned} C' = \lambda(\mathit{self}).\, & P'(\mathit{self})\ \mathbf{with} \\ \{\ & b = \mathit{true}, \\ & \mathit{eq} = \lambda(o).(o.i = \mathit{self}.i \text{ and } \\ & \qquad\qquad o.b = \mathit{self}.b) \\ \}& \end{aligned}$$

Before examining possible typings for C, consider the object it defines. The object $\mathbf{Y}(C')$ has a recursive type, τ_2'.

$$\tau_2' = \mu t.\{i : \mathit{int}, b : \mathit{bool}, \mathit{eq} : t \rightarrow \mathit{bool}\}$$

The simple typing of C' fails because $\tau_2' \not\leq \sigma'$. Simple types cannot be assigned to the definition of C' because P' of type $\sigma' \rightarrow \sigma'$ cannot be applied to *self* which has type τ_2' as required by the inheritance. It is important to note that in an untyped framework this use of inheritance is meaningful, but the type system is simply not expressive enough to describe the relevant constraints. The simple typing of inheritance, with its subtype assumption, cannot give a typing for this example.

3.2 Polymorphism and Inheritance

To overcome these problems we introduce polymorphism directly into the mechanism of inheritance. This is motivated by observing the type-dependency within a recursive definition: the type of object created depends on the type of *self*. We provide a more flexible typing by abstracting the type of *self* and replacing type recursion by type-dependency. The type recursion is reintroduced when the object is constructed. Let $F[t] = \{m_1 : \sigma_1, \ldots, m_j : \sigma_j\}$ be a type function defining a recursive type $\sigma = \mu t.F[t]$. An object with methods e_i of type σ_i is defined by an expression in which the type of *self* is polymorphic.

$$\begin{aligned} & P : \forall t \leq F[t].t \rightarrow F[t] \\ & P = \Lambda\, t \leq F[t]\,.\, \lambda(\mathit{self}: t). \\ & \quad \{m_1 = e_1, \ldots, m_j = e_j\} \end{aligned} \tag{3}$$

The F-bounded constraint $t \leq F[t]$ is central to the model. It provides information about the methods defined by the object denoted by *self*. For example, if $F[t] = \{m : t \rightarrow t, n : t\}$, the n method could return *self.m*(*self*). Of course, the exact type t is unknown — it is supplied by inheritors to indicate the type of the complete object into which the methods in P are being incorporated.

Object instantiation must now include the type of the object being created, as in $\mathbf{Y}(P[\sigma])$. The simple typing is recovered by forming $P[\sigma] : \sigma \rightarrow F[\sigma]$, which, by unrolling, is equal to $\sigma \rightarrow \sigma$.

For inheritance, it is necessary to define a type $\tau = \mu s.G[s]$ such that $G[t] \leq F[t]$. Any type satisfying $t \leq G[t]$ also satisfies $t \leq F[t]$.

The typing of inheritance involves defining a new polymorphic function that specializes its parent to the appropriate type before modifying its methods.

$$\begin{aligned} & C : \forall t \leq G[t].t \rightarrow G[t] \\ & C = \Lambda\, t \leq G[t]\,.\, \lambda(\mathit{self}: t). \\ & \quad P[t](\mathit{self})\ \mathbf{with}\ \{m_1' = e_1', \ldots, m_k' = e_k'\} \end{aligned} \tag{4}$$

The fields m_1' must be assigned values as specified in $G[t]$. The use of $P[t]$ is type-correct because $G[t] \leq F[t]$. The simple record combination operator, **with**, is sufficient because it is applied to values whose types are constant. Although it might seem reasonable to abstract over P to produce an ***abstract subclass***, or ***wrapper*** [21], the resulting function cannot be assigned a useful type without a more expressive record combination operator.

To illustrate the polymorphic typing of inheritance, the examples from Section 3.1 are combined into a single construct.

$$\begin{aligned} P = \Lambda\, t \le F[t]\,.\ &\lambda(self: t). \\ &\{\ i = 5, \\ &\quad id = self, \\ &\quad eq = \lambda(o: t).\,(o.i = self.i)\ \} \end{aligned}$$

The type function $F[t] = \{i : int, id : t, eq : t \rightarrow bool\}$ specifies the recursive type of the objects, and P has type $\forall t \le F[t].t \rightarrow F[t]$.

The inheriting definition adds a method b and redefines the equality method.

$$\begin{aligned} C = \Lambda\, t \le G[t]\,.\ &\lambda(self: t). \\ &P[t](self)\ \textbf{with} \\ &\quad \{\ b = true, \\ &\qquad eq = \lambda(o: t).\,(o.i = self.i \text{ and } \\ &\qquad\qquad\qquad o.b = self.b) \\ &\quad \} \end{aligned}$$

The new object has type $\tau = \mu t.G[t]$.

$$G[t] = \{i : int, id : t, b : bool, eq : t \rightarrow bool\}$$

The polymorphic application $P[t]$ is valid because $t \le G[t]$ and $G[t] \le F[t]$ imply $t \le F[t]$. Despite this relationship between G and F, their fixed points are not in a subtype relation.

4 Class Inheritance

4.1 Classes

A class is a parameterized object definition. In the previous section we used simple classes that were just descriptions of a single object. A more sophisticated notion of class includes instantiation parameters so that multiple objects, called the ***instances*** of the class, may be created. In this interpretation classes are functions that create object specifications. Classes may be inherited to define other classes.

A class is recursive if its instances use the class to make new instances. When a method using class recursion is inherited, the recursive use of the class is modified so that the method constructs subclass instances instead. Smalltalk is a good illustration of class recursion and inheritance: an object can determine the class that created it with the expression *self class*. To create a new instance like itself an object sends its class a *new* message: *self class new*. In Smalltalk, *new* messages are handled by metaclasses, which support specialization of object creation by inheritance.

A recursive class is defined using fixed points, just as objects are fixed points of mutually recursive method specifications. For objects, the functional argument represented *self*, to which recursive messages are sent. For classes, the argument represents the class to use in constructing new instances. This argument is called *myclass*. The general untyped form of a recursive class definition has two levels of recursion, *myclass* and *self*.

$$\begin{aligned} \mathcal{P} = \lambda(myclass).\,\lambda(x).\,\lambda(self). \\ \{m_1 = e_1, \ldots, m_j = e_j\ \} \end{aligned}$$

The argument x represents the instantiation parameter. The class recursion variable, *myclass*, is used in the expressions e_i to construct new instances of the class. Let $P = \mathbf{Y}(\mathcal{P})$ be the class associated with the class definition $\mathcal{P}$. An object is instantiated with parameter a by applying the class to a and then taking the fixed point: $\mathbf{Y}(P(a))$. The complete equation for making an instance from a recursive class definition involves a double fixed point: $\mathbf{Y}(\mathbf{Y}(\mathcal{P})(a))$. Two applications of the fixed-point function are used because class recursion and object recursion are independent.

In the child class definition, $\mathcal{P}$ is passed a new value for *myclass* so that the inherited methods m_i create instances of $\mathcal{C}$, not instances of $\mathcal{P}$.

$$\begin{aligned} \mathcal{C} = \lambda(myclass).\,\lambda(x).\,\lambda(self). \\ \mathcal{P}(myclass)(x)(self)\ \textbf{with} \\ \{m'_1 = e'_1, \ldots, m'_k = e'_k\ \} \end{aligned}$$

4.2 Typed Class Inheritance

The typing of class recursion uses the same technique of polymorphism introduced in Section 3.2. Although the scope of class-level recursion contains the scope of object recursion, the polymorphism associated with the type of *self* must be moved outside of the class recursion variable.

$$\begin{aligned} &\mathcal{P} : \forall t \le G[t].(\alpha \rightarrow (t \rightarrow t)) \rightarrow (\alpha \rightarrow (t \rightarrow F[t])) \\ &\mathcal{P} = \Lambda\, t \le F(t)\,.\ \lambda(myclass : \alpha \rightarrow (t \rightarrow t)). \\ &\quad \lambda(y : \alpha).\,\lambda(self: t). \\ &\qquad \{m_1 = e_1, \ldots, m_k = e_k\ \} \end{aligned} \tag{5}$$

Note that *myclass* produces values of type $t \rightarrow t$ rather than $t \rightarrow F[t]$ as in the final result-type of $\mathcal{P}$. This allows the fixed point of *myclass* to be used without complete knowledge of the final binding of t. The objects created by this class definition have type $\sigma = \mu t.F[t]$. Instantiation of a class definition with polymorphic typing, $P = \mathbf{Y}(\mathcal{P}[\sigma])$, involves binding the type argument the instance type and then taking the fixed point. The class P has type $\alpha \rightarrow (\sigma \rightarrow \sigma)$.

The typing of an inheriting class definition is straightforward.

$$\begin{aligned}
&\mathcal{C} : \forall t \leq G[t].(\alpha\rightarrow(t\rightarrow t))\rightarrow(\alpha\rightarrow(t\rightarrow G[t]))\\
&\mathcal{C} = \Lambda\, t \leq G(t)\,.\,\lambda(myclass : \alpha\rightarrow(t\rightarrow t)).\\
&\quad \lambda(x : \alpha).\,\lambda(self : t).\\
&\qquad \mathcal{P}[t](myclass)(x)(self)\\
&\qquad\quad \textbf{with } \{m'_1 = e'_1, \ldots, m_k = e'_k\}
\end{aligned} \tag{6}$$

4.3 Changing Instantiation Parameters

Class inheritance is complicated by the common need to change the form of the instantiation parameters of the subclass to be of some type β. The problem is that the inherited definition expects a value of *myclass* with type $\alpha\rightarrow(t\rightarrow t)$, but the subclass definition of *myclass* has type $\beta\rightarrow(t\rightarrow t)$. Unless $\alpha \leq \beta$, the types will not match. This condition is too restrictive: it is common for the subclass to require more information, not less.

The difference between the initialization parameters is bridged by two translation functions. The first translation, $Q : \beta\rightarrow\alpha$, converts child parameters to the form required by the parent class. The second translation, $T : \alpha\rightarrow\beta$, converts parent parameters to the form required by the child so that uses of *myclass* in parent methods will construct child instances. With these translation functions, the inheritance construct supports modification of instantiation parameters.

$$\begin{aligned}
&C : \forall t \leq G[t].(\beta\rightarrow(t\rightarrow t))\rightarrow(\beta\rightarrow(t\rightarrow G[t]))\\
&\mathcal{C} = \Lambda\, t \leq F(t)\,.\,\lambda(myclass : \beta\rightarrow(t\rightarrow t)).\\
&\quad \lambda(y : \beta).\,\lambda(self : t).\\
&\qquad \mathcal{P}[t](myclass \circ T)(Q(y))(self)\\
&\qquad\quad \textbf{with } \{m'_1 = e'_1, \ldots, m'_k = e'_k\}
\end{aligned} \tag{7}$$

T and Q are defined in a context in which y and *self* are bound. The context is particularly relevant in the case of T, since the additional information required for subclass instantiation is often computed from *self*.

5 Type Inheritance

As an operation on recursive definitions, inheritance can also be applied to recursively defined record types. Let $F[t] = \{x_1 : \sigma_1, \ldots, x_n : \sigma_n\}$ be a type function defining a recursive record type $\sigma = F[\sigma]$. Type inheritance allows the definition F to be modified to define a new type. A definition that inherits F has the form

$$G[t] = F[t] + \{x'_1 : \sigma'_1, \ldots, x'_n : \sigma'_n\}$$

G defines the type $\tau = G[\tau]$, a child of σ. Note that $G[t]$ need not be a subtype of $F[t]$ because the field types may be changed. The replacement of field types during type inheritance is analogous to the replacement of field values (methods) during object inheritance.

There is a close connection between type inheritance and class/object inheritance. In the polymorphic typing defined in Section 3.2, the type function G which specifies the type of the inheriting object may be expressed by inheriting F. The types of methods $m'_1, \ldots, m'_k$ that are changed can be identified in a type-function R for which $G[t] = F[t] + R[t]$ Thus the type of an inherited object is an inherited type.

The properties of types of inherited objects are analogous to those of subtypes. The constraint imposed by object inheritance, $G[t] \leq F[t]$, ensures that inherited objects can be used as arguments to F-bounded polymorphic functions just as values of subtypes can be used as arguments to subtype-bounded polymorphic functions. Thus F-bounded polymorphism is useful in object-oriented programming for writing functions that work uniformly over the subclasses of a class.

6 Example

The following example illustrates the recursive structure of objects, classes and types, and the typing of inheritance given in Sections 3 and 4. The class definition exhibits both object and class recursion, and gives an example of typed class inheritance. The type of the objects created from the inherited class is defined using the type inheritance operation described in Section 5. For a more complete discussion of a version of this example and the informal object-oriented notation used below, see [4].

A type *Point* specifies the interface of movable planar points. When a point is moved it returns a new point at the new location.

interface *Point*
    x : Real
    y : Real
    move(Real, Real) : Point
    equal(Point) : Boolean

More formally, *Point* is the fixed point of a type function derived from the interface definition.

$$\begin{aligned}
F[t] = \{\ & x : Real,\\
& y : Real,\\
& move : Real \times Real \rightarrow t,\\
& equal : t \rightarrow Boolean\ \}
\end{aligned}$$

Type inheritance can be used to extend the recursive type *Point*.

interface *ColorPoint* **inherits** *Point*
    color : Color

Type inheritance is explained as extension of type functions in Section 5. *ColorPoint* is the fixed point of G.

$$G[t] = F[t] + \{color : Color\}$$

ColorPoint $\not\leq$ *Point* because the equality method is contravariant. Intuitively, a *ColorPoint* can't be used where a *Point* is expected because it does not make sense to compare *Points* and *ColorPoints* for equality. The problem is that *Points* do not have color. The system could also have been designed to allow the comparison, but then the *ColorPoint* equality method could not determine the color of its argument.

On the other hand, *ColorPoint* does have the same recursive structure as *Point*: $ColorPoint \leq F[ColorPoint]$ and for all t, $G[t] \leq F[t]$. These are exactly the constraints required by inheritance.

The class *cart_point* implements objects of type *Point*. It has two initialization parameters, x and y, that specify the location of the point in cartesian coordinates.

```
class cart_point (x:Real, y:Real)
    implements Point
    method x : Real
        return x
    method y : Real
        return y
    method move(dx:Real, dy:Real) : Point
        return new myclass(self.x + dx, self.y + dy)
    method equal(p:Point) : Boolean
        return (self.x = p.x) and (self.y = p.y)
```

Instances of *cart_point* are recursive because they send messages to *self*. The class *cart_point* is also recursive because the *move* method uses *myclass* to create a new point at a given distance from itself. Both the *equal* method and the *move* method involve the type *Point* in association with object recursion, so there is an opportunity to encode these types so that they may be specialized. The definition above is easily translated into the format of Equation 5.

$$
\begin{aligned}
&\mathcal{P} = \Lambda\, t \leq F[t] \,.\, \lambda(myclass : (Real \times Real) \rightarrow (t \rightarrow t)).\\
&\quad \lambda(x : Real, x : Real).\, \lambda(self : t).\\
&\quad\ \{\ x = x,\\
&\qquad y = y,\\
&\qquad move = \lambda(dx : Real, dy : Real).\\
&\qquad\quad \mathbf{Y}(myclass(self.x + dx, self.y + dy)),\\
&\qquad equal = \lambda(o : t).\\
&\qquad\quad (self.x = o.x) \ \text{ and } \ (self.y = o.y)\\
&\qquad \}
\end{aligned}
$$

Instances of *cart_point* have type *Point*. The class associated with this definition is

$$cart\_point = \mathbf{Y}(\mathcal{P}[Point])$$

To illustrate, consider the point at location (2,5). Of course, the object has an infinite expansion, so it can only be written using fixed points.

$$
\begin{aligned}
p \ &= \ \mathbf{Y}(cart\_point(2,5))\\
&= \ \mathbf{Y}(\mathbf{Y}(\mathcal{P}[Point])(2,5))\\
&= \ \{\ x = 2,\\
&\qquad y = 5,\\
&\qquad move = \lambda(dx : Real, dy : Real).\\
&\qquad\quad \mathbf{Y}(cart\_point(2 + dx, 5 + dy)),\\
&\qquad equal = \lambda(o : t).\\
&\qquad\quad (2 = o.x) \ \text{ and } \ (5 = o.y)\,\}
\end{aligned}
$$

Using inheritance, a new class *color_point* is defined. Instances of *color_point* have an additional method *color* that is defined using an additional instantiation parameter for the class. The equality method is redefined so that two points are equal only if their colors match.

```
class color_point (x:Real, y:Real, c:Color)
    implements ColorPoint
    inherit cart_point(x,y)
        translating new myclass(x', y')
            to new myclass(x', y', self.color)
    method color : Color
        return c
    method equal(p:ColorPoint) : Boolean
        return super.equal(p) and
                (self.color = p.color)
```

The class *color_point* inherits *cart_point* and indicates the two translations. The first translation is given by *cart_point(x,y)*; it indicates how to instantiate the inherited point class. The second translation is more explicit; it indicates how recursive calls within *cart_point* are to be translated to construct *color_point* objects. In this example the moved point simply retains its color. It is also possible to define an arbitrary computation of the new color from the point's position and previous color. The modified *equal* method uses *super* to invoke the original notion of equality and add a new constraint. The treatment of *super* simply involves an additional let variable.

$$
\begin{aligned}
&\mathcal{C} = \Lambda\, t \leq G[t]\,.\\
&\quad \lambda(myclass : (Real \times Real \times Color) \rightarrow (t \rightarrow t)).\\
&\qquad \lambda(x : Real, y : Real, c : Color).\\
&\qquad\quad \lambda(self : t).\\
&\quad \textbf{let } super = \mathcal{P}[t](\lambda(x', y').\, myclass(x', y', c))\\
&\qquad\qquad\qquad (x, y)(self)\\
&\quad \textbf{in } super \textbf{ with } \{\ color = c\\
&\qquad\qquad\qquad equal = \lambda(o : t).\\
&\qquad\qquad\qquad\quad super.equal(o) \ \text{ and}\\
&\qquad\qquad\qquad\quad (self.color = o.color)\\
&\qquad\qquad\qquad \}
\end{aligned}
$$

Most of the work occurs in the value bound to *super*. First, the parent class definition is applied to t, a polymorphic type variable constrained by $t \leq G[t]$. This polymorphic application is legal because $\mathcal{P}$ accepts any type $t \leq F[t]$ and we know that $G[t] \leq F[t]$. The result is applied to a translated form of the class recursion variable *myclass*. Recursive class instantiation in *cart_point* are translated to construct colored points. Finally, the parent component is initialized by x and y, and *self* is bound to interpret method recursion properly.

Instances of *color_point* and *cart_point* cannot be intermixed in a program because their types are not subtype compatible (they cannot be compared for equality). However, it is possible to write F-bounded polymorphic functions that operate uniformly over either *Points* or *ColorPoints*.

7 Related Type Systems for Inheritance

7.1 Eiffel

Eiffel is based on an identification of classes with types and of inheritance with subtyping. Within this context, however, Eiffel is able to express many of the constructs described in this paper. The correspondence is not complete, because the identification of inheritance and subtyping makes Eiffel's type system insecure [12].

In Eiffel, the pseudovariable *Current* is used instead of *self* to indicate object recursion. Class recursion and a form of type inheritance are expressed by the type expression *Like Current*, which refers to the current class. That is, it denotes the class in which it appears or into which it is inherited. *Like Current* acts somewhat like the type variable t in the polymorphic typing of inheritance. The following code illustrates its use to implement the example from Section 3.2.

```
class P feature
    i : Integer is 5;
    id : Like Current is Current
    eq(other : Like Current) : Boolean is
        begin
            Result := (other.i = Current.i)
            end
end P

class C inherit P redefine eq feature
    b : Integer is 5;
    eq(other : Like Current) : Boolean is
        begin
            Result := (other.i = Current.i)
                and (other.b = Current.b)
            end
end P
```

This code illustrates the problem of assuming that subclasses are subtypes. With this assumption, one can assign an instance *c* of class *C* to a variable *v:P*, *v:=c*. It is then legal to send the *eq* message to *v* with parameter *p* of class *P*. However, *v.eq(p)* cannot execute properly because *p* does not have a *b* attribute. A modified conformance rule that eliminates this problem was proposed in [12].

Like Current also allows Eiffel to express class recursion. For example, the following *clone* method always creates an instance of the same class as the receiver of the message because it uses *Like Current* for the class. Unfortunately, Eiffel has no way to translate instantiation parameters uniformly.

```
class Copier feature
    clone : Like Current is
        local
            temp : Like Current;
        begin
            temp.Create;
            Result := temp;
        end
end
```

The typed model of inheritance presented in this paper provides a formal model in which Eiffel may be explained. It also indicates why Eiffel's type-system is insecure and how the language may be corrected and extended.

7.2 Other Languages

Like Eiffel, most other strongly-typed object-oriented languages, including Modula-3, C++, Trellis and Simula, are based on the identification of classes and types. Their subtype relations are based on their inheritance hierarchies. Unlike Eiffel, these languages are type-safe because they restrict inheritance to satisfy subtyping. In Modula-3, C++ and Simula, the types of methods may not be changed. Trellis allows the types to be changed according to the rules for function subtypes.

None of the languages provide support for class inheritance, as we define it. It can be simulated manually by defining a method in a root class P called *mynew* which simply executes **new** P. If *mynew* is redefined manually in each subclass C to return **new** C, then class inheritance can be achieved by always using *mynew* instead of **new**. Except in Trellis, the typing does not work correctly because *mynew* must have result-type P in all classes.

Simula and C++ are interesting because they provide mechanisms for translating the subclass instantiation parameters into a form appropriate for instantiation of the parent. This mechanism is explained by the Q translation introduced in Section 4.3.

7.3 Mitchell

Mitchell [20] has developed a typed object model based on extensible records and self-application. Object types are defined by a special **class** notation.

$$T = \textbf{class}\ t\ \{m_1 : \sigma_1, \dots, m_k : \sigma_k\}$$

Objects of type T are similar to records of functions representing methods, except that record component selection is replaced by a *message send* operation. The methods are polymorphic functions with an additional hidden argument. The polymorphism, over a domain of type functions, is used to specialize method types, while the extra argument represents *self*. They are both bound by the message sending operator, $\Leftarrow$, which binds the polymorphic argument to an empty type function, and binds the argument *self* to the object itself. Thus recursion in objects is implemented by self-application.

$$o \Leftarrow m \quad = \quad o.m[\lambda(t :: TYPE).\ \{\,\}](o)$$

A new object may be created by replacing methods or adding new methods in an existing object. Recursion in the types of existing methods are adjusted during extension.

There is a close relationship between Mitchell's work and the model presented in this paper. Although expressed in a different framework, the polymorphic typing of methods achieves the same effect. One advantage of Mitchell's framework is that the type of the *self* argument of each method can be different, giving a more precise typing of methods. Our system has the advantage of simplicity: it does not require special **class** types, extensible records, quantification of type functions, etc.

The systems also differ in their basic object models. Mitchell uses extensible objects, self-application, and a form of delegation [18, 27], while we use records, fixed points, inheritance, and recursive classes. In the models this difference manifests itself in the relative order of record construction and *self* abstraction. In our system, polymorphism (F-bounded) and *self* occur outside the record of methods, requiring type application and a fixed-point function to create an object, which is a simple record. Mitchell places the polymorphism (over type functions) and *self* within each method of an object and binds them during message passing.

Since record formation and *self* abstraction are independent, it appears that the two systems are isomorphic at the value level. Even so, there are tradeoffs. One advantage of the delegation system is that classes and objects are unified. However, class recursion does not seem to be supported, since all object creation is done by extending existing objects. This makes it difficult to construct objects with hidden state, as is possible with recursive classes. Our system has the advantage of a uniform treatment of inheritance on objects, classes and types. A formal investigation of relationship between the models may provide useful insights into implementation techniques and possible extensions.

7.4 Wand

Wand [29, 31] and Remy [23] have developed a type-inference scheme for dealing with records and record extension. Wand's system also allows recursive types. Record types are total functions from labels to a union of ***present*** and ***absent*** fields. The system uses ML-style parametric polymorphism instead of record subtype polymorphism: a record type has the form $\Pi[l_i : f_i]\rho$, where l_i are the explicitly defined fields and f_i is either ***present***(τ) or ***absent***. The ***record extension variable*** ρ represents any additional fields the record may have. In these systems there is no notion of subtyping; there is only parametric polymorphism. In a recent manuscript, Wand [30] proposes a type-inference rule in the style of ML for object inheritance with recursive types. With first-order extension variables and recursive types, his system can express F-bounded polymorphism. Thus his system can type the examples given in this paper.

A practical drawback to using first-order polymorphism to implement record subtype polymorphism is that records and record functions cannot be passed as arguments and then used with subtype polymorphism; the first-order constraint requires that the types be bound when the record is passed. A more serious problem arises in the presence of recursive types. In a traditional type-system with record subtypes, a subtype of a recursive type may have different fields at each level of unrolling in its infinite expansion. If restricted to first-order quantification, the extension variables in a recursive type cannot be instantiated at each level of unrolling. To achieve the effect of record subtyping, quantifiers would have to be included in the scope of recursion.

These problems could be solved by introducing explicit quantification, at the cost of making type-inference much more complex. A detailed comparison of Wand's type-inference approach based on record extension and our explicitly-typed approach with F-bounded polymorphism is an important topic for future research.

7.5 TS

The TS project [15, 16, 17] has recognized that inheritance does not necessarily produce subtypes. To type-check existing Smalltalk programs, which may contain ad-hoc combinations of code, their system copies the text of methods from parent to child before type-checking. These inherited method expressions are type-checked in their new context and may have very different types than they did in the parent class. As a result, a particular expression may be type-checked many times, depending on the depth of the inheritance hierarchy. They also perform type inference, using ML-style unification and first-order polymorphism.

The polymorphic typing of inheritance presented here supports a somewhat more abstract notion of typed inheritance, since a single (polymorphic) type must be assigned to a class. Our polymorphic typing of inheritance uses only the type of the parent, not its method expressions; the type of an inherited method may change but only according to its polymorphic typing. The polymorphic typing provides a degree of encapsulation not found in TS, but TS can type-check some programs that do not have polymorphic typings.

8 Conclusion

We present a typed model of inheritance that preserves more of the flexibility of inheritance in untyped object-oriented languages. Our typing applies to both object and class inheritance. In addition, a notion of inheritance for types is introduced. Type inheritance is analogous to subtyping, but is useful for object-oriented programming because inherited objects have inherited types, not subtypes.

Previously, typed languages allow either modification of instantiation parameters (Simula and C++) or provide class inheritance (Eiffel), but not both. Smalltalk provides both but is untyped. Our model provides a explicit higher-order formalism in which class inheritance, inheritance over recursive types, and modification of instantiation parameters are all supported.

Acknowledgement

We would like to thank Luca Cardelli, David Chase, Jim Donahue, John Mitchell and Benjamin Pierce for the discussions on this work. We would also like to thank Alan Snyder for his support and inspiration [26].

References

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

[2] A. H. Borning and D. H. Ingalls. A type declaration and inference system for Smalltalk. In *Proc. of Conf. on Principles of Programming Languages*, pages 133–141, 1982.

[3] P. Canning, W. Cook, W. Hill, J. Mitchell, and W. Olthoff. F-bounded polymorphism for object-oriented programming. In *Proc. of Conf. on Functional Programming Languages and Computer Architecture*, pages 273–280, 1989.

[4] P. Canning, W. Cook, W. Hill, and W. Olthoff. Interfaces for strongly-typed object-oriented programming. In *Proc. ACM Conf. on Object-Oriented Programming: Systems, Languages and Applications*, pages 457–467, 1989.

[5] P. Canning, W. Hill, and W. Olthoff. A kernel language for object-oriented programming. Technical Report STL-88-21, Hewlett-Packard Labs, 1988.

[6] L. Cardelli. A semantics of multiple inheritance. In *Semantics of Data Types, LNCS 173*, pages 51–68. Springer-Verlag, 1984.

[7] L. Cardelli. Amber. In *Combinators and Functional Programming Languages, LNCS 242*, pages 21–47, 1986.

[8] L. Cardelli. Structural subtyping and the notion of power type. In *Conf. Rec. ACM Symp. on Principles of Programming Languages*, pages 70–79, 1988.

[9] L. Cardelli, J. Donahue, M. Jordan, B. Kaslow, and G. Nelson. The Modula-3 type system. In *Conf. Rec. ACM Symp. on Principles of Programming Languages*, pages 202–212, 1989.

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

[11] W. Cook. *A Denotational Semantics of Inheritance.* PhD thesis, Brown University, 1989.

[12] W. Cook. A proposal for making Eiffel type-safe. In *Proc. European Conf. on Object-Oriented Programming*, pages 57–70. BCS Workshop Series, 1989. Also in *The Computer Journal*, 32(4):305–311, 1989.

[13] 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.

[14] A. Goldberg and D. Robson. *Smalltalk-80: the Language and Its Implementation.* Addison-Wesley, 1983.

[15] J. Graver. *Type-Checking and Type-Inference for Object-Oriented Programming Languages.* PhD thesis, University of Illinois, 1989.

[16] R. Johnson and J. Graver. A user's guide to Typed Smalltalk. Technical Report UIUCDCS-R-88-1457, University of Illinois, 1988.

[17] R. Johnson, J. Graver, and L. Zurawski. TS: An optimizing compiler for Smalltalk. In *Proc. ACM Conf. on Object-Oriented Programming: Systems, Languages and Applications*, 1988.

[18] H. Lieberman. Using prototypical objects to implement shared behavior in object-oriented systems. In *Proc. ACM Conf. on Object-Oriented Programming: Systems, Languages and Applications*, pages 214–223, 1986.

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

[20] J. C. Mitchell. Towards a typed foundation for method specialization and inheritance. In *Proc. of Conf. on Principles of Programming Languages*, 1989.

[21] D. Moon. Object-oriented programming with Flavors. In *Proc. ACM Conf. on Object-Oriented Programming: Systems, Languages and Applications*, pages 1–9, 1986.

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

[23] D. Rémy. Typechecking records and variants in a natural extension of ML. In *Conf. Rec. ACM Symp. on Principles of Programming Languages*, pages 77–88, 1989.

[24] J. Reynolds. User-defined data types and procedural data structures as complimentary approaches to data abstraction. In *New Advances in Algorithmic Languages*. INRIA, 1975.

[25] C. Schaffert, T. Cooper, B. Bullis, M. Kilian, and C. Wilpolt. An introduction to Trellis/Owl. In *Proc. ACM Conf. on Object-Oriented Programming: Systems, Languages and Applications*, pages 9–16, 1986.

[26] A. Snyder. Inheritance and the development of encapsulated software components. In B. Shriver and P. Wegner, editors, *Research Directions in Object-Oriented Programming*, pages 165–188. MIT Press, 1987.

[27] L. A. Stein. Delegation is inheritance. In *Proc. ACM Conf. on Object-Oriented Programming: Systems, Languages and Applications*, pages 138–146, 1987.

[28] B. Stroustrup. *C++*. Addison-Wesley, 1987.

[29] M. Wand. Complete type inference for simple objects. In *Proc. IEEE Symposium on Logic in Computer Science*, pages 37–44, 1987.

[30] M. Wand. Type inference for objects with instance variables and inheritance, 1989. manuscript.

[31] M. Wand. Type inference for record concatenation and multiple inheritance. In *Proc. IEEE Symposium on Logic in Computer Science*, pages 92–97, 1989.