Persistent Programming with Strongly Typed Linguistic Reflection

G. N. C. Kirby

University of St Andrews, North Haugh, St Andrews, Fife KY16 9SS, Scotland.

Abstract

The technique of linguistic reflection is of particular interest in persistent systems because it can allow long-lived data and programs to evolve in a type-safe manner. Existing reflective languages are hard to use because programs contain a mixture of several different kinds of code, with respect to their role in reflection. In some systems this problem is compounded by the presence of a high level of syntactic noise. The paper discusses some uses of strongly typed linguistic reflection in a persistent system and describes an attempt to improve the programmer's interface to reflection. This involves designing an extension to the strongly typed persistent language Napier88, called TemplateNapier. The paper also identifies some factors which make it difficult to write and to understand reflective programs.

1 Introduction

1.1 Linguistic reflection

A reflective language is one that has facilities for constructing and executing new code from within a program. The following shows a simple example of reflection in a hypothetical language:

let a = reflect("7") + 2

Here the function *reflect* is used at run-time to compile the representation "7" into executable code and to execute that code to produce the integer value 7. The identifier *a* is assigned the integer value 9 obtained using the result of the evaluation.

An alternative strategy is to interpret the representation, rather than to compile it and then evaluate it. An example of this is the *eval* function of Lisp [MAE62]. This paper however is concerned primarily with languages in which the reflective process involves compilation.

In general linguistic reflection involves taking some language value which represents source code and evaluating it to produce another language value. In the example a string was used to represent the code and an integer was produced. The reflection process can take place either at run-time, as in the example above, or at compile-time, and may or may not be strongly typed.

1.2 Strong typing

Linguistic reflection is of particular interest in persistent systems because it provides a way for long-lived data, including programs, to evolve in a type-safe manner. In any long-lived data-intensive application the form of the data will inevitably be subject to change over time. As strong typing is vital for maintaining the integrity of persistent data, this paper will be limited to the consideration of strongly typed reflection, which has been called type-safe linguistic reflection [SSS91]. With this form of reflection the type of any code produced by a program is checked before it is used, to ensure that it is consistent with the intended use. The technique has been used:

- to provide levels of genericity beyond conventional polymorphism [CW85,Str67] in programming languages [She90];
- to build adaptive object browsers [DB88,DCK89,KD90]; and
- as an implementation technique for data models [CAD87, Coo90].

1.3 Benefits of reflection

Consider two strongly typed languages A and B with the same type system, where A supports reflection and B does not. Given that they are both Turing complete, B can express any computation that can be expressed in A. However, some computations which describe highly generic problems may admit a more efficient implementation in language A than in B. These are the computations that cannot be fitted directly into the type system: any non-reflective implementation of the computations must involve interpretation. Using the reflective capabilities of language A may allow a more efficient compiled solution.

This can be illustrated with the natural join function. Assuming neither language has built-in support for relational operations, the most direct way to represent relations might be as sets of records. However in language B it is not possible to write a generic join function to operate over arbitrary sets, because the type of the result set depends on the combination of attribute names used in the records in the input sets but the program cannot compute over those names as they are not values in the language. The only solution is to use a more flexible representation for the tuples of the relations. Instead of records, arrays of (attribute-name,value) pairs can be used, where the attribute names are strings, bringing them into the domain of computation. The disadvantage of this approach is that some static type constraints are lost, so dynamic checks must be made on each call of the generic function to ensure that the inputs represent valid relations.

In language A, however, the set representation can be used. For every occurrence in the program of a join on a particular pair of relations, the representation of a join function specific to their types is generated. Using reflection the representation is transformed into an executable function which is applied to the relations. So long as the type of the function is checked when it is first produced it can be used repeatedly without further type checking.

1.4 Requirements for reflection

Mechanisms for the following activities are required to support strongly typed reflection in a language:

- building and manipulating representations of the constructs of the language;
- transforming those representations into executable form (often achieved by the compiler itself);
- and type checking the programs so constructed (often achieved by parts of the compiler).

For run-time linguistic reflection a further activity must be supported:

- strongly typed dynamic binding of the values created by reflection into running programs.

1.5 Goals

The goals of this paper are:

- to identify the advantages of strongly typed reflection and its relevance to persistent programming;
- to describe an extension of Napier88 [MBC89] which represents an attempt to integrate support for reflective programming into an existing language;
- and to identify factors affecting the understanding and construction of reflective programs.

2 Reflection and Persistence

It was stated earlier that reflection can allow solutions to a certain class of problems, highly generic functions, to be implemented more efficiently than would otherwise be possible. This section identifies the ways in which efficiency may be gained, describes an example of reflection in use, and examines the impact of persistence on a reflective language system.

2.1 Genericity with reflection

2.1.1 Features of reflective solutions

Section 1.3 gave an outline of the way in which reflection can be used to write a generic function. To recap, it involves using a generator which, for each call of the function, produces a representation of code for an instance of the function specialised for the appropriate types. This representation is compiled to produce the specific function required. The key features of this method are:

- Strong typing is preserved by a type check on the result so that no type errors can occur.
- The method is more efficient than interpretation because the type checking occurs only at the time of the compilation while the result can be used many times. Other computations may also be compiled away where they depend only on the types of the function arguments, for example the construction of the algorithm that determines whether two tuples match in natural join.
- The types of the values manipulated by the generic function may be specified more precisely than with the interpretive method, reducing the number of dynamic checks required and increasing efficiency. For example with the reflective implementation of natural join there is no need to check for duplicate attribute names in the input relations as each tuple is represented by a record. A check would be required with an interpretive implementation.
- Using reflection allows bindings, checks and other computations to be made earlier than with interpretation. This increases safety; the programmer can be more confident that the generic function will execute correctly.

2.1.2 Example: natural join

This section shows how a generic natural join function can be implemented using reflection in the language Napier88 [MBC89]. This language supports run-time reflection only, although it will be described in Section 2.2 how run-time reflection coupled with persistence gives all the power of compile-time reflection. Stemple et al. give a description of generic natural join using compile-time reflection in the language TRPL [SFS90].

The problem involves writing a generic natural join procedure which will work for relations with tuples of any record type. The difficulty for any non-reflective strongly typed language is in expressing the constraints that the join procedure should be defined only over sets of record types and that the type of the result relation depends on the types of the input relations. Another problem is expressing the algorithm which computes that result type.

Figure 2.1 shows the outline of a Napier88 program which defines a generator procedure that, given representations of the types of a pair of input relations, produces the representation of a join procedure specific to those types. This representation is then compiled and the resulting procedure can be used to perform a join on any relations of those types. The reserved words in the program are written in bold type.

The program starts with definitions of the types *typeRep* and *codeRep*. The details are not shown: any representations of Napier88 types and source code can be used, so long as they are agreed upon by the writers of the generator, the compiler and the code that constructs the type representations passed to the generator. The type definitions are followed by the definition of the procedure *joinGenerator*. The procedure takes two parameters *relation1Type* and *relation2Type* of type *typeRep*. The result of the procedure is a value of type **any** which is the infinite union of all Napier88 types. The body of the procedure constructs a representation of a procedure to perform the join for the particular relation types specified by the parameters, and assigns it to the identifier *joinProcedureRepresentation*. The code which constructs this representation has not been shown for brevity as it is fairly complex. It must analyse the type representations in order to synthesise the result type of the join and the algorithms to determine when pairs of tuples match on their common attributes and to construct new tuples from those which do match.

```
type typeRep is …              ! Representation of Napier88 types.
type codeRep is …  ! Representation of Napier88 code.

let joinGenerator = proc( relation1Type,relation2Type : typeRep → any )
begin
        ! Construct representation of a join procedure for this particular pair of relation types.
        let joinProcedureRepresentation =
                …       ! Of type codeRep.

        compile( joinProcedureRepresentation )        ! Return the compiled result.
end
```

Figure 2.1: A generic reflective procedure

The reflection is contained in the call to the procedure *compile* which compiles the code representation to produce an executable procedure and injects it into the infinite union type **any**. This allows *compile* to have a well defined type, **proc**(codeRep → **any**), even though the actual type of its result is not known statically. The **any** value is returned as the result of *joinGenerator*. To use that result it must be projected onto its specific type, shown in Figure 2.2 where the reflective procedure is used to perform a natural join on two relations. The program begins by defining the tuple types of the two relations to be joined, *part* and *supplier*, and the expected tuple type of the result, *partSupplier*. It is assumed that the type *Relation*, parameterised by a tuple type, has been previously defined. The program next constructs representations of the input relation types and passes them to the generator *joinGenerator* to obtain an **any** containing a join procedure specific to those types. The identifier *psJoin* is bound to that procedure. To do this the **any** must be projected onto a specific type: the program specifies the expected type of the procedure, written after the reserved word **onto**. If the type is correct it is the new procedure value, renamed *join*, which is bound to *psJoin*. Otherwise an error is reported and a dummy procedure (*dummyJoin*, assumed to have been defined earlier) is bound. Such a failure would only occur if there was an error in the generator procedure which produced the join procedure representation.

```
type part is structure( partName : string ; partNumber : int )
type supplier is structure( supplierName : string ; partNumber : int )
type partSupplier is structure(   partName : string ; partNumber : int ;
                                  supplierName : string )

! Representations of Relation[ part ] and Relation[ supplier ].
let partRelationRep = …
let supplierRelationRep = …

let wrap = joinGenerator( partRelationRep,supplierRelationRep )
let psJoin = project wrap as join onto
        proc( partRelation,supplierRelation → partSupplierRelation ) : join
        default : { write( "compilation failure" ) ; dummyJoin }

let parts = …                        ! Construct instance of Relation[ part ].
let suppliers = …                    ! Construct instance of Relation[ supplier ].

let partsAndSuppliers = psJoin( parts,suppliers )
```

Figure 2.2: Using a reflective procedure

A new program like that in Figure 2.2 must be written and compiled for each different pair of relation types to be joined. Though the types of the inputs and of the result must be written down in each case, the algorithms to perform the different joins are synthesised automatically.

The dynamic type check which takes place at the projection of the **any** value ensures that the reflection process is entirely strongly typed: even if the representation produced by *joinGenerator* was not valid code there would be no threat to the integrity of the language system as the **default** branch would be followed. This illustrates a restriction of this style of reflection. Although it is possible for a generator to produce the representation of some value whose type is not known statically, such a value cannot be used in a program because a static type assertion must be made at the point of the projection from **any**.[1]

2.2 The impact of persistence

Persistence allows the programmer to view data as having a single form throughout its life, removing the need for explicit translations between short-term program formats and long-term storage formats. The language Napier88 provides orthogonal persistence, that is any data value may persist for an arbitrary length of time irrespective of its type. This has the benefits that:

- it removes the need to write code to translate long-term data between different formats;
- and it allows values of all types including closures and abstract data types to persist beyond a single program execution. It is not possible to arrange this in a strongly typed language without a built-in persistence mechanism as there is no way to access all the details of such values in order to 'flatten' them to write them to the file system, or to reconstruct the values from flattened representations.

Persistence is relevant to reflection in several ways: reflection can be used to solve some problems associated with large persistent systems; the use of persistence allows some reflective algorithms to be implemented more efficiently; and persistence allows run-time reflective languages to simulate the capabilities of compile-time reflective languages. These will be illustrated in turn.

2.2.1 Data evolution

One problem with large, data-intensive, strongly-typed systems is that of data evolution: the form of the data will change as the applications which use it are inevitably modified. With large amounts of data there is a need for generic tools to organise the update of existing data and ensure that it remains consistent. Some of these tools can be built using reflection.

For example, it might be necessary to change an attribute name of a relation in the persistent store. A non-reflective procedure could be written to create a new relation with appropriate attribute names and copy the data into it from the existing relation. However, that procedure would have built into it the types of both the existing and new relations, giving it little potential for reuse.

Instead a generator can be written, parameterised by a representation of the type of the existing relation and the attribute name to be changed, which generates the representation of a procedure to perform the update. Reflection can then be used to convert that representation to an executable form. The benefit of this approach is that the same generic procedure can be used for changing any attribute name in any other relation.

2.2.2 Reflective algorithm implementation

The process of compilation is relatively expensive—so the reflective technique described gives

[1]In fact this is not quite true: the value could be stored or manipulated in its **any** form, thus browser technology [DB88,DCK89,KD90] could be used to discover its structure.

the greatest efficiency gains over interpretation in cases where the value produced by reflection is used many times after its creation. Persistence allows the compilation cost to be amortised over many program executions rather than a single one. This can be done by storing the compiled forms of all calls to a generator in a persistent table, keyed by the type representations used to create them. Before another call to the generator is made the table is scanned to discover whether the current type representations have been used before. If so the corresponding value in the table is used and no compilation is necessary.

This technique can be applied to the examples of natural join and data evolution given earlier. The degree of efficiency gained depends on several factors including the size to which the tables grow, the frequency of hits, and the cost of scanning the tables. Note that entries can be deleted from the tables if necessary without affecting anything other than the speed of future calls.

2.2.3 Run-time and compile-time reflection

Orthogonal persistence allows generators to be persistent, giving control over the time at which the compilation costs are incurred. There is as always a trade-off between flexibility and efficiency. Consider for example a generator which produces procedures to display values of any type, as might be used in a store browser application. One strategy is to keep the generator in the persistent store and evaluate it whenever the browser is executed, giving a procedure specific to the required type which can then be called. This gives high flexibility and low storage overhead, at the cost of a compilation on every execution.

Alternatively the generator can be evaluated repeatedly for selected types at the time the application is built, and only those procedures produced made persistent. This gives greater efficiency at execution time, but the browser is now less flexible: it will only work for the types which the programmer thought of at the outset. The storage required is also greater, with wastage if some of those types are never encountered. Finally a persistent cache can be used, as described in Section 2.2.2. This is a scheme which has been used to build browsers [DB88,DCK89,KD90].

There exist applications of run-time reflection which cannot be implemented with compile-time reflection: one example will be shown in Section 3. Furthermore, run-time reflection has all the power of compile-time reflection. Because generators can be persistent, the reflection can be performed as early or as late in the program construction process as required.

3 The Language TemplateNapier

A major problem with existing reflective languages is that it is difficult to write and to understand reflective programs in these languages. The messier details of the generator procedure in Figure 2.1 were omitted but further examples of reflective programming in several languages are given in Section 4, where issues affecting the programs' understandability are discussed. This section describes a new version of the Napier language designed to improve the interface to the reflective facilities.

The language TemplateNapier is derived by extending Napier88 with a new language construct, the **template**. A template is a generator: it takes as parameters both types and values and produces source code representations. When a template is evaluated at run-time the TemplateNapier source code produced is compiled and, if successful, the result bound into the program. In the current version of TemplateNapier the type **string** is used to represent source code and the form of the language reflects this. Future experiments will involve other representations such as abstract syntax graphs.

3.1 Template definitions

A template definition begins with the reserved word **template** followed by any number of type parameters enclosed in square brackets and value parameters in round brackets. Following the template header is a body whose type is **string** (or whatever other form of code representation is used). The last expression in the body defines the code representation produced by the template.

The example in Figure 3.1 shows the definition of a template that takes as parameters a structure type[2] and the name of one of its fields, and generates the representation of a procedure to write out the value of that field for given instances of the structure type. This could not be implemented without reflection in any language that requires structure field names to be known statically.

The program binds the identifier *mkWriteField* to a template with one type parameter, *T*, and one value parameter, *field*, of type *string*. Inside the body of the template the identifier *repn* is bound to a representation of the type parameter, obtained by writing the reserved word **repof** before the type identifier. The type representation obtained is of type **typerep**, a base type in the language. The programmer cannot examine the structure of a type representation directly; instead, a number of standard procedures which operate over type representations are brought into scope automatically at the beginning of each template definition. In the fifth line of the program the standard procedure *constructorName* is used to discover what kind of type is represented by the type parameter. If the parameter is not a structure type an error is reported, and the string produced by the template will not represent valid code.

The resulting code representation lies at the end of the template body. The first line of it contains a type definition for the type which was passed to the template, binding it to the name *T1*. The rest contains the definition of a procedure which takes a value of type *T1* as its parameter. Embedded within the code representation are two string expressions enclosed by the markers **code<** and **>**. When the template is evaluated these expressions are themselves evaluated and the resulting strings concatenated with the surrounding code. The first expression uses the standard procedure *typerepDefn* to obtain a string representation of the **typerep** while the second inserts the value of the parameter *field* into the code. The **code** markers thus provide syntactic sugar for string concatenation. Note that TemplateNapier, like Napier88, uses brackets to denote structure dereference e.g. a(b) to denote the field *b* of structure *a*.

```
let mkWriteField = template[ T ]( field : string )
begin
        let repn = repof T                                    ! Get representation of type.
        if constructorName( repn ) ~= "structure" do
                write( "not a structure type" )

        ! The source code produced.
        "type T1 is code< typerepDefn( repn ) >
        proc( instance : T1 )
                write( instance( code< field > ) )"
end
```

Figure 3.1: A template definition

3.2 Evaluating templates

Reflection is achieved in TemplateNapier by use of the **evaluate** clause; this is the only reflective construct in the language. Execution of an **evaluate** clause causes the execution of a template body to produce some source code, compilation of the source code to give a language value (or failure), a type check on the value, and if successful the binding of the value into the

[2]Structure type in Napier88 and TemplateNapier is synonymous with record type.

program.

An evaluation of the template defined in Figure 3.1 is shown in Figure 3.2. The program begins with a definition of the structure type *Person* which will be used as the type parameter. An instance of *Person*, *fred*, is created and the procedure *readString* is used to prompt the user for the name of the field to be written out, bound to the string identifier *desiredField*. The template is then evaluated. Note that this takes place at run-time. The template is passed the appropriate number of type and value parameters, in this case *Person* and *desiredField* respectively, and the name *writeTheField* is given to denote the result of the evaluation. The type *Person* is passed directly rather than the programmer having to obtain a value which represents it, a problem which was glossed over in Section 2.1.2.

Following the reserved word **to** is a list of type expressions, each followed by a program clause, and the word **default** also followed by a program clause. In the example there is only one type expression in the list, *proc(Person)*. The name for the result of the evaluation, *writeTheField*, is in scope in the program clauses up until **default**. The body of the template is executed to produce some source code representation which is then compiled and *writeTheField* initialised with the result. The type of the result is compared with the types in the list and the clause following the first type to match is executed. If no types match the result the code following **default** is executed. Thus a dynamic type check on the result of the reflection takes place, ensuring strong typing.

```
! The type the result procedure will work for.
type Person is structure( name,address : string )

let fred = Person( "alfred","32 south street" )     ! An instance of Person.
write( "Which field?" )                             ! Prompt user for a field name.
let desiredField = readString()                     ! Get the field name as a string.

evaluate mkWriteField[ Person ]( desiredField ) as
        writeTheField to

proc( Person ) :     writeTheField( fred )                      ! Result had expected type.
default :            write( "invalid field name entered" )      ! Compilation failed.
```

Figure 3.2: Evaluating a template

If this program is executed and the user types in “address” the template is evaluated to produce the code representation shown in Figure 3.3. If the user supplies any string other than “name” or “address” the code fails to compile and the error following **default** is reported.

```
"type T1 is structure( name,address : string )
proc( instance : T1 )
        write( instance( address ) )"
```

Figure 3.3: Code produced by evaluation of template *mkWriteField*

The code representation produced by the evaluation of the template in this example does not itself contain any template definitions or evaluations. However, templates are first class values in the language and may in general produce code that contains other templates, that are themselves evaluated when that code is executed.

In the current version of TemplateNapier the evaluation of templates always takes place at run-time. However it might be possible to implement a version in which evaluation could occur at compile-time in cases where the template parameters and all the identifiers referred to in the template body were manifest. This could allow compile-time and run-time reflection to be integrated using a single mechanism—this is the subject of current research.

3.3 Bindings within templates

It has been shown how the values produced by templates can be bound into running programs using the **evaluate** construct. It is also possible to bind values from a running program into a template, using the **eval** construct illustrated in Figure 3.4. If the template *t* is evaluated then when the execution of the template body reaches the point ① where the code representation is defined, the current R-value of the identifier *anInt* is bound into the code representation. Thus when the resulting procedure is executed the identifier *intValue* is initialised with the value *3* which is then written out.

```
let anInt = 3

let t = template[ T ]()
begin
        ...

①       "proc() ; begin
                        let intValue = eval< anInt >
                        writeInt( intValue )
                  end"
end
```

Figure 3.4: Use of the *eval* construct

The **eval** mechanism can be used to bind to the value of any identifier or expression in the static scope at the point of the definition of a template result. Note the difference between this and the **code** mechanism which enables pieces of code representation to be joined together.

3.4 Templates: miscellaneous

3.4.1 Viewing code generated by a template

For debugging purposes it may be useful to examine the code representation constructed in the evaluation of a template. This can be achieved using the **textof** construct shown in Figure 3.5. Executing this program results in the identifier *procedureText* being bound to the code string shown in Figure 3.3.

```
...     ! Code as in Figure 3.2.
let procedureText = textof mkWriteField[ Person ]( desiredField )
```

Figure 3.5: Obtaining the code produced by an evaluation

3.4.2 Equality of typereps

Equality over type representations, instances of the base type **typerep**, is defined as structural equivalence[3]. This allows the programmer to test a type parameter for equivalence to some other type, even though the definition of that type may lie in another program. For example a

[3]There are some exceptions to this; they are the same exceptions that are specified in the Napier88 type matching rules.

template definition might begin as follows:

```
        let x = template[ A ]()
        begin
                if repof A = repof structure( name : string )
                then …
                …
```

Here the type *A* is tested for structural equivalence with the given anonymous structure type.

3.4.3 Escapes in string literals

The strings "code<", "eval<" and ">" are used as markers to indicate special regions of string literals. They can be included without being interpreted as markers by preceding them with apostrophes. For example the string "a '> b" written in a template denotes the string "a > b".

3.4.4 Template types

In order to pass a template as a procedure parameter or bind to it in the persistent store it is necessary to be able to write its type. For example the types of the templates *mkWriteField* and *t* defined in Figures 3.1 and 3.4 respectively are written:

```
        template[ T ]( string )

and     template[ S ]()
```

Any names could be used here in place of *T* and *S*. Two templates have the same type if they have the same number of type parameters and value parameters and the corresponding value parameters in both templates have the same type. The names of the parameters do not have to be the same.

3.4.5 Standard procedures

The essence of reflection is the analysis of existing programs and the synthesis of new ones: many of the computations involved can be pre-coded and made available to the programmer. In TemplateNapier this is achieved by bringing a set of standard procedures into scope automatically at the beginning of each template. Some of these manipulate type representations; they are implemented at a level below TemplateNapier and cannot be written in the language. The other procedures are written in TemplateNapier and are provided for convenience rather than by necessity. Some manipulate other source code constructs besides type representations; others perform utility functions such as vector manipulation. A description of the full set of standard procedures is given in the Appendix.

3.4.6 Current implementation

The current prototype of TemplateNapier is written in Napier88 and is implemented as a reflective layer above Napier88. It translates TemplateNapier programs into Napier88 programs which are then compiled. This prototype version does not allow templates to produce reflective code.

4 Analysis of Reflective Programming

In this section an attempt is made to identify some factors that make it more difficult to write and understand reflective programs than conventional programs.

4.1 Code categories

A reflective program contains a mixture of several different kinds of code with respect to their role in reflection. The infinite set of valid TemplateNapier expressions can be grouped into a series of categories of decreasing generality. The first contains the entire set of TemplateNapier expressions, the second contains the non-reflective expressions from the first, while the last two sets contain only expressions which themselves represent other code. These categories will be referred to by the following names respectively: the **general** category, the **general non-reflective** category, the **codeRep** category and the **manifest codeRep** category. There is a subset relation between these categories:

manifest codeRep $\subset$ codeRep $\subset$ general non-reflective $\subset$ general

The definitions of the categories are now elaborated:

manifest codeRep

contains any manifest code which has the type used to represent source code, type **string** in the current version of TemplateNapier. Expressions in this category are themselves representations of code and because they are manifest, i.e. fixed at compile-time, they always produce the same code representation when evaluated.

example: "proc(x : int → int) ; x * x"

codeRep

contains all expressions which represent source code. The category includes arbitrary expressions and so the code represented by an expression may vary between evaluations.

example: "let x = y + " ++ readString()

Here the '++' indicates string concatenation, and the standard procedure *readString* returns a string input by the user. The code represented by this expression will vary depending on what string is input.

general non-reflective

This contains any code of any type so long as it does not include template definitions or evaluations. This category includes all legal Napier88 code.

example: **proc**(x : **int** → **int**) ; x * x

This is a procedure literal whereas the example of **manifest codeRep** code was a string representation of that literal.

general

contains all TemplateNapier code of any type. Code may contain reflective constructs.

example: **evaluate** x[**int**]() **as** result **to** …

4.2 Interactions between categories

A TemplateNapier program consists of **general** code. Template definitions within it contain a combination of **codeRep** and **manifest codeRep** code. They may also contain **general** code. When the **codeRep** and **manifest codeRep** code is evaluated, due to the execution of an **evaluate** clause at run-time, it produces the representation of new **general** code, which is then compiled and evaluated. If that new code itself contains reflective constructs its evaluation will involve further compile/evaluate cycles which continue until **general non-reflective** code is produced. The value obtained by evaluating that code is bound to by the running program. This is shown in Figure 4.1:

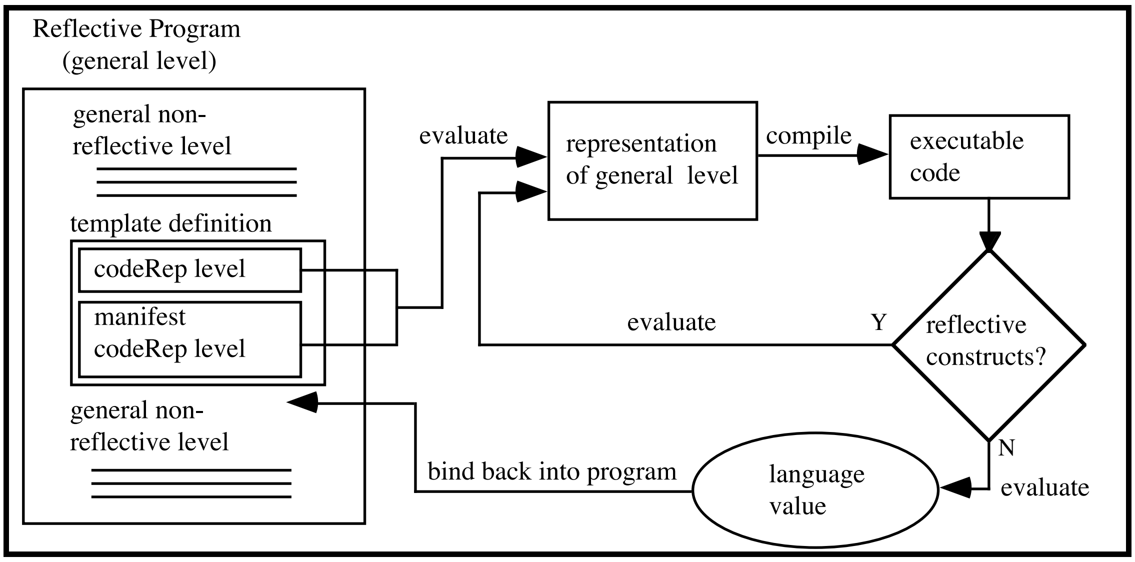


Figure 4.1: Interactions between code categories

In examining even a simple reflective program the user is presented with a mixture of 3 categories of code with different roles in the reflective process. Consider the categories of code in the template definition shown in Figure 4.2. The **manifest codeRep** code is shown in outline style, the **codeRep** code in italic and the **general** code in plain text. In order to reduce confusion the reserved words have not been emboldened.

```
let mkWriteField = template[ T ]( field : string )          ! Plain text is general code.
begin
    let repn = repof T

    "type T1 is code< typerepDefn( repn ) >                  ! Italic text is codeRep code.
    proc( instance : T1 )                                    ! Outline text is manifest codeRep code.
        write( instance( code< field > ) )"
end
…
evaluate mkWriteField[ Person ]( desiredField ) as writeTheField to
…
```

Figure 4.2: Code categories in a template definition

The string literal denoting the result of the template contains **manifest codeRep** code, except for the parts enclosed in **code** brackets, which contain **codeRep** code. The **general** code before the string literal is executed when the template is evaluated. The **codeRep** code is also executed then, to produce the representation of some new **general** code, but the **manifest codeRep** code is not executed at that stage. It is composed with the new code produced by the **codeRep** code.

Consider the evaluation, at run-time, of a template which produces non-reflective code. Two distinct code execution phases occur. The first is that described above, producing the representation of some **general** code. That representation is then compiled, and, if the compilation is successful, executed to give the result value. Thus in the first phase it is the existing **general non-reflective** code and the **codeRep** code which is executed, while in the second phase it is the **manifest codeRep** code and the new **general non-reflective** code. If the template constructs reflective code there is an evaluation cycle which continues until non-reflective code is produced.

The different code categories within a template definition appear different to the programmer. The **general** code and **manifest codeRep** code looks similar, the only difference being that the code in the two categories is executed during different evaluation phases. However, the **codeRep** code looks different because it may encode an infinite number of different segments of **general** code, depending on the environment in which it is evaluated. In trying to understand a template definition, the programmer attempts to visualise the code which it would produce for a particular set of input types and values. To do this it is necessary, mentally, to compute the resulting **general** code and to compose it with the **manifest codeRep** code. Even if the computation itself is not hard, the composition is, because of the discontinuity between the categories. Some parts of the resulting code are derived directly from the **manifest codeRep** code in the program, whereas the other adjoining parts are obtained only as a result of computation. The visualisation task is even more difficult where multiple levels of reflection take place and it is not clear whether that facility will find practical applications. It may be possible to produce tools which aid the programmer in understanding reflective programs: one idea is described in Section 4.5.

4.3 Code categories in other languages

4.3.1 TRPL

TRPL is a statically typed language that supports compile-time reflection [She90]. Figure 4.3 shows the code categories in a fragment of TRPL code which performs a function similar to that of the template *mkWriteField* shown in Figure 3.1:

```
macro WRITEFIELD(field);                        @ Plain text is general code.
EREP(fred.f, f := field);                       @ Italic text is codeRep code.
                                                @ Outline text is manifest codeRep code.
person = struct make_person( name:string,address:string);
variable fred : person := make_person("alfred","32 south street");

PRINT(WRITEFIELD(address));
```

Figure 4.3: Code categories in TRPL

The reflection occurs in the macro *WRITEFIELD*. The body of the macro consists of a call to the pre-defined macro *EREP*, which expands to a graph representation of the code passed to it. That code is *fred.f*, where *f* is substituted by whatever field name has been passed to *WRITEFIELD*. Although a graph form of code representation is used this is disguised by *EREP* which allows the **manifest codeRep** code to be written textually. The optional substitutions written after the main code *fred.f* provide the means for linking in the **codeRep** code. In this case there is one substitution for *f*. Following the macro definition there is the definition of type *person* and the creation of an instance *fred*. Finally the macro *WRITEFIELD* is called to produce code to dereference a field of *fred*, and the pre-defined macro *PRINT* used to write out the result. The expansion of these macros takes place at compile-time; after their expansion the program contains no reflective constructs. The type checking that is necessary for strongly typed linguistic reflection occurs as normal during compilation of the expanded program.

For simplicity this trivial example has been used; it does not demonstrate the full power of TRPL. Here the field name *address* has to be manifest so the field to be written out cannot depend on user input at run-time, unlike the TemplateNapier example. Some more useful examples of reflection in TRPL are given in [SS91].

4.3.2 PS-algol and Napier88

The language PS-algol [PS88] is a predecessor of Napier88. Both languages handle reflection in a similar way and reflective programs written in PS-algol and Napier88 have the same categories as TemplateNapier. Figure 4.4 shows how the field selection example can be written in PS-algol.

The program begins with the definition of the procedure *mkWriteField* which takes as parameters string representations of a structure class and a field name, and produces a string. The structure class representation plays the same role as the structure type representation in the TemplateNapier implementation. The string produced by *mkWriteField* is the representation of another procedure which itself takes a single parameter of type **pntr**, the infinite union of all structure types, and writes out the value of one of its fields.

```
let mkWriteField = proc( string structureClass,field → string )
begin
    let structureDefn = …                        ! Construct string representation of
                                                 ! structure definition using structureClass.

    "structure T1 " ++ structureDefn ++          ! Italic text is codeRep code.
    "proc( pntr instance )                       ! Outline text is manifest codeRep code.
       write( instance( " ++ field ++ " ) )"
end

structure Person( string name,address )
let fred = Person( "alfred","32 south street" )
write( "Which field?" )

let desiredField = readString()                  ! Get the field name as a string.

let procText = mkWriteField( class.identifier( fred ),desiredField )
structure procHolder( proc( pntr ) writeProc )
let dummyProcHolder = procHolder( proc( pntr a ) ; {} )
let resultHolder = compile( procText,dummyProcHolder )
let writeTheField = resultHolder( writeProc )    ! writeTheField is of type proc( pntr ).

writeTheField( fred )
```

Figure 4.4: Code categories in PS-algol

After the definition of *mkWriteField* the structure class *Person* is defined, an instance created, and a field name obtained from the user. The standard procedure *class.identifier* is used to obtain a representation of the structure class *Person* which is then passed to *mkWriteField* to produce a procedure representation. The interface to the compiler is more complex than in Napier88: as well as the source representation it takes a pointer to a structure of the appropriate type to contain the compilation result. The compiler checks the type of the result against it and then returns another instance of that structure type containing the result. Finally the structure is dereferenced to give the procedure *writeTheField* which is called with the parameter *fred.*

The main point of this example is to show how the categories are differentiated. The **manifest codeRep** code is that enclosed by quotes, and the **codeRep** code is composed with it using string concatenation. There is no convention dictating where the **codeRep** and **manifest**

codeRep code should occur within the program, and the user may not be able to tell from a superficial examination of the program which string expressions are involved in reflection and which are used in other ways.

4.3.3 PS-algol with place-holders

Cooper has used reflection in PS-algol for, among other things, constructing data models [CAD87,Coo90]. To improve the readability of reflective programs he created a variant of the language in which place holders can be used to indicate the variable parts of a generator, the **codeRep** code. Figure 4.5 shows the example from Figure 4.4 written in this way. The procedure *mkWriteField* now returns a pointer to a structure containing the compiled procedure, rather than the source text. When the 'procedure' *replace* is called the place holders, written in capital letters preceded by #, are replaced by the specified text.

```
structure procHolder( proc( pntr ) writeProc )
let mkWriteField = proc( string structureClass,field → pntr )
begin
      let structureDefn = ...                 ! Construct string representation of
                                              ! structure definition using structureClass.
      let program =
      begin
            structure T1 #STRUCTUREDEFN
            proc( pntr instance )
                  write( instance( #FIELD ) )
      end

      replace( program,#STRUCTUREDEFN,structureDefn )
      replace( program,#FIELD,field )

      let dummyProcHolder = procHolder( proc( pntr a ) ; {} )
      compile( program,dummyProcHolder )
end

...   ! Definitions of Person, fred and desiredField.

let resultHolder = mkWriteField( class.identifier( fred ),desiredField )
let writeTheField = resultHolder( writeProc )     ! writeTheField is of type proc( pntr ).
writeTheField( fred )
```

Figure 4.5: Code categories in PS-algol with place holders

The advantage of this approach is that the generator is easy to read: the **manifest codeRep** code within the definition of *program* looks just like normal **general** code. It is also easy to pick out the **codeRep** code, the parts of the generic definition which vary between instantiations, as each section is indicated by a place holder.

The cost is in complicating the type system. In the example *program* and the place holders do not have well-defined types, *replace* is a macro rather than a procedure and *program* is defined as though it were a procedure but treated as a string. This may be confusing to the programmer. In the design of TemplateNapier the intention was to use a similar method to improve readability while conforming to the spirit of the type system of the parent language.

4.4 Decisions in designing a reflective language

This section identifies desirable features in a reflective language and some of the choices to be made in designing such a language.

4.4.1 What are the goals?

The language should be powerful and understandable. This suggests the following:

- The type system should be coherent and simple.
- Code in different categories should be linked together without undue syntactic noise. At the same time it should be easy to identify the role of a given section of code in the reflective process, and to distinguish which code is fixed and which is produced as the result of computation.
- There should be mechanisms allowing programs to bind to values created by reflection, and allowing those values to contain bindings to values in the programs which created them. Ideally a simple mechanism will support a wide spectrum of binding times.

4.4.2 What are the choices to be made?

- In what form is code represented in the language? Some possibilities are the string, as used in PS-algol, Napier88 and TemplateNapier, the abstract syntax graph as used in TRPL, or a combination of both.
- When can reflection take place? The possibilities are at compile-time, at run-time, or a combination of both. It is not known of any languages which support both facilities; this is a current research topic.
- How is the code produced by generators distinguished from normal code and the different categories of generated code distinguished from one another?
- Does the definition of the code produced by generators appear at predictable points within a program? In the languages described this is under the control of the programmer who decides whether the definitions occur at the end of the generators (templates or procedures), which is probably the most understandable, or are distributed throughout the program.
- What pre-defined abstractions are available to the programmer for manipulating code representations? PS-algol and Napier88 have none built in, while the sets of abstractions provided by TRPL and TemplateNapier are broadly similar.
- Is meta-reflection possible? This occurs when the code produced by a generator itself contains reflective constructs. It is possible in all the languages that have been discussed but whether it is of practical use is not clear.

4.4.3 What factors affect understanding?

Practical experience in writing reflective programs has shown that they are considerably more difficult to write and understand than conventional ones. Some reasons for this are:

- A reflective program describes a potentially infinite class of programs rather than a single one. To understand it the programmer first must analyse the computation which constructs the resulting program and then abstract out the essential features of all the possible results. It is the existence of **manifest codeRep** code which makes this possible at all. It provides a constant framework around which the variable parts of the target computation are distributed.
- The **codeRep** and **manifest codeRep** categories appear different even though they both represent parts of the target computation. Once evaluated their results are integrated seamlessly but this is not apparent from the source program.
- Different code categories are evaluated at different times in the evaluation of a generator. In

the situation without meta-reflection there are two stages: during the first the **general** and **codeRep** code is evaluated to produce **general non-reflective** code, and during the second that code is evaluated together with the original **manifest codeRep** code to give a final result. The user has to remember that adjacent parts of the reflective program may not be evaluated together. They may also be evaluated in different environments; the **code** and **eval** mechanisms in TemplateNapier exist to allow values from the environment in which the source code is evaluated to be bound into the result code.

- The programmer must understand several mappings between code categories. These are:

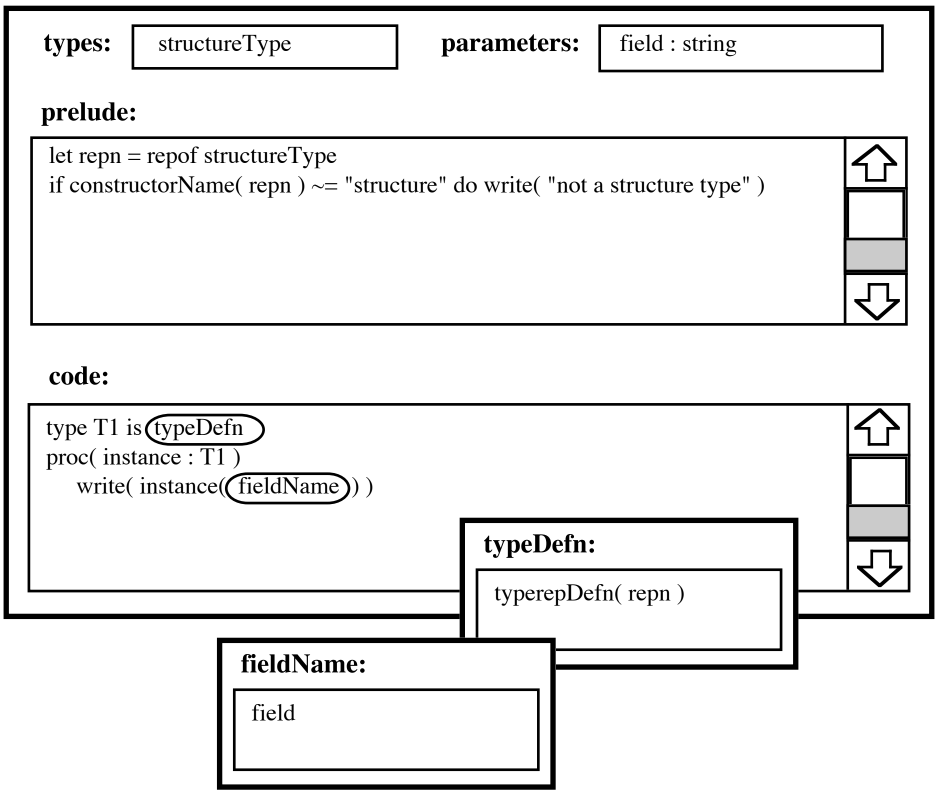
  - between **general** code and its representation as used in **manifest codeRep**;
  - between **general** code and its representation used in code manipulation functions;
  - and between **codeRep** code and the **general** code to which it evaluates.

  The first two are normally the same but they could be different. It might be more convenient to view the code as text in **manifest codeRep** and as abstract syntax for manipulating it.

- It is hard to read abstract syntax in textual form. This was used for the **manifest codeRep** code in early versions of TRPL, but the current version provides automatic conversion from a normal textual representation to abstract syntax.

4.5 Graphical interfaces

One way to reduce the syntactic noise in a reflective program might be through the provision of graphical tools. Figure 4.6 shows how a tool for editing template definitions in TemplateNapier might look (it has not yet been implemented). A group of windows displays the definition of the template from Figure 4.2. The main window contains edit-able sub-windows labelled *types*, *parameters* and *prelude* which show respectively the names of the type parameters, the names and types of the value parameters, and the main template code. The *code* sub-window shows the definition of the result code: the **manifest codeRep** code with a number of embedded buttons, one for each section of **codeRep** code. When one of these is pressed, another window containing the corresponding code appears.


types: structureType

parameters: field : string

prelude:

```
let repn = repof structureType
if constructorName( repn ) ~= "structure" do write( "not a structure type" )
```

code:

```
type T1 is typeDefn
proc( instance : T1 )
    write( instance( fieldName ) )
```

typeDefn:

```
typerepDefn( repn )
```

fieldName:

```
field
```

Figure 4.6: Window interface for a template

The advantages of using this interface to edit template definitions are:

- The generic form of the template is emphasised through the display of the **manifest codeRep** code in the main window. This shows the parts which are common to all instantiations of the template.

- Because that code is displayed in a separate window from the **general** code, it does not need to be enclosed in quotation marks. This makes it look more like real code and less like the representation of code.

- Using separate windows for the **codeRep** code removes the necessity for awkward markers embedded within the **manifest codeRep** code. The buttons within the main window make it obvious where the evaluation dependent code lies.

5 Conclusions

This paper has described how the technique of strongly typed linguistic reflection can be useful in implementing efficient generic code. This is relevant to persistent systems in that the technique can be used to build generic tools for manipulating strongly typed data. The facility of orthogonal persistence also allows the use of cacheing to further improve the efficiency of the technique.

Some short-comings in several existing reflective languages have been identified, and a new persistent reflective language based on Napier88 introduced. A prototype implementation of the language, TemplateNapier, has been completed. The design aim is to provide reflective constructs which are integrated harmoniously with the other language features inherited from Napier88.

The paper concludes with an attempt to analyse the reasons why reflective programs are hard to understand, and identifies a number of choices facing the designer of a reflective language.

6 Acknowledgements

I thank Ron Morrison, Richard Connor and Quintin Cutts at St Andrews, and Tim Sheard, Dave Stemple and Leo Fegaras at Amherst, for helpful comments, advice and discussions.

This work was supported by an SERC PhD studentship and by SERC Grant GR/F 02953.

7 References

[CAD87] R. L. Cooper, M. P. Atkinson, A. Dearle and D. Abderrahmane, "Constructing Database Systems in a Persistent Environment", Proc. 13th Int. Conf. on Very Large Data Bases pp 117-125 (1987).

[Coo90] R. L. Cooper, "On The Utilisation of Persistent Programming Environments", PhD Thesis, University of Glasgow Research Report CSC 90/R12 (1990).

[CW85] L. Cardelli and P. Wegner, "On Understanding Types, Data Abstraction, and Polymorphism", ACM Computing Surveys Vol 17 No 4 pp 471-523 (1985).

[DB88] A. Dearle and A. L. Brown, "Safe Browsing in a Strongly Typed Persistent Environment", Computer Journal Vol 31 No 6 pp 540-544 (1988).

[DCK89] A. Dearle, Q. I. Cutts and G. N. C. Kirby, "Browsing, Grazing and Nibbling Persistent Data Structures", In "Persistent Object Systems", J. Rosenberg and D. Koch (eds), Springer-Verlag pp 56-69 (1989).

[KD90] G. N. C. Kirby and A. Dearle, "An Adaptive Graphical Browser for Napier88", University of St Andrews Research Report CS/90/16 (1990).

[MAE62] J. McCarthy, P. W. Abrahams, D. J. Edwards, T. P. Hart and M. I. Levin, "The Lisp Programmers' Manual", M.I.T. Press, Cambridge, Massachusetts (1962).

[MBC89] R. Morrison, A. L. Brown, R. C. H. Connor and A. Dearle, "The Napier88 Reference Manual", Universities of Glasgow and St Andrews PPRR-77-89 (1989).

[PS88] "PS-algol Reference Manual, 4th edition", Universities of Glasgow and St Andrews PPRR-12-88 (1988).

[SFS90] D. Stemple, L. Fegaras, T. Sheard and A. Socorro, "Exceeding the Limits of Polymorphism in Database Programming Languages", Lecture Notes in Computer Science Vol 416, Springer-Verlag, pp. 269-285 (1990).

[She90] T. Sheard, "A user's Guide to TRPL: A Compile-time Reflective Programming Language", University of Massachusetts COINS Technical Report 90-109 (1990).

[SS91] T. Sheard and D. Stemple, "Examples in TRPL", University of Massachusetts COINS Technical Report (1991).

[SSS92] D. Stemple, R. B. Stanton, T. Sheard, P. Philbrow, R. Morrison, G. N. C. Kirby, L. Fegaras, R. L. Cooper, R. C. H. Connor, M. P. Atkinson and S. Alagic, "Type-Safe Linguistic Reflection: A Generator Technology", in preparation.

[Str67] C. Strachey, “Fundamental concepts in programming languages”, Oxford University Press, Oxford (1967).

Appendix: Standard Code Manipulation Procedures

This Appendix shows the types of the standard code manipulation procedures which are automatically brought into scope at the beginning of each template definition.

!*** Determines whether the type is a base type.
let baseType = **proc**(repn : **typerep** → **bool**)

!*** Returns type name if a base type, null string otherwise.
let baseName = **proc**(repn : **typerep** → **string**)

!*** Returns the text of a definition of the type.
let typerepDefn = **proc**(repn : **typerep** ; name : **string** → **string**)

!*** Returns constructor name if a constructed type, null string otherwise.
let constructorName = **proc**(repn : **typerep** → **string**)

!*** Returns the field names of a structure type.
let structureFieldNames = **proc**(repn : **typerep** → ***string**)

!*** Returns the field types of a structure type.
let structureFieldTypes = **proc**(repn : **typerep** → ***typerep**)

!*** Returns the constancies of the fields of the given structure.
let constancyMap = **proc**(structureInst : **any** → ***bool**)

!*** Returns the branch names of a variant type.
let variantBranchNames = **proc**(repn : **typerep** → ***string**)

!*** Returns the branch types of a variant type.
let variantBranchTypes = **proc**(repn : **typerep** → ***typerep**)

!*** Returns the type of the value injected into an any.
let injectedType = **proc**(a : **any** → **typerep**)

!*** Returns element type for a vector type, niltyperep otherwise.
let vectorElementType = **proc**(repn : **typerep** → **typerep**)

!*** Returns argument types for a procedure type, vector containing niltyperep otherwise.
let procArgumentTypes = **proc**(repn : **typerep** → ***typerep**)

!*** Returns result type for a procedure type, niltyperep for void or non-proc type.
let procResultType = **proc**(repn : **typerep** → **typerep**)

!*** Makes the representation of a structure type from vectors of names and types.
let mkStructType = **proc**(fieldNames : ***string** ; fieldTypes : ***typerep** → **typerep**)

!*** Makes the representation of a variant type from vectors of names and types.
let mkVarType = **proc**(branchNames : ***string** ; branchTypes : ***typerep** → **typerep**)

```
!*** Makes the representation of a procedure type from the argument and result types.
let mkProcType = proc( args : *typerep ; result : typerep → typerep )

!*** Makes the representation of a vector type from the element type.
let mkVectorType = proc( element : typerep → typerep )


!*** Concatenates the elements of the vector separated by the given string.
let iterate = proc( separator : string ; elements : *string → string )

!*** Concatenates the elements of the vector, after transformation, separated by given string.
let iterate2 = proc( separator : string ; elements : *string ;
                     transform : proc( string → string ) → string )

!*** Constructs new vector from all the first elements, another from all second elements, etc.
!*** Concatenates each new vector and concatenates the results, separated by the given string.
let iterate3 = proc( separator : string ; elements : **string ;
                     concat : proc( *string → string ) → string )

!*** Returns all the elements of the first vector which are also present in the second.
let intersect = proc[ T ]( first,second : *T → *T )

!*** Returns all the elements of the first vector which aren't present in the second.
let diff = proc[ T ]( first,second : *T → *T )

!*** Returns index at which target occurs in vector, or zero if not found.
let position = proc[ T ]( target : T ; vec : *T → int )

!*** Appends one vector to another.
let concat = proc[ T ]( first,second : *T → *T )
```