This paper should be referenced as:

Stemple, D., Morrison, R., Kirby, G.N.C., & Connor, R.C.H. “Integrating Reflection, Strong Typing and Static Checking”. In Proc. 16th Australian Computer Science Conference, Brisbane, Australia (1993) pp 83-92.

Integrating Reflection, Strong Typing and Static Checking

D. Stemple[1], R. Morrison[2], G.N.C. Kirby[2] & R.C.H. Connor[2]

[1]Department of Computer and Information Science,
University of Massachusetts, Amherst, MA 01038, USA
stemple@cs.umass.edu

[2]Department of Mathematical and Computational Sciences,
University of St Andrews, North Haugh, St Andrews KY16 9SS, Scotland
{ron,graham,richard}@dcs.st-andrews.ac.uk

Abstract

We define and present the computational structure of linguistic reflection as the ability of a running program to generate new program fragments and to integrate these into its own execution. The integration of this kind of reflection with compiler based, strongly typed programming languages is described. This integration is accomplished in a manner that preserves strong typing and does not unduly limit the amount of static type checking that can be performed. The benefits that accrue to linguistic reflection in the area of database and persistent programming languages are outlined and two examples are given.

1 Introduction

Linguistic reflection [SSS+92] is defined as the ability of a running program to generate new program fragments and to integrate these into its own execution. The focus of this paper is the integration of this style of reflection with compiled, strongly typed languages.

Three roles that type systems play in programming languages are data modelling, avoiding a class of errors in running programs and facilitating efficiency in implementations. Strong typing requires that all programming entities be typed before use and that all use be consistent with the type system. Static type checking is verifying type assertions prior to a program's execution. Strong typing ensures that a certain class of errors are cleanly detected and static typing improves efficiency by removing type checks from the run-time code. The goal of most type systems in this respect is therefore to make checking as static and thus as efficient as possible. However, some type checking cannot be performed statically. For example, user input must be checked when it is available, usually during the execution of an input statement.

Type systems also impose structure on the computations and therefore provide a data modelling facility. Systems require their data models to evolve through time and this is not well supported in static type systems. There is therefore a compromise between the roles of the type system in terms of safety and flexibility and the time of checking for type compatibility.

Linguistic reflection has the goal of allowing a program's behaviour to adjust dynamically in order to provide flexibility and high productivity. It thus extends the data modelling of the type system and it should not be surprising therefore to find a tension between type systems and reflection. The possibility that a program may significantly change its behaviour decreases the opportunity for static type checking and thus compromises some of the benefits of typing. In this paper, two means of integrating reflective computation with compilers for strongly typed languages are presented. The reflective facilities are controlled in a manner

designed to retain as much static type checking as possible. However the control is not so severe as to remove all the benefits of reflection.

Two techniques for type-safe linguistic reflection have evolved: compile-time linguistic reflection and run-time linguistic reflection. Compile-time linguistic reflection allows the user to define generator functions which produce representations of program fragments. The generator functions are executed as part of the compilation process. Their results are then viewed as program fragments, type checked and made part of the program being compiled.

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

The motivation for the work presented here has come from database and persistent programming. The benefits of linguistic reflection in database and persistent programming consist mainly of two capabilities. The first is the ability to implement highly abstract specifications, such as those used in query languages and data models, within a strongly typed programming language. The second is the ability to accommodate some of the continual changes in data-intensive applications without resorting to highly interpretative approaches or ad hoc restructuring methods. Both capabilities involve reflective access to the types of a system that is changing itself.

Both compile-time and run-time reflection have been provided in previous languages. Compile-time reflection appears in the macro facilities of Scheme [RC86] and POP-2 [BCP71] . Note that these differ from the macro systems of languages such as C [KR78] , where only a limited sub-language is available for defining the macros. Run-time reflection appears in the *eval* functions of Lisp [MAE+62] and SNOBOL4 [GPP71] and the *popval* function of POP-2.

Type-safe linguistic reflection is different for the following reasons:

- More information is available to the reflective computation, in the form of systematically required types. This information can be used to automatically adjust to implementation details and system evolution. Linguistic reflection without strong typing has little systematic information available about the structures involved in computation.

- The type safety of all newly generated program fragments is checked before they are allowed to be executed. Such type discipline is highly advantageous in a database programming environment in which the integrity of long-lived data must be supported.

It is somewhat ironic that strong typing, which makes it difficult to integrate reflection with typed programming languages, is what makes linguistic reflection effective as an amplifier of productivity.

Type-safe linguistic reflection has been used to attain high levels of genericity [SFS+90, She91] and accommodate changes in systems [DB88, DCK89] ; two examples of these are given. It has also been used to implement data models [Coo90a, Coo90b, CQ92] , optimise implementations [CAD+87, FS91] and validate specifications [FSS92, SSF92] . The importance of the technique is that it provides a uniform mechanism for production and evolution that exceeds the capabilities of present database programming languages.

This paper is structured as follows: Section 2 contains the definition and the anatomy of the two kinds of linguistic reflection. Section 3 describes two examples of the use of type-safe linguistic reflection, abstraction over types and accommodating evolution in strongly typed persistent systems.

2 Definition and Anatomy of Linguistic Reflection

2.1 Linguistic Reflection

Linguistic reflection occurs whenever an expression in a language is evaluated and the result of that evaluation is itself evaluated as an expression in the language in the ongoing computation. This will be described in terms of an evaluation function, *eval*, which is used to translate sentences in the language into values of some form. This translation occurs in both compilation and interpretation.

Given a language, *L*, and a domain of values, *Val*, the type of the evaluation function *eval* is:

$$eval : L \rightarrow Val$$

The domain of values, *Val*, differs for different languages, for example, 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*. 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 has two parts: the first is to invoke a generator to produce a program fragment; the second is to cause the generated fragment to be evaluated. 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 relationships among the domains are:

$$L_R \subset L$$
$$Val_{L_R} \subset Val_L \subseteq Val$$
$$L_{Gen} \subseteq L$$

Two more 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*. The second, *raise*, takes a value in Val_L and produces an expression in *L*:

$$drop : L_R \rightarrow L_{Gen}$$

$$raise : Val_L \rightarrow L$$

Linguistic reflection is defined as the occurrence of the pattern of computation shown in Figure 1 within the *eval* function, in the evaluation of a program in *L*. The function inL_R tests whether a given language construct lies within the reflective sub-language L_R.

```
procedure eval (e : L) → Val
        case
        ...
        inL_R (e) => eval (raise (eval (drop (e))))
        ...
```

Figure 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 in the ongoing computation.

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*.

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. The type checking of 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. A major significance of compilation in this context is that the parsing phase of compilation also includes type checking. This is the principal difference between compiled languages and those such as Lisp, Scheme, POP-2, SNOBOL4 etc., where compilation and type checking of the reflectively generated code do not occur. This lack of type system protection reduces the suitability of their reflective mechanisms for use in a database programming language context.

The anatomy given so far must be further refined to describe reflection in a compiled language. In Figure 2 the structure of *eval* in an environment with a compiler is shown as a composition of two functions, *compile* and *eval'*:

```
procedure eval (e : L) → Val
        eval' (compile (e))
```

Figure 2 *eval* as Function Composition

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
```

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, that is static, 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*. 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 shown in Figure 3.

```
procedure eval (e : L) → Val
        eval' (compile (e))

procedure compile (e : L) → L'
        if inL_R(e)  then compile (raise (eval' (compile (drop (e)))))
                     else translate (e)
```

Figure 3 *eval* in Compile-Time Linguistic Reflection

The reflective pattern here is

```
eval' (compile (raise (eval' (compile (drop (e))))))
```

2.3.1 Optimised Compile-Time Linguistic Reflection

An optimised variant of the previous architecture can be produced by dividing the compiler into two parts: a *parser* that translates the source level program into an intermediate abstract syntax form, and a *post-parse compiler* that takes abstract syntax and completes the compilation. This choice of Val_L as abstract syntax representations 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 produces a compiled version of the generator in the target language generator subset L_{Gen}. This optimised *drop* function is denoted by $drop_{Opt}$. 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 shown in Figure 4.

```
procedure parse (e : L) → Val_L
    ...

procedure eval (e : L) → Val
    eval' (compile (e))

procedure compile (e : L) → L'
    postParseCompile (parse (e))

procedure postParseCompile (e_v : Val_L) → L'
    if inL_Rv (e_v) then postParseCompile (raise_Opt (eval' (drop_Opt (e_v))))
                    else translate (e_v)
```

Figure 4 *eval* in Optimised Compile-Time Linguistic Reflection

Here the reflective evaluation pattern is

```
eval' (postParseCompile (raise_Opt (eval' (drop_Opt (e_v)))))
```

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'*.

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 forms of L_R, to initiate reflection. The original expression e is in the process of being evaluated by

```
eval (e)
=> eval' (compile (e))
=> eval' (e) ! where e is the compiled form of e
```

The pattern of *eval'* in this case is shown in Figure 5.

```
procedure eval' (e : L') → Val
        case
        ...
        inL_R' (e) => eval (raise_Run (eval' (drop_Run (e))))
        ...
```

Figure 5 *eval* in Run-Time Linguistic Reflection

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 $L_R' \rightarrow L_{Gen}'$. The reflective evaluation pattern here is

```
eval (raise_Run (eval' (drop_Run (e))))
```

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. Run-time reflection with type checking performed after any new fragment is generated contains a form of dynamic type checking. However, the form described here limits the checking to the incremental addition and its conformity with the program it increments. Since the new code is checked prior to its execution this could also be thought of as a case of incremental static type checking. The main point is that the requirement for new checking is kept to a minimum and the benefits of static checking are preserved as much as possible while the flexibility afforded by reflection is made available in the strongly typed environment.

2.5 Dimensions of Linguistic Reflection

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 is the result of the generation bound into the original computation?

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

- When is the type checking performed?

The models of compile-time and run-time linguistic reflection presented above represent three sets of choices along these dimensions. Combining compile-time and run-time reflection in a single language and adding a persistent store to the environment provide opportunities to explore further the space of linguistic reflection.

3 Uses of Linguistic Reflection

Here two examples of linguistic reflection are presented, in order to show how the reflection mechanisms that have been described appear at the level of a programming language. The examples are abstraction over types and accommodating evolution in strongly typed persistent systems. Following these examples other applications of linguistic reflection are given.

3.1 Abstraction Over Types

3.1.1 Natural Join

A generic natural join function provides an example of abstraction over types that is beyond the capabilities of most polymorphic type systems. 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 code 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. A call to a polymorphic generic join can be generated. The generic join takes two relations, a function for testing that two tuples match (called *match* below) and a function that constructs the output tuples by concatenating matching input tuples (called *concat* below). Here reflection is used to specialise the generic join, which requires the match and concatenation functions as input, to a generic natural join that requires only the two input relations. The natural join call in TRPL can be of the form:

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

and generate a call to the generic join of the form:

```
join (r, s, match, concat)
```

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

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

stype = set (struct make_a_d (a : integer, d : boolean));
```

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*.

The call to *NATJOIN* is a TRPL reflective construct and its inner *eval* generates a 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 as illustrated below. The @ symbol is used to indicate a comment line.

```
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,                                  @ TRPL form of lambda function
        [x, y] → make_a_b_c_d (x.a, x.b, x.c, y.d))
```

It is the ability of a TRPL macro to determine the types of the relations *r* and *s* that allows the generic natural join to be written. This ability is lacking from the reflective facilities of the languages listed in Section 1.

3.1.2 TRPL Optimised Compile-Time Reflection

In TRPL, Val_L comprises values of two types, one for representing types, *type_rep*, and one for expressions, *exp_rep*. Figure 6 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 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),
                   function_call : struct make_function_call (
                                        function_name : string,
                                        parameter_list : list (exp_rep)),
                   ...)
```

Figure 6 TRPL Types to Define Values in Val_L for TRPL

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 generator functions invoked by the macro calls are defined in macro definitions and are functions from the parsed macro input (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 completion 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 7 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 for output type
2    compute the unique and overlap component names 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 7 Outline of a TRPL Natural Join Macro Definition

3.2 Evolution in Strongly Typed Persistent Systems

Linguistic reflection may also be used in accommodating the evolution of strongly typed persistent object stores. A characteristic of such stores is 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.

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 that 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] 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. The browser program takes as parameter an infinite union type, an **any**, 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, and 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.

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.

To display a value the browser needs to be able to construct and display a menu window with an entry for each field. 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 8.

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, for example a structure type. The browser displays structures using a generic method that 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.

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 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.

3.3 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.

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

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

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

        else use similar methods for other type constructors
    end
end
```

Figure 8 Browsing Using Run-Time Linguistic Reflection

4 Conclusions

This paper discusses how the integration of linguistic reflection, strong typing and static checking can provide a uniform mechanism for the production and evolution of data and programs that exceeds the capabilities of present database programming languages. In particular, two examples of the technique, natural join and the persistent store browser, illustrate how linguistic reflection may be used in practice to overcome the limits of current polymorphic type systems with a minimum of dynamic type checking. Two kinds of reflection have evolved: compile-time and run-time. A semi-formal description of both mechanisms is given and it is demonstrated that they are manifestations of the same beast, that is the

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

evaluation pattern. Reflective facilities abound in other programming languages but we believe this to be the first reporting of the integration of reflection, strong typing and static checking together with the definition of the mechanism of reflection.

ACKNOWLEDGEMENTS

We are grateful for many useful discussions with Robin Stanton, Paul Philbrow, Leo Fegaras, Richard Cooper, Malcolm Atkinson and Suad Alagic. 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).

[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).

[Coo90a] Cooper, R.L. “Configurable Data Modelling Systems”. In Proc. 9th International Conference on the Entity Relationship Approach, Lausanne, Switzerland (1990) pp 35-52.

[Coo90b] Cooper, R.L. “On The Utilisation of Persistent Programming Environments”. Ph.D. Thesis, University of Glasgow (1990).

[CQ92] Cooper, R.L. & Qin, Z. “A Graphical Data Modelling Program With Constraint Specification and Management”. In Proc. 10th 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.

[DCK90] Dearle, A., Cutts, Q.I. & Kirby, G.N.C. “Browsing, Grazing and Nibbling Persistent Data Structures”. In **Persistent Object Systems**, Rosenberg, J. & Koch, D.M. (ed), Springer-Verlag (1990) pp 56-69.

[FS91] Fegaras, L. & Stemple, D. “Using Type Transformation in Database System

Implementation". In Proc. 3rd 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. 11th International Conference on Automated Deduction (CADE-11), Saratoga Springs, New York (1992).

[GPP71] Griswold, R.E., Poage, J.F. & Polonsky, I.P. **The SNOBOL4 Programming Language**. Prentice-Hall, Englewood Cliffs, New Jersey (1971).

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

[KR78] Kernighan, B.W. & Ritchie, D.M. **The C programming language**. Prentice-Hall (1978).

[MAE+62] 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).

[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 Notices 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**, Bancilhon, F., Thanos, C. & Tsichritzis, D. (ed), 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.

[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.

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