Type-Safe Linguistic Reflection: A Generator Technology

Stemple, D.[1], Stanton, R.B.[2], Sheard, T.[3], Philbrow, P.[4], Morrison, R.[5], Kirby, G.N.C.[5], Fegaras, L.[1], Cooper, R.L.[4], Connor, R.C.H.[5], Atkinson, M.P.[4] & Alagic, S.[6]

[1]Department of Computer and Information Science, University of Massachusetts, Amherst, MA 01038, USA

[2]Department of Computer Science, Australian National University, Acton, ACT 2601, Australia

[3]Department of Computer Science and Engineering, Oregon Graduate Institute, Beaverton, OR 97006, USA

[4]Department of Computing Science, University of Glasgow, Lilybank Gardens, Glasgow G12 8QQ, Scotland

[5]Department of Mathematical and Computational Sciences, University of St Andrews, North Haugh, St Andrews KY16 9SS, Scotland

[6]Department of Computer Science and Electrical Engineering, University of Vermont, Burlington, VT 05405, USA

Abstract

Reflective systems allow their own structures to be altered from within. In a programming system reflection can occur in two ways: by a program altering its own interpretation or by it changing itself. Reflection has been used to facilitate the production and evolution of data and programs in database and programming language systems. This paper is concerned with a particular style of reflection, called linguistic reflection, used in compiled, strongly typed languages. Two major techniques for this have evolved: compile-time reflection and run-time reflection. These techniques are described together with a definition and anatomy of reflective systems using them. Two illustrative examples are given and the uses of type-safe reflective techniques in a database programming language context are surveyed. These include attaining high levels of genericity, accommodating changes in systems, implementing data models, optimising implementations and validating specifications.

1 Introduction

Reflective systems allow their own structures to be altered from within. In a programming system reflection can occur in two ways: by a program altering its own interpretation or by it changing itself. The first of these, which is common in object-oriented systems, we call behavioural reflection and the second we call linguistic reflection. Linguistic reflection allows a program to generate code that is integrated into the program's own execution. Given that a language is Turing complete, adding linguistic reflection to the language does not add any computational power but can cut out a level of interpretation and provide more succinct notation. Such reflection can be used to facilitate both the production and evolution of programs. It is of special interest in database programming languages since in these languages supporting evolution is a major requirement. Linguistic reflection provides a mechanism that may encompass changes to the data, the programs that manipulate the data, and the schema. An example of the use of linguistic reflection is the automatic generation of user interface forms. In current systems it has also been used to attain high levels of genericity [SFS+90], accommodate changes in systems [DB88, DCK89], implement data models [Coo90a, Coo90b, CQ92], optimise implementations [CAD+87, FS91] and validate specifications [FSS92, SSF92].

A number of independent mechanisms for reflection have evolved in programming systems, the first of which appeared in Lisp [MAE62]. Here the concern is with the integration of strong typing, compilation systems and reflective expression. Two techniques for type-safe linguistic reflection have evolved: compile-time linguistic reflection and run-time linguistic reflection. The importance of type-safe linguistic reflection is that it provides a convenient, efficient and uniform mechanism for the production and evolution of database systems and programs.

The introduction of a statically checked type system into a programming language reduces run-time errors by restricting the class of programs that can be written. Where such a type system has been found to be over-restrictive language designers have introduced constructs to alleviate the restriction for some well-identified classes of problem. A good example of this is polymorphism [Mil78] where an infinite class of computations is re-introduced using a single mechanism. Like polymorphism, type-safe linguistic reflection extends the class of algorithms that can be written in a type-safe manner. However, this extension can be made without extending the type system itself.

Linguistic reflection involves defining representations of the syntactic structures of a language within the same language. Compile-time linguistic reflection allows the programmer to define generators which produce representations of program fragments. The generators are executed as part of the compilation process. After the generators execute, their results are then viewed as program fragments and become part of the program being compiled.

In run-time linguistic reflection the mechanism is concerned with the construction and binding of new components with old in an environment. The technique involves firstly the use of a compiler that can be called dynamically to compile newly generated program fragments, and secondly a linking mechanism to bind these new program fragments into the running program.

In order to maintain type-safety each generated program must be type checked in both compile-time and run-time linguistic reflection. Type checking the generators, as opposed to the generated programs, for type correctness of all their generated programs is a second order type checking problem and is undecidable in general.

This paper is concerned with a particular style of reflection which is constrained to be type-safe and is used in compiled, strongly typed languages. Section 2 contains the definition and the anatomy of linguistic reflection. Section 3 examines the dimensions of linguistic reflection. Section 4 describes two examples of the use of type-safe linguistic reflection in the context of the anatomy described. The examples are abstraction over types and the accommodation of evolution in strongly typed persistent object stores. The section also surveys some of the existing uses of linguistic reflection in database programming languages.

2 Definition and Anatomy of Linguistic Reflection

2.1 Linguistic Reflection

Linguistic reflection is defined as the ability of a program to generate new program fragments and to integrate these into its own execution. Given a language, *L*, and a domain of values, *Val*, the nature of execution of a program in *L* will be discussed. The function *eval* is the evaluation function:

$$\boxed{\text{eval} : L \to \text{Val}}$$

The domain of values, *Val*, differs for different languages. Examples of *Val* include numbers, character strings, final machine states, the state of a persistent object store, and the set of bindings of variables produced by the end of a program's execution.

For linguistic reflection to occur, there must be a subset of *Val*, called Val_L, that can be mapped into *L*. For example, Val_L could be the set of character strings containing syntactically correct *L* expressions. Since Val_L is a subset of *Val* that may be translated into the language *L* it may be thought of as a representation of *L*.

A subset of *L* consisting of those language constructs that cause reflective computation is denoted by L_R. L_R is called the reflective sub-language and Val_{L_R} stands for its representation. An evaluation of an expression in L_R invokes a generator. In linguistic reflection the generators, the programs that produce other programs, are written in a subset of the language *L* which will be denoted by L_{Gen}. L_{Gen} may include all of *L* but the programs written in L_{Gen} must produce results in Val_L. The major relationships among the language and value sets are:

$$\boxed{\begin{array}{l} L_R \subset L \\ Val_{L_R} \subset Val_L \subseteq Val \\ L_{Gen} \subseteq L \end{array}}$$

The significance of the proper subset relationships is explained below. Two functions are required for a full description of linguistic reflection. The first, *drop*, takes a construct in L_R and transforms it into a generator in L_{Gen}. The second, *raise*, takes a value in Val_L and produces an expression in *L*:

$$\boxed{\begin{array}{l} \text{drop} : L_R \to L_{Gen} \\ \text{raise} : Val_L \to L \end{array}}$$

Linguistic reflection is defined as the occurrence of the following pattern of computation, within the *eval* function, in the evaluation of a program in *L*:

```
procedure eval (e : L) -> Val ! This types e as L and
                                   ! eval as L -> Val
        case e of
        ...
        ConstructOfL_R => eval (raise (eval (drop (e))))
        ...
```

Figure 2.1 The Linguistic Reflective Nature of *eval*

where the ellipses cover all the non-reflective evaluations. The construct *eval (raise (eval (drop (e))))* represents the intuition that during the evaluation of a reflective expression the result of the evaluation is itself evaluated as an expression in the language.

The expression produced by *drop* is a generator that is evaluated by the inner *eval*. The type of a generator *g* in L_{Gen} is:

```
g : Val -> Val_L
```

The result of the generator is an expression in Val_L which is then translated into *L* by *raise*. The result is finally evaluated by the outer *eval*. This is illustrated in Figure 2.2. The whole diagram represents the *eval* function as does the box containing *eval* within the diagram. This structure is a consequence of representing the recursive function *eval* by a flow diagram and will be a feature of other diagrams.

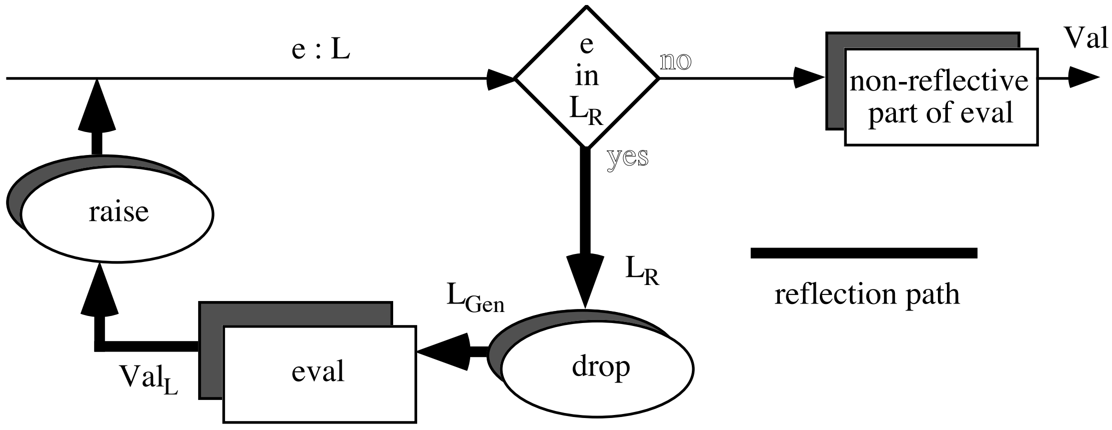


Figure 2.2 *eval* in Linguistic Reflection

In order to make these concepts more concrete an example language is introduced. In this language:

- Val_L is the set of character strings that represent sentences in *L*;
- L_R contains the single verb **execute** which initiates reflection;
- *drop* presents a string expression to the inner evaluation function;
- *raise* maps a string representation to the corresponding sentence in *L*;
- L_{Gen} is the set of expressions in *L* that result in character strings in Val_L.

In this example language, *drop* and *raise* are defined by:

```
drop (execute (stringExpression)) = stringExpression

raise ("expression") = expression
```

Figure 2.3 gives an example of linguistic reflection in this language. The symbol ++ denotes string concatenation. The inner *eval* concatenates the strings to produce the string "*2+3*", while the outer *eval* evaluates the expression ***execute*** *(*"2+" ++ "3"*)* by the following sequence:

```
eval (execute ("2+" ++ "3"))

        ! the reflection is recognised
=> eval (raise (eval (drop (execute ("2+" ++ "3")))))
=> eval (raise (eval ("2+" ++ "3")))
=> eval (raise ("2+3"))
=> eval (2+3)
=> 5
```

Figure 2.3 An Example of Linguistic Reflection

The above example shows that some reflective expressions may be evaluated statically, at compile-time, since here all the information to perform the inner *eval* may be found statically. Uses of this style of reflection are described later. This is not always the case however and some reflective computation may have to be delayed until run-time. Figure 2.4 shows such a computation in which run-time input is solicited by the *readString* procedure. The inner *eval* is thus constrained to execute at run-time.

```
execute ("2+" ++ readString ())
```

Figure 2.4 Example of Run-Time Linguistic Reflection

A reflective computation is well formed if it terminates and the output of each inner *eval* is syntactically correct and typed correctly. Termination requires that the inner *eval* must eventually result in a value in $Val_{L\text{-}L_R}$, the set of values that represent non-reflective program constructs. Syntactic correctness requires that the result of *eval (drop (e))* is in Val_L for all reflective expressions. A generated expression must be internally type consistent as well as typed correctly for its context.

In general, type correctness must be checked for each individual generated expression. Type checking generators for the types of all their possible outputs is a topic for further research but it is undecidable in general.

2.2 Compilation

This paper is concerned with the mechanisms for linguistic reflection in compiled languages and the anatomy given so far must be further refined to describe these. Figure 2.5 shows the structure of *eval* as a composition of two functions: *compile* and *eval'*. The function *compile* takes an expression in language *L* and produces another in a target language *L'*. The function *eval'* is the evaluation function for *L'*. The types of the functions are defined by:

```
compile : L -> L'

eval' : L' -> Val
```

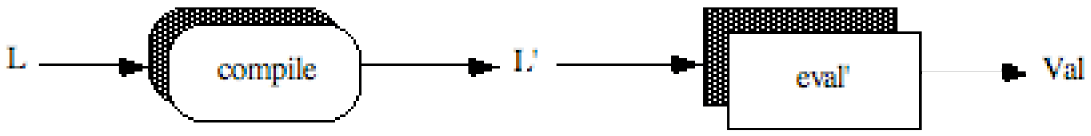


Figure 2.5 *eval* as Function Composition

2.3 Compile-Time Linguistic Reflection

One way in which linguistic reflection can be accomplished in a compilation environment is for reflective constructs to be compiled and executed during the compilation of a program containing them. This is limited to cases where the reflection is over compile-time information and cannot be used for reflection that depends on values that are only available at run-time.

In such a system, generators are used to express computations over the syntactic elements of a program. As in any form of linguistic reflection, the computations are expressed in the subset L_{Gen} of the language L. The reflective sub-language L_R contains the calls to the generators. That is, the pattern of evaluation that defines L_R is only initiated by these reflective calls. A possible *drop* function in this architecture is a function that takes a reflective call, finds its generator definition and uses the definition and the call arguments to form a call to the generator. The inner *eval* executes the call at compile-time to produce a new expression in Val_L. This in turn is transformed to an L expression by *raise* and presented to the outer *eval*. Figure 2.6 illustrates this model. Such a pattern of reflection is called *static* or *compile-time linguistic reflection* since the reflection is performed at compile-time even though the evaluator, *eval'*, is called. The type checking of the generated expressions is performed by the compiler. The pattern of *eval* is given by:

```
procedure eval (e : L) -> Val          ! This types e as L and
        eval' (compile (e))            ! eval as L -> Val.

procedure compile (e : L) -> L'

        case e of
        ConstructOfL_R => compile (raise (eval' (compile (drop (e)))))
        notConstructOfL_R => translate (e)
```

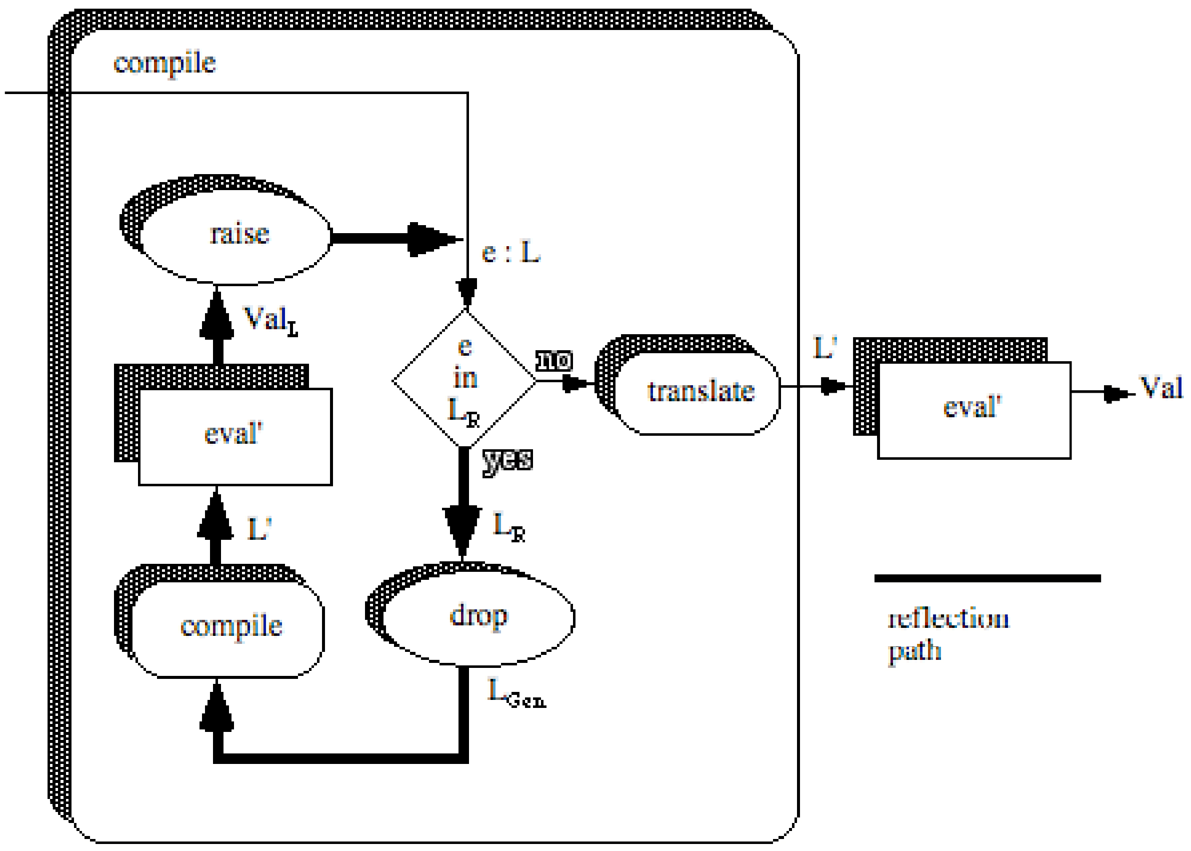


Figure 2.6 *eval* in Compile-Time Linguistic Reflection

The macro facilities in POP-2 [BCP71] and Scheme [RC86] contain this style of compile-time reflection without the type checking.

2.3.1 Optimised Compile-Time Linguistic Reflection

An optimised variant of the previous architecture can be produced by having the parser generate abstract syntax as values in *Val*. This choice of Val_L allows the result of the inner *eval* to be passed directly to the post-parse compiler called *postParseCompile*. The *raise* function reduces to the identity function in this optimisation. The *drop* function here produces a compiled version of the generator in the target language generator subset L_{Gen}. This optimised *drop* function is denoted by $drop_{Opt}$. The structure of *eval* in this case is shown in Figure 2.7. Here e_v denotes the parsed form of *e* expressed in Val_L and L_{R_v} the parsed forms of L_R. The pattern of *eval* is given by:

```
procedure eval (e : L) -> Val              ! This types e as L and
        eval' (compile (e))                ! eval as L -> Val.

procedure compile (e : L) -> L'
        postParseCompile (parse (e))

procedure postParseCompile ($e_v$ : $Val_L$) -> L'
        case $e_v$ of
        $ConstructOfL_{R_v}$ => postParseCompile ($raise_{Opt}$ (eval' ($drop_{Opt}$ ($e_v$))))
        $notConstructOfL_{R_v}$ => translate ($e_v$)
```

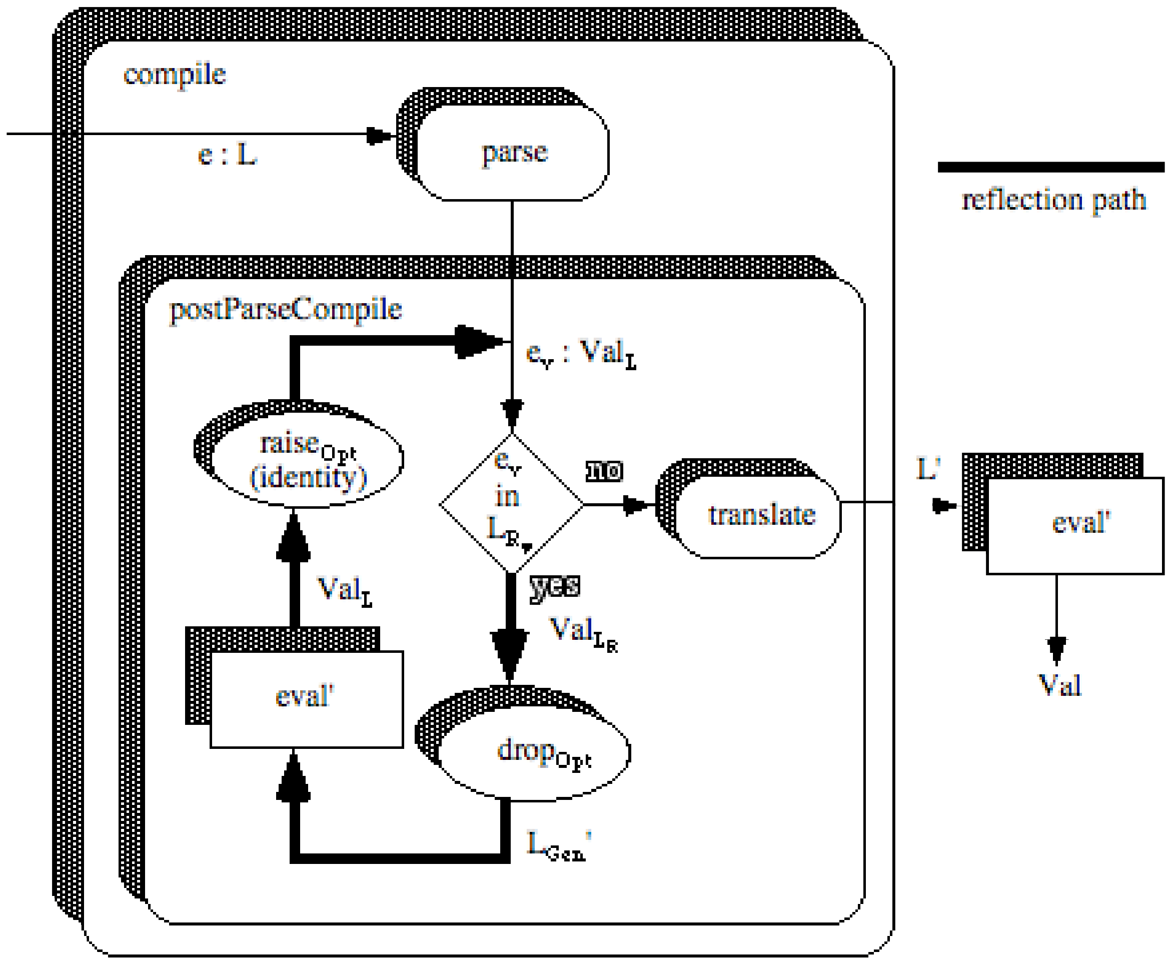


Figure 2.7 *eval* in Optimised Compile-Time Linguistic Reflection

The two versions of the same *eval*' function in both Figure 2.6 and Figure 2.7 highlight some implementation choices. For example, although both *eval*' functions are semantically the same they may be implemented differently. The *eval*' within the compiler could be an interpreter function and the right hand *eval*' could be the machine executing machine code. The details of these implementations are not germane to our description.

An example of such an architecture is the implementation of TRPL [She90]. The TRPL reflective constructs are TRPL context sensitive macro calls, the elements of L_R. The $drop_{Opt}$ function takes the parsed arguments of a macro call and passes them to the macro definitions which have been compiled into target language functions (generators) ready for *eval*'. Thus a call of the compiler is avoided in the reflective *eval*. The result of executing the compiled macro definitions is to produce new TRPL code expressed in the parsed form Val_L. This code can contain new function, type and even macro definitions. This new code is presented to the post-parse compiler for compilation and evaluated using *eval'*. Type checking is performed after each inner *eval'*.

Figure 2.8 gives an example of optimised compile-time reflection as it occurs in TRPL. e_v denotes the Val_L form of *e*, while $\underline{e}$ denotes its compiled form.

eval (**execute** ("2+" ++ "3"))
=> eval' (compile (**execute** ("2+" ++ "3")))
=> eval' (postParseCompile (parse (**execute** ("2+" ++ "3"))))
=> eval' (postParseCompile ((**execute** ("2+" ++ "3"))$_{\mathrm{v}}$))

! the reflection is recognised
=> eval' (postParseCompile (raise$_{\mathrm{Opt}}$ (eval' (drop$_{\mathrm{Opt}}$ ((**execute** ("2+" ++ "3"))$_{\mathrm{v}}$)))))

! $drop_{Opt}$ produces $\underline{\text{"2+" ++ "3"}}_{Gen}$ which denotes the compiled generator of $(2+3)_v$
=> eval' (postParseCompile (raise$_{\mathrm{Opt}}$ (eval' ($\underline{\text{"2+" ++ "3"}}_{\mathrm{Gen}}$))))
=> eval' (postParseCompile (raise$_{\mathrm{Opt}}$ ((2+3)$_{\mathrm{v}}$)))

! *raise* is the identity function
=> eval' (postParseCompile ((2+3)$_{\mathrm{v}}$))
=> eval' ($\underline{2+3}$)
=> 5

Figure 2.8 Optimised Compile-Time Linguistic Reflection in TRPL

The original expression, ***execute*** *("2+" ++ "3")*, is parsed and then examined by the post-parse compiler which recognises that it is a parsed form of a reflective construct. A generator previously compiled into its *L'* form from a definition of ***execute*** is produced by $drop_{Opt}$ using the parsed form of ***execute***'s input. This generator, $\underline{\text{"2+" ++ "3"}}_{Gen}$, evaluates to the Val_L form of *2 + 3*. The inner *eval'* executes the generator and the parsed form $(2+3)_v$ is produced. This is passed to the post-parser compiler, which completes its compilation. It is eventually evaluated in its compiled form by *eval'* as the completion of the original *eval.*

2.4 Run-Time Linguistic Reflection

Where reflection occurs at run-time the expression in L_R, which causes the reflection, has already been compiled. That is, it is the *eval'* function that recognises the expression in L_R', the compiled form of L_R, to initiate reflection. The original expression *e* is in the process of being evaluated by

eval (e)
=> eval' (compile (e))
=> eval' ($\underline{e}$)   ! where $\underline{e}$ is the compiled form of *e*

The pattern of *eval'* in this case is shown by

```
procedure eval' (e : L') -> Val          ! This types e as L' and
                                         ! eval' as L' -> Val.
        case e of
        ...
        ConstructOfL_R' => eval (raise_Run (eval' (drop_Run (e))))
        ...
```

Notice that the outer evaluation function is *eval* whereas the inner one is *eval'*. The outer *eval* encompasses the compiler since it expands to *eval' (compile (...))*. The *drop$_{Run}$* function has the type L_R' -> L_{Gen}'. This is illustrated in Figure 2.9.

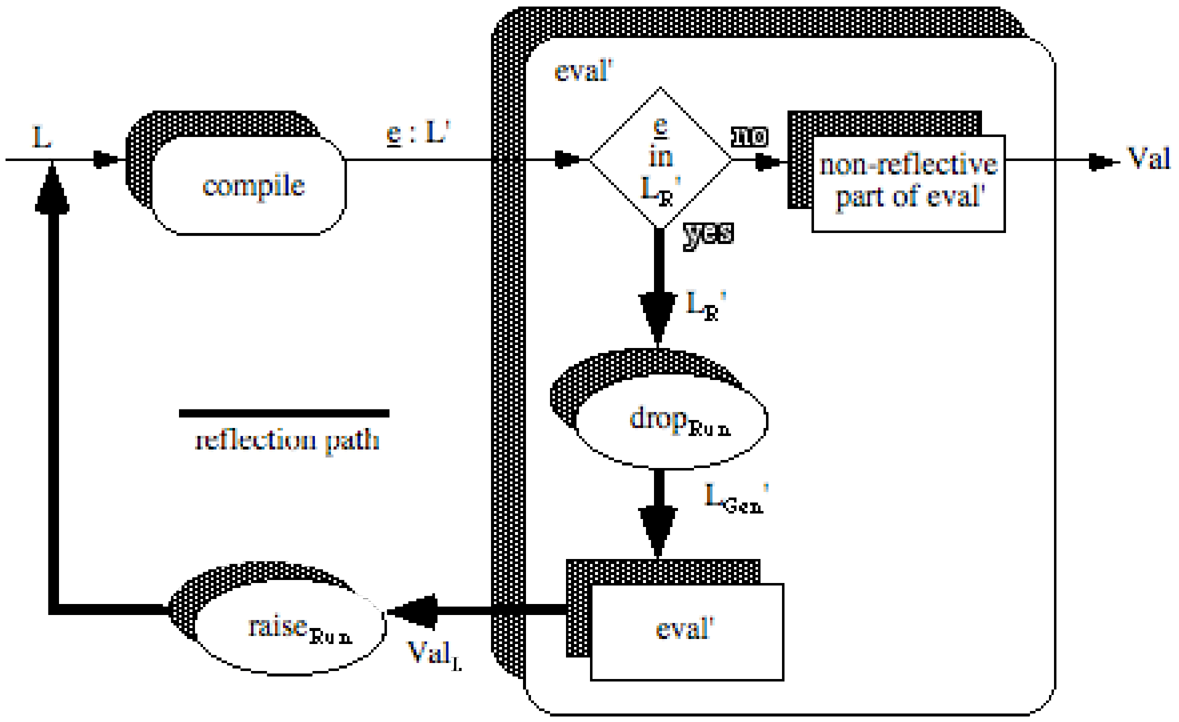


Figure 2.9 *eval* in Run-Time Linguistic Reflection

An example of this form of reflection is the use of a run-time callable compiler together with the ability to bind and execute newly compiled program fragments within the running program. PS-algol [PS88] and Napier88 [MBC+89] with their callable compilers and incremental loaders are examples of languages that provide run-time linguistic reflection. The function *eval* in Lisp and the function *popval* in POP-2 are early examples of untyped run-time reflection.

Figure 2.10 shows the evaluation of ***execute*** *("2+" ++ readString ())* in run-time reflection:

eval (**execute** ("2+" ++ readString ()))
=> eval' (compile (**execute** ("2+" ++ readString ())))
=> eval' (**execute** ("2+" ++ readString ()))

! now the reflection is recognised
=> eval ($\text{raise}_{\text{Run}}$ (eval' (drop_{Run} (**execute** ("2+" ++ readString ())))))
=> eval ($\text{raise}_{\text{Run}}$ (eval' ("2+" ++ readString ())))

! if "3" is input for the call of *readString*
=> eval ($\text{raise}_{\text{Run}}$ ("2+3"))

! applying $\textit{raise}_{\textit{Run}}$ and expanding *eval*
=> eval' (compile (2+3))
=> eval' (2+3)
=> 5

Figure 2.10 Run-Time Linguistic Reflection in Napier88

The original expression is first compiled and is in the process of being evaluated by *eval'* when the reflection is discovered. The compiled form ***execute*** *("2+" ++ readString())* is presented to $\textit{drop}_{\textit{Run}}$ which removes the **execute** verb. The inner *eval'* reads in the string and concatenates it with "*2+*". If the string read in is "*3*" then the result of the concatenation is "*2+3*". This expression is in Val_L and is transformed into *L* by $\textit{raise}_{\textit{Run}}$. Finally the expression *2+3* is compiled and evaluated by *compile* and *eval'*.

3 Dimensions of Linguistic Reflection

So far three subtly different forms of linguistic reflection have been described. The pattern:

eval (raise (eval (drop (e))))

within the *eval* function represents the intuition given in the definition of linguistic reflection. With optimised compile-time linguistic reflection the pattern of evaluation is given by:

eval' (postParseCompile ($\text{raise}_{\text{Opt}}$ (eval' (drop_{Opt} (e_v)))))

with the reflection recognised within the post-parse compiler. The inner evaluation is performed by *eval'* and $\textit{drop}_{\textit{Opt}}$ operates on the parsed form of the expression, e_v.

In run-time linguistic reflection the pattern of evaluation is given by:

eval' (compile ($\text{raise}_{\text{Run}}$ (eval' (drop_{Run} ($\underline{e}$)))))

within the *eval'* function. The *eval'* function operates over the compiled form of the expression $\underline{e}$ and calls *eval* to compile and evaluate the result of the inner *eval'*. Here the reflection is recognised during run-time (target language) evaluation, i.e., in *eval'*, but entails a call of the compiler in performing the outer *eval*.

The inner evaluation characterises linguistic reflection. This we have called generation and as can be seen above the nature of generators can vary. Notice, however, that the generators are always written in the subset L_{Gen} of the language, *L*. It is the nature of *drop* that differs in that the form of the expression presented to it may be in the language itself (*drop*), its parsed form ($drop_{Opt}$) or its compiled form ($drop_{Run}$).

The dimensions of linguistic reflection can be categorised by the following:

- What initiates linguistic reflection?
- How are the generators written?
- When are the generators executed?
- In what environment are the generators executed?
- How are the generated results bound?

For type-safe linguistic reflection there is one other dimension, namely

- When is the type checking performed?

3.1 What Initiates Linguistic Reflection?

Linguistic reflection is initiated by an expression in the reflective sub-language, L_R, being evaluated. The form of L_R may be a simple verb, such as **execute**, or a more sophisticated function or macro call as will be seen in section 4.

3.2 How are the Generators Written?

Since the generators are all written in the language subset L_{Gen}, it is the nature of the language forms that they manipulate that distinguishes different linguistic reflective languages. The generators compute over and produce expressions in Val_L. In some systems this may simply be strings.

Where some processing of the expressions has already taken place, there is a possibility of using more structured forms for Val_L. In optimised compile-time linguistic reflection the generators operate over parsed forms of *L*. Thus Val_L can be the abstract syntax trees constructed by the parser. The generators then have the possibility of computing over these abstract syntax trees forming new ones to construct new program fragments.

3.3 When are the Generators Executed?

In current implementations, type-safe linguistic reflection takes place at fixed points in the evaluation process. To allow the reflective evaluation time to be under the user's control at least two L_R constructs are necessary. Both perform the same function but differ temporally and therefore in the environments in which they operate. They are:

- force which forces the reflective evaluation on the first encounter and replaces the *force* construct with the generated result. It therefore performs the inner *eval* and the *drop*.
- delay which delays reflective evaluation. That is the inner *eval* and *drop* are not performed until the program is executing after the initial compilation phase.

Compile-time linguistic reflection uses *force* implicitly whereas run-time linguistic reflection uses *delay*. Further investigation of the means of bringing the reflective evaluation time under users' control is needed.

3.4 In What Environment are the Generators Executed?

The time of reflective evaluation affects the environment that is available to the generator. There are two environmental issues here. First of all the generators may need access to the details of the compilation such as a symbol table containing type, scoping and identifier definitions. This is trivially available in compile-time linguistic reflection but it is also possible to parameterise the generators, with an environment, and to arrange that the compiler environment is preserved and available at run-time for run-time linguistic reflection.

The second issue is that generators may bind to existing values in a database or persistent store. This may be to R-values, by copy, or L-values, by reference, and may be immediately resolved, by *force*, or delayed until run-time, by *delay*. Means of implementing and exploiting such bindings are under investigation.

3.5 How are the Generated Results Bound?

In compile-time linguistic reflection the result of the generation is bound into the compilation taking place. In run-time linguistic reflection the result of the generation is bound into the executing program. The mechanisms chosen for this in both TRPL and Napier88 are quite simple and only accommodate the binding of generated fragments into the programs containing them. Other aspects involve the binding of free identifiers within both the generated fragment and the original program. The range of binding choices along with their use and implementation are topics for research.

3.6 When is the Type Checking Performed?

In optimised compile-time linguistic reflection the result of the generation is integrated into the program being compiled. The internal type consistency of the new program fragment and its type compatibility with the environment into which it is placed are both checked by the post-parse compiler before execution. In the implementation of TRPL the control of reflection is located in the type checker.

In run-time linguistic reflection the result of the generation is type checked when it is presented to the compiler as part of the outer *eval*. This checks for the fragment's internal consistency. The type compatibility of the fragment with its environment is checked when it is incrementally bound. Thus type checking forms part of the binding phase.

4 Uses of Linguistic Reflection

Here two examples of linguistic reflection are presented in detail. They are examples of:

- abstraction over types, and
- accommodating evolution in strongly typed persistent systems.

Both examples involve reflective access to types in order to achieve the required behaviour. It is somewhat ironic that strong typing, which makes it difficult to integrate reflection with strongly typed programming languages, plays a key role in making linguistic reflection effective in cases exemplified in this section. Linguistic reflection without strong typing, such as in Lisp macro evaluation, has little systematic information available about the structures involved in computation. Types in a strongly typed language constitute systematically required information about all computations. This information can be used in linguistically reflective programming to automatically adjust to specific details such as the names of attributes and to the creation of new types during system evolution.

Following the detailed explanation of the two examples other applications of linguistic reflection are given.

4.1 Abstraction Over Types

4.1.1 The Example

In this section an example of creating an abstraction over types is given using the TRPL style of optimised compile-time linguistic reflection. Abstractions over types are useful when the details of a computation depends significantly on the details of its input types. A generic natural join function provides an example of such abstraction. Here the details of the input types, particularly the names of the tuple components, significantly affect the algorithm and the output type of the function, determining:

- the result type,
- the algorithm to test whether tuples from both input relations match on the overlapping fields, and
- the code to build a relation having tuples with the aggregation of fields from both input relations but with only one copy of the overlapping fields.

The specification of a generic natural join function may be achieved by compile-time linguistic reflection as long as the types of the input relations are known at compile-time. One approach is to generate individual natural join functions for each distinct call. A second approach is to write a generator that produces a call to a parametric polymorphic join function for each unique call to natural join. The reflective ability required for this approach is the same as for the first but the details are simpler. Thus the second approach is chosen to facilitate presentation. A parametric polymorphic join is one in which both the match function and the concatenation function are given as arguments.

The following gives the partial specification of such a join function using TRPL syntax:

```
function (alpha, beta, gamma)
  join(     r1        : set (alpha),
            r2        : set (beta),
            match     : [alpha, beta] -> boolean,
            concat    : [alpha, beta] -> gamma) : set (gamma);
```

This function is polymorphic over types *alpha*, *beta* and *gamma*. It uses the *match* input function to qualify pairs of tuples for inclusion in the result and the *concat* function to construct elements in the result set. With an arbitrary *match* function this function is a theta-join [Cod79]. By supplying the correct *match* and *concat* functions it can be specialised to a natural join. The point to notice is that all the information necessary for constructing *match* and *concat* for a natural join is obtainable from the types *alpha* and *beta*, which in TRPL are inferred. Linguistic reflection allows the generation of the correct *match* and *concat* functions from the representations of these types.

Consider computing the natural join between variables of types *rtype* and *stype*, defined by the equations

```
rtype = set (struct make_a_b_c (a : integer, b : boolean, c : integer));
stype = set (struct make_a_d (a : integer, d : boolean));
```

The TRPL type equations involving the definition of **struct** types define constructor functions for values (tuples) of the types, in this case *make_a_b_c* and *make_a_d*, and selector functions, e.g., *a*, *b*, *c* and *d*. For variables *r* and *s* of types *rtype* and *stype* the following expression for the natural join of *r* and *s* may be written:

```
NATJOIN (r, s)
```

This is a TRPL reflective construct, recognised by virtue of its being fully capitalised, and its inner *eval* generates an appropriate call of the generic join, in its Val_L form. The linguistic reflective process also generates a new type equation to define the type of the join's output and then generates the appropriate *match* and *concat* functions. For example:

```
jointype = set (struct make_a_b_c_d (a : integer, b : boolean, c : integer, d : boolean));

join(   r, s,
        [x, y] -> x.a = y.a,
        [x, y] -> make_a_b_c_d (x.a, x.b, x.c, y.d))
```

Forms such as *[x, y] -> x.a = y.a* represent lambda functions in TRPL. This example stands for the boolean function of two variables that returns *true* if the *a* components are equal.

4.1.2 Details of TRPL Optimised Compile-Time Reflection

In this section the definition of a natural join context sensitive macro in TRPL is presented in considerable detail and related to the general picture of linguistic reflection presented in Section 2. In TRPL, Val_L comprises values of two types, one for representing types, *type_rep*, and one for expressions, *exp_rep*. Figure 4.1 gives the TRPL type definitions for these. Both are defined as unions of choices for the different types and syntactic categories. The type constructors include *struct*, as described above, *list*, *pair*, and *singleton*, for constructing a type consisting of a single value such as the empty list *nil*. Expression categories are the syntactic categories of the language and include identifier, integer constant, selection of a structure component and function call.

```
type_rep =
union (   int_type : singleton int_rep,

          struct_type : struct struct_rep ( constructor_name : string,
                                            struct_components : list (pair (string, type_rep))),

          parametric_type : struct parametric_rep ( parametric_constructor_name : string,
                                                    parameters : list (type_rep)),
          ...)

exp_rep =
union (   identifier :        struct make_identifier (identifier_name : symbol),
          integer_constant : struct make_integer_constant (integer_value : integer),
          selection :         struct make_selection (  structure_value : exp_rep,
                                                       attribute : string),
          function_call :     struct make_function_call (  function_name : string,
                                                           parameter_list : list (exp_rep)),
          ...)
```

Figure 4.1 Types of TRPL Val_L for Representing the Language

The TRPL reflective sub-language consists of calls of context sensitive macros such as *NATJOIN* above. Calls of these macros initiate linguistic reflection. Macros are called context sensitive since they have access to the types defined at the point of their compilation. The generators invoked by the macro calls are defined in macro definitions and are functions from the parsed input to the macro calls (in Val_L) and types contained in the compiler environment (also in Val_L). They generate inline expansions as well as new function and type definitions. The new definitions augment the compiler environment at the time of the generation, i. e., at the time of the inner *eval.*

A TRPL macro definition consists of three parts: the header, the *units* and the inline expansion. The *units* section generates the new function and type definitions. Figure 4.2 shows the outline of a TRPL macro for a natural join function.

```
macro NATJOIN (r, s) ;
1 get and expand types for r and s; generate new names
2 compute the set of unique and overlapping components of r and s
3 compute the output type definition and add it to environment via units
4 compute the representations of the match and concat functions
5 build the representation of the inline expansion
```

Figure 4.2 Outline of a TRPL Natural Join Macro Definition

Figure 4.3 shows the header and a segment for accomplishing task 1 in a TRPL macro definition for a natural join function.

```
macro NATJOIN (r, s) ; env e;
let  ertype :=  type_of (r, e),            @ get the types of r and s
     estype :=  type_of (s, e),
     rtype :=   expandtype (ertype, e),    @ expand set types to
     stype :=   expandtype (estype, e),    @ remove any type variables

@ build component lists for r and s
     rcomps := case rtype
                {parametric_rep ("set", cons (struct_rep (?, rcompslist), nil))
                         -> rcompslist,      @ ? indicates tuple constructor name unimportant
                others  -> warning ("first argument not a set of tuple", nil)},
     scomps := case stype
                {parametric_rep ("set", cons (struct_rep (?, scompslist), nil))
                         -> scompslist,
                others  -> warning ("second argument not a set of tuple", nil)},

@ generate symbols for new type
     tn := genstring ("type$"),

@ and constructor function for output tuples
     constr := genstring ("constr$"),
```

Figure 4.3 Natural Join Definition Segment for Type Extraction

First the types of *r* and *s* are extracted from the current compilation environment *e* using a built-in function *type_of*. This uses an environment variable defined in the header as the current compiler environment. These types are expanded using another built-in function *expandtype*. This expands all type variables contained in a type representation into their structural forms. The next two equations extract the list of component names by using pattern recognition on the representation of the input types. A representation of a legal type for this macro call is of the form *parametric_rep ("set", cons (struct_rep ("constrName", componentList), nil))*. The case statement either matches this for each type representation or returns an error. When a match is made the variables in the pattern are bound to their matched components and the case body is evaluated. Question marks stand for parts of the value to be matched by anything and ignored. The case bodies here are just the extracted list of component name and type pairs. This section ends with the generation of new names for the output type and a constructor function for its tuples.

Figure 4.4 shows tasks 2 and 3: the computation of the unique and overlapping components of the two input relations, along with the *units* section containing the generation of the output type definition.

```
    runique :=
        set_difference (rcomps, scomps, [x & ?, y & ?] -> string_eq (x, y)),
    sunique :=
        set_difference (scomps, rcomps, [x & ?, y & ?] -> string_eq (x, y)),
    overlap :=
        set_intersection (scomps, rcomps, [x & ?, y & ?] -> string_eq (x, y))
in
units
    LIST (  @ the new type definition
            define_type ( tn,
                          parametric_rep ("set", LIST (struct_rep (constr,
                              append3 (overlap, runique, sunique))))))
```

Figure 4.4 Computing Component Overlap and the Output Type

This code uses pattern matching lambda expressions, the expressions starting with *[x & ?, y & ?]*. In these functions the input arguments are first matched with the patterns in the brackets. The patterns here are pairs since *&* is the infix pair construction operator. The values are *struct_component*s defined as pairs in Figure 4.1. As before, successful pattern matching causes the variables in the patterns to be bound to the matching components of the values. In this case *x* and *y* are bound to the names of the components.

The unique and overlapping components are computed by *set_difference* and *set_intersection* utilising the lambda functions over the component lists. Pattern matching lambda expressions capture the criterion that components are equal when their names represented as strings are equal. If the names are equal but the types are not, the *match* function will produce a type error when it is passed to the compiler. The *units* section contains only the output type definition using another built-in function *define_type.* The first parameter gives the computed type name and the second supplies the representation of the type expression including the tuple constructor function name, bound to *constr*. Note the use of the constructor functions, *parametric_rep* and *struct_rep*, to construct the typed representation of the new type.

Figure 4.5 gives the code for generating the representations of the *match* and *concat* function bodies. The *match* body is an expression of the form *rt.a=st.a && rt.b=st.b && ... && true*, where && denotes logical *and*. It is to be used in the inline expansion as the body of a lambda function having *rt* and *st* as variables standing for the tuples of the input relations, *r* and *s*.

```
@ build bodies of match and concat
@ first a representation for the body of the match lambda
@ expression which looks like rt.a=st.a && rt.b=st.b && ... && true

let  eqterm := [x & ?] -> EREP ((rt.field) = (st.field), field := s2id (x)),
     match := listreduce (  listmap( overlap, eqterm),
                            [term, exp] -> EREP (t && e, t := term, e := exp),
                            EREP (true)),

@ build a representation for the body of the concat
@ lambda expression which looks like
@ construct (rt.common1, … rt.unique1, … st.unique1, …)

concat := EREP (  con ( …args),
                  con := s2id (constr),
                  args := append3 (  listmap (overlap, [x & ?] -> EREP (rt.f, f := s2id (x))),
                                     listmap (runique, [x & ?] -> EREP (rt.f, f := s2id (x))),
                                     listmap (sunique, [x & ?] -> EREP (st.f, f := s2id (x)))))
```

Figure 4.5 Generating the *match* and *concat* Representations

This portion of the definition uses a macro, *EREP*, to facilitate the generation of expression representations. *EREP* takes as its first argument an expression which gives a pattern for the representation it generates. Optional arguments may follow which give values to be substituted in the representation of the first argument. This allows computed representations to be inserted into constant expressions. A simple example of this is *EREP (f (x), x := s2id ("y"))*, where *s2id* is a function that converts a string value into the representation of an identifier. This evaluates to the representation of *f (y)*. The *match* body is produced by mapping the *eqterm* function over the overlapping component name and type pairs. The *eqterm* function takes a component pair, extracts the component name and constructs an equality expression that compares the named projection of *rt* and *st* tuples. The list of these terms is used to construct a boolean expression *and*ing all the equality terms with *true*. This uses a reduction function over the mapped list. The reduction uses a binary lambda function and *EREP* to build the representation of the *and* expression. Starting the reduction with *true* defines the base case of no common component names to be the cartesian product.

The *concat* body is generated by using *EREP* and *listmap*, together with a feature that allows variable length constructs in the pattern used in *EREP*. The ellipsis before *args* marks it as a parameter that accepts a list for its substitution. The list of representations of component names is produced by the *append3* and *listmap* functions, the former a function that appends three lists. An example of a *concat* body is *make_a_b_c_d (rt.a, rt.b, rt.c, st.d).*

Figure 4.6 gives the definition of the inline expansion to be generated. It uses *EREP* and the computed bodies of *match* and *concat* to generate the representation of a call to *join.*

```
@ the inline expansion is a call to join with lambda functions for match and
concat
in
        EREP ( join (r, s, [rt, st] -> mtch, [rt, st] -> cnct),
                  mtch := match,
                  cnct := concat)
```

Figure 4.6 Defining the Inline Expansion

Figure 4.7 shows the evaluation of a call to *NATJOIN*. The types of *r* and *s* are *rtype* and *stype* as given in Section 4.1.1.

NATJOIN (r, s)
=> eval (NATJOIN (r, s))
=> eval' (compile (NATJOIN (r, s)))
=> eval' (postParseCompile (parse (NATJOIN (r, s))))
=> eval' (postParseCompile ((NATJOIN (r, s))$_v$))

  ! the reflection is recognised
=> eval' (postParseCompile (raise (eval' (drop ((NATJOIN (r, s))$_v$)))))

  ! *drop* produces a call to the compiled NATJOIN definition
=> eval' (postParseCompile (raise (eval' (NATJOIN_generator (r$_v$, s$_v$)))))
=> eval' (postParseCompile (raise ((join (r, s,[x, y] -> x.a = y.a,
          [x, y] -> make_a_b_c_d (x.a, x.b, x.c, y.d)))$_v$)))

  ! *raise* is the identity function
=> eval' (postParseCompile ((join (r, s, [x, y] -> x.a = y.a,
          [x, y] -> make_a_b_c_d (x.a, x.b, x.c, y.d)))$_v$))
=> eval' (join (r, s, [x, y] -> x.a = y.a,
        [x, y] -> make_a_b_c_d (x.a, x.b, x.c, y.d)))
=> natural join of *r* and *s*

Figure 4.7 Natural Join Using Optimised Compile-time Linguistic Reflection

By the time the compilation of *NATJOIN* starts, the definition given in Figures 4.3 through 4.6 has been compiled into a generator comprising the call to *define_type* and the code that constructs the inline expression. *NATJOIN_generator* stands for the compiled form of this generator. *drop* produces a call to this generator with the parsed form of the macro's arguments as input. The output of this generator is bound into the original computation simply by being passed to the post-parse compiler.

While the types of the input of a natural join are inferred by the underlying type system, which is non-reflective, the inference of the output type is beyond the type system's capabilities. Reflection has been, in effect, used to perform this inference. Note however that neither having the programmer provide the output type nor building the ability to do this particular inference into the type system solves the problem. The *match* and *concat* functions would still need to be synthesised and this is beyond the scope of type inference. Natural join is not the only problematical operation in relational database systems. The *nest* and *unnest* operations of nested relational systems make similar demands on a programming language [JS82].

4.2 Evolution in Strongly Typed Persistent Systems

Another way linguistic reflection may be used is in accommodating the evolution of strongly typed persistent object stores. Characteristics of such stores are that the type system is infinite and that the set of types of existing values in the store evolves independently from any one program. This means that when a program is written or generated some of the values that it may have to manipulate may not yet exist, and their types may not yet be known for inclusion in the program text. For strong typing these values must have a most general type but in some applications their specific types can only be found once they have been created.

An example of such a program is a persistent object store browser [DB88, DCK89] which displays a graphical representation of any value presented to it. The browser may encounter values in the persistent store for which it does not have a static type description. This may occur, for example, for values which are added to the store after the time of definition of the browser program. For the program to be able to denote such values, they must belong to an infinite union type, such as Amber's **dynamic** [Car85], PS-algol's **pntr** [PS88] or Napier88's **any** [MBC+89].

Before any operations may be performed on a value of an infinite union type it must be projected onto another type with more type information. This projection typically takes the form of a dynamic check of the value's type against a static type assertion made in the program that uses it. A projection of a Napier88 **any** value is shown in the example below:

```
proc (val : any)
    project val as specific onto
    int :        writeInt (specific )
    string :     writeString (specific )
    default :    writeString ("not a string or int")    ! specific not in scope here
```

This shows a procedure that takes a parameter *val* of type **any**. In the body of the procedure the specific type of *val* is matched against two alternatives. If a match occurs the name *specific*, denoting the value with the specific type, enters scope in the corresponding branch. If the type of *val* does not match either of the statically specified types **int** or **string** then the specific type of *val* is unknown and the default branch is executed.

As stated earlier the browser program takes as parameter an infinite union type to allow it to deal with values whose types were not predicted at the time of implementation. However the program cannot contain static type assertions for all the types that may be encountered as their number is unbounded. There are two possibilities for the construction of such a program: it may either be written in a lower-level technology [KD90] or else be written using linguistic reflection.

To allow a reflective solution the program must be able to discover dynamically the specific type of a value of the union type. Such functionality may be provided in a strongly typed language without compromising type security by defining representations of types within the value space of the language, i.e., within Val_L. An example of such a representation in TRPL was shown in Figure 4.1. Access to type representations may be provided by a function such as the Napier88 procedure

```
getTypeRep : proc (any -> TypeRep)
```

which allows a program to discover type description information by the manipulation of values of the representation type.

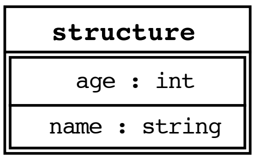
4.2.1 Details of Browser Implementation

The linguistic reflective implementation of the browser program has a number of components. First of all the value of the union type passed to the program is interrogated to yield a representation of its specific type. Using this information the browser constructs a representation of some appropriate Napier88 code. The compiler is called dynamically with this code representation as its argument, and returns some executable code which is capable of performing the appropriate projection of the union type, along with the required operations to browse the value. This new code is type-safe since it has been checked by the compiler. A different program will need to be generated for each different type of value which is encountered during the browsing of the persistent store.

An example of the operation of a Napier88 browser program will now be given in the context of the anatomy of run-time reflection defined in Section 2.4. Assume that a value of the following type, injected into the union type **any**, is passed to the browser:

```
type Person is structure (name : string ; age : int)
```

To display the value the browser needs to be able to construct and display a menu window such as that shown below:



It must also be able to extract the field values for further browsing should the user select one of the menu entries. The browser has built into it methods for displaying instances of the base types such as **string** and **int**. An outline of the browser code is shown in Figure 4.8.

```
let browser = proc (val : any)
begin
    let valTypeRep = getTypeRep (val)

    if valTypeRep denotes a base type then use built-in method else
    begin
        case true of
        valTypeRep denotes a structure type :
        begin
            let new = evaluate (makeCode (valTypeRep))

            ! new is of type any.
            ! makeCode builds up a string program representation through
            ! analysis of valTypeRep.

            ! For the example the result will be
            ! "type T is structure (name : string ; age : int)
            !    proc (x : any)
            !    project x as specificX onto
            !    T : menu2 ('"name : string'", '"age : int'",
            !                proc() ; browser (any (specificX (name))),
            !                proc() ; browser (any (specificX (age))))
            !    default : writeString ('"error'")"

            ! single quote is used as an escape to allow the inclusion
            ! of double quotes in the string.

            project new as newDisplayer onto
            proc (any) :   newDisplayer (val)
            default :      writeString ("error in compilation")
        end

        other cases : use similar methods for other type constructors
    end
end
```

Figure 4.8 Browsing Using Run-Time Linguistic Reflection

When the browser program is called it first obtains a representation of the type of the value passed to it. If it is one of the base types the browser has built-in knowledge of how to display it. Otherwise the type must be an instance of one of a fixed number of type constructors. In the example it is a structure type. The browser displays structures using a generic method. The method involves constructing a program that defines a procedure to display instances of the particular structure type, evaluating it and calling the resulting procedure to display the structure.

For brevity the definitions of the procedures *getTypeRep*, *makeCode*, *menu2* and *writeString* have not been shown. Note that the program produced by the generator itself contains a call to the browser program. This is achieved by binding the browser program into the persistent store where it can be accessed by the generated program. The details of this access have also been omitted.

Figure 4.9 shows the mode of evaluation of the reflective part of the browser program, the call to the *evaluate* procedure.

```
evaluate (makeCode (valTypeRep))
=> eval (evaluate (makeCode (valTypeRep)))
=> eval' (compile (evaluate (makeCode (valTypeRep))))
=> eval' (evaluate (makeCode (valTypeRep)))

    ! now the reflection denoted by the call to evaluate is recognised
=> eval (raise_Run (eval' (drop_Run (evaluate (makeCode (valTypeRep))))))
=> eval (raise_Run (eval' (makeCode (valTypeRep)))

=> eval (raise_Run ("type T is … writeString ('"error'")"))
=> eval' (compile (type T is … writeString ("error")))
=> eval' (type T is … writeString ("error"))

    ! the procedure value produced
=> proc( x : any ) ; …
```

Figure 4.9 Reflective Evaluation within the Browser

The algorithm shown is potentially inefficient as it requires reflection to be performed on every encounter with a structure type. In practice the persistent store is used to cache the results of reflection so that the code generation and reflection need not occur for types encountered previously.

This style of reflection can be analysed in the context of the dimensions of reflection described in Section 3, as follows. The linguistic reflection is initiated at run-time when the evaluator *eval'* encounters the compiled form of the L_R construct *evaluate*. Generators are procedures that produce strings. In the cases that the generators execute without errors the strings represent fragments of Napier88 code, i.e., they are in Val_L. The generators are executed at run-time and may access values in the persistent store but have no direct access to compilation information. In the example type information is passed to the generator as a parameter; this is obtained using a pre-defined procedure that produces a representation of the type of any value injected into an **any** union. The result of the generation is compiled using the run-time callable compiler and the result of executing the new compiled code bound into the original computation using Napier88's **any** projection mechanism. This mechanism supports dynamic incremental binding. Finally, type checking occurs in two stages. In the first the internal type consistency of the generated program fragment is verified during the operation of the compiler at run-time. In the second the type compatibility of the existing program and the value produced by evaluation of the compiled fragment is checked during evaluation of the **any** projection clause. Note that two uses of the infinite union are required. One is to give a type to the *getType* procedure so that it may be statically typed and yet permit type inquiry over values of any type, and the other to give a type to the evaluate procedure so that it can evaluate any expression.

This example illustrates the use of linguistic reflection to define programs that operate over values whose type is not known in advance. These programs potentially perform different operations according to the type of their operands but without endangering the type security of the system. The requirement for such programs is typical of an evolving system where new values and types must be incrementally created without the necessity to re-define or re-compile existing programs.

4.3 Applications of Linguistic Reflection

Applications of reflection in the context of database programming languages have stimulated the development of the technology described above. These applications address the following problems:

- attaining high levels of genericity,
- accommodating changes in systems,
- implementing data models,
- optimising implementations, and
- validating specifications.

4.3.1 Attaining High Levels of Genericity

The examples given in sections 4.1 and 4.2 both address the problem of attaining a high level of genericity. In each, the type dependent details of instances of a family of functions are generated. Thus the generators can be thought of as highly generic abstractions over the functions. Another example of this approach is a set of four traversal functions over recursive data types [She91]. These functions generalise the list map and fold functions allowing them to be applied to any recursive data type. Sheard has also used the technique to define a deep equality test for any type [She90]. Similarly, forms systems for data entry and access can be automatically generated from type definitions. Cooper has used such a technique to provide a rich repertoire of interaction modes over any structures that may be defined in a range of data models [Coo90a]. There is frequently a greater range of type dependent algorithms required than can sensibly be provided by built-in system programs. Linguistic reflection allows programmers to tailor their own style of presentation without requiring them to use a separate language or to penetrate the internal properties of the system. Ease of use of reflective systems remains a significant problem; there have been several attempts to develop more suitable notations for expressing generators and the calls to them [Kir92, She90].

As demonstrated by the examples, the genericity achievable via linguistic reflection has often depended on the ability of a generator to access type details and generate program fragments that are tailored to the types given when the generator is executed. This constitutes a form of *ad hoc* polymorphism [Str67], but the genericity attained in these examples exceeds the capabilities of current polymorphic type systems [SFS+90]. In most polymorphic systems, the behaviour of polymorphic functions must be essentially invariant over the range of input types. The examples listed above have behaviour that varies too much to be accommodated by polymorphic systems.

4.3.2 Accommodating Changes in Systems

The browser described in section 4.2 illustrates the way in which programs can adjust to system evolution, in this case the creation of values of previously unencountered types. Linguistic reflection can be used to accommodate a wide range of system changes. For example the schema changes of typical database applications become type changes in database programming languages, and reflective programs that are based on type details can regenerate code whenever a schema changes. If algorithms such as joins or form generation are systematically derived from the type information these derivations will be re-computed. With run-time reflection this happens lazily which may save computation since many systems undergo a sequence of changes between runs of many of their applications. In contrast the hand-crafted method of providing the same functionality requires that a programmer locate all the places where changes are necessary, perform all the changes correctly and then re-validate the software. The reflective method gains particularly well in this case as it may avoid the need for re-validation as is discussed below.

4.3.3 Implementing Data Models

A data model is typically defined by a data description language and by one or more data manipulation languages (including query languages). Linguistic reflection allows these languages to be implemented efficiently, avoiding any additional levels of interpretation.

Sentences in the data description language introduce new model constructs. The reflective generator translates these sentences into type declarations and declarations of associated procedures and introduces these into the computational context. Sentences in the data manipulation language are then translated into corresponding algorithms against these representational types and executed via reflection. In a persistent language this provides a very rapid means of prototyping and evaluating a data model [Coo90a, Coo90b, CQ92]. With the optimisation strategies discussed below this can be developed into a reasonable quality implementation of a DBMS for the data model.

This use of reflection to implement languages is not confined to data models. The technique is applicable to any language and has been used in a commercial system to develop requirements analysis tools based on process modelling [BPR91, GGR92, War89]. Philbrow has used the same technique to provide polymorphic indexing mechanisms over arbitrary collections [Phi90].

4.3.4 Optimising Implementations

Using linguistic reflection to avoid a level of interpretation is a form of optimisation. In addition to this optimisation, a generator that develops concrete code for high level abstractions can choose from implementation strategies in order to minimise costs [CAD+87]. Relational query optimisation, for example, can be integrated directly into the compilation process via linguistic reflection. Run-time reflection allows re-compilation and new optimisation as the statistics of the database change. More general transformations of high level specifications into implementations can also be accomplished using linguistic reflection [FS91].

4.3.5 Validating Specifications

There are various ways linguistic reflection can be used to support validation of programs. The first derives from the fact that generated program fragments are stereotyped in their form. This stereotyping can be aimed toward producing forms that facilitate verification efforts [FSS92, SSF92]. Generators themselves can be analysed in order to verify properties of all generated expressions. Though this is a second order problem, there is the possibility of stereotyping the generator programs themselves to produce sub-languages that support the second order reasoning. Validating generators would be especially useful since it would mean that programs that were regenerated as a result of system evolution such as changes to types would not need to be re-validated.

Theorem proving itself can be integrated with compilation using linguistic reflective capabilities. A version of the Boyer-Moore theorem prover kernel has been implemented in TRPL working over the parsed form of TRPL's functional core language. Using this kernel validation of properties of TRPL functional programs can be performed as a part of the compilation process. For example, the problem of verifying that database integrity constraints are invariants of transactions can be addressed by this approach [SS89].

5 Conclusions

A style of reflection appearing in strongly typed programming languages has been identified, defined and described. This style of reflection, termed type-safe linguistic reflection, can extend the class of algorithms that can be written in a type-safe manner. Linguistic reflection is characterised by the ability of a program to generate code in its language that is to be integrated into its own execution. This ability provides a base for generator technology that can be integrated with a programming language in a uniform and type-safe manner. While this capability has been a feature of many interpreter based languages with weak type systems, it is relatively new in compiler based, strongly typed systems. Two styles of linguistic reflection have arisen in database programming languages, compile-time and run-time. Both have been described in detail, allowing a comparison of the mechanisms as currently implemented.

Many uses have been found for linguistic reflection in the database programming area. These uses are characterised by a need for a high level of genericity in specifying data and procedures, a requirement that has proved problematical to meet using programming language type systems alone. Two such uses have been detailed and several more discussed.

Type safety has been achieved in PS-algol, Napier88 and TRPL by type checking each generated program segment, which is necessary when the complete programming language can be used to write generators. Limiting the language subset available for writing generators may allow the generators to be type checked for the type of all output at one time. This is a topic for future research. Other work to be done includes combining the two styles of reflection presented here, finding well engineered means of writing linguistically reflective code, and exploring the relationship of linguistic reflection with other kinds of reflection.

ACKNOWLEDGEMENTS

This paper was started at a one week workshop on reflection held in St Andrews in January 1991. The work was supported by ESPRIT II Basic Research Action 3070 – FIDE, SERC grants GR/H 15219 and GR/F 02953, and National Science Foundation grants IRI-8606424 and IRI-8822121. Richard Connor is supported by SERC Postdoctoral Fellowship B/91/RFH/9078.

REFERENCES

[BCP71] Burstall, R.M., Collins, J.S. & Popplestone, R.J. **Programming in POP-2**. Edinburgh University Press, Edinburgh, Scotland (1971).

[BPR91] Bruynooghe, R.F., Parker, J.M. & Rowles, J.S. “PSS: A System for Process Enactment”. In Proc. First International Conference on the Software Process: Manufacturing Complex Systems (1991).

[CAD+87] Cooper, R.L., Atkinson, M.P., 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.

[Car85] Cardelli, L. “Amber”. AT&T Bell Labs, Murray Hill Technical Report AT7T (1985).

[Cod79] Codd, E.F. “Extending the relational model to capture more meaning”. ACM Transactions on Database Systems 4, 4 (1979) pp 397-434.

[Coo90a] Cooper, R.L. “On The Utilisation of Persistent Programming Environments”. PhD Thesis, University of Glasgow (1990).

[Coo90b] Cooper, R.L. “Configurable Data Modelling Systems”. In Proc. Ninth International Conference on the Entity Relationship Approach, Lausanne, Switzerland (1990) pp 35-52.

[CQ92] Cooper, R.L. & Qin, Z. “A Graphical Data Modelling Program With Constraint Specification and Management”. In Proc. Tenth British National Conference on Databases, Aberdeen (1992).

[DB88] Dearle, A. & Brown, A.L. “Safe Browsing in a Strongly Typed Persistent Environment”. Computer Journal 31, 6 (1988) pp 540-544.

[DCK89] Dearle, A., Cutts, Q.I. & Kirby, G.N.C. “Browsing, Grazing and Nibbling Persistent Data Structures”. In **Persistent Object Systems**, Springer-Verlag (1989) pp 56-69.

[FS91] Fegaras, L. & Stemple, D. “Using Type Transformation in Database System Implementation”. In Proc. Third International Conference on Database Programming Languages, Nafplion, Greece (1991) pp 289-305.

[FSS92] Fegaras, L., Sheard, T. & Stemple, D. “Uniform Traversal Combinators: Definition, Use and Properties”. In Proc. Eleventh International Conference on Automated Deduction (CADE-11), Saratoga Springs, New York (1992).

[GGR92] Greenwood, R.M., Guy, M.R. & Robinson, D.J.K. “The Use of a Persistent Language in the Implementation of a Process Support System”. ICL Technical Journal 8, 1 (1992) pp 108-130.

[JS82] Jaeschke, G. & Schek, H.J. “Remarks on the Algebra of Non First Normal Form Relations”. In Proc. First Symposium on the Principles of Database Systems (1982) pp 124-138.

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

[Kir92] Kirby, G.N.C. “Persistent Programming with Strongly Typed Linguistic Reflection”. In Proc. 25th International Conference on Systems Sciences, Hawaii (1992) pp 820-831.

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

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

[Mil78] Milner, R. “A Theory of Type Polymorphism in Programming”. Journal of Computer and System Sciences 17, 3 (1978) pp 348-375.

[Phi90] Philbrow, P.C. “Indexing Strongly Typed Heterogeneous Collections Using Reflection and Persistence”. In Proc. ECOOP/OOPSLA Workshop on Reflection and Metalevel Architectures in Object-Oriented Programming, Ottawa, Canada (1990).

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

[RC86] Rees, J. & Clinger, W. “Revised Report on the Algorithmic Language Scheme”. ACM SIGPLAN 21, 12 (1986) pp 37-43.

[SFS+90] Stemple, D., Fegaras, L., Sheard, T. & Socorro, A. “Exceeding the Limits of Polymorphism in Database Programming Languages”. In **Lecture Notes in Computer Science**, 416, Springer-Verlag (1990) pp 269-285.

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

[She91] Sheard, T. "Automatic Generation and Use of Abstract Structure Operators". ACM Transactions on Programming Languages and Systems 19, 4 (1991) pp 531-557.

[SS89] Sheard, T. & Stemple, D. "Automatic Verification of Database Transaction Safety". ACM Transactions on Database Systems 12, 3 (1989) pp 322-368.

[SSF92] Stemple, D., Sheard, T. & Fegaras, L. "Linguistic Reflection: A Bridge from Programming to Database Languages". In Proc. 25th International Conference on Systems Sciences, Hawaii (1992) pp 844-855.

[Str67] Strachey, C. **Fundamental Concepts in Programming Languages**. Oxford University Press (1967).

[War89] Warboys, B. "The IPSE 2.5 Project: Process Modelling as the Basis for a Support Environment". In Proc. First International Conference on System Development Environments and Factories, Berlin, Germany (1989).