21. Wegner, P. and Zdonik, S. B. "Inheritance as an Incremental Modeification Mechanism of What Like is and Isn't Like", *Lecture Notes in Computer Science*, vol 322, Springer - Verlag, pp. 55 - 77, 1988.

4. Atkinson, M. P., Bailey, P. J., Chisholm, K. J., Cockshott, W. P. and Morrison, R. "An Approach to Persistent Programming", *The Computer Journal*, vol 26, 4, pp. 360 - 365, 1983.

5. Atkinson, M. P. and Morrison, R. "Types, Bindings and Parameters in a Persistent Environment", *Data Types and Persistence*, Springer - Verlag, pp. 3 - 20, 1985.

6. Cardelli, L. "A Semantics of Multiple Inheritance", *Lecture Notes in Computer Science*, vol 173, Springer - Verlag, pp. 51 - 67, 1984.

7. Cardelli, L. "Amber", AT&T Bell Labs., Tech. Report, 1985.

8. Cardelli, L. "Typeful Programming", DEC Systems Research Centre, Palo Alto, 45, 1989.

9. Cardelli, L. and MacQueen, D. "Persistence and Type Abstraction", *Data Types and Persistence*, Springer-Verlag, Heidelberg, pp. 31 - 42, 1988.

10. Cardelli, L. and Wegner, P. "On Understanding Types, Data Abstraction and Polymorphism", *Association for Computing Machinery Computing Surveys*, vol 17, 4, pp. 471 - 523, 1985.

11. Connor, R. C. H. "Types and Polymorphism in Persistent Programming Systems", Ph. D. Thesis, St Andrews, 1990.

12. Connor, R. C. H., Brown, A. B., Cutts, Q. I., Dearle, A., Morrison, R. and Rosenberg, J. "Type Equivalence Checking in Persistent Object Systems", *Implementing Persistent Object Bases*, Morgan Kaufmann, pp. 151 - 164, 1990.

13. Connor, R. C. H., McNally, D. J. and Morrison, R. "Subtyping and Assignment in Database Programming Languages", *Proc. 3rd International Workshop on Database Programming Languages,1991*, Morgan Kaufmann, To appear.

14. Jones, A. K. and Liskov, B. "A Language Extension for Expressing Constraints on Data Access", *Communications of the Association for Computing Machinery*, vol 21, 5, pp. 358 - 367, 1978.

15. Leroy, X. and Cardelli, L. "Using the Dot Notation for Abstract Data Types", DEC Systems Research Centre, Palo Alto, Report 56, 1990.

16. Matthews, D. C. J. "Poly Manual", Unviversity of Cambridge, U.K., Technical Report 65, 1985.

17. Milner, R. "A Theory of Type Polymorphism in Programming", *Journal of Computer and System Sciences*, vol 17, pp. 348 - 375, 1978.

18. Mitchell, J. C. and Plotkin, G. D. "Abstract Types have Existential Type", *Association for Computing Machinery Transactions on Programming Languages and Systems*, vol 10, 3, pp. 470 - 502, 1988.

19. Morrison, R., Brown, A. L., Connor, R. C. H. and Dearle, A. "The Napier88 Reference Manual", University of St Andrews, PPRR-77-89, 1989.

20. Ohori, A., Tabkha, I., Connor, R. C. H. and Philbrow, P. "Persistence and Type Abstraction Revisited", *Implementing Persistent Object Bases*, Morgan Kaufmann, pp. 141 - 153, 1990.

6.4 Dynamic type equivalence rules

A common model of persistence in programming languages relies upon the use of an infinite union type for late binding to previously prepared program and data[4]. Examples of such types are Napier88's type **any**[19] and Amber's type **dynamic**[1, 7, 8]. There are problems associated with the injection and projection of open quantifier types with such dynamic union types[9], which have up to now been only partially addressed[11, 12, 20]. Since these open quantifiers are bounded they will suffer from at least all of these problems, and possibly more, and provide yet another active research area.

7 Conclusions

This paper proposed a type system with an accurate specification of subtyping through bounded quantification. Both bounded universal and bounded existential quantification were described and shown to preserve static type checking while retaining the expressiveness of other forms of subtyping. One advantage of the proposed system is that the well known incompatibility between static checking and mutability in subtyping systems that preserve identity has been avoided.

The research into this type system is at an early stage and its expressibility is not clear. It is proposed to build this type system on top of Napier88 to evaluate it in the context of a persistent programming environment.

Work on producing a type inference algorithm and its associated failure difficulties will also be pursued.

Acknowledgements

This work was supported by SERC Grant GR/F 28571 and ESPRIT II Basic Research Action 3070 - FIDE. The authors would also like to thank Craig Baker, Quintin Cutts, Graham Kirby and Dave Munro for proof reading and making many suggestions for improving the paper. Richard Connor is supported by SERC Postdoctoral Fellowship B/91/RFH/9078, and Ron Morrison is currently a Visiting Fellow at the Australian National University.

References

1. Abadi, M., Cardelli, L., Pierce, B. C. and Plotkin, G. "Dynamic Typing in a Statically Typed Language", *Association for Computing Machinery Transactions on Programming Languages and Systems*, vol 13, 2, pp. 237 - 268, 1991.
2. Albano, A. "Type Hierarchies and Semantic Data Models", *ACM SIGPLAN 83: Symposium on Programming Language Issues in Software Systems*, San Francisco, pp. 178 - 186, 1983.
3. Albano, A., Dearle, A., Ghelli, G., Marlin, C., Morrison, R., Orsini, R. and Stemple, D. "A Framework for Comparing Type Systems for Database Programming Languages", *Database Programming Languages*, Morgan Kaufmann, pp. 170 - 178, 1989.

$A : \exists t \leq X.t$

$\exists t \leq X.(A : t)$

It would appear that, for any type X, each of the three types shown above may all be safely related to each other by a subtype relation. This would have the rather strange result that the subtype relation would no longer represent a partial ordering. Although this may seem unusual, it is not necessarily untenable, and indeed there do exist languages where the subtype relation is not a partial ordering[16].

6.3 Type inference

One disadvantage of the system as described is that it contains a great deal of syntactic noise. It is therefore interesting to consider a system of type inference in which all locations are inferred with closed existentially quantified types and all procedures are inferred to be bounded universally quantified procedures.

```
type society is {president : person}

let newPresident = proc (s : society ; p : person)
  s.president := p

newPresident (aSociety, aPerson)
newPresident (aSociety, aStudent)
newPresident (aStudentSociety, aStudent)

!* only this call would fail statically *!
newPresident (aStudentSociety, aPerson)
```

Figure 7 A Full Inference System

The *newPresident* example could then be written as in Figure 7, without the syntactic noise. The rather complex procedure shown in Section 4 with its related quantifiers could be inferred from this textual form, as this is the only way the assigment statement could be correctly typed. Of the four calls shown, the correct specialisations and widening operations could also all be inferred for the three good calls, but the one bad call would cause a static type error.

It is not clear, however, how far the inference could be taken with such a system, nor is it clear what further loss of flexibility may be encountered by the removal of concrete type judgements from the system.

One further difficulty of using type inference to hide these complicated types from the user is how to explain type errors.

Finally it should be possible to devise a typing system where the subtyping may only be used where explicitly stated. Other uses of type information would be exact. Thus the complicated types would be restricted to such cases.

```
let temp := widen aSociety to ∃t ≤ society.t
temp := widen aFraternity to ∃t ≤ society.t

temp.president := widen aPerson to ∃t ≤ person.t
```

is not allowed, as the expression *temp.president* is given an open quantifier type, and is not compatible with the closed quantifier type on the right hand side of the assignment.

6 Further research topics

The type system which has been introduced is at a first stage of research. The only claim made is that it is able to accurately express the types of computations similar to those given in our examples, and that it is sound. The second of these claims indeed still lacks a formal proof. It is therefore proposed only as a possible contender to allow the safe and static combination of subtyping and assignment without the introduction of mutability constraints in a subtype relation. Any further claim will require some more investigation of the system. In particular some of the first areas to be investigated are given below.

6.1 Expressibility

As always the extra degree of static safety gained by this system will have a drawback in a loss of flexibility. That is, the class of incorrect programs that has been ruled out is accompanied by a class of correct programs. Exactly what programs are within this class is not yet clear.

One restriction for example is caused by the dereference rule for values of a closed existentially quantified type. This would disallow, for example, the following perhaps reasonable program:

```
let temp = widen aSociety to ∃t ≤ society.t
temp.president := widen aPerson to ∃t ≤ person.t
```

One possible extension to the system would be related existential quantifiers, which may potentially be used to allow safe assignment within such contexts. Another possible approach would be the introduction of a construct similiar to the open clause identified by Mitchell and Plotkin[18], or its avoidance by the techniques discussed in[15].

6.2 Subtyping rules

The subtyping rules given above are based on intuition and pragmatic requirements. However, it is also believed that they are a subset of the possible rules. Discounting assignment, the same operations are available on the value denoted by *A* for any of the following type judgements:

A : X

$$\frac{A : X,\ B : Y,\ X\ \mathbf{R}\ Y}{A := B \text{ legal}}$$

$$\frac{A : \exists t \leq X.t,\ B : \exists t \leq Y.t,\ X\ \mathbf{R}\ Y}{A := B \text{ legal}}$$

$$\frac{\exists t \leq X.(A,B : t)}{A := B \text{ legal}}$$

5.4 Justification of soundness

The soundness of the proposed system relies upon the combination of the substitution compatibility rules and the dereference rules for types which are abstracted over. Types which are abstracted over within universally quantified procedures are represented as open existential quantifiers, as is the type of any value obtained by dereference of such a value. This means that for an assignment to be allowed statically, there must exist a type judgement which asserts that the values on both sides of the assignment are typed as abstractions of the same type. The dereference rule for open quantifiers with such a judgement allows the compatibility of such values to be judged recursively. In the example of *newPresident* shown earlier

```
let newPres = proc [u ≤ person ; t ≤ {president : u}] (s : t ; p : u)
  s.president := p
```

these rules are used to correctly type the assignment.

Free assignment is allowed among locations and values sharing the same closed quantifier types, but this is made safe by the dereference rule for such types, which judges the dereferenced fields to have open quantifier types, and therefore allows assignment to these fields again only where they may be statically deduced to share the same type. Therefore the dangerous program

```
let temp := aSociety
temp := aStudentSociety
temp.president := aPerson              ! fails here
```

which causes loss of soundness in a naïve subtyping system is not possible to write in the bounded quantification system. A program such as

```
let x = proc [t ≤ S, u ≤ {x : t}](A : t ; B : u)
  begin
      !*    ∃t ≤ S.(∃u ≤ {x : t}.(B : u) ; A, B.x : t)          *!
```

Closed quantifier dereference

The dereference of a field within a value of a closed quantifier type results in a value whose type is an open quantifier, as shown by the following rule:

$$\frac{A : \exists t \leq \{x : X\}.t}{\exists t \leq X.(A.x : t)}$$

Open quantifier dereference

The dereference of a field within a value of an open quantifier type results in a value whose type is another open quantifier. However, if two values are known to be of the same open quantifier type, then their respective fields may also be determined to be of the same open quantifier type:

$$\frac{\exists t \leq \{x : X\}.(A : t)}{\exists s \leq X.(A.x : s)}$$

$$\frac{\exists t \leq \{x : X\}(A,B : t)}{\exists s \leq X.(A.x, B.x : s)}$$

5.2 Subtype relation

The following straightforward additions are made to the subtype relation to incorporate the new type judgements:

- Each concrete type is a subtype of its corresponding closed quantifer:

$$X \leq \exists t \leq X.t$$

- Each open quantifier is a subtype of its corresponding concrete type:

$$\frac{\exists t \leq X.(A : t)}{\text{typeof } (A) \leq X}$$

- Closed quantifier types are related according to their corresponding concrete types

$$\frac{X \leq Y}{(\exists t \leq X.t) \leq (\exists t \leq Y.t)}$$

- Open quantifier types are related according to their corresponding concrete types

$$\frac{X \leq Y,\ \exists t \leq X.(A : t),\ \exists s \leq Y.(B : s)}{\text{typeof } (A) \leq \text{typeof } (B)}$$

5.3 Assignment compatibility

Assuming that a type equivalence relation **R** is already defined over the concrete type system, the only rules for compatibility at assignment are as follows:

5.1.1 Closed quantifiers:

Declaration

The type ∃t ≤ X.t may be defined within the programming language's type algebra. For example

type society **is** {president : ∃t ≤ person.t}

Widening

A value of a closed quantifier type may be created by using the widen operation. This operation preserves identity and its only effect is a type coercion.

$$\frac{A : T,\ T \leq S}{(\textbf{widen}\ A\ \textbf{to}\ \exists t \leq S.t) : \exists t \leq S.t}$$

For the uninitiated the above type judgement should be read as: if *A* is of type *T* and *T* ≤ *S* then the statement ***widen*** *A* ***to*** *∃t ≤ S.t* is of type *∃t≤ S.t*

5.1.2 Open quantifiers

Bounded universal quantification application

Within the context of a bounded universally quantified procedure, a value of the type of a universal quantifier may be judged with an open existential judgement as in the case of the following rule:

$$\frac{A : T,\ T \leq S}{\exists t \leq S.(A : t)}$$

```
let x = proc [t ≤ S] (A : t)
  begin
      !*    ∃t ≤ S.(A : t)  *!
```

The static type checking rule ensures that the actual parameter is a subtype of the formal. Furthermore, if the universal quantification shows two values to be the same type, this may be reflected by an open existential judgement as a case of the following rule:

$$\frac{A, B : T,\ T \leq S}{\exists t \leq S.(A, B : t)}$$

```
let x = proc [t ≤ S] (A, B : t)
  begin
      !*    ∃t ≤ S.(A, B : t)  *!
```

The static typechecking of procedure application ensures that at any call the actual parameters corresponding to *A* and *B* are the same type and that type is a subtype of *S*.

The use of related quantifiers leads to nested open existential judgements:

evaluated to a value. Every value and every denotation has a single associated type. The type of a value is the type with which it was created and the type of a denotation is the type statically associated with that denotation. The type of a denotation may therefore be different from the type of the value obtained by its evaluation. A type judgement is an expression in a meta-language which associates a denotation with a type.

To describe the type rules of the system outlined above two forms of existentially quantified type judgement will be introduced. The first, which will be referred to as a closed quantifier judgement, is used to model the bounded existential quantification described in Section 4. For example

$$P : \exists t \leq X.t$$

may be read as "the value denoted by *P* is of a type *t* which is bounded by *X*".

The second will be referred to as an open quantifier judgement and allows the denotation as well as the type to be quantified over. An example of this is

$$\exists t \leq X.(P : t)$$

which may be read as "there exists a type *t* bounded by *X* such that the value denoted by *P* is of type *t*". These judgements will be used to model both bounded universal quantification and the type of free quantifiers as introduced in Section 3.

Although both type judgements offer similar information about the type of the value denoted by *P*, there is an important difference between them. This is that the closed quantifier judgement means that the evaluation of *P* may result in a value of any number of different types, so long as each of these is bounded by *X*. A consequence of this, if *P* denotes a location, is that values of different types may be assigned to it, so long as each type is bounded by *X*.

The open quantifier, however, means that *P* may evaluate to a value of only a single type, and that this type is bounded by *X*. This means that if *P* denotes a location values of different types bounded by X may not be assigned to it.

The open quantifier judgement may be extended to assert that two different denotations refer to values of the same type. For example, notice the difference between the following two judgements:

$$P, Q : \exists t \leq X.t$$

$$\exists t \leq X.(P, Q : t)$$

The first of these states that *P* and *Q* both denote values whose type is bounded by *X*. There is no relative information however about these two types. The second judgement states that there exists a single type *t* bounded by *X* such that both *P* and *Q* are of type *t*. The types of the values denoted by *P* and *Q* may be different under the first judgement, but not under the second. In the presence of such judgements assignment of values to locations judged as the same open quantifier type is freely allowable.

5.1 Quantifier introduction

The above type judgements may be introduced as follows:

Here the reserved word **widen** is introduced to perform a type widening operation to emphasise the fact that the type ∃t ≤person.t is a supertype of *person*. The **widen** operation may succeed as long as the value fits the bounded existential signature; for simple bounded types of this form this is true so long as the value is a subtype of the quantifier bound.

Using the existential types the example given in Figure 2 can now be extended to more legal statically typed cases. This is shown in Figure 6.

```
type person is {name : string}
type student is {name : string ; matricNo : int}
type eSociety is {president : ∃t ≤ person.t}
type eStudentSociety is {president : ∃t ≤ student.t}

let newPresident := proc [P ≤ (∃t ≤ person.t), S ≤ {president : P}] (s : S ; p : P)
                        s.president := p

let anESociety := eSociety (aPerson)
let anEStudentSociety := eStudentSociety (aStudent)

! make a person a society's president
newPresident [∃t ≤ person.t, eSociety] (anESociety,
                                        widen aPerson to ∃t ≤ person.t)

! make a student a society's president
newPresident [∃t ≤ person.t, eSociety] (anESociety,
                                        widen aStudent to ∃t ≤ person.t)

! make a student a student society's president
newPresident [∃t ≤ student.t, eStudentSociety] (anEStudentSociety,
                                        widen aStudent to ∃t ≤ student.t)

! make a person a student society's president
! This is illegal and will be caught at compile time
newPresident [∃t ≤ person.t, eStudentSociety] (anEStudentSociety,
                                        widen aPerson to ∃t ≤ person.t)
```

Figure 6 Existentially Quantified Types as Bounds

Figure 6 allows the second call in Figure 2 to become legal by widening the type of the parameter to the existential supertype. Thus the *student* will only behave as a *person* when acting as *president* of the *society*. The *newPresident* procedure in Figure 6 could be used without change in Figure 2 with the same effect as the existing one.

5 Type rules

In order to describe the type rules for bounded quantification some terms are first introduced. A value is either a denotable value or a location which contains one. A denotation is an expression within a programming language which may eventually be

example, as students have all of the attributes of persons then it is reasonable for a student to be the president of a non-student society (in computational, rather than sociological terms):

```
type person is {name : string}
type student is {name : string ; matricNo : int}
type society is {president : person}

let aPerson := person ("Joe Doe")
let aStudent := student ("Jim Dim", 12345)
let aSociety := society (aPerson)

aSociety.president := aStudent
```

Figure 4 Subtype Assignment

The desired operational semantics is clear in a simple example, and this may again be achieved by the use of either subtype substitution or bounded quantification. This time, however, the bounded quantification must be existential rather than universal. Once more different syntax will be introduced to clarify the different systems.

In a subtype substitution system, *society* is defined to be a record type with a single field, *president*, which may be a value of any type which is a subtype of *person*. This is written more clearly as

type society **is** {president : ≤ person}

On the assignment to the *president* field of *aSociety*, the left hand side is asserted to be of type ≤ *person* and the right hand side is asserted to be of type ≤ *student*. Therefore the assignment is guaranteed to be type safe so long as both assertions are correct.

Using bounded existential quantification the semantics are more subtle. To allow the *president* field of type *society* to be assigned a value of any subtype of *person*, it must be defined as a bounded existentially quantified type:

type eSociety **is** {president : ∃t ≤ person.t}

This may be read as the type *eSociety* has a single field, *president*, which has a value of a type which is a subtype of *person*. This allows the *president* field to refer to a value of any subtype of *person*, but both the creation of *aSociety* and the assignment to its field mean something subtly different, for which the following syntax is introduced:

```
type eSociety is {president : ∃t ≤ person.t}

let anESociety := eSociety (widen aPerson to ∃t ≤ person.t)
anESociety.president := widen aStudent to ∃t ≤ person.t
```

Figure 5 Existentially Quantified Types

```
let newPres := proc (s : ≤ society ; p : ≤ person)
                    s.president := p

let newPres := proc [t ≤ society, u ≤ person] (s : t ; p : u)
                    s.president := p
```

Again, the second procedure is not well typed. The expressions *s.president* and *p* are both known to be some subtype of *person*, but as there is no way of telling that they are the same subtype the assignment is not correctly typed. It may perhaps now be seen more clearly why the first procedure can result in error, by comparing the more explicit type assertions of the bounded quantified procedure. As has been shown in Figure 2 related quantifiers are required to solve this problem using bounded universal quantification.

The type of the expression *s.president* in the bounded quantification system is somewhat problematic and it is necessary to extend the bounded quantification system so that such expressions may be attributed with types. A first intuition is that such types are existentially quantified and that the type of *s.president* is ∃t ≤ person.t. ∃t ≤ person.t should be read as there exists a type *t* such that *t ≤ person*. Although this type is reasonable in terms of the set model of type, it would be problematic in a programming language with structural type equivalence. This is because the same type judgement may be made for the expression *p*, and as it is important for these expressions not to be type compatible some special type equivalence rules would need to be introduced.

A type may be found to represent the expression *s.president*. It is a free quantifier of the procedure type. An identifier may be introduced to denote this type by rewriting the procedure as an equivalent pair of nested procedures as follows:

```
let newGetPres := proc [u ≤ person] () → proc [t ≤ {president : u}] ( t ) →
u
                        proc [t ≤ {president : u}] (s : t) → u ; s.president
```

In this example *newGetPres* is a procedure which is quantified by a type *u* which is bounded by *person*. It produces as its result another procedure which is quantified by *t* which is bounded by a record with one field named *president* of type *u*. The result is the value of the expression *s.president*. Notice that *u* and *t* are related quantifiers and that it has already been shown how to make these recursive. The important point is that the type of *s.president* can be named.

The types and type equivalence rules for such free quantifier types are discussed in section 5.

4 Bounded existential quantification

As well as allowing polymorphism over a class of procedures, subtyping may also be used for partially abstracting over the type of other data. This use of subtyping occurs when the substitution operation used is assignment rather than parameter passing. For

```
rec type convenor is {name : string ; headOf : society}
& society is {president : convenor}

rec type studentConvenor is {name : string ; headOf : studentSociety ;
                                                    matricNo : int}
& studentSociety is {president : studentConvenor}

let newPresident = proc [u ≤ {headOf : t}, t ≤ {president : u} (s : t ; p : u)
begin
        s.president := p
        p.headOf := s
end

newPresident [convenor, society] (aSociety, aConvenor)
newPresident [studentConvenor, studentSociety] (aStudentSociety,
aStudentConvenor)
```

Figure 3 Recursive Related Quantifiers

3.1.1 Free quantifiers

The main difference between the type systems of subtype substitution and bounded universal quantification is in the compatibility of types within a polymorphic context. In the bounded quantified system a static type assertion means that any value is of a precise type rather than a subtype. Subtyping may only be achieved where a type is explicitly abstracted over. That is, in the call of the bounded universally quantified procedure. This leads to a system where more accurate type information is available, but which is less flexible. For example, consider the following attempt to write the *getPres* procedure in the two systems:

```
let getPres := proc (s : ≤ society) → ≤ person
                s.president

let getPres := proc [t ≤ society] (s : t) → person
                s.president
```

The first of these uses subtype substitution and is a procedure that takes a parameter which is a subtype of *society* and returns a value which is a subtype of *person*. The second procedure, however, is not well typed. This is because the result value is stated to be of precisely type *person* and this is not the type of the expression *s.president* which could be any subtype of *person*. It is precisely this lack of type compatibility that is a sufficient restriction to avoid the type accuracy problem noted earlier in Figure 1. Consider an attempt to write the procedure *newPres* in the two systems:

```
type person is {name : string}
type student is {name : string ; matricNo : int}
type society is {president : person}
type studentSociety is {president : student}

let newPresident := proc [P ≤ person, S ≤ {president : P}] (s : S ; p : P)
                        s.president := p

let aSociety := society (aPerson)
let aStudentSociety := studentSociety (aStudent)

! make a person a society's president
newPresident [person, society] (aSociety, aPerson)

! make a student a society's president
! fails since society does not have a field of type student
newPresident [student, society] (aSociety, aStudent)

! make a student a student society's president
newPresident [student, studentSociety] (aStudentSociety, aStudent)

! make a person a student society's president
! This is illegal and will be caught at compile time
newPresident [person, studentSociety] (aStudentSociety, aPerson)
```

Figure 2 Related Quantifiers

In Figure 2, the procedure *newPresident* has related quantifiers in that the second quantifier is a subtype of any record type that has a field called *president* of the first quantifier type. The illegal fourth call of the procedure can now be detected at compile time since *studentSociety* does not have a field of type *person* and cannot therefore be substituted for the quantifier *S*. For the moment the ability to perform this static checking appears to be at some cost since the second call is also illegal. We will return to this later.

In some cases mutually recursive specification of quantifiers is required. This is demonstrated in Figure 3.

as string have no subtypes. Consider again the example procedure *personName* given above.

let personName := **proc** (it : person) → **string** ; it.name

More accurately it could be stated that the type of the parameter *it* is any subtype of person and written

let personName := **proc** (it : ≤ person) → **string** ; it.name

The equivalent bounded universal quantified procedure is

let personName := **proc** [t ≤ person] (it : t) → **string** ; it.name

That is, *personName* is quantified by *t* which is bounded by *person* and returns the *name* field of the record *it*. It may be called by

```
let aPersonName := personName [person] (aPerson)
let aStudentName := personName [student] (aStudent)
```

One immediate advantage of bounded universal quantification is that there is now no loss of type information. For example

```
let personId := proc [t ≤ person] (it : t) → t ; it

personId (aStudent)   ! is of type student
```

ensures that the output type is the same as the input type.

The descriptive powers of bounded universal quantification may be increased by the use of related quantifiers. For example, in order to write the procedure *newPresident* using bounded universal quantification it is necessary to introduce related quantifiers. This allows the exact type match in the assignment and is shown in Figure 2.

The problem just described occurs because the application of the principle of substitutability may not be statically checked for correctness in the language of the example. Solutions to this problem appear in different languages. They fall into the categories described below.

2.2 Categorising the solutions

The problem described in Figure 1 is manifested as a conflict between two apparently independent intuitions. The first of these is type accuracy; that is, any static type description associated with a value is an accurate description of that value's attributes. The second intuition, the principle of substitutability, always allows subtype values to be used in substitution operations where supertype values are specified. These intuitions are in fact interdependent and must be considered together.

To maintain type accuracy the system must prevent any dangerous update from occurring. Modifications which are sufficient to preserve the overall type accuracy of the system may be made in any of the following categories[13]:

- substitution context limitation:

  limit the contexts in which substitution using inclusion may occur,

- substitution mutability limitation:

  model mutability within the type system, and restrict type inclusion in some appropriate manner, or

- substitution dynamic failure:

  check the validity of substitution dynamically, and accept that a failure may occur at the time of update.

This paper describes a solution to the above problem that limits the context in which substitution may be made. The system uses a more accurate specification of type than that obtained by using the principle of substitution. This is achieved by the use of bounded quantification[10] in the absence of any other subtyping mechanism.

3 Bounded Quantification

Subtyping facilitates the specification of generic code and the manipulation of heterogeneous collections of data. The specification of generic code is called inclusion polymorphism and is made more precise here by using bounded universal quantification. The manipulation of heterogeneous collections of data such as a set of *person*s that includes subtype of *person* is handled by bounded existential quantification. In bounded quantification the type specifications are exact.

3.1 Bounded universal quantification

In bounded universal quantification subtyping is only allowed on the parameters at the call of a bounded universally quantified procedure. All other substitution such as in assignment requires an exact type match. For the present it is assumed that base types such

```
let personId:= proc (it : person) → person ; it

personId (aStudent)   ! is of type person
```

In the above a type widening operation has taken place during the dynamic substitution in the procedure call *personId (aStudent)*. That is, the type of the result, *person*, is a wider or supertype of *student*. Cardelli originally called this a type checking anomaly. It has also been referred to as a lack of closure in the type system[3].

The problem of mutable values is more important. Jones and Liskov[14], Albano[2] and Wegner and Zdonik[21] have independently discovered in different contexts a more serious loss of type information which may lead to a program failure in a strongly typed language.

In Figure 1 the type *studentSociety* is a subtype of *society*. The procedure *newPresident* takes a *society* and a *person* as parameters and updates the *president* field of the *society* with a value of type *person*. The four calls are all type correct in that only subtypes are substituted for supertypes. However the last call updates the *president* field of *aStudentSociety* to a *person*. This is illegal since only *student*s can be *president*s of *studentSociety*s. Statically all the individual actions appear to be type safe; however, the program is clearly in some sense incorrect.

```
type person is {name : string}
type student is {name : string ; matricNo : int}
type society is {president : person}
type studentSociety is {president : student}

let newPresident := proc (s : society ; p : person) ; s.president := p

let aSociety := society (aPerson)
let aStudentSociety := studentSociety (aStudent)

! make a person a society's president
newPresident (aSociety, aPerson)

! make a student a society's president
newPresident (aSociety, aStudent)

! make a student a student society's president
newPresident (aStudentSociety, aStudent)

! make a person a student society's president
! This causes problems
newPresident (aStudentSociety, aPerson)
```

Figure 1 Problems With Subtyping

solutions to this problem can be categorised into restricting the context of substitution, restricting mutability and tolerating dynamic type checking[13].

This paper proposes a solution to this incompatibility by more accurate specification of subtyping through bounded quantification. Both bounded universal and bounded existential quantification are described and shown to preserve static type checking while retaining the expressiveness of subtyping.

2 Subtyping

Cardelli[6] has proposed a semantics for subtyping in which a type is a subtype of another if the operations allowed on the second type are also allowed on the first. This subtype relation defines a partial ordering of the types, which may therefore be described in terms of a lattice. For simplicity only record types will be considered in order to concentrate on the interaction between subtyping and mutable values. It is noted that Cardelli has also given a semantics for subrange, variant and function subtyping.

The discussion here assumes structural equivalence and implicit inheritance, for maximum generality. A record type τ is defined to be a subtype (written $\leq$) of type τ' if τ has all the fields of τ', possibly some more, and the common fields of τ and τ' are in the $\leq$ relation. These ideas will be introduced by example. The declarations

type person **is** {name : **string**}
type student **is** {name : **string** ; matricNo : **int**}

define two record types *person* and *student*. The type *person* is a record with one field called *name* of type string. The type *student* is also a record but has two fields, one called *name* of type string and the other called *matricNo* of type integer. In some of the following examples the type names will be overloaded as constructor functions:

let aPerson := person ("Joe Doe")
let aStudent := student ("Jim Dim", 42)

The subtyping rule implies that *student* is a subtype of *person*. The primary intuition behind subtyping, the principle of substitutability, is that a value of a subtype can be substituted anywhere a value of a supertype is expected. Thus a procedure written for *person*s should work for *student*s. For example

let personName := **proc** (it : person) $\rightarrow$ **string** ; it.name

declares a procedure called *personName* which takes as a parameter a value *it* of type *person* and returns a string which is the *name* field of *it*. It is expected that this procedure will work equally well for *person*s and all its subtypes. The calls

personName (aPerson)
personName (aStudent)

are both legal and return the *name* of the respective record values.

2.1 Problems with Subtyping

Cardelli[6] has shown that the accuracy of static type assertions can be lost even without mutable values. Consider for example

SUBTYPING WITHOUT TEARS

R.C.H. CONNOR
Department of Mathematical and Computational Sciences, University of St Andrews, North Haugh, St Andrews, KY16 9SS, Scotland

and

R. MORRISON
Department of Mathematical and Computational Sciences, University of St Andrews, North Haugh, St Andrews, KY16 9SS, Scotland

ABSTRACT

The introduction of a statically checked type system into a programming language restricts the class of computations that can be performed, in order to gain safety. Where a type system has been found to be over-restrictive language designers have introduced constructs to alleviate the restriction for some well-identified classes of problem. One such example is subtyping which facilitates the specification of generic code and the manipulation of heterogeneous collections of data.

There is a well known incompatibility between static checking and mutability in subtyping systems that preserve identity. This paper proposes a solution to this incompatibility by the more accurate specification of subtyping through bounded quantification. Both bounded universal and bounded existential quantification are described and shown to preserve static type checking while retaining the expressiveness of other forms of subtyping.

1 Introduction

Type systems provide two important facilities within programming languages – data modelling and protection[5]. The type system allows the programmer to partition the universe of discourse of the application into well defined sets of entities that may then be manipulated in a consistent manner. The user models the application in terms of these sets of entities which are the value sets of the types.

Traditionally it is expected that consistent use of a type system may be checked by a static scan of the program text. This has a number of advantages including discovering errors earlier and, by eliminating run time checks, making programs execute more efficiently.

There is a design tension between the expressive power of a language and static type checking. A type system is introduced to bring discipline to a language and therefore restricts the class of computations that can be expressed. Where this has been found to be over-restrictive language designers have introduced constructs to alleviate the restriction for some well-identified classes of problem. Parametric polymorphism[17] is one such example. Another is subtyping.

There is a well known incompatibility between static checking and mutability in subtyping systems that preserve identity[2]. The principle of substitutability[21] states that a subtype may be substituted for a supertype in any expression. Where the language has mutable values the incompatibility is that the subtyping cannot be checked statically. Partial

This paper should be referenced as:

Connor, R.C.H. & Morrison, R. “Subtyping Without Tears”. In Proc. 15th Australian Computer Science Conference, Hobart, Tasmania (1992) pp 209-225.