Adding Axioms to Cardelli-Wegner Subtyping

A J H Simons, Department of Computer Science, University of Sheffield,

Regent Court, 211 Portobello Street, Sheffield S1 4DP, United Kingdom.

Email: A.Simons@dcs.shef.ac.uk

Abstract

Cardelli and Wegner developed a simple theory of object subtyping which was later to form the basis for a second-order theory of bounded quantification [Card84, CW85, Ghel90] and the higher-order theory of F-bounded quantification explored by Cook and others [CCHO89a, CHC90]. In all of these presentations, the abstract type of objects is only expressed syntactically, in terms of an external interface of function signatures. Here, we re-introduce semantic descriptions for objects, in terms of sets of axioms constraining the operation of some invocations of their functions. We use the well-understood technique of *definition by comprehension* to motivate subtyping rules for object axioms and prove how these rules interact properly with Cardelli-Wegner style subtyping rules. For languages like Eiffel [Meye88, Meye92] and Sather [Omoh94] in which programmers can write object axioms, rules governing the addition of preconditions, postconditions and data type invariants can now be motivated from the type-theoretic standpoint. By translating between semantic and syntactic modes of definition, we show how our new subtyping rules for axioms will have their counterparts in the second and higher-order theories of strongly-typed inheritance.

Introduction

The Cardelli-Wegner theory of subtyping [Card84, CW85] popularised a view of objects as record instances whose components are methods and gave rise to terms such as *covariance* and *contravariance* to describe the typing relationships which have to exist between two objects' methods for the objects to enter into subtyping relationships. Many interesting and useful theories describing strongly-typed inheritance in object-oriented programming have been developed from this work [CW85, Card86, Card88a, Card88b, CL91, Cook89a, CCHO89a, CCHO89b, CP89, CHC90], although not all of these treat inheritance as as subtyping. We shall show how subtyping has a role to play in each of these theories.

In all these theories, the type of objects is expressed in a *syntactic* way, as a set of exported function signatures owned by the type. We see advantages in extending this theory to include *semantic* descriptions of types, expressed in terms of axioms, to take account of those object-oriented languages such as Eiffel [Meye88, Meye92] and Sather [Omoh94] in which semantic assertions may be made about object behaviour, but about which no inferences may yet be drawn as to the effect this has on object type. Furthermore, a proper incorporation of axioms into object-oriented type theory is long overdue. Algebraic descriptions of abstract types have for long used an axiomatic style of declaration [Gutt75, Gutt77, FGJM85]. We introduce an algebraic style of specification to capture the semantics of an object's methods.

We develop an axiom subtyping rule to add to the syntactic subtyping rules given by Cardelli and Wegner in [CW85]. The integration is made possible by appealing to the simple notion of *definition by comprehension*. For example, given that a simple non-recursive integer point type has the syntactic specification:

$$\text{INTEGER\_POINT} = \{\ x : \rightarrow \text{INTEGER};\ y : \rightarrow \text{INTEGER}\ \}$$

a subtype of integer points can be defined by comprehension using two axioms on x and y:

$$\text{POSITIVE\_POINT} = \{\ \forall p \in \text{INTEGER\_POINT} \mid p.x \geq 0 \wedge p.y \geq 0\ \}$$

The fact that this defines a subtype is easily shown by plotting the set of INTEGER_POINTs in the Cartesian plane. All POSITIVE_POINTs are a subset occupying the first quadrant. By giving this type a new name, we are able to translate between axiomatic and syntactic modes of defining types. We go on to show what additions and modifications to sets of axioms result in subtypes. By interpreting pre- and postconditions in this model, we show how it is possible to include subtyping rules for preconditions, postconditions and data type invariants, as used in Eiffel and Sather [Meye88, Meye92, Omoh94, SOM93].

The Cardelli-Wegner Calculus of Subtyping

Cardelli and Wegner [Card84, CW85] developed one of the first typed models for object-oriented programming based on a calculus of subtyping. We present their syntactic subtyping calculus here, since our axiom subtyping calculus is a simple extension to this approach. According to these authors:

> "a type A is included in, or is a subtype of another type B when all the values of type A are also values of B, that is, exactly when A, considered as a set of values, is a subset of B" [CW85], p508.

It is clear that subtyping in this model is identified with the subset relationship:

$\sigma \subseteq \tau \Leftrightarrow \forall x\ (x \in \sigma \Rightarrow x \in \tau)$ [Rule 0: subset; and subtype]

One immediately useful property of having set-theoretic types is that it allows us to construct a complete partial order (CPO) relating all types (ie sets) in a lattice by the relation $\subseteq$.

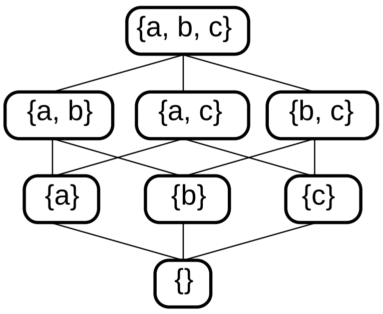

Figure 1: CPO for sets

Figure 1 illustrates this for a small domain of values {a, b, c}. CPOs allow us to infer more general types for objects by navigating upwards in the lattice. For example, an object *a* of type {a} also has the types {a, b}, {b, c} and {a, b, c}. Put more formally, a CPO satisfies the mathematical properties of *reflexivity*, *transitivity* and *antisymmetry*:

- reflexivity: $\forall\sigma\ (\sigma \subseteq \sigma)$
- transitivity: $\forall\sigma\ \forall\tau\ \forall\upsilon\ (\sigma \subseteq \tau \wedge \tau \subseteq \upsilon \ \Rightarrow\ \sigma \subseteq \upsilon)$
- antisymmetry: $\forall\sigma\ \forall\tau\ (\sigma \subseteq \tau \wedge \tau \subseteq \sigma \ \Rightarrow\ \sigma = \tau)$

This means that a type is a subtype of itself; a type is a subtype of some eventual ancestor type if a linking path can be found through the subtype lattice; and two different types can only be subtypes of each other if in fact they are the same type.

We can also construct CPOs for subrange types s..t, where $s \in$ NATURAL; $t \in$ NATURAL; and the total ordering $s \leq t$ holds. The set of all subranges has a useful partial order $\subseteq$ among its elements.

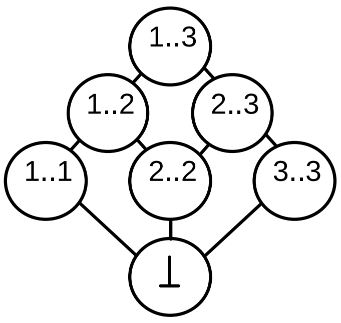

Figure 2: CPO for subranges

Figure 2 illustrates this for all subtypes of 1..3, where $\perp$ denotes the empty subrange. This graph has fewer nodes and arcs than the CPO for set-types, due to the fact that subranges may not contain discontinuous sequences of numbers such as {1, 3}. To express this constraint, we assert the following equivalence:

$$s..t \subseteq \sigma..\tau \iff s \geq \sigma \wedge t \leq \tau$$

Henceforward, we shall use the (weaker) implication $\Rightarrow$ and denote this using:

$$\frac{s \geq \sigma,\ t \leq \tau}{s..t \subseteq \sigma..\tau} \qquad \text{[Rule 1: subtyping for subranges]}$$

which says that the subrange type s..t is a subtype of σ..τ if the start of the range s is at least as big as σ and the end of the range t is no greater than τ. This is the standard form of type-inference rule found elsewhere in the literature. It is read from top to bottom: the antecedent conditions appear above the line and the deduced consequent below the line.

A function type is constructed using $f : \sigma \rightarrow \tau$, where σ is the *domain* (the type of the argument) and τ is the *codomain* (the type of the result). In order to motivate the construction of CPOs between function types, Cardelli and Wegner [CW85] use subranges to model the domain and codomain of functions. By expanding the *codomain,* we see that a function can be considered to belong to the following increasingly more general types:

$$f : (2..5 \rightarrow 3..4) \subseteq (2..5 \rightarrow 2..5) \subseteq (2..5 \rightarrow 1..6) \ ...$$

since any function mapping NATURALs into the codomain 3..4 will also map them into 2..5 and 1..6. However, a symmetrical expansion of the *domain* of a function does not result in more general function types:

$$g : (3..4 \rightarrow 2..5) \not\subseteq (2..5 \rightarrow 2..5) \not\subseteq (1..6 \rightarrow 2..5) \ ...$$

since a function accepting NATURALs in 3..4 will not accept values outside this range, such as 5 or 1. In fact, an antisymmetrical condition applies - the domain must shrink in order to obtain a more general function type. Combining these considerations, in order to obtain a more general function type, the domain must shrink and the codomain must expand:

$$h : (1..6 \rightarrow 3..4) \subseteq (2..5 \rightarrow 2..5) \subseteq (3..4 \rightarrow 1..6) \ ...$$

Turning the rule around, in order to obtain a more specific subtype function, the domain must expand and the codomain must shrink. We express this in the function subtyping rule:

$$\frac{s \supseteq \sigma, \; t \subseteq \tau}{s \to t \subseteq \sigma \to \tau}$$ [Rule 2: subtyping for functions]

This rule says that for two function types S and T, S $\subseteq$ T if S is *covariant* with T in its result type (ie the result of S is also a subtype of the result of T) and S is *contravariant* with T in its argument type (ie the argument of S is a supertype of the argument of T). This is an important result, whose significance for object-oriented programming we shall observe later.

Records are aggregated objects composed of named fields containing values, which can be modelled as a set of mappings from labels to values. Record types are similarly modelled as a set of mappings from labels to types. This allows us to construct typed records of the form:

{ x = 2; y = -3 } : { x : INTEGER; y : INTEGER }

Cardelli first identified record subtyping [Card84, Card88a] as a way of constructing CPOs among structured types. Consider the simple, non-recursive record types INTEGER_POINT and COLOUR_POINT, where COLOUR_POINT objects have an additional field indicating one of two possible colours:

INTEGER_POINT = { x : INTEGER; y : INTEGER }

COLOUR_POINT = { x : INTEGER; y : INTEGER; c : BOOLEAN }

To determine the subtyping relationship, Cardelli appeals to the substitutability criterion. Wherever a program expects an object of type INTEGER_POINT, a COLOUR_POINT object may be substituted, since it has all the fields of INTEGER_POINT. This means that a COLOUR_POINT can be coerced to an INTEGER_POINT, but not vice-versa. This suggests a subtyping relationship:

p : COLOUR_POINT $\subseteq$ INTEGER_POINT and leads to the first part of the record subtyping rule dealing with monotonic extensions to record types:

$\{ x_1:\sigma_1, \dots x_k:\sigma_k, \dots x_n:\sigma_n \} \subseteq \{ x_1:\sigma_1, \dots x_k:\sigma_k \}$ [Rule 3.1: record extension]

which says that for two record types S and T, $S \subseteq T$ if S has the same number, or strictly more fields than T and those fields that it shares with T are in the same types. If two records simply share a common subset of fields, neither one is in a subtype relationship with the other.

The second part of the record subtyping rule comes from considering what happens if the fields of S are not in the same types as T. Consider the type:

POSITIVE_POINT = { x : POSITIVE; y : POSITIVE }

which has the same structure as INTEGER_POINT, yet the type of its fields is restricted. By plotting all INTEGER_POINTs in the Cartesian plane, it is easy to see that the set of all POSITIVE_POINTs forms a subset which occupies the first quadrant, defined by:

POSITIVE_POINT = { $\forall p \in$ INTEGER_POINT | $p.x \geq 0 \wedge p.y \geq 0$ }

and we have already identified a subset with a subtype. This intuition leads to the second part of the record subtyping rule dealing with modifications to the field types:

$$\frac{\sigma_1 \subseteq \tau_1, \dots \sigma_n \subseteq \tau_n}{\{ x_1:\sigma_1, \dots x_n:\sigma_n \} \subseteq \{ x_1:\tau_1, \dots x_n:\tau_n \}}$$ [Rule 3.2: record overriding]

This rule says that for two record types S and T, $S \subseteq T$ if they have the same number of fields and the type of each field σ_i of S is a subtype of the corresponding field τ_i of T. There is no relationship between records whose fields are in a mixture of super- and subtype relationships.

The two parts of the rule are combined in the record subtyping rule:

$$\frac{\sigma_1 \subseteq \tau_1, \dots \sigma_k \subseteq \tau_k}{\{ x_1{:}\sigma_1, \dots x_k{:}\sigma_k, \dots x_n{:}\sigma_n \} \subseteq \{ x_1{:}\tau_1, \dots x_k{:}\tau_k \}} \quad \text{[Rule 3: record subtyping]}$$

which says that for two record types S and T, S $\subseteq$ T if S has n-k more fields than T; and the first k fields of S are subtypes of those in T. The general rule reduces to rule 3.1 if the first k fields of S are in fact the same types as those in T (allowed by the reflexivity of $\subseteq$) and reduces to rule 3.2 if n=k.

In object-oriented programming, record types are constructed from fields having function types, representing the types of an object's methods. Since methods may refer to each other, or even the whole object *self*, an object is invariably an instance of a recursive record type [BL90, BM92] whose existence is established using the fixed-point theory of recursion [Scot76, Stoy77]. A recursive record type has the form:

$$\begin{aligned}&\text{INTEGER\_POINT} = \text{Rec pnt . } \{ \text{x} : \rightarrow \text{INTEGER}; \ \ \text{y} : \rightarrow \text{INTEGER};\\ &\qquad \text{equal} : \text{pnt} \rightarrow \text{BOOLEAN}; \ \ \text{move} : \text{INTEGER} \times \text{INTEGER} \rightarrow \text{pnt} \}\end{aligned}$$

in which *Rec* is used to quantify the recursion variable *pnt*. We need to introduce a slight addition to the record subtyping rule which will allow us to create subtype records in the presence of recursion. A subtyping rule for recursive types is given by [Card86]:

$$\frac{s \subseteq t \ \triangleright \ \sigma \subseteq \tau}{\text{Rec } s.\sigma \subseteq \text{Rec } t.\tau} \quad \begin{array}{l}\text{s free only in } \sigma\text{, t free only in } \tau.\\ \text{[Rule 3x: recursive record subtyping]}\end{array}$$

which says that if assuming s $\subseteq$ t allow us to derive "$\triangleright$" that $\sigma \subseteq \tau$ then the recursive type Rec s.σ is a subtype of the recursive type Rec t.τ. Within the limits of co- and contravariance, we can express subtyping between recursive record types on the assumption that their syntactic recursion variables enter into a subtyping relationship.

A Calculus of Axiom Subtyping

The machinery we have so far will allow us to infer subtyping relationships between object types expressed syntactically as sets of function signatures. The only area not addressed by [Card84, CW85, Card86, Card88a, Card88b] is the effect of axioms on types. We have considered this informally in [SC92, Simo94] but provide a definitive treatment here.

Axioms help define the meaning of an abstract object type, by expressing invariant properties of the type in terms of relationships pertaining between executions of some of its functions. This is a time- and implementation-independent view, for the moment avoiding notions such as *preconditions* and *postconditions* [Jone86]. It should be possible, by appealing only to the principle of induction, to infer the precise abstract structure and behaviour of a type from its axioms.

To motivate our calculus of axiom subtyping, we appeal to the simple notion of *definition by comprehension*. If a base type is further qualified by an axiom, then the effect is always to generate a type whose objects form a subset of the base type; and we have already identified a subset with a subtype. Any set defined by comprehension has the property:

$$\{ \forall x \in \sigma \mid p(x) \} \subseteq \sigma$$

since p(x) is either already an axiom of σ or restricts σ to a proper subset. Conventionally, sets defined by comprehension must indicate the base type for which the restricting predicate is well defined. Axioms are expressed in terms of operations that a given base type must possess; otherwise their meaning is undefined. Here, we examine the effect of adding or substituting axioms over a single base type.

A certain primitive collection of objects might have the partial specification:

```
COLLECTION = Rec col . {  add : ELEMENT → col;
        rem : ELEMENT → col;  has : ELEMENT → BOOLEAN }

new : → COLLECTION;                    // The constructor function for this type
```

$\forall$c : COLLECTION, $\forall$d, e : ELEMENT

$\neg$ new.has(e);

c.add(e).has(e);

c.add(d).add(e) = c.add(e).add(d);

new.rem(e) = new;

new.add(e).rem(e) = new;

This type of COLLECTION is empty when created, contains an element that has been added, is unordered and removes an initial element if one is present. The latter axiom deliberately underspecifies the behaviour of *rem()*. It is an open issue whether COLLECTIONs contain single, or multiple occurrences of each element; or whether *rem()* removes one, or all occurrences of an element.

Let us now consider the type obtained by adding some new axioms:

$\forall$e : ELEMENT . { $\forall$c $\in$ COLLECTION |

c.add(e).add(e) = c.add(e);

c.add(e).rem(e) = c.rem(e) }

This definition comprehends only those COLLECTIONs which behave like SETs. It rules out those for which a double application of *add()* results in a semantically different COLLECTION; and rules out those for which an application of *rem()* leaves some occurrence of the element in the COLLECTION.

This leads to the first part of the axiom subtyping rule dealing with monotonic additions to the axioms of a type:

$$\{ \forall x \in \sigma \mid \alpha_1, \ldots \alpha_k, \ldots \alpha_n \} \subseteq \{ \forall y \in \sigma \mid \alpha_1, \ldots \alpha_k \}$$

[Rule 4.1: axiom addition]

which says that for two types S and T defined by comprehension on a common base type σ, S $\subseteq$ T if S has the same number, or strictly more, distinct axiomatic properties than T. The axioms are implicitly conjoined with $\wedge$. Distinctness means that the properties are judged primary and cannot be derived from other properties.

Consider now the type obtained by adding a different axiom:

$$\forall e : \text{ELEMENT} \,.\, \{\ \forall c \in \text{COLLECTION} \mid c.\text{add}(e).\text{rem}(e) = c\ \}$$

which comprehends all those COLLECTIONs for which applications of *add()* and *rem()* are symmetrical, ie those which behave like BAGs. It turns out that this new axiom for BAGs actually subsumes a previous axiom of COLLECTION:

$$c.\text{add}(e).\text{rem}(e) = c \;\Rightarrow\; \text{new}.\text{add}(e).\text{rem}(e) = \text{new}$$

The previous axiom is in fact a ground instance of the new BAG axiom. A *ground instance* is a special case of a logical formula, obtained by substituting a particular value into a general variable. Here, the new BAG axiom, expressed in general terms of $\forall c \in$ COLLECTION, also includes the old COLLECTION axiom, expressed in terms of a specific COLLECTION instance, *new*. This is what is meant by subsumption.

Looking back, it is now apparent that one of the SET axioms introduced above also entails the old COLLECTION axiom. We can show this in a two-step proof involving another of COLLECTION's axioms:

$$\frac{c.\text{add}(e).\text{rem}(e) = c.\text{rem}(e)}{\text{new}.\text{add}(e).\text{rem}(e) = \text{new}.\text{rem}(e)} \quad \text{UNIV\_INSTANTIATION}\ \{\ \text{new} / c\ \}$$

$$\frac{\text{new}.\text{add}(e).\text{rem}(e) = \text{new}.\text{rem}(e) \;\wedge\; \text{new}.\text{rem}(e) = \text{new}}{\text{new}.\text{add}(e).\text{rem}(e) = \text{new}} \quad \text{EQUAL\_TRANS}$$

So, from both SET and BAG axioms, we have been able to derive one of the earlier COLLECTION axioms. We call this relationship between axioms one of *entailment*.

Typically, axioms are selected for economy, with the aim of capturing precisely the semantics of a type. It is undesirable to overspecify or to underspecify the semantics of types. It would be redundant to include, in the specification of SETs or BAGs, any axiom from COLLECTION which was automatically entailed by other SET or BAG axioms. Although subtyping always involves *adding* to the logical properties of a type, in some cases we may achieve this with less redundancy by *modifying* the syntactical form of axioms, provided that these entail the axioms of the supertype. This motivates the second part of the axiom subtyping rule governing axiom substitution:

$$\frac{\{\ \alpha_1, \ldots \alpha_m\ \} \Rightarrow \{\ \beta_1, \ldots \beta_n\ \}}{\{\ \forall x \in\ \sigma \mid \alpha_1, \ldots \alpha_m\ \} \subseteq \{\ \forall y \in\ \sigma \mid \beta_1, \ldots \beta_n\ \}} \qquad \text{[Rule 4.2: axiom substitution]}$$

which says that for two types S and T defined by comprehension on a common base type σ, $S \subseteq T$ if the m syntactically modified axioms α_i of S necessarily entail the n original axioms β_i of T. Axiom substitution is not one-to-one, but on the basis that the new set of axioms entails the original set. The size m of the substituted set may therefore arbitrarily grow or shrink with respect to the size n of the original set, so long as this entailment obtains.

The two parts of the rule are combined in the axiom subtyping rule:

$$\frac{\{\ \alpha_1, \ldots \alpha_k\ \} \Rightarrow \{\ \beta_1, \ldots \beta_n\ \}}{\{\ \forall x \in\ \sigma \mid \alpha_1, \ldots \alpha_k, \ldots \alpha_m\ \} \subseteq \{\ \forall y \in\ \sigma \mid \beta_1, \ldots \beta_n\ \}} \qquad \text{[Rule 4: axiom subtyping]}$$

which says that for two types S and T defined by comprehension on the same base type σ, $S \subseteq T$ if S has m-k more distinct axiomatic properties than T and the first k axioms of S necessarily entail all n axioms of T. The general rule reduces to rule 4.1 if the first k axioms of S are in fact identical to the n axioms of T (allowed by the reflexivity of $\Rightarrow$) and reduces to rule 4.2 if m=k.

Using the axiom subtyping rule, we can derive syntactically similar, but semantically disjoint subtypes of a common, partially specified base type. Familiar examples of these include STACKs and QUEUEs, whose syntactic types are identical, in terms of the signatures of *push()*, *pop()* and *top()* functions [Amer90]. Expressing this shared syntactic specification as a property of the type RANKING:

$$\text{RANKING} = \exists\ \text{ran} . \{\text{empty} : \to \text{BOOLEAN};$$
$$\text{push} : \text{ELEMENT} \to \text{ran};\ \ \text{pop} : \to \text{ran};\ \ \text{top} : \to \text{ELEMENT}\ \}$$

new : → RANKING   // The constructor for this type

we can give a partial semantic specification for RANKING, describing for some functions those properties which are common to both STACKs and QUEUEs:

$\forall$r : RANKING, $\forall$e : ELEMENT

new.empty;

$\neg$ r.push(e).empty;

new.push(e).pop = new;

new.push(e).top = e;

new.pop = $\perp$;

new.top = $\perp$;

where $\perp$, pronounced "bottom", stands for the undefined value. The introduction of $\perp$ is a necessary trick that allows us to model partial functions (defined for only some inputs in the domain) as total functions, which either deliver a valid result or $\perp$.

A STACK is now a type defined by comprehension on RANKING, obtained by providing two axioms which subsume two earlier ones in RANKING:

STACK = $\forall$e : ELEMENT . { $\forall$s $\in$ RANKING |

s.push(e).pop = s;

s.push(e).top = e }

This comprehends all those RANKINGSs for which all applications of *push()* and *pop()* are symmetrical, ie those which behave like STACKs, and for which all applications of *top()* return the last item entered with *push()*, ie those which exhibit the last-in, first-out property of STACKs. A QUEUE is another type defined by comprehension on RANKING, obtained by adding the alternative axioms:

$$\begin{aligned}&\text{QUEUE} = \forall d, e : \text{ELEMENT} . \{ \forall q \in \text{RANKING} \mid \\ &\quad q.push(d).push(e).pop = q.push(d).pop.push(e); \\ &\quad q.push(d).push(e).top = q.push(d).top \}\end{aligned}$$

to express the fact that a *push()* and a *pop()* carried out on a non-empty QUEUE will yield the identical QUEUE no matter in which order they are executed (the two operations commute and this ensures the sequential property of QUEUEs); and that pushing an item onto a non-empty QUEUE does not affect the *top()* item, thereby ensuring the first-in, first-out property of QUEUEs. These are all additional to RANKING's axioms, which are still useful to define what should happen if the QUEUE contains a single item.

This style of definition works well for families of subtypes which have an identical syntactic specification. However, it is more common for axioms to be introduced at the same time as new operations. In this case, we are trying to relate two sets defined by comprehension over two syntactically different base types. We can determine the appropriateness of this on a case-by-case examination. Clearly, we may derive:

$$\{ \forall x \in \sigma \mid \alpha_1, \ldots \alpha_m \} \subseteq \{ \forall y \in \tau \mid \beta_1, \ldots \beta_n \}$$

in the trivial case $\sigma = \tau$. If σ has fewer functions, or more general function types than τ, then for syntactic reasons $\sigma \supset \tau$. By constraining some of σ's operations in $\{ \forall x \in \sigma \mid \alpha_1, \ldots \alpha_m \}$ more strictly than $\{ \forall y \in \tau \mid \beta_1, \ldots \beta_n \}$, we generate two types by comprehension, neither one of which is a subtype of the other. If σ has more functions, or more specific function types than τ, then for syntactic reasons $\sigma \subset \tau$. In this case, defining $\{ \forall x \in \sigma \mid \alpha_1, \ldots \alpha_m \}$ more strictly than $\{ \forall y \in \tau \mid \beta_1, \ldots \beta_n \}$ still generates two types in a subtyping relationship.

We conclude that a modified version of the axiom subtyping rule can apply in cases where the base types $\sigma \subseteq \tau$ are covariant with the types defined on them by comprehension:

$$\frac{\sigma \subseteq \tau \;\wedge\; \{\, \alpha_1, \dots \alpha_k \,\} \Rightarrow \{\, \beta_1, \dots \beta_n \,\}}{\{\, \forall x \in \sigma \mid \alpha_1, \dots \alpha_k, \dots \alpha_m \,\} \subseteq \{\, \forall y \in \tau \mid \beta_1, \dots \beta_n \,\}} \qquad \text{[Rule 4x: axiom subtyping]}$$

However, such an extended formulation of the axiom subtyping rule may be considered redundant, since we can derive these typings from existing simpler rules. Consider that, in general, we can interleave syntactic and semantic subtyping stages:

$$\{\, \forall x \in \sigma \mid \alpha_1, \dots \alpha_m \,\} \subseteq \sigma \subseteq \{\, \forall y \in \tau \mid \beta_1, \dots \beta_n \,\} \subseteq \tau$$

τ is a syntactic type whose semantics is then given by $t = \{\, \forall y \in \tau \mid \beta_1, \dots \beta_n \,\}$. We define the syntactic subtype $\sigma \subseteq t$ by adding one more operation *f()* to τ's syntactic interface. So far, *f()* has no semantics. If we now define a set $s = \{\, \forall x \in \sigma \mid \alpha_1, \dots \alpha_m \,\}$ to give *f()* a meaning in relation to other operations, this in turn creates a subtype of the partially specified type σ.

One of the consequences of our axiom subtyping rule is that it allows us to define a notion of "axiom inheritance" for object types. By this, we mean that a particular type will collect together the axioms of all its ancestors, forming a consistent and non-redundant set. In the same way that inherited functions may be replaced by subtype functions, it is permissible for axioms to be replaced by more general axioms (ie which entail the redundant inherited axioms).

Object Subtyping with Preconditions and Postconditions

The mathematical style of declaration treats all axioms as invariant properties of the type. In practical programming languages such as Eiffel [Meye88, Meye92] and Sather [Omoh94], it is more common to express these axioms as *preconditions*, *postconditions* and *data type invariants*. We interpret the first two in the following way:

- Preconditions: are a necessary consequence of allowing partial functions (ie which are not defined for every element of their type). Instead of saying that *pop()* is undefined $\perp$ for an empty STACK, we restrict *pop()* so that it may not be applied to an empty STACK. This is rather like defining a subtype of STACK by comprehension to which *pop()* may legally be applied:

  $\{ \forall s \in \text{STACK} \mid \neg\, s.\text{empty} \}$

- Postconditions: are a necessary consequence of the representation- and time-dependent implementation strategies of a programming language. Instead of saying that a STACK s satisfies an axiomatic property *s.push(e).pop = s*, we assert after *pop()* that some counter has decremented; and after *push()* that some counter has incremented. Nonetheless, postconditions have the effect of restricting the possible results a function could have. As such, they restrict the meaning of the type to which they belong.

Using Cardelli's calculus of simple subtyping [Card84, Card88a] and our own extension of this theory to include axiom subtyping, we now describe all the properties that simple object types must possess for a subtyping relationship to obtain between them. Combining the above Rules 0 - 4, we find the following:

Any two related object types σ and τ, modelled as records containing sets of functions, are in a subtype relation $\sigma \subseteq \tau$ if:

- extension: σ adds monotonically to the functions inherited from τ (Rule 3); and
- overriding: σ replaces some of τ's functions with subtype functions (Rule 3); and
- restriction: σ has a stronger data type invariant than τ (Rule 4) or is a subrange (Rule 1) or subset (Rule 0) of τ.

A function $\sigma.f$ is a legal subtype replacement for another $\tau.g$ only if:

- contravariance: the arguments of σ.f are more general supertypes than those of τ.g (Rule 2); and therefore preconditions are weaker (Rule 4);
- covariance: the result of σ.f is a more specific subtype than that of τ.g (Rule 2); and therefore postconditions are stronger (Rule 4).

These findings may be applied in different ways in programming languages, depending on their adoption of simple subtyping, bounded quantification or F-bounded quantification to explain inheritance. We discuss the differences between these approaches in our conclusion.

Trellis [SCBK86] is one of the earliest languages treating classes uniformly as types to handle subtype relations correctly. Together with the type rules of POOL-I [Amer90] and of Emerald [BHJL86] Trellis observes both the covariant rule for function results and contravariant rule for function arguments. Contravariance is a counter-intuitive finding for inheritance-as-subtyping because it prevents the uniform specialisation of function arguments and results. It forbids the replacement of a function $g:\tau\rightarrow\tau$ closed over a type τ by a function $f:\sigma\rightarrow\sigma$ closed over a subtype $\sigma \subseteq \tau$.

> Contravariance "has the unfortunate effect of making argument type redefinition almost useless, since it is usually not very useful to allow a redefined method to accept a large class of arguments" [Cook89b], p62.

Generally, Eiffel identifies *class* with *type* and *inheritance* with *subtyping* [Meye88, Meye92]. In its typing rules for function replacement, it follows the covariant rule for results, but curiously ignores the contravariant requirement for arguments [Cook89b]. Instead, a "global system validity check" is performed retrospectively at assembly time to catch type errors. Intuitively, Eiffel is reaching for a polymorphic typing rule which allows inherited functions to be closed over subclasses in the self-type *like Current* which depend on other arguments *like <anchor>*, which an F-bounded interpretation of inheritance would provide.

Eiffel is best known for its *assertions*, executable axiomatic statements defining the semantics of its routines. Coincidentally, Eiffel obeys our Rule 4 for axiom subtyping, although this is justified in terms of Meyer's *programming by contract* metaphor, rather than from subtyping considerations. All services that are guaranteed by one class must also be guaranteed by its descendants, therefore the postcondition on which the success of the service depends must not be weaker, but may be stronger. On the other hand, all messages which one class understands must also be understood by its descendants, therefore the precondition on which acceptance of a message is contingent must not be stronger, but may be weaker. If Eiffel were to convert to an F-bounded type system, the axiom rules would have to be changed to allow stronger preconditions on functions accepting arguments closed over *like Current* or *like <anchor>*.

Sather [Omoh94] obeys all our syntactic rules. It has a polymorphic type variable *SAME*, referring to the type of self, over which it could construct an F-bounded interpretation of inheritance. However, it adopts a simple covariant interpretation of the inherited self-type *SAME* where this occurs as a receiver of messages and a contravariant interpretation in other cases [SOM93]. This means that inherited functions in the type *SAME* are considered subtype functions, so long as *SAME* does not also occur as an argument type (due to dynamic dispatch, this is always safe). Inherited functions closed over arguments in the type SAME are not considered subtype functions. Although nothing is said about the effect adding pre- and postconditions has on types, Sather should observe the current Eiffel rules.

Conclusions on Typed Inheritance

In the simple subtyping approach [Card84, Card88a], object classes are identified with types and inheritance is considered to be the same thing as subtyping. In simple subtyping, we say that an object of type X has methods in the type $\sigma(X)$ since in general they may refer to self and are therefore based on the type of X. The expectation is that other objects of some type $Y \subseteq X$ can be passed legally to these methods. In a statically-bound system, this may always be done safely, albeit with a loss of type information since a method in the type $\sigma(X)$ can only return an object of type X, even when passed an object of type Y. However, it is a common practice in object-oriented programming to replace some of a class X's methods by redefined versions for a subclass Y, in the type $\sigma(Y)$ with the intention that these methods will be invoked dynamically where the original methods in the type $\sigma(X)$ were expected. For this reason, it is important to ensure that the type signature of every redefined method in $\sigma(Y)$ is a subtype of its counterpart in $\sigma(X)$. This ensures that objects of type Y behave in a conformant manner and can safely be passed to variables of type X.

Due to the loss of type information when a class Y inherits methods in the parent type $\sigma(X)$, later work [CW85, Card88b, Ghel90] explored the addition of bounded universal quantification to express the type of polymorphic methods. In bounded quantification, an object's methods are given the type $\forall t \subseteq X.\sigma(t)$ to express the idea that they may acquire more precise type signatures based on all subtype objects of a given type. This work was founded on a simple generalisation of the second-order λ-calculus [CW85, DT88]. The expectation was that a class Y, inheriting a method from X with the type $\forall t \subseteq X.\sigma(t)$, would obtain a version typed in $\sigma(Y)$ by the replacement of the type parameter t. Unfortunately, this expectation was only fulfilled in the context of non-recursive types. In λ-calculus explanations of type recursion, the recursive type of t has to be fixed at $t=X$, making bounded quantification no more expressive than simple subtyping for methods inherited by the subtype Y. Apart from the addition of polymorphism to the type model, a class Y was still expected to conform to a class X according to simple subtyping. Cardelli has since moved away from λ-calculus models in favour of his own object-calculus [Card90, Card92, CL91, CM92], in which it is easier to model pure subtyping. Expressions in the λ-calculus can be translated into object-calculus, in which the primitive operation is record field selection, rather than function application.

At around the same time, work by Cook, Hill, Canning, Olthoff and Mitchell [CCHO89a, CCHO89b, CP89, CHC90] showed that bounded quantification did not deliver useful type signatures for polymorphic functions in the context of recursive types. Their F-bounded quantification, a higher-order subtyping theory, delivers more appropriate types for the same polymorphic functions. In F-bounded polymorphism, an object's methods are given the type $\forall t \subseteq F[t].\sigma(t)$ to express the fact that the type of *self* in the inherited part of an object must change before being combined with new methods. A class is not a type X, but a type generator $\forall t \subseteq F_X[t]$, a polymorphic construct from which the simple types of objects can be recovered when they are created. The simple type X of an object created from a class $\forall t \subseteq F_X[t]$ is given by $X = F_X[X]$, which is explained in the λ-calculus in terms of the fixed point theory of recursion. The result of applying $F_X[t]$ to X delivers a set of methods for the object in the type $\sigma(X)$. An object of type Y inheriting methods in the type $\forall t \subseteq F_X[t].\sigma(t)$ obtains versions which are retyped in $\sigma(Y)$, exactly as desired. As a result, inheritance is not the same thing as simple subtyping, since it is nearly always the case that methods $\sigma(Y)$ are not subtypes of methods $\sigma(X)$, therefore Y is not a subtype of X. Instead, different type checking rules are used, based on a point-wise comparison of types created from generators. Instead of insisting that $Y \subseteq X$, F-bounded quantification requires all possible instantiations of the two generators to be pointwise in a subtype relationship, in other words, where the class $\forall t \subseteq F_Y[t]$ inherits from the class $\forall t \subseteq F_X[t]$, we require that $\forall t \subseteq F_Y[t].t \subseteq F_X[t]$.

In view of this, it is clear that subtyping still has an important role to play in object-oriented programming, although inheritance of type and subtyping are now usually recognised as quite distinct notions. Our modest extension to the Cardelli-Wegner subtyping calculus restores to object-oriented programming the full power of abstract type algebras.

References

[Amer90] P America (1990), 'Designing an object-oriented language with behavioural subtyping', *Proc. Conf. Foundations of Object-Oriented Lang.*, 60-90.

[BHJL86] A Black, N Hutchison, E Jul and H Levy (1986), 'Object structure in the Emerald system', *Proc. 1st ACM Conf. Object-Oriented Sys., Lang. and Appl.*, 78-86.

[BL90] K Bruce and G Longo (1990), 'A modest model of records, inheritance and bounded quantification', *Information and Computation, 87(1-2),* 196-240.

[BM92] K Bruce and J Mitchell (1992), 'PER models of subtyping, recursive types and higher-order polymorphism', *Proc. 19th ACM Symp. Principles of Prog. Lang.*, 316-327.

[Card84] L Cardelli (1984), 'A semantics of multiple inheritance', in: *Semantics of Data Types, LNCS 173*, eds. Kahn, MacQueen and Plotkin, Springer Verlag, 51-68.

[Card86] L Cardelli (1986), 'Amber', *Combinators and Functional Programming Languages, LNCS, 242*, 21-47.

[Card88a] L Cardelli (1988), 'A semantics of multiple inheritance', *Information and Computation, 76*, 138-164.

[Card88b] L Cardelli (1988), 'Structural subtyping and the notion of power type', *Proc. 15th ACM Symp. Principles of Prog. Langs.*, 70-79.

[Card90] L Cardelli (1990), 'Notes about $F_{\leq}$', unpublished manuscript.

[Card92] L Cardelli (1992), 'Extensible records in a pure calculus of subtyping', *Research Report 81, DEC Systems Research Center,* January. Reprinted in: [GM94].

[CCHO89a] P Canning, W Cook, W Hill, W Olthoff and J Mitchell (1989), 'F-bounded polymorphism for object-oriented programming', Proc. 4th Int. Conf. Func. Prog. Lang. and Arch., Imperial College London, September, 273-280.

[CCHO89b] P Canning, W Cook, W Hill and W Olthoff (1989), 'Interfaces for strongly-typed object-oriented programming', *Proc. 4th ACM Conf. Object-Oriented Lang., Sys. and Appl.*, 457-467.

[CHC90] W Cook, W Hill and P Canning (1990), 'Inheritance is not subtyping', *Proc. 17th ACM Symp. Principles of Prog. Lang.*, 125-135.

[CL91] L Cardelli and G Longo (1991), 'A semantic basis for Quest', *J. of Func. Prog., 1(4)*, 417-458.

[CM92] L Cardelli and J Mitchell (1992), 'Operations on records (summary)', *Proc. 5th Int. Conf. Math. Found. Prog. Lang. Semantics*, pub. *LNCS, 442*, Springer Verlag, 22-52.

[Cook89a] W Cook (1989), *A denotational semantics of inheritance*, PhD Thesis, Brown University.

[Cook89b] W Cook (1989), 'A proposal for making Eiffel type-safe', *Proc. 3rd European Conf. Object-Oriented Prog.*, 57-70; reprinted in *Computer Journal 32(4)*, 305-311.

[CP89] W Cook and J Palsberg (1989), 'A denotational semantics of inheritance and its correctness', Proc. 4th ACM Conf. Object-Oriented Prog., Sys., Lang. and Appl., 433-443.

[CW85] L Cardelli and P Wegner (1985), 'On understanding types, data abstraction and polymorphism', *ACM Computing Surveys, 17(4)*, 471-521.

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

[FGJM85] K Futatsugi, J Goguen, J-P Jouannaud and J Meseguer (1985), 'Principles of OBJ2', *Proc. 12th ACM Symp. Principles of Prog. Lang.*, 52-66.

[Ghel90] G Ghelli (1990), 'Modelling features of object-oriented languages in second-order functional languages with subtypes', *LNCS, 489*, 311-337.

[Gutt75] J W Guttag (1975), 'The specification and application to programming of abstract data types', *Technical Report CSRG-59, University of Toronto,* September.

[Gutt77] J W Guttag (1977), 'Abstract data types and the development of data structures', *Comm. ACM 20(6)*, 396-404.

[Jone86] C B Jones (1986), *Systematic Software Development Using VDM*, Prentice-Hall.

[Meye88] B Meyer (1988), *Object-Oriented Software Construction*, Prentice-Hall.

[Meye92] B Meyer (1992), *Eiffel: The Language*, Prentice-Hall.

[Omoh94] S M Omohundro (1994), *The Sather 1.0 specification*, International Computer Science Institute, Berkley CA.

[SOM93] C Szypersky, S Omohundro and S Murer (1993), 'Engineering a programming language: the type and class system of Sather', Technical Report TR-93-064, International Computer Science Institute, Berkley CA.

[SC92] A Simons and A Cowling (1992), 'A proposal for harmonising types, inheritance and polymorphism for object-oriented programming', *Dept. Comp. Sci. Research Report CS-92-13*, Univ. Sheffield, UK.

[SCBK86] C Schaffert, T Cooper, B Bullis, M Kilian and C Wilpolt (1986), 'An introduction to Trellis/Owl', *Proc. 1st ACM Conf. Object-Oriented Prog., Sys., Lang., and Appl.*, pub. *ACM Sigplan Notices, 21(11)*, 9-16.

[Scot76] D Scott (1976), 'Data types as lattices', *SIAM J. Computing, 5(3)*, 523-587.

[Simo94] A J H Simons (1994), *Exploring Object-Oriented Type Systems*, OOPSLA-94 Tutorial 28, ACM Press.

[Stoy77] J Stoy (1977), *Denotational Semantics: The Scott-Strachey Approach to Programming Language Theory*, MIT Press, Cambridge MA.