

Type-Safe Linguistic Run-time Reflection A Practical Perspective

Richard Cooper and Graham Kirby*

Dept. of Computing Science, University of Glasgow, Glasgow G12 8QQ, Scotland

*Division of Computer Science, University of St Andrews,
North Haugh, St Andrews, Fife KY16 9SS, Scotland

Abstract

Reflection is a property of application development systems which permits programs to change their own behaviour. Linguistic reflection is a variety in which this is carried out by extending the program with extra modules which are created, compiled and linked in by the program itself. With run-time reflection this happens during the running of the program. Typically this occurs by including commands in the program to create the new module as a string and then call either the compiler or interpreter for the language to create an executable form. When combined with persistence, the executable code can then be stored for reuse. However, writing programs which exploit reflection can be a daunting task, since keeping track of multiple representations of a program can involve copious amounts of intricate string manipulation. This paper sets out to analyse some of the issues which make for difficulty, to outline some support tools which have been created and then to discuss the possibilities of an underlying language-independent theory for reflection.

1 Introduction

The development of programming languages for building persistent applications has been motivated by the need to support the secure description of computation over long-lived values with complex structures [1]. Since the programs and data are both expected to be long lasting and therefore run-time errors may not be detected for some considerable time, it is essential that there is as much checking of the program at compile time as possible. Hence there is a great benefit if database programming languages are strongly typed and, wherever possible, statically typed.

There is, however, an inevitable tension between type security and programming flexibility. In an inflexibly typed language, such as Pascal, there needs to be much repetitive programming and some kinds of computation can only be expressed in cumbersome and inefficient ways. Modern type systems attempt to ameliorate this situation by providing type constructors which allow more general programming.

Parametric types allow the programming of code against values drawn from a range of types which vary only in the type of some component value which the code does not touch [2]. Dynamic types package a range of types so that they appear to be the same type [3]. Using these techniques a great many programs can be written which are general to a range of types – polymorphic programs. However, all such polymorphic programming works either by assuming that detail in the type information can be abstracted over or, in the case of *ad hoc* polymorphism, that the range of types is known at the outset and can thus be catered for.

This leaves a range of applications in which: the program needs to make use of the detail of the type information; this type detail varies between the types that the program has to deal with; and the number of types to be dealt with is not known by the programmer at the time that the program is written. Consider, for instance the creation of a program which is to display the value of a variable which is passed to it. If this program is to cope with values which may be scalars, then it will be possible to write a piece of *ad hoc* polymorphic code. If arrays are also to be included, then it is possible to write generic code as long as the type of an array does not depend on its size. If, however, the values may also include records, simple *ad hoc* polymorphism will not work. Take for instance an attempt to print out the value of a *Person* record where the record has a *name* and an *age* field. Using Pascal notation, there needs to be a line in the code of the form:

```
writeln( P.name, P.age )
```

In other words, the program has to have type specific detail (the names of the fields) embedded in it.

This is the situation in which reflection becomes a suitable technique. A **reflective language** is one in which it is possible to write programs which can adapt their behaviour to cope with novel data [4]. There are essentially two ways in which a program can adapt its behaviour, either by changing its execution environment or by changing itself. The former case we call **behavioural reflection** and this includes systems in which the program can affect internal structures of the compiler or run-time environment which cause the program to work differently [5]. Some object oriented systems, such as Modulex, provide a form of behavioural reflection by use of meta-classes [6]. Meta-classes are classes whose members are themselves classes. The meta-class to which an object belongs determines a great deal about the form and facilities associated with that object. If an application can change a meta-class in the same way as it can change an ordinary class, then the program can change the behaviour of its classes.

The other form of reflection is **linguistic reflection**, in which the program changes itself directly. Lisp [7] was the first language to provide this with the *eval* function and POP2 [8] was another language in which mini-programs could be constructed and executed at run-time. Both of these languages were untyped and so the problem of matching individually typed incarnations of the code caused no problems. TRPL [9] applied linguistic reflection to a strongly typed language, by making the reflection happen at compile time. TRPL provides macros which are

evaluated at compile time and these macros can allow the compiled program to vary depending upon the context in which it is compiled.

The particular form of reflection with which this paper concerns itself is **type-safe run-time linguistic reflection**. That is, the program modifies itself at run-time, but always through the inclusion of code which is type checked. The paper will discuss two systems which provide this facility. Napier88 [10] does so by supplying the compiler as a system function which can be called at any time during the run of a program to transform a string containing source code into a new fragment of the program. O₂ [11] provides reflection in the form of a system function which calls the interpreter – again the interpreter is passed a string and the result can be a modification to the application.

The problem with the use of linguistic reflection is that it requires the use of a daunting amount of manipulation of code representations and the careful control of different program representations in the same environment. The program contains fragments which are normal code, pieces which are code representations and pieces which connect the two. To this end a number of proposals have been made to simplify the problem of writing reflective programs [12, 13, 5], but no firm solution is yet in place. In this paper, tools are described which make reflective programming significantly easier. The paper concludes with insights into the interaction between the nature of the type system and the ways in which reflection can be produced. To start with, however, the technique will be described with examples and the uses of reflection will be categorised.

2 Type-Safe Run-Time Linguistic Reflection

A language, L , exhibits type-safe run-time linguistic reflection if the run-time environment of the language contains 1) a module, callable at run-time, which is capable of generating code and checking its validity, and 2) a method of linking that code into the running program. For the purposes of this paper, the module will be restricted to being a version of the compiler or interpreter used by the system.

Thus, in general, if there is a module, M , which takes source code in the form of a string – either from a file or from the terminal and produces code, i.e:

$$M: \text{string} \rightarrow \text{code}$$

then the kinds of system being dealt with here provide a version of M which takes a string within the program and creates code.

2.1 Type-Safe Run-Time Linguistic Reflection in Napier88

In Napier88, this module is a version of the compiler, callable at run-time, which takes in a string containing a block of code and returns a compiled form of the block. A block of code in Napier88 is a sequence of void clauses (i.e. clauses which return no value) possibly ending with a clause which does return a value. Thus:

$$I := 1$$

```
J := 2
I+J
```

is a block of type **int** in Napier88. The run-time compiler returns the compiled form of the code, but as this might be of any legal Napier88 type, the return type of the compiler is the dynamic type, **any**, which is the infinite union of all Napier88 types. To derive the actual value out of the dynamic type, Napier88 provides the **project** clause which reveals the actual type. The detail of this is slightly different for void and non-void blocks.

If *source* is a string containing a void block, the following code will cause it to be compiled and executed:

```
object := compile( source )
project object as O onto
  string: writeString( "Errors: " ++ O )      ! If the compilation fails.
  proc() : O()                                ! Executes the program.
  default: writeString( "Internal compiler error" ) ! Should never
happen.
```

The **project** statement projects the compiled form, *object*, either as a string containing error messages or as a parameterless procedure (i.e. a compiled form of the block in *source*) which can then be executed. So if the only visibly effecting part of *source* was the line "writeString("ABC")" then the execution of the above code would result in "ABC" being displayed.

Blocks which return values are handled slightly differently. For instance, if *source* contains just the expression: "1+2", then the following code assigns the sum to the variable *result*:

```
object := compile( source )
project object as O onto
  string: writeString( "Errors: " ++ O )      ! If the compilation fails.
  proc( $\rightarrow$  any): project O() as OO onto ! Evaluates the program.
  int: result := OO
  default: writeString( "Unexpected result" ) ! Should not happen.
  default: writeString( "Internal compiler error" ) ! Should never happen.
```

The result of compiling a block which returns a value is a procedure which when will called will return the value of the expression wrapped up into an **any** value. When this is projected out, the integer appears.

Using one of the two forms above it is possible to create stand-alone programs, evaluate expressions or to generate procedures with their parameters. The latter is just an instance of the expression evaluation – in this case the type of the expression is a procedure type. Thus if *source* contains the string: "proc(X, Y: int \rightarrow int); X+Y" which is a procedure which adds two numbers together and returns the result, then the following code:

```

object := compile( source )
project object as O onto
  string: writeString( "Errors: " ++ O )      ! If the compilation fails.
  proc(→ any): project O() as OO onto      ! Evaluates the program.
    proc( int, int → int ): result := OO
    default: writeString( "Unexpected result" ) ! Should not happen.
  default: writeString( "Internal compiler error" ) ! Should never happen.

```

sets *result* to the compiled form of this procedure.

By using the compiler in this way, the programmer can generate code which is both type-safe and efficient. Once generated and integrated with the program there is no way of distinguishing code which has been generated reflectively from that which has been written in the normal way.

2.2 Type-Safe Run-Time Linguistic Reflection in O₂

O₂ [11] is an object oriented database system built around an object-oriented model which pays more than usual attention to support for orthogonal persistence. Programs in O₂ can manipulate both values (which may be of base type or be sets, bags, lists or records) and objects, where objects are members of classes which are arranged into a multiple inheritance graph. Each class is built on top of a type, usually a record type, and has methods associated with it. Application development is built in terms of a schema and each schema can be associated with a set of databases, a set of persistent roots, a set of application programs and a set of free standing functions, as well as sets of types and classes. Orthogonality is a guiding principal in that equivalent facilities are provided for values of all types and for objects of all classes.

O₂ comes with a distinguished schema called *Meta_schema* and this contains a number of classes whose members are meta-data. Note, first of all, that these are not true meta-classes in the sense mentioned in the introduction, since these classes hold, not classes as instances, but meta-information about classes and types. There is a class, *Meta*, which holds meta-information about schemata, and classes which hold meta-information about types and classes. O₂ also provides a method which makes it reflective. The class *Meta* has a method called *command* which sends an O₂ instruction to the O₂ interpreter. Since the O₂ system provides incremental schema development, this means that the program can change the schema at run-time, adding more code as new values are input.

To illustrate this with a trivial example, the following method can be executed against any object and will then add to the class of the object, a method which returns the value 123. The reflective method is added to the most general class, *Object*, in order to make it available to all objects and is:

```

method body add123Method in class Object
{ o2 Meta_class theClass = Schema→class( self );
  /* Returns class meta-data */

  o2 string className = theClass→Name;
  o2 string source =
    "method public Method123: integer in class " + className + ";" +
    "method body Method123: integer in class " + className + ";" +
    "{ return (123); }"
  o2 integer noErrors = Schema→command( source )
}

```

Since the meta-data can be accessed, the technique shown here can be used to **add** methods which are tailored to suit particular features of the type of the class. Reflection can also be used to write methods which add new classes or persistent roots automatically and also to write methods for the class *Object* which vary depending upon the type of the object. For instance, it has been found possible to write a method which displays all of the base data which is a component of the object.

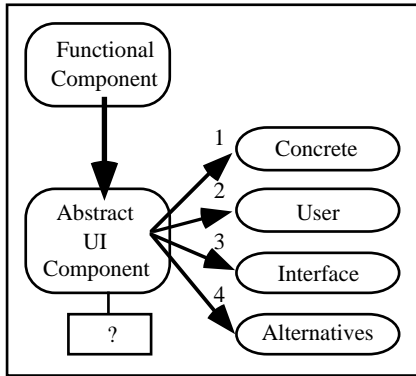
3 Uses of Reflection

The examples given in Section 2 have been restricted to trivial examples of compiling at run-time programs which could be written statically. Reflection comes into its own when creating code which cannot be written statically, but depends upon some information which will only be available at run-time. There are two sources of such information, which require rather different handling. Firstly, the extra information may come from input from the user during the run of the program. Secondly, the extra information may come from the meta-information about novel types of data being encountered by the program.

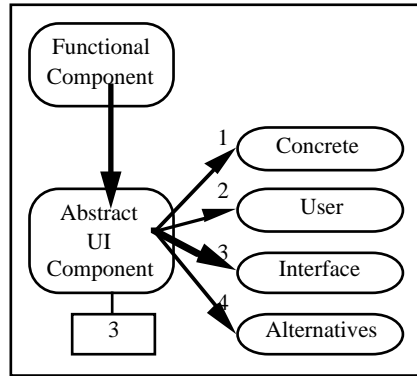
The first is primarily met in those systems which are intended to be user configurable. In such systems, the functionality is programmed in the form of a **template**, which the user may then customise. The template contains **placeholders** which are replaced by user choices. The configurable system is supported by a superstructure which permits multiple versions of the functionality to co-exist.

The second use is for programs which must support an *ad hoc* style of programming in the face of a potentially infinite set of types. In such cases, the basic functionality is again provided as a template, but now the placeholders are automatically filled by meta-information derived from the particular type encountered. For instance, the field names from records might be embedded in a placeholder reserved for the purpose. This use of reflection is inherently more difficult to manage than the user-configurable form since there is a possible complicating factor of recursion added to the process.

Examples will now be given of the two kinds of use, in the context of which the nature of the difficulties facing the programmer will be discussed.

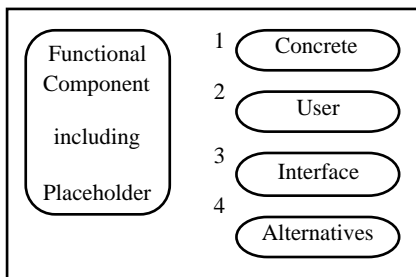


Unconfigured

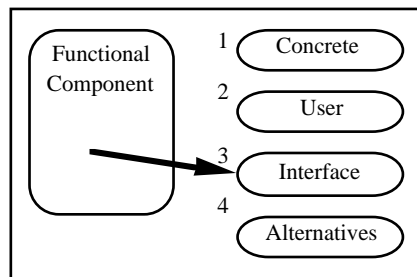


Configured

Configurability by Indirection

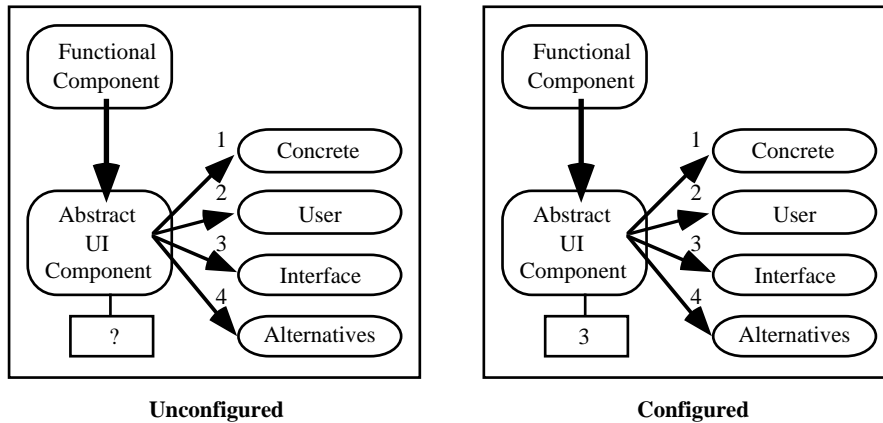


Unconfigured



Configured

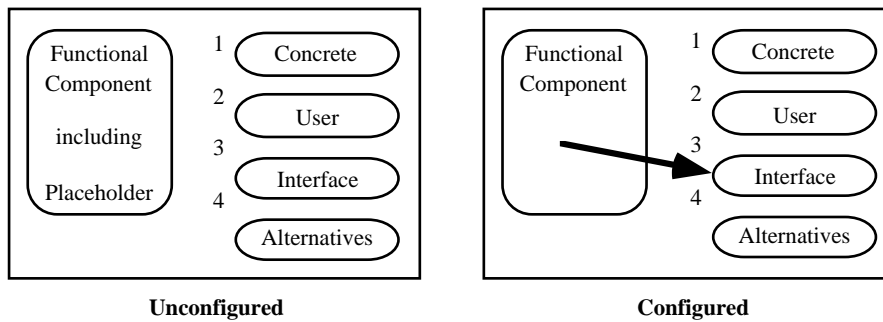
Configurability by Reflection



Unconfigured

Configured

Configurability by Indirection



Unconfigured

Configured

Configurability by Reflection

Figure 1 Two Ways of Providing Configurability

3.1 An Example of Using Reflection for Configurable Programs

The first class of uses for reflection comprises those programs which are provided as an extensible set of parallel versions. To take a common example, let us suppose that the user interface to a program is to be made customisable. The process of customising the interface consists of binding together two sets of software – the underlying functionality and a set of concrete interaction primitives. Presuming a system component which allows the person configuring the interface to identify and pair the components, there are two ways in which the configured interface can be built, illustrated respectively by the top and bottom halves of Figure 1:

- i) by **indirection** – the interaction needs of the functional components are met by calls to abstract modules – these modules include a switching command which chooses between the concrete versions of the interface component – the user supplies a choice and the abstract module uses this.
- ii) by **reflection** – the functional components are stored as source templates, which include place-holding slots where calls to the UI components should be – the user selection results in these being replaced by source which directly calls the selected component – the completed template is compiled and stored for future use.

The version resulting from reflection appears to be superior to that resulting from indirection in at least two respects. Firstly, the code is simpler. It is not difficult to imagine that there might be many different user interface components being connected in various ways. In this case the number of interconnections would grow quite quickly into a complex web. More importantly, however, the indirect approach has in-built inefficiency – for every call to an interface components, there must be an extra procedure call and a case statement. The disadvantage of the reflective approach is a loss of flexibility. In the example the interface could no longer be reconfigured simply by updating the switch value.

Another problem with the reflective approach is that the code for the functional component is really being written in a somewhat different language than that for a normal program and suitable support needs to be provided for this. To illustrate the point, a particular usage of reflection for configurability will now be discussed.

The Configurable Data Modelling System (CDMS) has been created as a prototype system in which user access mechanisms can be configured [14]. CDMS tackles the problem that whereas DBMS are designed for a wide range of user, the user interfaces provided are few in number and often poor in quality [15]. CDMS provides a component which allows novel access mechanisms to be created without recourse to repetitive low level programming. To this end CDMS allows the configuration of **user access mechanisms** (UAMs), where each UAM consists of a conceptual model tied to a concrete user interface. The conceptual model is built as an instantiation of a highly abstract generic model and each conceptual model can be associated with many user interfaces. The user interface style may be graphical, form-based or textual and the system is being built to bring all aspects of data management into a common framework. To this end, the generic conceptual is being enhanced with facilities to manage constraints and active values [16].

One aspect of CDMS is the part which configures textual interfaces – i.e. query languages [17]. The process of configuration takes in the following steps:

- i) The human configurer selects, names and constrains the constructs which make up the conceptual model – the constructs are either some kind of data value (for instance, entity, record or base value) or some relationship between (attribution, part-of or inheritance).

- ii) The system builds data structures to hold such values and primitive operations to manage them automatically – the operations allow values to be created, deleted, edited and displayed.
- iii) The configurer designs compound operations out of the primitive ones using a menu driven interface – for instance it is simple to create an operation to create an entity together with its attributes, or an operation to update selected values from a relation, or the **select .. from .. where** operation of SQL.
- iv) The system automatically builds parameterised procedures to implement the configurer’s design. The parameters are derived from the parts of the design that the configurer leaves unspecified.
- v) The configurer inputs a syntactic description for each operation which the query language is to support directly. This is entered in a flavour of Backus Naur Form.
- vi) The configurer associates the syntaxes with the operations, tying up slots in the syntax with parameters to the operation. There is a direct manipulation interface to achieve this.
- vii) A parser/interpreter is built to manage the syntax structures provided.
- viii) Finally the system takes all of the configurer’s decisions and builds and compiles an interpreter for the query language. This is stored in the set of available interfaces for the data model.

A typical pairing of a compound operation with syntax is the following which creates a relation. The syntax:

create table <rname> (<aname> : <adomain> { , <aname> : <adomain> }*)

is paired with the operation:

createRelation = **proc**(Rname:string; Anames:*string; Atypes:*string)

The software is built in Napier88 around an interpreter whose template looks as follows:

```

let interpreter = proc()  ! all interpreters are parameterless procedures
begin
    ...hardwired modules for expression management and persistence

    %$DataModelPrimitives%$      ! placeholder filled in at stage (ii)

    %$Compound Operations%$      ! placeholder filled in at stage (iv)

    ...hardwired code to manage user input and provide syntax analysis

    %$PARSER%$                    ! placeholder filled in at stage (vii)

    ... (mostly) hardwired evaluator  ! completed at stage (viii)
end

```

Each of the strings "%\$XXXXX%\$" denotes a placeholder which is replaced by strings derived from configuration decisions. The data modelling primitives are themselves built out of templates. Each kind of value and relationship has a template with placeholders for such information as the name of the construct in the configured conceptual model. The compound operations are built by glueing together pieces of code in accordance with the design. The parser is generated using standard techniques.

So, we see here the interpreter being written in a subtly different language from the normal. In order to make the coding tractable, an informal extension to the normal Napier88 syntax has been added. Placeholders always appear surrounded in "%\$" braces, so that a uniform set of support tools can be provided for instantiating them. Worse is the management of strings within the program. Since the whole program must be held in a string, strings inside the program must have their delimiters escaped, thus changing the syntax of the program. No completely satisfactory answer to this has been found, but perhaps the least inadequate response is to replace the usual string characters (" is the delimiter and ' is the escape character) with different characters which have no particular meaning in Napier88 (for instance, "£" for the delimiter and "\" for the escape character).

3.2 Using Reflection for Self-Adaptive Programs

The second set of uses to which reflection has been put is for program modules which are required to adapt themselves to varying types of input. This mechanism has been used extensively in Napier88 and its predecessor language PS-algol [18]. Among the uses have been the PS-algol browser [19] and a strongly typed relational system [20]. The former allows the user to navigate about values in the PS-algol persistent store, displaying an automatically generated menu for the current value, from which the user can traverse to a related value. The browser works from a template source module for the menu which it fills in with type specific information. By use of reflection, the browser is written entirely in PS-algol, without having to resort to type-unsafe access to the underlying structures in the

implementation environment. The relational system produced an abstract data type of algebraic operations for relations which managed relation-specific record structures. For instance, the *select* operation was generated specifically for each kind of relation encountered, thus avoiding the inefficiencies associated with an indirect approach.

To demonstrate how the technique is used, consider a procedure which is to display the value of any type which it is passed. To understand how this is programmed, it is necessary first to describe the subset of the Napier88 type system which the procedure can handle.

Napier88 supports the following types:

- a set of base types including **int**, **real**, **string** and **bool**;
- a constructor for arrays (called **vectors** in Napier88) – for any type *t*, the type **t* is the type of a vector whose values are of type *t*;
- a constructor for records – for instance, the type **structure**(name: **string**; age: **int**) may be the type of a record holding information about a person;
- a constructor for variants – for instance, the type **variant**(number: **int**; text: **string**) might be a general type for identifying "numbers", which are either integers or strings;
- the type **any** is the infinite union of all types. Thus **any**(123) and **any**("ABC") have the same type **any**. For example a value *X* of type **any** could be used as follows:

project X as Y onto

```

int:      { writeString( "int:" ) ; writeInt( Y ) }
string:  { writeString( "string:" ) ; writeString( Y ) }
default: writeString( "not int or string" )

```

The language also supports graphical types, extensible environments, procedures and abstract data types, but these will be ignored in the following discussion. Napier88 also allows the description of parameterised type operators, for instance it is possible to define a type of homogeneous pairs:

```

type pair[t] is structure( one, two: t )

```

meaning that any instance of this type is a record of two values with the same type. Type operators must be specialised before use so any reflective procedure does not have to deal explicitly with such values.

The requirement in this example is for a procedure *Print*, which takes a value of type **any** and then prints out its value in the following way:

- if the value is an instance of a base type, print it;
- if it is a vector, print out the values of its components;
- if it is a record, print out the names and values of its fields;

- if it is a variant, print out which variant it is and its value.

The following code handles the case if the value injected into the any is an integer:

```
let intPrint = proc( X: any )
  project X as Y onto
    int: writeInt( Y )
    default: writeString( "error: not an int" )
```

To complete the handling of base type values, equivalent procedures are produced for **string**, **real** and **bool**.

There is now a need for a context in which these alternative procedures are built. To manage this, an associative access mechanism such as the *Map* bulk type [21], is useful. A Map of **string** to **proc(any)** is created and initially populated with the pairings "int" to intPrint, "string" to stringPrint, etc.

The overall strategy for our *Print* procedure can now be described:

- i) obtain a string representation of the underlying type of the value;
- ii) look up the Map to find if a version for this type already exists – if so retrieve and use it;
- iii) if it is not there, build a procedure, similar in structure to *intPrint*, compile it, store it in the Map and then use it.

To support the building of these procedures, templates are created for each of the type constructors for vectors, records and variants. Outlines of these now follow:

```
vectors for i = lwb[%$TY%$(Y) to upb[%$TY%$(Y) do Print( any(Y(i)) )
```

```
records for each field: writeString( "%$FN%$" ); Print( any(Y(%$FN%$)) )
```

```
variants project Y as Z onto
  %$VN%$: Print( any(Z) )
```

In these we see placeholders for the type of the components ("%\$TY%\$") of the vector, the names of fields ("%\$FN%\$") in the record template and the variant branches ("%\$VN%\$") in the variant template. Appendix A shows the full details.

To summarise, the need for reflection of this kind arises in the context of programs which are expected to handle a potentially infinite number of types of values and yet do so in a type secure manner. The technique of reflection resolves the conflict between requiring both generality and type security by supporting the creation of a program which generates the versions which the programmer would have written had he or she been able to anticipate the types which actually arise. However, a glance at Appendix A will demonstrate that the writing of such programs is a complex task – and remember this is only a print procedure, hand picked since it is a simple and useful reflective program. Yet notice, that if another reflective procedure was required, *a very similar program would result*. In essence

the only pieces of the code which specify that this is a print routine are the name ("Print"), the signature ("proc(X:any)") and the pieces of code shown in outline font – i.e. one piece of code for each base type and one for each type constructor. This points the way both towards some tools for simplifying the task of writing reflective programs and also towards an underlying theory of what is actually required for reflection in other programming systems.

4 Support Tools for Reflection

This section describes the tools that have been implemented to make writing reflective programs easier. The context will once more be Napier88, but similar tools will be required in any equivalent environment. The first subsection outlines the tools which support both uses of reflection in that they make easier the management of the supporting structures – the set of equivalent versions and the templates. The second subsection describes a more powerful tool which allows programs such as the one shown in Appendix A to be generated from the essential code fragments which distinguish different reflective programs.

4.1 Basic Support Tools

The first important technique which has been introduced is the notion of a template [13]. A template is a string containing mostly Napier88 code with two exceptions:

- the delimiters and escape characters of strings are replaced by different characters, since, as they need to be embedded in strings they would otherwise perturb the normal compilation of the code – "£" and "\" are used for the delimiter and escape character respectively;
- at some points where type specific or user supplied detail is required, the code is replaced with a placeholder – which is identified as being a string surrounded by "%\$" delimiters.

The first requirement is for procedures which manage these differences:

stringReplace: **proc(template: string → string)** is a procedure which takes in a template and replaces all instances of "£" and "\" with appropriately escaped Napier88 versions.

replace: **proc(this, that, template: string → string)** is a procedure which takes in a template and replaces all instances of "%\$"+*this* ++ "%\$" by *that*.

The second technique that has been introduced is the Map which holds the versions, and an important aspect of this is the generation of appropriate unique values for the keys. This requirement is met by one of the procedures which are needed to generate and analyse the type description for embedding in the template:

getType: **proc**(X: any → typeRep) generates a structured value holding all of the component information of the type of the input value;

describeType: **proc**(T: typeRep → **string**) generates a legal Napier88 type expression for the type – it is essential that it be possible to introduce complete type specifications into the generated code;

componentType: **proc**(T: typeRep → **string**) generates a type description of the component type of a vector;

fieldNames: **proc**(T: typeRep → ***string**) generates a vector of strings being the names of fields of a record type;

variantNames: **proc**(T: typeRep → ***string**) generates a vector of strings being the names of variants of a variant type.

makeDefault: **proc**(typeRep → **string**) generates a string containing a default value for the type – for instance if the type were **real**, this would return "0.0". This is required to handle procedures which return a result – see below.

In fact the output of *describeType*, being a unique description of the type, is used for the key to the Map. The construction of an appropriate structure to hold type representations is discussed at length in [22].

4.2 A Reflective Program Generator

This section tackles the problem of creating a generator for reflective programs. In essence, it employs the technique described in section 3.1 to provide a support tool for generating the kinds of program discussed in section 3.2. The basic component is a procedure, called *makeReflect*, which takes in a number of strings which describe the name and signature of the procedure, and one string each containing the required functionality when each of the base types and one or more strings for each of the type constructors¹ is encountered.

To illustrate this, the following call would be sufficient to generate the program in Appendix A:

```
makeReflect( "Print",           ! the name
            "proc( X: any )",   ! the signature
            "writeInt( Y )",    ! integer functionality
            "writeBool( Y )",   ! boolean functionality
            "writeReal( Y )",   ! real functionality
```

¹There is also one other parameter in which the programmer must provide details of how to retrieve required values (including the system functions) which are used in the code fragments. This is omitted for brevity, but is just a single piece of pure Napier88 code.

```

"writeString( Y )",           ! string functionality
"",                           ! vector functionality before the loop
"Print( any( Y(i) ) )",      ! vector functionality inside the loop
"",                           ! vector functionality after the loop
"",                           ! vector functionality before the fields
"writeString( £%$FN%$: £ )   ! functionality for each record field
Print( any( Y( %$FN%$ ) ) )
writeString( £\n£ )",
"",                           ! vector functionality after the fields
"writeString( £%$VN%$: £ )   ! functionality for each variant
Print( any(Z) )
writeString( £\n£ )" )

```

Use of the procedure *makeReflect* greatly simplifies the creation of reflective procedures since the programming task has been reduced to its essential core – i.e. what code should be provided for each type of data encountered. In order to use it though, certain conventions must be met when writing the code fragments:

- there is only one reflective variable, which must be called *X* and be declared to be of type **any**;
- where the underlying type of *X* is used, it must be called *Y*; and
- if *X* is itself a variant, then the projected value must be called *Z*.

The procedure works by turning the program in Appendix A into a configurable template by replacing all of the functionality (i.e. the pieces of code in outline font), the procedure name, signature, type and parameter lists (i.e. those pieces of code which are underlined) by place holders which are substituted by the arguments of the procedure. The program has also to be modified because now some parts of the code are doubly reflective, since *makeReflect* generates a program which itself generates programs. This means that we have strings inside strings inside strings and this requires careful handling.

This template works only for procedures which do not return a result and a different template is required for those which do – since the syntactic context of these are different. In essence, the difference is that a result value is declared at the top of the procedure (the creation of a default value for any type is a non-trivial piece of code) and each block of functional code is turned into a block whose value is passed to the return variable. Thus the following would be created for a procedure which returns a real value:

```

let result := 0.0 ! generated by makeDefault
let theMap = m_empty[string, proc( any → real ) ]() ! create map
...
let intVersion = proc( X: any → real )
    project X as Y onto
        int: result := begin
            writeInt( Y ) ! sample functionality for an
            5.3 * float( Y ) ! integer parameter
        end
        default: writeString( "Will not get here" )
...
etc.

```

5 Conclusions

5.1 Towards a Theory of Reflective Programs

This section attempts to bring together the previous discussion and place it in a more general context. Reviewing the previous sections:

- Applications which manage long-lived data are best programmed in a type secure environment, since a great number of run-time errors, which potentially could occur many years after the program was written, can be eliminated.
- Many programs which require type specific information as part of their code are difficult to write efficiently in strongly typed languages.
- Similarly, programs which are intended to be user-configurable, if programmed normally, will tend to become inefficient.
- Linguistic reflection is a technique which attempts to solve these problems by allowing programs to be written which extend themselves. The technique allows type-specific versions of procedures to be generated as new types of data are encountered. It also allows template programs to be configured from user input. In both cases the program generated is that which would have been written had a programmer been on hand to write it.
- The technique involves building new code fragments as strings which merge template versions of the code with type-specific or user-provided input. The strings are then compiled and stored for later re-use.
- Support tools are required to make the creation of such programs tractable. Tools were discussed which manage templates with placeholders, the extraction of type information and the storage of versions of compiled programs.
- Finally, a tool, *makeReflect*, was discussed which enables reflective programs to be built more easily, by providing just the functional code.

The structure of the procedure *makeReflect* was largely determined by the type system of Napier88 (or rather that part of the type system which it can currently manage). Essentially, *makeReflect* can be analysed as comprising the following parts:

- a store of versions, indexed by the type they manage;
- a non-type-specific template which provides an abstract structure for housing any application;
- type-specific templates which are embedded into the general template; and
- a program which takes in one piece of functional code for each construct in the type system and embeds it in its type-specific template.

The possibility therefore exists of using this structure to generate a theory of reflective programs, which may proceed as follows.

Let $T = \{ \text{TOP}, \text{BT}_1, \dots, \text{BT}_m, \text{TC}_1, \dots, \text{TC}_n \}$ be a type system with m base types and n type constructors and a type, **TOP**, which is the union of all other types. It then becomes possible to provide reflection by supplying two system functions:

typeDesc: **TOP** \rightarrow **string** returns a unique syntactically correct description of the type; and

compile: **string** \rightarrow **TOP** returns a compiled form of the string.

It is then possible to use these for the creation of programs of the form:

program: **PROGRAM**

where type **PROGRAM** = **TOP**, $T_1, \dots, T_p \rightarrow \mathbf{RT}$

in which the first parameter is a reflective, there are p other parameters of fixed types and one result type, **RT**. It is also possible to supply support tools which perform as follows:

enter: **string**, **PROGRAM** which stores the pairing of a type description and a program version for that type;

baseTypeVersion _{i} : **PROGRAM** for $i = 1$ to m , which instantiates the program for that base type;

typeConstructorTemplate _{j} : **string** for $j = 1$ to n , which contains a generalised template for handling values of each constructed type;

*typeConstructorGenerator*_j: **string** → **PROGRAM** for $j = 1$ to n , which takes in a type description and returns a version for the type described by the string;

generator: **PROGRAM** which brings the *baseTypeVersion*_i; and *typeConstructorGenerator*_j into a common program which can handle all types;

makeReflect: **string, string ... string** which takes in one code fragment for each base type and type constructor and returns a program generator for the supplied functionality.

Keeping this as a general model, it should become possible to provide linguistic reflection for any language which has a most general type, such as **any**, and the ability to manage code fragments as first-class values.

5.2 Research Directions

The discussion so far has centred on linguistic reflection using string representations of program fragments. The advent of hyper-programming [5], which allows programs in a persistent environment to contain direct links to values and types in that environment, raises further possibilities for reflection. The ability of generators to manipulate hyper-program representations rather than strings provides a safe, convenient and efficient mechanism by which generated code may access values and types in the generator evaluation environment. This also removes the need to flatten complex type representations to strings [23].

Another area for research is the incorporation of the structured program representations and pattern matching found in TRPL into a run-time reflective system.

Acknowledgements

The work has been developed in the context of collaborative research between the Persistent Programming Research Groups led at the University of St Andrews by Professor Ron Morrison and at the University of Glasgow by Professor Malcolm Atkinson. Among those whose influence have been vital to the development of these ideas have been Al Dearle, Fred Brown, Richard Connor, Quintin Cutts, Paul Philbrow and Zhenzhou Qin. Miguel Mira da Silva provided very useful comments on an earlier draft of this paper. The work has been supported by ESPRIT Basic Research Activity 6309, FIDE2, and by SERC Research Grants H17671, Configurable Data Modelling, and GR/F 02953.

References

1. Atkinson MP, Bailey PJ, Chisholm KJ, Cockshott WP, Morrison R. An Approach to Persistent Programming. *Comp. J.* 1983; 26,4:360-365

2. Cardelli L, Wegner P. On Understanding Types, Data Abstraction and Polymorphism. *ACM Comp. Surveys* 1985; 17,4:471-523
3. Cardelli L. Amber. AT&T Bell Labs, Murray Hill Report AT7T, 1985
4. Stemple D, Stanton RB, Sheard T et al. Type-Safe Linguistic Reflection: A Generator Technology. ESPRIT BRA Project 3070 FIDE Report FIDE/92/49, 1992
5. Kirby GNC. Reflection and Hyper-Programming in Persistent Programming Systems. Ph.D. thesis, University of St Andrews, 1992
6. Wirth N. Programming in Modula-2. Springer-Verlag, 1983
7. McCarthy J, Abrahams PW, Edwards DJ, Hart TP, Levin MI. The Lisp Programmers' Manual. M.I.T. Press, Cambridge, Massachusetts, 1962
8. Burstall RM, Collins JS, Popplestone RJ. Programming in POP-2. Edinburgh University Press, Edinburgh, Scotland, 1971
9. Sheard T. Automatic Generation and Use of Abstract Structure Operators. *ACM ToPLaS* 1991; 19,4:531-557
10. Morrison R, Brown AL, Connor RCH et al. The Napier88 Reference Manual (Release 2.0). University of St Andrews Report CS/93/15, 1993
11. Bancilhon F, Delobel C, Kanellakis P. The Story of O₂: Building an Object-Oriented Database System. Morgan Kaufmann, 1992
12. Cooper RL. On The Utilisation of Persistent Programming Environments. Ph.D. thesis, University of Glasgow, 1990
13. Kirby GNC. Persistent Programming with Strongly Typed Linguistic Reflection. In: Proc. 25th International Conference on Systems Sciences, Hawaii, 1992, pp 820-831
14. Cooper RL. Configurable Data Modelling Systems. In: Proc. 9th International Conference on the Entity Relationship Approach, Lausanne, Switzerland, 1990, pp 35-52
15. Stonebraker M, Agrawal R, Dayal U, Neuhold EJ, Reuter A. DBMS Research at a Crossroads: The Vienna Update. In: Proc. 19th International Conference on Very Large Databases, Dublin, 1993, pp 688-692
16. Cooper RL, Qin Z. A Generic Data Model for the Support of Multiple User Access Mechanisms. *submitted*
17. Cooper RL. Configuring Database Query Languages. *submitted*
18. PS-algol Reference Manual, 4th edition. Universities of Glasgow and St Andrews Report PPRR-12-88, 1988
19. Dearle A, Brown AL. Safe Browsing in a Strongly Typed Persistent Environment. *Comp. J.* 1988; 31,6:540-544

20. Cooper RL, Atkinson MP, Dearle A, Abderrahmane D. Constructing Database Systems in a Persistent Environment. In: Proc. 13th International Conference on Very Large Data Bases, 1987, pp 117-125
21. Atkinson MP, Lécluse C, Philbrow P, Richard P. Design Issues in a Map Language. In: P. Kanellakis and J. W. Schmidt (ed) Bulk Types & Persistent Data. Morgan Kaufmann, 1991, pp 20-32
22. Connor RCH. Types and Polymorphism in Persistent Programming Systems. Ph.D. thesis, University of St Andrews, 1990
23. Kirby GNC, Connor RCH, Morrison R. START: A Linguistic Reflection Tool Using Hyper-Program Technology. To Appear: Proc. 6th International Workshop on Persistent Object Systems, 1994

Appendix A - A Reflective Program to Print Any Value

The core of the program is the template, assigned to the variable *source*. This holds two placeholders - one for a description of the type and one for the detailed code which is required for this type. This code is built up by extracting information from the type description. There is a different piece of code for managing each kind of type (chosen by the **case** statement at the top of the second page). Eventually, the variable *source* holds a complete procedure for handling values of the type and this is compiled, stored and used.

! Set up the context - the map and the base type versions

```
let theMap = m_empty[string, proc( any ) ]() ! Create the map
let enter = proc( S:string; P:proc( any ) ) ! A procedure to add an entry
    m_isu_insert[string, proc( any )]( theMap, S, P )
let intVersion = proc( X: any ) ! Create the integer version
    project X as Y onto
    int: writeInt( Y )
    default: writeString( "Will not get here" )
enter( "int", intVersion ) ! Enter the integer version
let boolVersion = proc( X: any )
    project X as Y onto
    bool: writeBool( Y )
    default: writeString( "Will not get here" )
enter( "bool", boolVersion ) ! Enter the boolean version
let realVersion = proc( X: any )
    project X as Y onto
    real: writeReal( Y )
    default: writeString( "Will not get here" )
enter( "real", realVersion ) ! Enter the real version
let stringVersion = proc( X: any )
    project X as Y onto
    string: writeString( Y )
    default: writeString( "Will not get here" )
enter( "string", stringVersion ) ! Enter the string version

let error = proc( S: string ) ! An error message procedure
    writeString( "n**** ERROR: " ++ S ++ "n" )
```

! Templates for the constructed types.

```
let vectorTemplate =
    "for i = lwb[%$TY%$]( Y ) to upb[%$TY%$]( Y ) do Print( any( Y(i) ) )"
let structureTemplate =
    "writeString( £%$FN%$ £ );
```

```

Print( any(Y(%$FN%$)) );writeString(£\n£)"
let variantTemplate = "%$VN%$: begin
                        writeString(£%$VN%$: £ )
                        Print( any(Z) )
                        writeString( £\n£ )
                        end"

```

! The main procedure.

```

let Print = proc(X:any)
begin
  let T = getType( X )           ! Get the type of the value
  let S = describeType( T )     ! Get a string description
  if m_contains[string, proc( any )]( theMap, S )
  then begin                    ! If there is a version for this type
    let P = m_find[string,proc( any )](theMap,S) ! extract and use it.
    P( X )
  end
  else begin ! Otherwise
    let source := "proc( X: any ) ! Start with a template for the
begin ! procedure
  rec type t1 is %$TYPEDESC%$
  project X as Y onto
  t1: begin
    %$DETAIL%$
  end
  default: writeString( £Will not get here£ )
end"
    source := replace( "TYPEDESC", S, source ) ! put in actual type

  case kind(T) of
    "base": begin
      error( "Unknown Base Type" )
      source := ""
    end
    "vector": begin ! insert the template and component type
      source := replace( "DETAIL", vectorTemplate, source )
      source := replace( "TY", componentType( T ), source )
    end
    "structure": begin
      let heart := "" ! build a string containing
      let FNS = fieldNames( T ) ! one line for each
      for i = lwb[string](FNS) to upb[string](FNS) do
        heart := heart ++
          replace( "FN", FNS(i), structureTemplate )
      ! put this in the source

```

field

```

        source := replace( "DETAIL", heart, source )
    end
"variant": begin
    ! build a string containing
    ! one line for each variant
    let heart := "project Y as Z onto'n"
    let VNS = variantNames( T )
    for i = lwb[string](VNS) to upb[string](Ftypes) do
        heart := heart ++
            replace( "VN", VNS(i), variantTemplate)
        heart := heart ++ "default: writeString( £Bad Variant£ )"
        ! put this in the source
        source := replace( "DETAIL", heart, source )
    end
    default: begin
        error( "Sorry - cannot deal with this type" )
        source := ""
    end
end

if source ~= "" do
begin
    let object = compile( source )          ! compile the source
    project object as O onto
    string: writeString( "COMPILE FAILED'n" ++ O ++
        source )          ! error
    proc( → any ):          ! success
        project O() as P onto
        proc( any ):      ! extract the procedure
            begin
                enter( S, P )          ! store it
                P( X )                ! and use it
            end
        default: writeString( "Will not get here'n" )
        default: writeString( "Will not get here'n" )
    end
end
end
end

```