ELECTRONIC WORKSHOPS IN COMPUTING

Series edited by Professor C.J. van Rijsbergen

Paolo Atzeni and Val Tannen (Eds)

Database Programming Languages (DBPL-5)

Proceedings of the Fifth International Workshop on Database
Programming Languages, Gubbio, Umbria, Italy, 6-8 September 1995

Paper:

Investigating Extension Polymorphism

Richard Connor, Dharini Balasubramaniam and Ron Morrison

Published in collaboration with the
British Computer Society

©Copyright in this paper belongs to the author(s)

Investigating Extension Polymorphism

Richard Connor, Dharini Balasubramaniam and Ron Morrison

School of Mathematical and Computational Sciences,
University of St Andrews, St Andrews, Fife, Scotland KY16 9SS

Abstract

This paper examines a polymorphism mechanism which allows abstraction over evolving types. Although inclusion polymorphism is often quoted as being such a mechanism, this holds only for record types, for which constructor the inclusion relation matches the way types evolve in a long-lived system. This paper formalises the idea of evolution in the type space by the notion of extension, based upon both intuition and observation of evolving systems in a persistent programming environment. Extension is proposed as a type abstraction mechanism to allow smooth evolution in statically typed systems, and programming language mechanisms are introduced which maintain soundness and static checking in the presence of this abstraction. Extension over some function types is related to the concept of type matching, but a more general treatment is given as our interest is in extension over the whole type system.

1 Introduction

The major perceived uses of inclusion polymorphism in DBPLs are system evolution and simple polymorphism within programs. In the first instance this is because the extension of record types, by the addition of fields or the refinement of existing fields, is observed to be a common pattern of evolution. This activity of extension also corresponds to the notion of type inclusion. However the overloaded use of inclusion to encompass the refinement of type definitions has not always been well understood and the concepts of inclusion and extension, often in the joint guise of various forms of inheritance, have sometimes been confused. This confusion can lead to unsound or at best dynamically checked languages.

In this paper a new and more cautious approach to the problem of system evolution is introduced. Instead of attempting to adapt a subtyping mechanism to allow evolution, we investigate from first principles the way in which type definitions are typically changed by programmers in an evolving system. This is based mainly on experience in observing the evolution of applications in persistent programming languages [1, 2]. The most common evolutionary paradigm we define as *extension*, which is best summarised as the process of refining type definitions to form a more accurate model of the real system. The notion is formalised by the definition of an *extension relation*. The underlying hypothesis is that, by providing mechanisms which allow polymorphism over this relation, programmers may write code which will continue to be correctly (soundly and statically) typed as the types of the data over which it operates evolve.

This paper reflects the first step in an experiment of providing explicit language-level support for type evolution. The full experiment is designed to be structured in four main phases:

1. Define a formal model of common evolutionary activity
2. Derive appropriate language mechanisms
3. Implement such mechanisms, and provide them to users
4. Observe the reactive patterns of evolution, and iterate until stable

The investigation is currently positioned somewhere between the second and third phases of the first iteration. A set of mechanisms which preserve sound and static typing over extended types has been identified, and the mechanisms are described here. It is not yet clear how expressive they are, but as they allow at least some useful activity they are considered worthy of further investigation. In particular, they do not disallow any other polymorphism mechanism from co-existing in the same language, and so can only help in the quest for mechanisms to allow painless evolution in database programming languages.

It should perhaps be stressed that the investigation so far is confined to that of typing mechanisms that support the evolution process, and has not reached the stage of developing programming methodologies. Much of the code introduced as example appears baroque in structure; however it is important to first establish sound mechanism before programmer interfaces are considered.

This work was in progress at the same time as some of the more recent work on type matching, notably that by Cardelli and Bruce [3-5]. Type matching and extension share much common motivation: the motivation of matching is to provide sound typings for certain patterns of inheritance within recursive object types, and these patterns closely correspond to those of extension. However our requirement is more general, and we develop

extension over a whole type system. The matching systems of Cardelli and Bruce are outlined in a later section on related work.

2 The Evolution of Types

In this section the patterns for evolution are introduced for each constructor; these patterns are formalised by the extension relation defined in the following section.

For each of the major type constructors we examine the pragmatics of their evolution based upon the experience of building evolving applications in a persistent programming environment [6, 7]. To keep the discussion simple at this point we examine a core language with no other polymorphism mechanisms, and therefore consider only records, variants, locations and procedures. One point which should be made is that mutability, for the moment, is explicitly modelled in the type system. That is, an updatable location containing a value of type *A* is typed as *loc(A)*, rather than as *A*.

2.1 Records

Record type evolution is possibly the most commonly investigated, and patterns of record type evolution are relatively well established. Record types typically evolve by the addition of fields, or by the refinement of existing fields. Thus the type *Person* defined below as a record of a *name* of type **string** and an *address* of type *Address*

```
type Address is { street : string }
type Person is { name : string; address : Address }
```

might evolve through refinement of the data model to, for example

```
type Person2a is { name : string; address : Address; age : int }
```

or, by refinement of the definition of *Address*, to

```
type Address2 is { street : string; number : int }
type Person2b is { name : string; address : Address2 }
```

since *Address2* is an evolutionary refinement of *Address*.

If record types are the only types considered then evolutionary refinement is contained within the inclusion relation over types. Subtyping is often exemplified by record type subtyping; perhaps this is partly why the different concepts of system evolution (often encompassed in definitions of inheritance) and subtyping have sometimes been confused.

2.2. Variants

In practice, variant types have been observed to evolve in the same way as record types; that is, as the data model becomes more refined, variants tend to grow more branches. Again, the refinement can take place in two different ways; by the addition of a branch, or by the refinement of the type of one of the branches. For example,

```
type Herbivore is ...
type Competitor is [ horse : Herbivore; rider : Person ]
```

might evolve to

```
type Competitor2a is [ horse, pony : Herbivore; rider : Person ]
```

or, by refinement of the type of *rider*,

```
type Competitor2b is [ horse : Herbivore; rider : Person2a ]
```

Evolutionary refinement over variant types is partly at odds with the usual intuition of inclusion, where *Competitor2a* would be a supertype of *Competitor*. Notice however that *Competitor2b* would indeed be a subtype of *Competitor*, so when a variant type is evolved by refinement of one of its branches, rather than by the addition of a new branch, the extension and inclusion relations may coincide.

2.3. Locations

Locations may or may not be typed explicitly within a type system. Here we deal with typed locations, which is the more general case.

The type of a location in an evolving data model is simply a recursive application of the evolution relation to the type of the value contained within the location. Thus, for example

```
type AddressContainer is loc( Address )
```

may be refined by evolution to

```
type AddressContainer2 is loc( Address2 )
```

A further pattern of evolution which is frequently observed is for a component type which is not originally defined as updateable to become so. Thus, for example,

```
type AddressRecord is { address : Address }
```

may be refined to

```
type AddressRecord2 is { address : loc( Address ) }
```

These definitions of extension do not imply inclusion, although the fact that this discovery had been remade a number of times in the literature [8-10] shows that it may comprise a non-trivial departure from intuition. Acceptable rules for inclusion over typed locations are trivial subtyping only ($\text{loc}(A) \leq \text{loc}(A)$) in a language where locations are dereferenced by an explicit operation (e.g. Galileo [11]), which may be augmented by $\text{loc}(A) \leq A$ if location dereference is implicit (e.g. Quest [12]). Any more liberal subtype relation than this may lead to a loss of soundness unless the language algebra is changed to deal with it, which will lead to the loss of decidable static typechecking [13].

2.4. Procedures

The observation of evolving systems shows that procedure types evolve by the process of evolution of their parameter and result types. Thus the type

```
type GetAddrProc is proc( Person  $\rightarrow$  Address )
```

has the following among its extended forms:

```
type GetAddrProc2a is proc( Person  $\rightarrow$  Address2 )
type GetAddrProc2b is proc( Person2a  $\rightarrow$  Address )
type GetAddrProc2c is proc( Person2a  $\rightarrow$  Address2 )
```

It is thus seen that extension coincides with the concept of “covariant subtyping”. This term is used to describe inheritance between object classes where the types of method parameters become more specialised, this being contrary to the inclusion relation over functions [14].

The example often quoted is to do with cows, herbivores, plants, and grass. One motivation for inheritance is exactly the same as our motivation for evolution: the ability to refine definitions. However the notion of inheritance is also normally linked with inclusion polymorphism. If types other than records are abstracted over, the same mechanisms are not strictly applicable to both concepts.

If the notion of inheritance is allowed to correspond to that of specialisation, then the type of *Cow* in the following example may be inherited from the type of *Herbivore*:

```
type Plant is { rootDepth : real }
type Grass is { rootDepth, yield : real }

type Herbivore is { colour : string; eat : proc( Plant ) }
type Cow is { colour : string ; milkYield : real; eat : proc( Grass ) }
```

This gives a problem with static typechecking, in that a value denoted as a *Herbivore* may not necessarily be provided with any *Plant* parameter to eat, as the actual herbivore may be a *Cow*. In this case the *eat* method may require to access the *yield* field of its *Plant* parameter, which it will only have if it happens to be *Grass*. Thus dynamic checking may be required to ensure soundness.

Languages such as Eiffel [15] use this definition of inheritance; whilst Eiffel still allows subsumption among values of such classes static typechecking in general is lost, and Eiffel compilers may require to plant dynamic checks in some programs. However, the common opinion amongst Eiffel users seems to be that this is not problematic, such programs being rare, and the language is often quoted as being statically typed. A similar language, Sather [16], defines separate inclusion and inheritance hierarchies. The mechanism of type matching, discussed in a later section, is another mechanism used to address this problem.

3 Extension definition

Based on the intuitions in the previous section, the extension relation may now be defined. The relation is denoted by the symbol “ $\leftarrow$ ”, the notation $A \leftarrow B$ may be read as “ A is extended from B ”, meaning A may be formed, by extension, from B .

3.1 Reflection:

For any type t , we define $t \leftarrow t$. In fact the relation turns out also to be transitive and anti-symmetric, making it a partial ordering as might be expected.

3.2 Records:

$$\{l_1 : t_1, \dots, l_n : t_n\} \leftarrow \{l'_1 : t'_1, \dots, l'_m : t'_m\} \\ \text{iff } m \leq n \text{ and, for } i \text{ in } 1..m, l_i = l'_i, \text{ and } t_i \leftarrow t'_i$$

3.3 Variants:

$$[l_1 : t_1, \dots, l_n : t_n] \leftarrow [l'_1 : t'_1, \dots, l'_m : t'_m] \\ \text{iff } m \leq n \text{ and, for } i \text{ in } 1..m, l_i = l'_i, \text{ and } t_i \leftarrow t'_i$$

3.4 Locations

$$\text{loc}(p) \leftarrow \text{loc}(p') \\ \text{iff } p \leftarrow p'$$

$$\text{loc}(p) \leftarrow p$$

3.5 Procedures:

$$\text{proc}(p \rightarrow q) \leftarrow \text{proc}(p' \rightarrow q') \\ \text{iff } p \leftarrow p' \text{ and } q \leftarrow q'$$

4 Polymorphism Mechanisms

In this section the polymorphism mechanisms which allow static typing over extended forms of types are introduced. To keep the discussion simple, a language with no other polymorphism mechanism is considered. However there is no reason, as we will justify later, why the mechanisms described here could not coexist with others. In particular, an interesting possibility is for a language which has both extension and inclusion; there are some interesting interactions between these mechanisms, as will be outlined later.

We first introduce typings for extension types. To obtain polymorphism, it is necessary to introduce a mechanism which can cause expressions to be typed as “some type that is extended from T ”, where T is an arbitrary type in the language.

The most obvious way to achieve this is to introduce $\leftarrow$ as an explicit type constructor, allowing the syntactic form $\leftarrow T$ to signify extension. However, this introduces problems with static typing as the judgements $e1 : \leftarrow T$ and $e2 : \leftarrow T$ do not signify that $e1$ and $e2$ are semantically compatible. To overcome this we introduce an algebra which is precisely typed except where type is explicitly abstracted.

The major new typing to be introduced is based on the notion of an extension type variable. As well as the usual type and value environments, we introduce a third environment for these variables. This environment contains judgements of the form $t \leftarrow T$, where t is any identifier and T stands for any type, including types composed from other extension type variables. Extension type judgements are restricted to the form $e : t$, where t is a member of the extension variable environment, and two such expressions are deemed to be type equivalent if and only if they share the same type variable.

The type rules given in the rest of this section explicitly show these three environments. For example the following shows the rule for standard procedure typing. In this example, τ stands for the environment of type variables, ϖ for the environment of expression variables, and ε for the environment of extension variables.

$$\frac{\tau, \varpi; x : t_1, \varepsilon \vdash e : t_2}{\tau, \varpi, \varepsilon \vdash \text{proc}(x : t_1 \rightarrow t_2) ; e : \text{proc}(t_1 \rightarrow t_2)}$$

There are two ways of introducing extension type variables: explicitly, through the use of a form of bounded universal quantification, and implicitly, through the application of operations to expressions already typed as

extension variables. The former is introduced as a new language construct, and the variables are introduced explicitly by the programmer; in the latter the variables are not explicit in the program text, and are introduced implicitly by the typechecker.

4.1 Explicit extension variable introduction

Extension type variables may be introduced by a form of bounded universal quantification, based on the extension abstraction rather than on inclusion. Such forms appear in the language as, for example,

```
proc [  $t \leftarrow X$  ] (  $x : t \rightarrow t$  ) ; ...
```

It is worth restating that this type abstraction does not imply inclusion, unlike the similar syntactic form used for traditional bounded universal quantification. The abstraction does not therefore imply that the value denoted by x in the procedure body has all of the operations defined on X available; it has only those which are defined on any type which may be derived from X by extension. Operations available on denotations typed as extension variables will be explained later; in the meantime it should be noted that typing of the procedure bodies is very much more restrictive than for bounded universal quantification based upon inclusion.

The type rule corresponding to the introduction of a quantified procedure shows how the extension type environment is explicitly enriched, with the type of the procedure body being checked in the context of an extension type variable introduction:

$$\frac{\tau, \mathcal{W}; x:t_1, \varepsilon; t \leftarrow T \vdash e : t_2}{\tau, \mathcal{W}, \varepsilon \vdash \text{proc}[} t \leftarrow T](x : t_1 \rightarrow t_2) ; e : \text{proc}[} t \leftarrow T](t_1 \rightarrow t_2) }$$

Notice that the ordering of the three typechecking environments is not significant here: for example, the types t_1 and t_2 may involve instances of the extension variable t .

4.2 Implicit extension variable introduction

The full range of operations available over denotations typed as extension variables will be described later. For now we demonstrate the implicit variable introduction by means of an example based on a record type extension. As record types may be extended only by the addition of more fields, or the refinement of existing ones, the same dereference operations may be applied to extensions as may be applied to the base type. Therefore the following code should be well typed:

```
type Address is { street : string }
type Person is { name : string ; address : Address }

let derefAddr = proc [  $t \leftarrow \text{Person}$  ] (  $p : t$  )
begin
  let x = p.address
end
```

The reason for implicit variable introduction is explained by considering the type of the new variable x . The structure of the extension relation means that the type is not *Address*, but may in fact be any type which is extended from *Address*.

This is captured by the introduction by the typechecker of a new variable to the extension environment. The essence of this is shown by the following type rule:

$$\frac{\tau, \mathcal{W}, \varepsilon; t \leftarrow \{ \dots, l : T, \dots \} \vdash x : t}{\tau, \mathcal{W}, \varepsilon; t \leftarrow \{ \dots, l : T, \dots \}; \$\text{new} \leftarrow T \vdash x.l : \$\text{new}}$$

In fact the identifier generated by the typechecker captures the derivation of the new identifier also, so that other static instances of dereferences in the same scope will be type compatible. Thus the actual string generated within the typechecker will appear something like $\$t_l$, signifying that the new type variable has been implicitly created by the dereference of the l field of a denotation typed as the extension type variable t . As any denotation typed as an extension has available a statically known, finite set of operations, the new identifiers may be safely generated without ambiguity. The use of the $\$$ symbol simply ensures a distinction between implicitly and explicitly generated variables. This mechanism allows the correct typing of, for example, the following program:

```

type Address is { street : string }
type Person is { name : string ; address : loc( Address ) }

let shareAddr = proc[ t ← Person ]( p,q : t )
begin
  p.address := q.address
end

```

We introduce one further set of rules to increase the correct typings available over implicitly introduced quantifier variables. As currently defined, there is no typing for the following program:

```

type Address is { street : string }
type Person is { name : string ; address : Address }

let getName = proc[ t ← Person ]( p : t → string )
begin
  p.name
end

```

The program is semantically well-defined as, by inspection of the extension relation, the type `string` has no proper extensions defined; therefore any type extended from `string` (the derived type of the procedure body) must in fact be equal to `string`. For each type with this property we introduce a rule which effectively allows coercion from the extension variable to the type itself, for example:

$$\frac{\tau, \mathcal{W}, \varepsilon; t \leftarrow \text{string} \vdash e : t}{\tau, \mathcal{W}, \varepsilon \vdash e : \text{string}}$$

An important instance of this rule occurs over extension variables themselves:

$$\frac{\tau, \mathcal{W}, \varepsilon; s \leftarrow t; t \leftarrow T \vdash e : s}{\tau, \mathcal{W}, \varepsilon; t \leftarrow T \vdash e : t}$$

This rule allows the useful introduction related quantifier variables, such as:

```

type Address is { street : string }

let getAddr = proc[ s ← Address, t ← { address : s } ]( pers : t ; addr : s → s )
  if addr = pers.address then addr else pers.address

```

To type this procedure it is necessary to ascertain that the type of `pers.address` is equal to `s`. We are given `pers : t`, with `t ← { address : s }`; from the type rule for structure dereference over extension types, `pers.address` is typed as some fresh variable `new`, after `new ← s` is placed in the extension environment. The last rule given then allows the retyping of `pers.address` as `s`, and so the whole procedure can be usefully typed.

We now examine how these mechanisms may be used with respect to the other basic type constructors of the language. For each type constructor, we examine the operations allowed upon the abstracted form, depending upon its bound.

4.3 Records

As already discussed, the extension and inclusion relations are the same for record values, and all the operations available on the bound are therefore available on the abstraction. Normal record subtyping may therefore be achieved in almost exactly the same manner as with bounded universal quantification. Thus for example a procedure to dereference the `age` field of a value of any type extended from `Person` could be

```

let getName = proc[ t ← Person ]( p : t → string ); p.name

```

We can also write a procedure to dereference the `address` field of a person, typed as follows:

```

let getAddress = proc[ s ← Address, t ← { address : s } ]( p : t → s )
  p.address

```

This procedure is applicable to any type which extends from `person`, including where the `address` field is typed as a location; notice that the precise result type may be deduced from the calling context.

4.4 Variants

An abstracted variant type may belong to any branch of the bounding type, but also any other branch of the extended type, which is not known statically. A straightforward solution to this is to restrict the use of abstracted variant values to within a multi-branch case project statement, which is terminated by a default case.

```
project X as
  a : ...
  b : ...
default : ...
```

Based on experience in Napier88 [17] this is how variants are used in an evolving system. In most cases semantic knowledge of the variant value ensures that the default branch does not get executed even when the variant type is not abstracted over. The only pragmatic difference between the methodology proposed here and that currently used is that procedure written with the extension polymorphism in mind would not have to be recompiled in the context of the changed type definitions.

The type of the projected variant is treated in the same way as the result of a structure dereference operation, again due to the structure of the extension relation.

4.5 Locations

Updates to locations may only be allowed when the right and left hand sides of the assignment statement may be statically deduced to be the same type. The result of dereferencing an abstracted location is given as follows (where ! is used to denote the dereference operation, which may be implicit or explicit in the language)

$$\frac{\tau, \varpi, \varepsilon; t \leftarrow \text{loc}(T) \vdash e : t}{\tau, \varpi, \varepsilon; \$_t_deref \leftarrow T \vdash !e : \$_t_deref}$$

Notice again how the mechanical creation of implicit extension variable identifiers allows type equivalence to be deduced with different static instances of dereference.

4.6 Procedures

```
proc[ t ← proc(X → Y) ]( x : t ) ; ...
```

The value x in this context cannot in general be applied! The only operation on the procedure is application; all that is known about its parameter type is that it is extended from X . The only context in which the procedure could therefore safely be applied is when there are no extensions of X , for example if X is a base type, in which case the form of the quantified procedure is not particularly useful.

Extended procedure types may be useful when related quantifiers are used, as in the following example:

```
let eatIt = proc[ herb ← { eat : proc( p ) }, p ← plant ]( h : herb; veg : p )
  h.eat (veg)
```

which may be applied as

```
eatIt[ Herbivore, Plant ]( aHerbivore, aPlant )      ! or
eatIt[ Cow, Grass ]( aCow, aBitOfGrass )             !, but not

eatIt[ Cow, Plant ]( aCow, aPlant )
```

Notice that *eatIt* can't be applied to *aHerbivore* and *aBitOfGrass* either, although this would be semantically correct; to achieve this the type of the field *eat* has to be defined as a quantified procedure itself.

4.7 Quantified procedures

Having introduced a new type constructor, the quantified procedure, it is necessary to revisit the definition of the extension relation. A new rule is introduced:

```
proc[ t ← X ]( p → q )  ←      proc[ t ← Y ]( p' → q' )
    iff X ← Y, p ← p' and q ← q'
```

It appears at this point that the only useful abstraction by extension over these forms is through related quantifiers as above, although this is a matter for further investigation.

The rule for application of quantified procedures is as follows:

$$\frac{\tau, \bar{w}, \varepsilon \vdash e_1 : \mathbf{proc}[t \leftarrow T](t_1 \rightarrow t_2) \quad \tau; t:T_{app}, \bar{w}, \varepsilon \vdash e_2 : t_1 \quad \tau, \varepsilon \vdash T_{app} \leftarrow T}{\tau; t:T_{app}, \bar{w}, \varepsilon \vdash e_1[T_{app}](e_2) : t_2}$$

Notice that the application is only well typed if the actual parameter is typed as the formal parameter with instances of the extension type variable consistently substituted by the specialisation type, rather than by some abstraction of the specialisation type. This is of particular importance with respect to the typing of applications of procedures with related quantifiers; notice, for example, the consequence on typing the following procedure applications:

```

type address1 is { street : string }
type address2 is { street : string; number : int }
type person1 is { name : string; addr : loc( address1 ) }
type person2 is { name : string; addr : loc( address2 ) }

let aProc = proc[ s ← address1, t ← { addr : loc( s ) } ]( a : t ; b : s → t ) ; ...

let a = aProc[ address1, person1 ]( anAddress1, aPerson1 ) !* correctly typed
let a = aProc[ address2, person2 ]( anAddress2, aPerson2 ) !* correctly typed
let b = aProc[ address1, person2 ]( anAddress1, aPerson2 ) !* incorrect

```

5 Soundness and Decidability

We have sketched a proof of soundness for the system described, based upon the set of type rules from which the examples given in this paper are taken. We assume soundness for the core language without extension, and use the following observations to guide the proof through the added rules:

1. the explicit introduction of extension type variables (through bounded quantification forms) is matched exactly in the type rules
2. the rules which allow the application of inbuilt operations to denotations typed as extension variables are based on the structure of the extension relation, therefore it is easy to prove that these applications have well-defined meanings
3. the result of applying these inbuilt operations are themselves typed as extension variables; again case analysis over the operations and the structure of the extension relation shows the validity of this
4. the implicit variable name introduction is justified by the uniqueness of generated names within a context, shown by considering all valid operations for each type constructor
5. the promotion of an extension type to a real one, where the real one has no proper extensions, is clear; that a particular type has this property is justified by the structure of the extension relation
6. the application rule for quantified procedures simply ties together the typing of the procedure operand and result types

Therefore soundness is a relatively clear property of the system. We conjecture that the type judgements made by the system are relatively clear to programmers, even although the full set of type rules is perhaps rather complex.

Although the investigation so far has been in the context of a language without any other form of polymorphism, it is believed that the explicit nature of type abstraction in the system prevents any harmful interaction with other polymorphism mechanisms (with the exclusion of any based upon the mechanism of type inference, such as ML-style parametric polymorphism.) In particular, the operation-based justification of soundness over individual type constructors give a clear intuition that it may happily co-exist with implicit subtyping mechanisms based on inclusion.

We have not addressed the problem of decidable typechecking for this system. Indeed the system of type rules we have produced is rather more complex, and less obviously suited to an algorithmic conversion, than other systems which are known to be undecidable. We have written an algorithm which has, so far, always terminated; however, we understand the dangers of relying on intuition in such cases.

6 Interaction with inclusion polymorphism

There are some interesting interactions between the system outlined above and one which has also inclusion polymorphism based on subsumption. To give an idea of what interactions are possible, the concepts of *positive* and *negative* types are first introduced. These describe subsets of the type space which have the following properties:

positive = {A ∈ Type | X ← A implies X ≤ A} (where ≤ is the subtype relation)
 negative = {A ∈ Type | X ← A implies A ≤ X}

For example,

```
type Elephant is { a : int; b : { c : bool ; d : real } }
```

is positive, as any type extended from it is also a subtype of it, and

```
type Castle is [ a : int; b : [ c : bool ; d : real ] ]
```

is negative.

Notice that these sets, which are not disjoint, both include many non-trivial members. It is relatively straightforward to characterise the notions of positive and negative by examination of the combination of the structures of the inclusion and extension relations over each type constructor. The result is that some statically known extension relationships may be used to imply inclusion relationships, and therefore the application of the subsumption rule.

To bring the concept into the mechanism of the typechecker, we can introduce new rules into a system which already contains a subsumption rule based on the inclusion relation. The first is simple, and allows us to coerce a positive extension variable to the type it abstracts over:

$$\frac{\tau, \overline{\omega}, \varepsilon; t \leftarrow T \vdash e : t \quad T \text{ positive}}{\tau, \overline{\omega}, \varepsilon \vdash e : T}$$

The use of negative types is rather more subtle, and is restricted to the application of procedures whose type is abstracted over by extension:

$$\frac{\tau, \overline{\omega}, \varepsilon; t \leftarrow \text{proc}(S \rightarrow T) \vdash e_1 : t \quad \tau, \overline{\omega}, \varepsilon \vdash e_2 : S \quad S \text{ negative}}{\tau, \overline{\omega}, \varepsilon; \text{new} \leftarrow T \vdash e_1(e_2) : \text{new}}$$

The use of these two rules in combination allows the typing of a new class of procedures; for example the following is well-typed in a system with these rules as well as subsumption:

```
let proc[ t ← proc( Castle → Elephant ) ]( aProc : t ; c : Castle → Elephant )
  aProc( c )
```

This part of the typechecker has not yet been implemented, and the expressive power of this combination is not yet completely clear as it becomes a substantial task to decide whether programs are well typed without some mechanical help!

7 Related Work

Some of the recent research on type matching in object oriented languages is strongly related to the work presented here. Although the primary concern of type matching is to do with recursive object types, the motivation for the work is the same as that behind extension polymorphism, namely the failure of subtyping to capture the refinement, by specialisation, of types. In object oriented languages, this means that type safe inheritance of binary methods by subclassing is not possible. The concept of matching was introduced to address this problem.

Abadi and Cardelli present an investigation into the formal semantics of subtyping and matching in [3]. To explain matching, the notion of protocol, which characterises the interface of an object, is introduced. A protocol extension relation is then defined as

$$v_i : B_i \text{ } i \in I, m_j^+ : C_j\{\mathbf{Self}\} \text{ } j \in J \text{ extends } v_i : B_i \text{ } i \in I', m_j^+ : C_j'\{\mathbf{Self}\} \text{ } j \in J' \\ \text{if } C_j\{\mathbf{Self}\} <: C_j'\{\mathbf{Self}\} \text{ for all } j \in J', \text{ with } I' \subseteq I \text{ and } J' \subseteq J$$

If the protocol of A extends the protocol of B then A matches B . Two interpretations of this relation are higher-order and F-bounded subtyping. Polymorphism is provided by match-bounded quantification which complements subtype-bounded quantification.

In PolyTOIL [4], occurrences of **self** in the bodies of methods refer to the receiver of the message, and the keyword **MyType** denotes the external type (where instance variables are hidden) of **self**. Its meaning changes automatically during subclassing to refer to the new object type, just as the meaning of **self** changes to correspond to an element of the subclass. **MyType** plays two distinct but related roles. In an object, **MyType** is simply the type of that object. In classes, **MyType** will have a flexible meaning. The definition of subtyping in PolyTOIL is :

$$\text{ObjectType}\{m_j : T_j\} \text{ } 1 \leq j \leq m <\# \text{ObjectType}\{m_i : T_i\} \text{ } 1 \leq i \leq n \\ \text{if } \begin{array}{l} 1) \text{ } n \leq m \text{ and for each } i \leq n, T_i <: T_i \\ 2) \text{ no method } m \text{ of } T \text{ has a contravariant occurrence of } \mathbf{MyType} \end{array}$$

The matching relation between two object types is defined by

$$\begin{aligned} \text{ObjectType}\{m_j : T'_j\} \ 1 \leq j \leq m <\# \text{ObjectType}\{m_i : T_i\} \ 1 \leq i \leq n \\ \text{if } n \leq m \text{ and for each } i \leq n, T'_i <: T_i \end{aligned} \quad (D1)$$

No assumption is made on the meaning of **MyType** while determining whether two method types are in the subtype relation. A subclass will always match its superclass but is not necessarily a subtype; subtypes always match. If the only occurrences of **MyType** in method types are covariant then the two types match iff they are subtypes. Methods of an object type are type checked under the assumption that **MyType** matches that object type. Polymorphism is provided by subtyping and bounded matching, which is a mechanism for restricting type parameters.

An ultimate aim of the PolyTOIL project is to replace subtyping with generalised matching [5]. In order to make the matching relation primitive, its definition is refined to

$$\text{ObjectType}\{m_i : T_i\} \ 1 \leq i \leq k <\# \text{ObjectType}\{m_j : T_j\} \ 1 \leq j \leq n \quad \text{iff } n \leq k \quad (D2)$$

All uses of subtyping in parameters which are of object type are replaced by polymorphic functions using bounded matching. A new constructor $\#$ is introduced to simplify the notation such that if $a : S$ and $S <\# T$ then $a : \#T$. This also allows a form of subsumption: if $S <\# T$ and $a : \#S$ then $a : \#T$. It is instructive to compare this rule with that given in Section 4.2 for extension variables.

The language Tool [18] has a definition of matching which is equivalent to D1 in the PolyTOIL system. An object type A matches another object type B (the syntax $A <*: B$ is used) if they are subtypes under the assumption that the corresponding **Self** types are equal. The programmer has explicit control over the way subclasses relate to superclasses, in subtyping, matching or equivalence relations. Tool provides bounded quantification with subtyping or matching.

8 Conclusions and Future Work

We believe that we have demonstrated the feasibility of a new class of polymorphism, based on extension, as a mechanism to allow smooth evolution in DBPLs. We have included the framework of a soundness proof for the system. The development of the system over recursive typings is in progress, and is not reported in this document.

The expressivity of the system is not clear and demands further investigation; however we have shown that at least some new useful programming activity is possible. In particular, the interaction with inclusion demands further investigation.

As with all new mechanisms there is a danger of complicating a language to a point where it is not worthwhile. The concepts introduced are not particularly difficult, but some arguably obscure new syntax has been introduced to express them. A further avenue of investigation is to simplify the presentation of the new types to a programmer, or possibly even to introduce an inference mechanism to work over them.

9 Acknowledgements

Richard Connor is supported by EPSRC Advanced Research Fellowship B/94/AF/1921. Dharini Balasubramaniam is partly supported by a UK Government ORS award.

10 References

- [1] Morrison, R., Connor, R.C.H., Cutts, Q.I., Kirby, G.N.C. & Stemple, D. "Mechanisms for Controlling Evolution in Persistent Object Systems". *Journal of Microprocessors and Microprogramming* 17, 3 (1993) pp 173-181.
- [2] Connor, R.C.H., Cutts, Q.I., Kirby, G.N.C. & Morrison, R. "Using Persistence Technology to Control Schema Evolution". In *Proc. 9th ACM Symposium on Applied Computing*, Phoenix, Arizona (1994) pp 441-446. Available from <http://www-fide.dcs.st-and.ac.uk>
- [3] Abadi, M. and Cardelli, L., "On Subtyping and Matching", *Proc. Ecoop95*, Springer Verlag (1995).
- [4] Bruce, K.B., Schuett, A. & van Gent, R., "PolyTOIL: a Type-safe Polymorphic Object-Oriented Language", *Proc. Ecoop95*, Springer Verlag (1995).
- [5] Bruce, K.B., "Typing in Object-Oriented Languages: Achieving Expressibility and Safety" (November 1995), internet communication from <http://www.cs.williams.edu/~kim/>
- [6] Atkinson, M.P., Buneman, O.P. & Morrison, R. "Delayed Binding and Type Checking in Database Programming Languages". *Universities of Glasgow and St Andrews Technical Report PPRR-45-87* (1987).

- [7] Atkinson, M.P. & Morrison, R. "Orthogonally Persistent Object Systems". VLDB Journal 4, 3 (1995) pp 319-401.
- [8] Jones, A.K. & Liskov, B. "A Language Extension for Expressing Constraints on Data Access". Communications of the ACM 21, 5 (1978) pp 358-367.
- [9] Albano, A. "Type Hierarchies and Semantic Data Models". In **ACM SIGPLAN 83: Symposium on Programming Language Issues in Software Systems** (1983) pp 178-186.
- [10] Wegner, P. & Zdonik, S.B. "Inheritance as an Incremental Modification Mechanism or What Like is and Isn't Like". In **Lecture Notes in Computer Science 322**, Springer-Verlag, Proc. ECOOP'88 - European Conference on Object-Oriented Programming (1988) pp 55-77.
- [11] Albano, A., Cardelli, L. & Orsini, R. "Galileo: a Strongly Typed, Interactive Conceptual Language". ACM Transactions on Database Systems 10, 2 (1985) pp 230-260.
- [12] Cardelli, L. "Typeful Programming". DEC Technical Report 45 (1989).
- [13] Connor, R.C.H., McNally, D.J. & Morrison, R. "Subtyping and Assignment in Database Programming Languages". In **3rd International Workshop on Database Programming Languages**, Kanelakis, P. & Schmidt, J.W. (ed), Morgan Kaufmann, Proc. 3rd International Workshop on Database Programming Languages, Nafplion, Greece (1991). Available from <http://www-fide.dcs.st-and.ac.uk>
- [14] Cardelli, L. "A Semantics of Multiple Inheritance". In **Lecture Notes in Computer Science 173**, Kahn, G., MacQueen, D.B. & Plotkin, G. (ed), Springer-Verlag, Proc. International Symposium on the Semantics of Data Types, Sophia-Antipolis, France (1984) pp 51-67.
- [15] Meyer, B. **Eiffel - the Language** Prentice-Hall (1992).
- [16] <http://www.icsi.berkeley.edu:80/Sather/>
- [17] Morrison, R., Brown, A.L., Connor, R.C.H. & Dearle, A. "The Napier88 Reference Manual". Universities of Glasgow and St Andrews Technical Report PPRR-77-89 (1989). <http://www-fide.dcs.st-and.ac.uk>
- [18] Gawecki, A. and Matthes, F., "TooL: A Persistent Language Integrating Subtyping, Matching and Type Quantification", FIDE Technical Report no. FIDE/95/135, University of Hamburg