This paper should be referenced as:

Connor, R.C.H., Atkinson, M.P., Berman, S., Cutts, Q.I., Kirby, G.N.C. & Morrison, R. “The Joy of Sets”. In **Database Programming Languages**, Beeri, C., Ohori, A. & Shasha, D.E. (ed), Springer-Verlag (1993) pp 417-433.

The Joy of Sets

Richard Connor, Malcolm Atkinson†, Sonia Berman‡, Quintin Cutts, Graham Kirby and Ron Morrison

Department of Mathematical and Computational Science,
University of St Andrews, St Andrews, KY16 9SS, Scotland

†Department of Computer Science, University of Glasgow,
Glasgow G12 8QQ, Scotland

‡Department of Computer Science, University of Capetown, South Africa.

{richard, quintin, graham, ron}@dcs.st-and.ac.uk
mpa@dcs.glasgow.ac.uk
sonia@cs.uct.ac.za

Abstract

The semantics of many bulk data models depends on user-defined attributes such as definitions of element equality, ordering, and other domain predicates. While these attributes are an intrinsic part of the data model, they are not normally treated as part of the static type description. This leads to the occurrence of data modelling errors which are not statically detectable, such as a union operator accidentally being applied to two sets which have different equality semantics.

Here we introduce a model of bulk types which includes such attributes as part of the type definition, along with a typechecking scheme which is statically decidable. The model relies upon the value dependencies being manifest to the typechecker; one neat way of achieving this is through the new paradigm of hyper-programming. For cases where the static typechecking introduced is unnecessarily restrictive a polymorphism scheme which provides controlled flexibility is introduced.

1 Introduction

The semantics of many bulk data models depends on user-defined attributes such as definitions of element equality, ordering, and other domain predicates. While these attributes are an intrinsic part of the data model, they are not normally treated as part of the type description. This may lead to the occurrence of data modelling errors such as a union operator accidentally being applied to two sets which have different semantics for the equality of their elements.

Attributes such as the definition of element equality are typically held with instances of bulk data types, rather than with their type [ALP+91]. When a

function such as union is applied, an arbitrary choice is made between the two possible equality functions. The chosen equality is used not only as an attribute of the newly constructed bulk value, but also to implement the application of the union function itself, as this depends upon the definition of element equality. Thus the semantics of union and similar set functions is compromised as, among other reasons, commutivity is lost.

By including these attributes as part of the type, however, it should be possible to define function types over bulk values such that the functions they describe are limited in application to bulk values where the element domain attributes are the same. Any attempt to apply these functions to bulk values with different attributes will be detected by the type system and the execution will be disallowed. A dynamically typed system will generate a type error or exception when the operator is erroneously applied, and a statically typed system will detect such errors before the program which contains them starts to execute.

A directory of Scottish names is used as an example of the inclusion of element attributes in the type. People may take a different view as to whether the names “MacFarlane”, “Macfarlane”, and “McFarlane” are really different names, but their owners are usually protective of their different forms. If they are used to retrieve data however it is likely that the retriever will not wish to distinguish between them. A telephone company’s applications may require a number of directories, some of which distinguish between different spellings and some of which do not. For example a printed version of a telephone directory may distinguish between them whereas an electronic version used by an operator enquiry service may not. These directories are characterised in the system by bulk data dictionaries; whether different spellings are distinguished or not is characterised by different domain equality rules in these dictionaries.

Bulk operators such as union and intersection over directories whose elements have different equality rules are undefined and should be considered as an error. If such errors are not detected, and an arbitrary choice of equality is used, then no further mechanical error will result and the program will appear to execute correctly. The only way of detecting that an error has occurred is by analysing the directory which results. By including the equality definition in the type the error will be detected earlier in a strongly typed system, ideally during the program’s typechecking.

Figure 1 shows a possible syntax for type definitions where domain equality attributes are described as a part of the type. The types *customerDirectory* and *operatorDirectory* are defined as specialisations of the type operator *Dictionary*, which has been previously defined. The structural attributes of the two dictionary types are the same, but different domain equality rules are associated with the types.

These different rules for domain equality mean that the types *customerDirectory* and *operatorDirectory* are not equivalent types within this system.

```
type customerDirectory is Dictionary[ string, phoneNo ] with
        ! simple (system-defined) equality
        domainEq = proc( a,b : string -> bool ) ; a = b

type operatorDirectory is Dictionary[ string, phoneNo ] with
        ! some complex equality rules
        domainEq = proc( a,b : string -> bool ) ; ...
```

Figure 1 Attributes associated with type

This restriction of bulk type equivalence must be carefully introduced to avoid losing flexibility within the programming system. The telephone company software may include a procedure *printDirectory*, which has a single operand of type *customerDirectory*. The application of *printDirectory* to a value of type *operatorDirectory* is an error, and the inequivalence of these types will lead to the reporting of a type error in such circumstances. However, there is also a large class of functions which may be correctly applied to bulk data values with any equality rule, such as a membership test. Figure 2 shows examples of such typings.

```
printDirectory( aCustomerDirectory )                  ! typing is correct
printDirectory( anOperatorDirectory )                 ! typing is incorrect

let x = member( "McPhee",aCustomerDirectory )         ! typing is correct
let y = member( "McPhee",anOperatorDirectory )        ! typing is also correct
```

Figure 2 Correct typing of different functions

The importance of the examples is that they demonstrate not only the importance of treating domain equality as an attribute of type in some circumstances, but also the importance of an appropriate form of polymorphism to allow abstraction over domain equality where it is not required. In fact the type of the functions union and intersection also requires polymorphism over element equality: the correct restriction is that the operands must have the same definition of equality, but the functions are applicable to any such pair of bulk values.

Equality of elements is not the only value that might be associated with bulk types; for instance some bulk types require order, others may require a characteristic function that restricts the values in the domain. The rest of this paper considers in

detail the example of a set with an associated element equality function only; the mechanisms outlined are sufficiently general to extend easily to other examples of bulk types that depend on function values.

2 Name and structural equivalence

In general, the equivalence of types may be defined either by name or by structure. In name equivalence regimes two types are considered to be the same if and only if they share the same instance of a definition. Thus in the above example *customerDirectory* and *operatorDirectory* are de facto different types irrespective of the particular equality semantics defined.

Name equivalence thus gives a trivial solution to the requirement that sets with different element equality rules should be regarded as different types. However there are a number of problems in the use of a strict name equivalence regime in the context of database programming languages; these may be solved, but only by the use of structural equivalence testing in some circumstances [CBC+90]. In particular, the ability to abstract over part of the type definition, as required in the typing of functions such as *member*, can not be achieved without the use of structural equivalence.

The structural type equivalence of bulk types whose specification includes attributes such as domain equality requires the following to be true:

- the two bulk constructors are the same,
- the element types are the same, and
- the two equality tests are the same.

Traditional type checking takes care of the first two but the final test is problematic, and is the subject of this paper. The first obvious problem is the definition of "sameness" over values which in this case are function types. The second problem is that to achieve static typechecking the particular instances of the values must be available to the typechecker during its analysis of the program text.

Thus to achieve the static structural checking of these types in the most general context it is necessary to solve two well known intractable problems: the equality of functions and the static checking of dependent types. However, the careful introduction of certain restrictions may be used to allow partial solutions to both problems. The good news is that these restrictions may be easily explained to a programmer and give a useful programming algebra which allows precisely the desired static typechecking over bulk types with value dependencies.

3 Decidable Static Checking

There are two major problems with the decidable static typechecking of types which depend upon function values. The first is that dependent types in general pose difficulties with decidable static checking; the second is that function values in general do not have a good decidable notion of equality.[1]

An angle of attack is outlined here which provides partial solutions to both of these problems. The conjunction of these partial solutions gives a useful degree of static checking over dependent set types.

3.1 Dependent Types

For dependent types to be structurally equivalent it is necessary for the values upon which they depend to be equal. For the equivalence of dependent types to be statically decidable it is therefore necessary for the values upon which the types depend to be statically available to the typechecker. For some simple examples this is not a problem. For example, array types in Pascal are dependent upon their bounds; as these values are constrained to being manifest constants, the values are available to the typechecker and their equality is statically decidable.

For types which are dependent on more complex values statically decidable typechecking causes more difficulties. There are however two circumstances in which static typechecking can be maintained: either when a manifest constant binding is set up between the types and the values upon which they depend , or else when the system can arrange for the values upon which the type depends to be available to the typechecker during static analysis. The first of these is of limited use in the context of database programming languages, as will be shown; the second is more useful, however, and may be provided by systems which support incremental compilation or hyper-programming.

3.1.1 Value equality deduced by constant binding

Where the full equivalence of dependent types is not statically decidable, a syntactic rule may be introduced to constrain the use of dependent types to contexts where a more restrictive rule is statically enforceable. This mechanism relies upon the use of values being constrained to contexts where their equality is a consequence of that

---

[1]To avoid confusion the term *equality* will be used to mean the usual equivalence relation as defined over values, and *equivalence* will be used to mean the usual equivalence relation as defined over types.

context. Such a mechanism is commonly used to constrain a value dependency which arises with existentially quantified types [MP88].

One way of enforcing this with bulk types is to allow type definitions only to use functions referred to by identifiers with constant bindings. This allows identity to be used to determine that the equality functions are the same. Figure 3 shows an example of equivalent set type definitions which have associated equality functions. Because the identifier *stringEq* is a constant binding, it is possible to deduce statically that any other set type which defines equality as the same binding will also share the same procedure. Notice that this is a different consideration from the procedure itself being manifest; *stringEq* may be bound to the result of an arbitrary computation.

```
let stringEq = proc( a,b : string -> bool ) ; ...
        !** = means constant binding **!

type dictType1 is set[ string ] with stringEq
type dictType2 is set[ string ] with stringEq
```

Figure 3 Dependent set type definitions

That sound static typing can be achieved here is clear; however such a system is of limited use. The main problem is that only a subset of the desired equivalence semantics is achieved. Figure 4 shows a program where equivalent types cannot be determined statically.

```
let dictStringEq = if caseSensitive() then stringEq else
ignoreCase

type dictType1 is set[ string ] with stringEq
type dictType2 is set[ string ] with dictStringEq
```

Figure 4 Correct typechecking not statically decidable

This may be of limited consequence within a single program; a programmer would most likely use a single definition for all set types which share equality semantics, thus avoiding in any case the problems of structural type equivalence. The real limitation of this mechanism becomes apparent when separate compilation and the merging of independently prepared subsystems are considered; the ability to share a constant binding does not exist in these cases. To achieve correct type equivalence it is necessary to make the values available to the typechecker.

3.1.2 Values available to typechecker

Some programming language environments allow for the possibility of values of non-trivial types to be available in the typechecking domain. For example, the incremental compilation strategy of languages such as ML [MTH89] and Galileo mean that structured values are potentially available to the compiler at typechecking time. The hyper-programming system of Napier88 [MBC+89] allows arbitrary bindings to be performed at composition time, thus making the values potentially available to the typechecker. If all values upon which the types depend are available before the compilation of a program, and the typechecker has the ability to perform the required equality tests over them, then static typechecking may be achieved. An essential part of this mechanism is that, once references to dependent values are made, the system preserves these for the lifetime of the programs; that is, the system guarantees referential integrity.

The essential requirement is that any values upon which a type depends are evaluated before any equivalence testing is performed upon the type. This restriction of the general type description can be enforced in incremental compilation systems by restricting the stated dependencies to free identifiers which are resolved in the external environment, and in a hyper-programming system by restricting the dependencies to being hyper-links.

Figure 5 shows how the program shown in Figure 4 may be adapted for use in an incremental compilation system. Representing the same text as two compilation/evaluation sessions meets the requirement that the type dependencies appear as free identifiers with respect to the compilation unit. The point is that in this case, the function value *dictStringEq* has been evaluated, as has the value *stringEq*, before the compilation of the second program. The typechecker can therefore access these values during the static analysis of the program fragment and perform the necessary equality testing on them whenever a structural comparison of the two types is required.

```
let dictStringEq = if caseSensitive() then stringEq else
ignoreCase
```

```
type dictionary1 is set[ string ] with stringEq
type dictionary2 is set[ string ] with dictStringEq
```

Figure 5 Correct typechecking statically decidable by incremental compilation

The same technique may be used in a hyper-programming system where type dependencies are constrained to being represented as hyper-links; this guarantees in a subtly different manner that the required values have been evaluated before typechecking time. This will be explained in more detail later. There are two other situations discussed later where such values may be available at compilation time: these are where the value is a manifest literal, and where compile-time linguistic reflection [SSS+92] is used.

There are two major advantages of this solution over that of constant binding. Firstly it allows a fully general type equivalence check, rather than a restricted subset of it. The second is a consequence of this particularly important in database programming languages: it is possible to merge independently constructed subsystems whilst preserving the type equivalence semantics.

The problem of how to determine whether two functions have the same behaviour as each other is now examined.

3.2 Functional Equality

The most general notion of functional equality is that, for any two functions, they are equal if they produce an equal result for any equal operands. It is well known that there exists no decidable algorithm which tests this notion for any given pair of functions. As a consequence, many programming languages provide no notion of equality over values of function type.

There are however other less general equivalence relations which may be usefully defined over functions. In some languages for example the concept of identity may be defined for functions; the identity of a function is created when its closure is formed [MBC+89]. Another possibility is that the code, perhaps normalised according to some semantics preserving rule, is equal, and the bindings from that code are equal [DD85]. The point here is that there do exist a number of equivalence relations which may be tested by fully decidable algorithms and which are sub-partitions of the most general relation defined by functional equality. Thus although it is not possible in general to determine the equivalent behaviour of arbitrary functions, it is possible to determine the equivalent behaviour of particular subsets of functions. Figure 6 shows three pairs of equal functions, the detection of whose equality is of varying complexity.

Without further elaboration we define a notion of weak functional equality, which will detect that both the second and third pair of functions are equal. The important attribute of this equality over simple identity is that it does not preclude the equality of pairs of functions which are independently constructed in different environments. The usefulness of this weak notion of equality may be seen in two

situations: the merging of sub-systems, and the copying of values between persistent environments.

```
let first = proc( a : int -> int ) ; if a = 0 then 1 else 37
let second = proc( x : int -> int )
begin
        let res := 37
        if x = 0 do res := 1
        res
end

let difficultToDecide = ( first = second )
```

```
let first = proc( a : int -> int ) ; if a = 0 then 1 else 37
let second = first

let easyToDecide = ( first = second )
```

```
let first = proc( a : int -> int ) ; if a = 0 then 1 else 37
let second = proc( x : int -> int ) ; if x = 0 then 1 else 37

let quiteEasyToDecide = ( first = second )
```

Figure 6 Varying degrees of complexity for functional equality testing

The merging of sub-systems, including the merging of independently prepared program and data, may only be sensibly achieved in cases where the type of the data has been agreed beforehand. In the case of dependent set types, the type includes the description of the element equality function. The importance of weak equality in this context is that as long as the equality is agreed beforehand, different instances of the same equality function may be prepared in the sub-systems, and then mechanically detected as equal during the system merge.

Copying values between typed object stores is really a special case of merging. In a typed object store every object must have a strong association with a representation of its type, to allow the typechecking of programs which bind to it. When values are copied between stores their associated type representations must also be copied. The importance of weak equality over identity in this case is that

the equality of functions is preserved over copying, thereby preserving type equivalence over values independently propagated from a common source.

4 Hyper-programming interface and examples

Hyper-programming is a new style of programming made possible when source code representations are stored inside a typed persistent store. A hyper-program is a source representation which contains, in addition to the normal linear text, embedded links to other values in the store. Since the linked values can be accessed by scanning the hyper-program, they are available to the compiler. If the language definition allows the values on which types depend to be denoted by links, those values will be accessible by the typechecker for static checking. This may only be allowed in cases where the links are known to refer to immutable values, so that a constant binding is maintained from type to value as above.

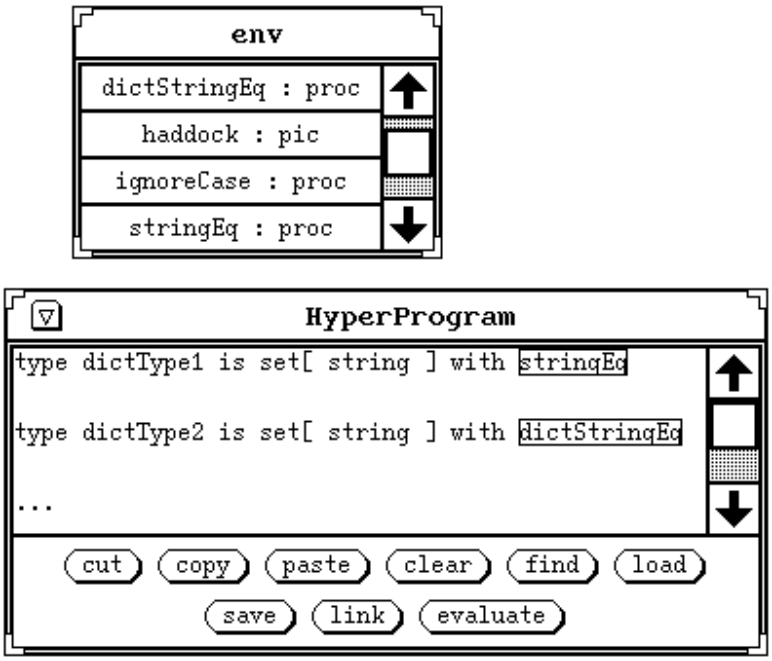

Figure 7 An example hyper-program

Figure 7 shows the programmer's view of a simple hyper-program in which dependent set types may be checked statically. The values on which they depend are denoted by light-buttons which represent direct links to values already present in the persistent store. Although it is not possible to tell from the text of the program

whether the types are the same, the typechecker is able to follow the links into the persistent store and perform arbitrary computation over the values in their closure.

Although the example above serves to illustrate the mechanism, it is unlikely to be used in practice since the programmer would not normally write separate equivalent type definitions within a single program. Figure 8 shows two separate compilation units using equivalent dependent types. They may be compiled in entirely separate environments, or in the same environment, so long as the function values are already present in the compilation environment.

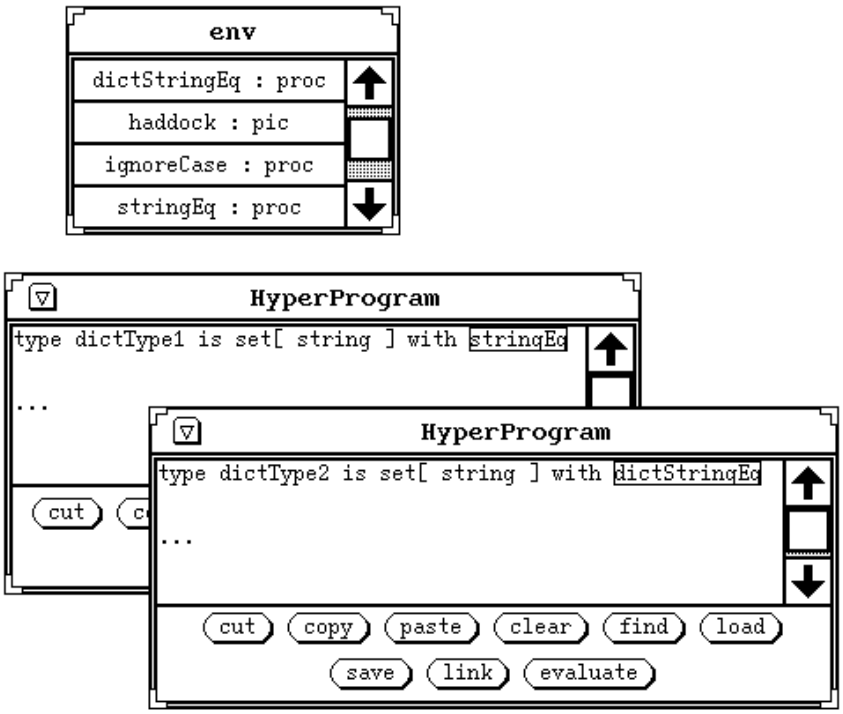

Figure 8 Separate compilation units

If the types declared in the programs above are equivalent to each other, then data of these types which is created and placed in the persistent store will be seen to be typed equivalently in later use. Figure 9 shows an example of a hyper-program which applies the union operator to two values created by the programs in Figure 8. Note that these are typed set values whose type equivalence is checked at compilation time. It is not possible to see whether this program is typed correctly by a simple examination of the program text; however the typechecker is again able to access the closure of the embedded hyper-links, which contains the necessary

information. It perhaps should be pointed out that in a hyper-programming system the programmer should also be given this ability!

One further point to make is that the light buttons named as *d1* and *d2* may refer to locations, rather than instances of the bulk types. After the program has been compiled it is possible for these locations to be updated with other values of the same type; however the program will work correctly with the new versions since their types, including dependencies, must be the same as the old versions. The significance of this in database terms is that the programs may be typechecked after the schema has been defined, rather than after it has been populated.

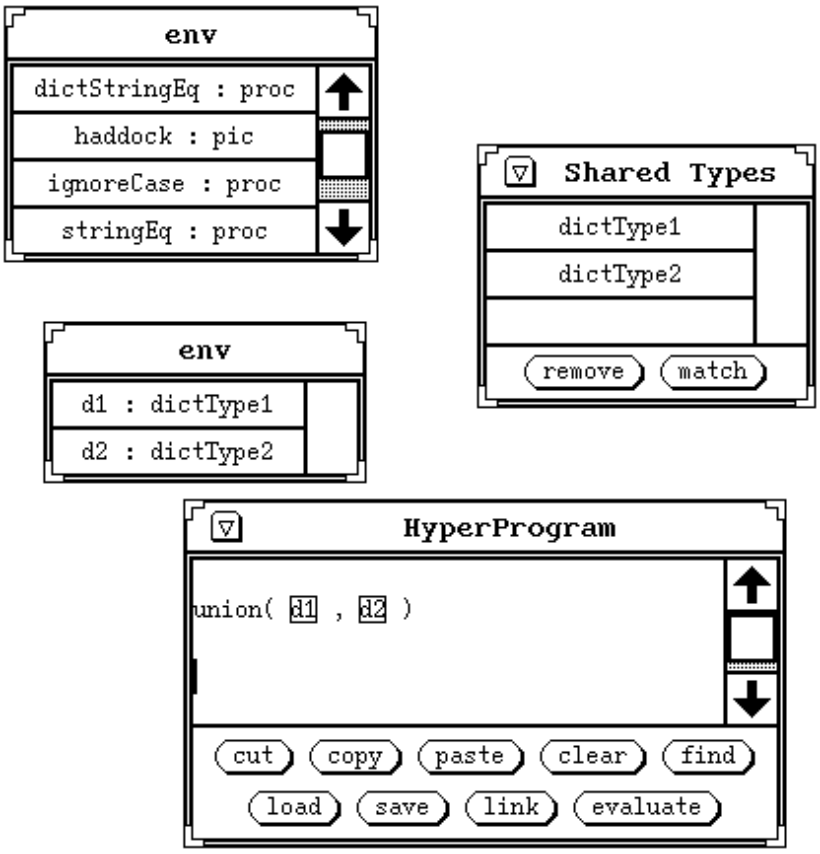

Figure 9 Statically checkable union application

5 Polymorphism

The main motivation for introducing value dependencies to bulk data types is to make a class of errors statically detectable. As always, however, the restriction introduced decreases the class of correct programs, as well as the class of incorrect

ones. Polymorphic forms are required to re-introduce some of the lost flexibility in a controlled manner.

The notation used here for polymorphism is deliberately verbose in order to expose all of the concepts being introduced. It is expected that a programming language with these concepts would elide much of the unnecessary information to reduce noise. The notation will be introduced by example in the style of Napier88 polymorphic forms. Figure 10 shows a polymorphic procedure in Napier88, and a call to that procedure. Notice that the parametric form of the polymorphism is made explicit, and that the type parameter is explicitly supplied at the call.

```
let id = proc[ T ]( x : T -> T ) ; x

let three = id[ int ]( 3 )
```

Figure 10 The polymorphic identity procedure in Napier88

To help understanding, formal identifiers are introduced within the square brackets for every type, and aspect of type, which is being abstracted over. To abstract over a set type, a notation must therefore introduce formal parameters for the type of the element, and the value of the equality function. Figure 11 shows a possible notation for the type of the procedure *cardinality*. This type should be read as: "a procedure which, for every type *T*, and for every type *S* which is a set with element type *T* and equality value *e*, takes a parameter of type *S* and returns an integer."

The only new concept introduced here is that the formal parameter *e* stands for a value; as this value is part of the type, however, it is included in the universal quantification. The value denoted by *e* must be locally accessible, as it is required to check any insertion into a value of the set type; there is no reason why *e* should not be used as a value within the procedure body.

```
cardinality : proc[ T, S : set[ T ] with e ]( S -> int )
```

Figure 11 The polymorphic *cardinality* procedure type

Figure 12 shows a call to the *cardinality* procedure. Notice that although the set type including the equality value is passed explicitly in this example, this could be inferred according to the type of the set parameter.

```
let x = cardinality[ string, set[ string ] with ignoreCase ](
mySet )
```

Figure 12 A call of cardinality

Enough abstraction has been introduced to write some different forms of the set union operation. Figure 13 shows a union procedure for sets with the same equality.

```
let union = proc[ T, S : set[ T ] with e ]( a , b : S -> S )
begin
        let new = createSet[ T , S ]()
        for i in a do new := insert[ T, S ]( new , i )
        for i in b do new := insert[ T, S ]( new, i )
        new
end
```

Figure 13 A set union implementation

It is also useful to consider the case where the operand sets have different models of equality. Typically where equality is not part of the type information and is instead associated with the bulk type instances one or other function is chosen arbitrarily, or the programmer may supply a new equality function. Each of these may be programmed using the notations introduced; the difference is that the programmer has fine control over the outcome, and must specify the static types involved at each stage.

```
let union = proc[ T , S1 : set[ T ] with e1 , S2 : set[ T ] with e2 ]
                    ( a : S1 ; b : S2 -> set[ T ] with
                            proc( a,b : T -> bool ) ;
                                    e1( a , b ) and e2( a , b ) )
begin
    type resType is set[ T ] with proc( a,b : T -> bool ) ;
                                    e1( a , b ) and e2( a , b )
    let new = createSet[ T, resType ]()
    for i in a do new := insert[ T, resType ]( new , i )
    for i in b do new := insert[ T, resType ]( new, i )
    new
end
```

Figure 14 A more interesting union operator

One further possibility is that the type of the result may be constructed in terms of the equality functions used for the operands. This requires relaxing the rules for the construction of set types to allow the use of manifest literals as well as constant hyper-links. Figure 14 shows a union procedure in which the type of the result uses an equality relation which is the conjunction of those of the operand types, and which is still statically checkable. Successive calls to this procedure with equivalent operand types will result in equivalent result types, and the result type can be determined statically.

This typing relies upon the recursive application of weak equality over function values. Notice that the example does not introduce general value dependencies, as the definition of the equality function relies only upon values which are part of type parameters to the procedure, rather than value parameters. The use of arbitrary values within the body of the function would preclude the static checking which is possible here as the functional equality could not be determined statically.

6 Conclusions and further work

This paper has outlined how bulk data types that depend upon value based attributes may be statically type checked. The technique depends upon the equality relation over the value based attributes being decidable statically. One way of achieving this is by hyper-programming, a programming technique that allows links to persistent objects to be part of the source representation of a program. These links are available to the type checker and therefore can be used as part of the type equivalence relation for bulk types. To restore the flexibilty that has been lost by the introduction of static type checking, a polymorphic abstraction mechanism is introduced over these value dependent bulk types.

One expected result not discussed in this paper is that major efficiency gains may be made by using static knowledge of these functions in the implementation of the bulk values.

7 Acknowledgements

The title of this paper derives from an inspiration by Fred Curtis of Sydney University whilst on a chance visit to St Andrews, and is in no way connected with the authors' knowledge of literature [Deu90]. Dave Stemple of the University of Massachusetts at Amherst is to be thanked for his constructive comments on the paper. This work was supported by Esprit III Basic Research Action 6309 - FIDE 2, and Richard Connor is supported by SERC Postdoctoral Fellowship B/91/RFH/9078.

8 References

[ACO85] Albano, A., Cardelli, L. & Orsini, R. “Galileo: a Strongly Typed, Interactive Conceptual Language”. ACM Transactions on Database Systems 10, 2 (1985) pp 230-260.

[ALP+91] Atkinson, M.P., Lécluse, C., Philbrow, P. & Richard, P. “Design Issues in a Map Language”. In **Bulk Types & Persistent Data**, Kanellakis, P. & Schmidt, J.W. (ed), Morgan Kaufmann (1991) pp 20-32.

[BL84] Burstall, R. & Lampson, B. “A Kernel Language for Abstract Data Types and Modules” Proc. International Symposium on theSemantics of Data Types, LNCS Vol. 173, Springer-Verlag (1984).

[CBC+90] Connor, R.C.H., Brown, A.L., Cutts, Q.I., Dearle, A.,Morrison, R. & Rosenberg J. “Type Equivalence Checking in Persistent Object Stores”. In **Implementing Persistent Object Bases - Principles and Practice**, Morgan - Kaufmann, pp 154-170.

[DD85] Donahue, J. & Demers, A. “Data Types are Values”. ACM Transactions on Programming Languages and Systems 7, 3 (1985) pp 421-445.

[Deu90] Deux, O.O. “The Story of O_2”. IEEE Transactions on Knowledge and Data Engineering 2, 1 (1990).

[KCC+92] Kirby, G.N.C., Connor, R.C.H., Cutts, Q.I., Dearle, A., Farkas, A.M. & Morrison, R. “Persistent Hyper-Programs”. In Proc. 5th International Workshop on Persistent Object Systems, San Miniato, Italy (1992) pp 73-95.

[MBC+89] Morrison, R., Brown, A.L., Connor, R.C.H. & Dearle, A. “The Napier88 Reference Manual”. University of St Andrews Technical Report PPRR-77-89 (1989).

[MP88] Mitchell, J.C. & Plotkin, G.D. “Abstract Types have Existential Type”. ACM Transactions on Programming Languages and Systems 10, 3 (1988) pp 470-502.

[MTH89] Milner, R., Tofte, M. & Harper, R. **The Definition of Standard ML**. MIT Press, Cambridge, Massachusetts (1989).

[SSS+92] Stemple, D., Stanton, R.B., Sheard, T., Philbrow, P., Morrison, R., Kirby, G.N.C., Fegaras, L., Cooper, R.L., Connor, R.C.H., Atkinson, M.P. & Alagic, S. “Type-Safe Linguistic Reflection: A Generator Technology”. ESPRIT BRA Project 3070 FIDE Technical Report FIDE/92/49 (1992).

[Tur36] Turing, A. “On Computable Numbers with an Application to the Entscheidungs-problem”. Proceedings of the London Mathematical Society 42 (1936) pp 230-265.