This paper should be referenced as:

Kirby, G.N.C., Connor, R.C.H., Morrison, R. & Stemple, D. “Using Reflection to Support Type-Safe Evolution in Persistent Systems”. University of St Andrews Technical Report CS/96/10 (1996).

Using Reflection to Support Type-Safe Evolution in Persistent Systems

G.N.C. Kirby, R.C.H. Connor, R. Morrison and D. Stemple*

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

*Department of Computer and Information Science,
University of Massachusetts, Amherst, MA 01038, USA.

Abstract

Reflection has been used to address many different problem areas, and the term *reflection* has itself been used to describe several distinct processes. This paper identifies three simple operations, *generation*, *raising* and *dynamic rebinding*, which may be composed to yield several varieties of reflection. These can be used to allow a self-contained programming system to evolve, through the incorporation of new behaviour into either the application programs or the interpreter which controls their execution.

Reflection is a powerful mechanism and potentially dangerous. Used in the context of persistent programming systems, safety is an important consideration: the integrity of large amounts of data may be at stake. This has led to the use of type checking in conjunction with reflection in such systems to provide some guarantees of safety. The paper describes the nature of reflection in persistent systems and identifies some example applications.

1 Introduction

Reflection may be defined loosely as the ability for a programming system to change its own behaviour. In general, the behaviour of any system may be captured by a definition of that system with respect to a framework for explaining the definition. For example the behaviour of a program may be captured by the definition of the program itself, and a semantics for the language in which it is written. There are two corresponding mechanisms by which a system can support its own evolution: by changing its own definition, or by changing the interpretation of its definition. The former may be termed *linguistic reflection* and the latter *behavioural reflection*.

Linguistic reflection is of particular interest in large, long-lived systems and applications. The importance of the mechanism is that it allows evolutionary change within such systems; it is distinguished from other mechanisms by the fact that no lower-level or external semantics are required to effect the change. The ultimate goal of linguistic reflection is to allow self-supporting evolving systems: no matter how the system is required to evolve in the future, it may always be achieved from within, without stepping out of the system into a different semantic domain.

Linguistic and behavioural reflection are both potentially dangerous mechanisms, and should be treated with great care. This is particularly important in strongly typed persistent programming systems which may contain large amounts of highly structured and valuable data. One approach to achieving the flexibility provided by reflection in such systems, while imposing sufficient restrictions for safety, is to combine linguistic reflection with type checking. In this way all changes to the system definition are verified by the type checker before use. This is in line with the already extended role of the type checker in such persistent systems, where it is used to control the use of data throughout the lifetime of the data.

The remainder of the paper contains a brief description of the distinguishing features of persistent programming systems, a description of the linguistic and behavioural reflective processes in such systems, some examples of their use, and finally a summary of related work. An appendix contains a more detailed description of the reflective processes.

2 Persistent Programming Systems

2.1 Persistence

The persistence of data is the length of time for which it exists; this may range from the very short to the very long. Some examples at the extremes of the spectrum are the temporary data created during the evaluation of an expression in a program and the long-term data stored in a company's customer database. In the first case the data persists for a brief fraction of a single program execution, while in the second the data may outlive the programs that operate on it.

Persistence technology is at the core of database programming languages. In an orthogonally persistent programming system, the manner in which data is manipulated is independent of its persistence. The same mechanisms operate on both short-term and long-term data, avoiding the traditional need for separate systems to control access to data of different degrees of longevity. Thus data may remain under the control of a single persistent programming system for its entire lifetime [ABC+83, AM85, Con90] . Since translation between storage formats is automatic and hidden from the programmer, a conceptually simpler programming system results. The principal gains provided by orthogonal persistence are:

- improved programming productivity from simpler semantics;
- avoidance of the need for *ad hoc* arrangements for long term data storage, and for data format translations; and
- type checking protection mechanisms operate over the whole environment.

2.2 Typing in Persistent Systems

Type systems are historically viewed as mechanisms which impose static safety constraints upon a program. Within a persistent environment, however, the type system takes on a wider role.

Data manipulated by a programming language is governed by that language's type system. In non-persistent languages, however, data which persists for longer than the invocation of a program may only be achieved by the use of an operating system or database interface which is shared by all applications. As a consequence of this, such data passes beyond the jurisdiction of the type system of any one language. Mechanisms which govern long term data, such as protection and module binding, must be dealt with at the level of this interface. Historically this has the consequence that the type system may not be enforced, and knowledge of the typed structure of data may not be taken advantage of.

In a persistent system, the storage of data beyond a single program invocation is handled by programming language mechanisms, and no common operating system interface is necessary. The only route by which data may be accessed is through the programming language, and so the type system of a single language may be used to enforce protection upon both transient and permanent data. High-level modelling may be relied upon for the entire lifetime of the data, as it never passes outside the language system. The universality of the persistent type system has consequences in terms of both the modelling and protection provided by the type system itself, and also presents some new challenges in terms of implementation.

With respect to modelling, the persistent type system must be sufficiently flexible to allow the modelling of activities normally provided by untyped support systems. Such activities

include, for example, the linking of separately prepared program units, and file system access protection.

With respect to protection, the increased role of a type system means that any protection mechanisms programmed at a high level may be fully relied upon to protect the data for its lifetime, as access from outside the constraints of the type system is not possible. In particular, various high level information hiding techniques may be used to restrict data access, instead of relying upon the normally coarse grain and typeless control provided by outside technologies.

2.3 Longevity and Evolution

One of the goals of persistent language research is to develop self-contained programming systems in which all aspects of the software process are represented within a uniform and consistent framework. A programming system will however become obsolete if it can no longer meet the changing needs of the applications that it supports.

Evolution is inevitable in a long running system as the people who use the data, the data and the uses to which the data is put all change. This is reflected within the programming system by changes to the data, the programs which use the data and the meta-data (the types of the programs and data). Changes to program and data with the invariant of fixed meta-data may be handled relatively simply by updates to programs and data respectively. In fact an orthogonally persistent system will support persistent first-class procedures, so that programs and other data need not be differentiated—a program may be stored in the persistent system as a procedure value.

A particularly difficult problem of evolution is to change the meta-data while keeping all the existing programs and data consistent with the semantics of the change. Here the constraints of the type system, introduced in order to improve safety, act to hinder the required evolution. For example if the definition of type *A* has been evolved to type *B*, it may be desirable to make all persistent values of type *A* now have type *B*. However it is just such a potentially dangerous operation that the type system is designed to prevent, since programs already bound to those values may rely on the assumption that they have type *A*.

One approach to resolving this conflict is the development of increasingly sophisticated type systems [CM88, Con90, ACP+91, BO91] . An alternative is to use reflection, combined with a relatively simple type system, to allow the system to alter its own behaviour as the need arises. The next section describes the mechanisms involved.

3 Behavioural and Linguistic Reflection

3.1 Underlying Mechanisms

Several forms of reflection may be described in terms of three operations which will be referred to as *generation*, *raising* and *dynamic rebinding*.

Generation involves the manipulation of program representations by a running program, so that it may generate representations of new program fragments. This requires the definition of some representation form in the program's value space; more details are given in the Appendix. The new program fragments may be created by composing other fragments, or from scratch. Although not essential, utilities which analyse and extract information from existing program fragments are useful in practice. In particular, type information derived from existing program fragments and other values may be used to generate new programs which will operate on them.

Raising involves transforming a generated program representation (which is a language value) into an executable program. In a strongly typed persistent system this must involve

type checking to ensure that the representation denotes a valid program, thus the operation will fail if presented with an invalid representation.

Dynamic rebinding involves a run-time linking of a value into a program so that the value may be accessed by that program. This might be achieved by assigning the value to a location accessible to the program. In strongly typed languages this also requires type checking to ensure that the new value is compatible with the code into which it is bound. Of particular interest is the special case where the value bound is a first-class procedure value, meaning that a new fragment of executable code (procedure) becomes available to the running application.

These three operations may be composed to give both linguistic and behavioural reflection as described in the next section.

3.2 Forms of Reflection

Figure 1 shows a representation of the possible anatomy of a reflective programming system. It consists of an application program with associated data, and an interpreter which executes the program. The interpreter itself consists of a number of levels of interpretation above an immutable system kernel. In turn, each level above the kernel contains programs, with associated data, whose behaviour is defined with respect to the interpreter defined by the levels below. This structure is similar to that of the infinite tower of interpreters [Smi82, FW86, DM88] , except that here the number of interpreters is finite and the languages interpreted by them need not be the same.

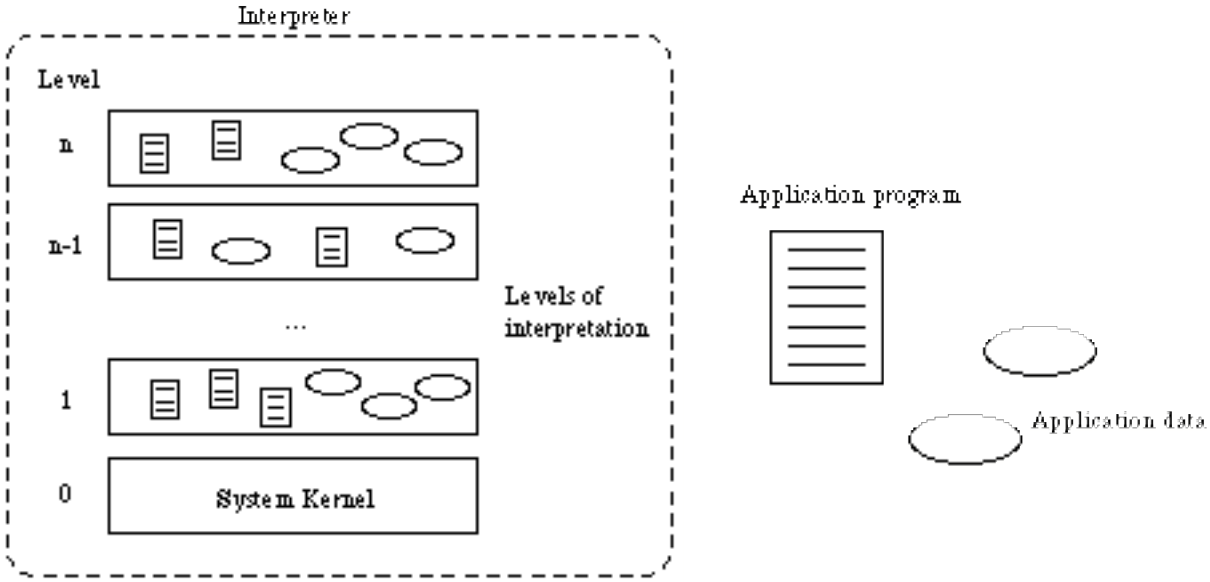

Figure 1. Anatomy of a reflective programming system

There are a number of ways in which an executing application program may alter its future behaviour. It may:

- update its associated data (case 1);
- alter its own code (case 2);
- update the data within a level of the interpreter (case 3); or
- alter the code of the programs within a level of the interpreter (case 4).

The change in each of these cases involves (or may involve) the combinations of operations shown below:

	generation	raising	dynamic rebinding
Case 1			•
Case 2	•	•	•
Case 3			•
Case 4	•	•	•

Case 1 is not reflective and may involve either dynamic rebinding where new application data is created, or simple update where existing data are modified.

Case 2, representing *linguistic reflection*, involves the application program creating a new program fragment and incorporating it into itself. This is achieved with the following combination of operations:

- generation, to create the fragment representation;
- raising, to check the representation and transform it into an executable fragment; and
- dynamic rebinding, to link the new fragment into the application.

Cases 3 and 4 represent *behavioural reflection* in which the application modifies the interpreter rather than itself. Case 3 is similar to case 1, except that the dynamic rebinding or update takes place at the interpreter level rather than the application level. Likewise, case 4 is similar to case 2 except that the generation and raising operate on the language of a particular interpreter level, which may be different from the application language.

Figure 2 illustrates the paths of change invoked in the various cases when a reflective sub-program within an application is executed.

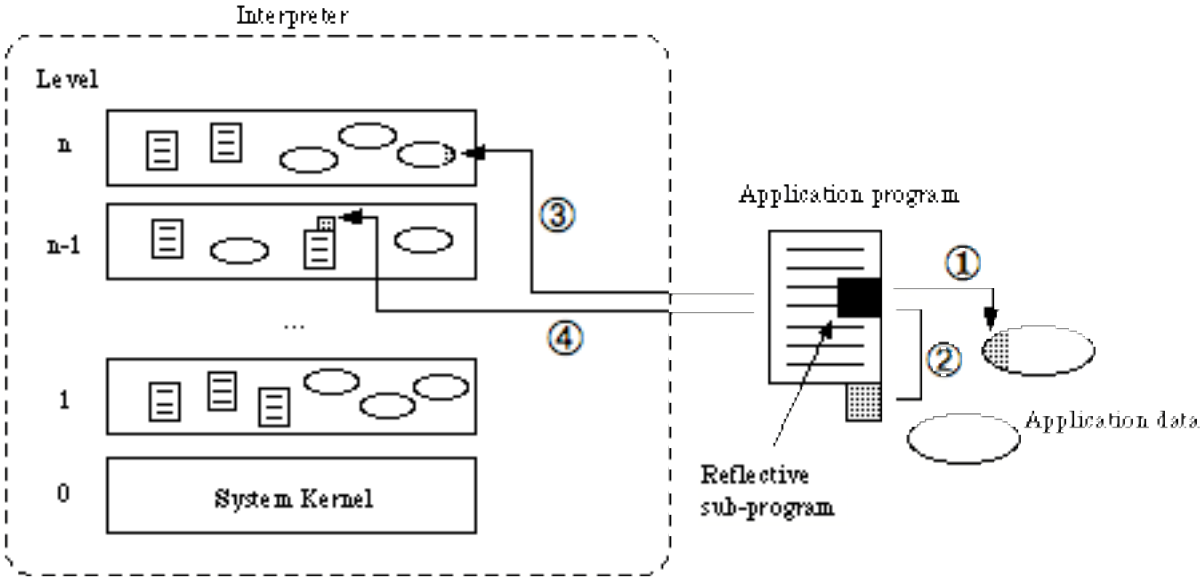

Figure 2. Evolution mechanisms

Systems which support linguistic reflection are described in [BCP71, DB88, She90, SSF92, FD93, HS94, KCM94, SH94] , while behavioural reflective systems are described in [Smi82, FW84, Smi84, FW86, LRN86, Mae87, DM88, PL89, Ala90, Mul92] .

3.3 Safety Checking

Uncontrolled reflection is a flexible and powerful mechanism. It is also potentially dangerous in that it may allow operations which compromise the integrity of the data in the system. This would be unacceptable in a persistent programming system. For this reason, linguistic reflection has been adopted as the appropriate flavour of reflection for persistent systems, since the raising and dynamic rebinding operations can incorporate type checking. In general, the problem of verifying whether a proposed change to a level of the interpreter could result in subsequent unsafe behaviour by the application is undecidable.

Persistent programming systems may thus support *strongly typed linguistic reflection* [SSS+92] which guarantees that any new code generated during the execution of reflective sub-programs is type correct.

Analysis of the reflective generators themselves to determine whether their output is well-typed is an open research issue [SH94] , and the systems described in this paper do not achieve this; rather, the code produced is tested for type correctness after it is generated. Thus dynamic failure is possible within these systems; however the type integrity of the overall system is guaranteed. In large scale programming systems this is of far greater importance than the elimination of run-time errors within an individual program. In particular, the points at which reflective generation occurs within code are typically well-defined and are generally programmed in such as way as to allow for the possible failure of the generator to produce well-structured code.

3.4 Inputs to Generation

The discussion so far has concentrated on how, given some representation of a desired system change, that representation is passed back into the system to effect the change. This happens either through transformation of the representation into executable application code or through incorporation of the representation into the interpreter.

Although not fundamental to reflection itself, another important issue is how a given representation is generated. It may be produced from scratch, but more often it is based on existing information, so in the case of behavioural reflection the generated representation may be based on a representation of the current state of the interpreter. In linguistic reflection the new representation may be based on a representation of some existing part of the application. For example a reflective browser might generate a program to display a value of a particular type, given a representation of that type.

Although a strongly typed programming system places restrictions on the use of reflection, in order to provide safety, such a system has the advantage that it contains much systematically acquired type information about its components, which is available for use by generator programs. Thus one of the utilities supported in a type safe linguistic reflection system is a **typeOf** operator which returns a representation, as a value, of the type of a given value. Another, **sourceOf**, returns a representation of the source code of a given executable program. Others may extract sub-parts and other information from program representations to assist in the analysis of existing program fragments.

Behavioural reflection systems may provide a **reify** operator which returns a program level representation of the current interpreter state.

4 Uses of Reflection

This section outlines two applications of type safe linguistic reflection to supporting evolution in persistent programming systems.

4.1 Schema Evolution

The problem of schema evolution in a long-lived strongly typed system was alluded to earlier. The search for general solutions is still a research issue but linguistic reflection may be used to provide a useful degree of support in some simple common cases. As an example of meta-data evolution consider Figure 3 which describes a partial schema for an application implementing a parts/suppliers database.

```
type address is ...

type Supplier is structure( s_name : string ; s_address : address )
type Part is structure( p_name : string ; p_no : int ; suppliers : set[ Supplier ] )

entry PARTS is set[ Part ]
entry SUPPLIERS is set[ Supplier ]
```

Figure 3. A partial schema—parts have suppliers

In Figure 3, a supplier, represented by the type *Supplier*, has a name *s_name* and address *s_address*. A part, represented by the type *Part*, has a part name *p_name*, a part number *p_no* and a set of suppliers *suppliers*. There are two entries in the schema, the set of parts *PARTS* and the set of suppliers *SUPPLIERS*. This schema is designed to model each part as having a set of suppliers.

As the system evolves it may become appropriate to change the organisation of the data to model each supplier as supplying a set of parts. This could be represented by the schema shown in Figure 4.

```
type address is ...

type Part is structure( p_name : string ; p_no : int )
type Supplier is structure(     s_name : string ; s_address : address ;
        supplied_parts : set[ Part ] )

entry PARTS is set[ Part ]
entry SUPPLIERS is set[ Supplier ]
```

Figure 4. A partial schema—suppliers supply parts

The databases represented in Figures 3 and 4 contain the same information. However, the data has different structure, the programs that operate over the data are different and the meta-data is different. Given that a user has specified the meta-data change it is now necessary for the existing data and programs to be changed consistently. Type safe linguistic reflection can be used to assist this process.

The first step is to locate all existing Parts and Suppliers, and all the programs which operate on them. This can be achieved by following links, which were automatically maintained, from the initial schema. Now the source code of those programs can be obtained, since it is also maintained automatically, and used together with details of the schema change to generate representations of equivalent programs which use the new definitions of *Part* and *Supplier*. Reflection is used to transform the representations into executable programs which then replace the old ones. Finally several strategies may be used to replace the data values with their new equivalents, either automatically or under user direction. This entire change to

the system structure can be made transactionally so that it may take place in a live system. More details are given in [CCK+94] .

4.2 Process Modelling

Another application of type safe linguistic reflection arises in process modelling systems such as PSS [GGR92] . Here the requirement is for software entities whose behaviour models real world processes. These may persist for years or decades, for example where a process is the maturation of a life assurance policy or an employee's progress within an organisation. The ability for process representations to evolve through time is thus crucial.

The central entity in PSS is the role, a persistent process which communicates with other roles through message passing. Each role is specified in the process modelling language PML and has an initial behaviour which may be modified dynamically in a type safe manner. This is achieved with the **BehaveAs** action which takes a string containing a specification of a new behaviour written in PML. The string is compiled using a run-time accessible compiler, yielding either an error if syntax or type checking failed, or a set of new procedures which implement the new behaviour. A particular role is then updated with the procedures so that its behaviour is subsequently altered. The role's internal data is now accessible by the new procedures.

5 Related Work

A number of programming systems support behavioural reflection. Some, such as 3-Lisp [Smi82] , Brown [FW84] and Blond [DM88] allow programs to access interpreter data structures directly. Object-oriented systems such as Smalltalk-80 [GR83] and 3-KRS [Mae87] provide meta-objects which define part of the behaviour of their corresponding objects, and allow objects to reflect by passing messages to their meta-objects.

Untyped linguistic reflection is present in Lisp [MAE+62] and POP-2 [BCP71] . Strongly typed linguistic reflection is supported in the persistent languages PS-algol [PS88] and Napier88 [MBC+94] , while TRPL [She90] and CRML [HS94] are languages in which the reflection takes place during program compilation. Recent work with Meta-ML [SH94] addresses the problem of static checking of non-manifest generator applications.

6 Conclusions

This paper has attempted to characterise two forms of reflection, linguistic and behavioural, as mechanisms by which a self-contained system may evolve. Due to the importance of strong typing in persistent programming systems, which may support large bodies of long-lived data, linguistic reflection combined with type checking is the form used in persistent systems. This is because the proposed addition of a programming fragment to an application program may be verified for safety relatively easily, whereas in general the checking of a change to the system's interpreter with respect to the application's subsequent behaviour is undecidable.

A number of interesting research areas remain, including:

- the judicious restriction of linguistic generators to allow static verification, in the sense that it could then be guaranteed that all possible program fragments output by a generator would be type-safe;

- similar restriction of behavioural reflection so that it could be verified whether a proposed change to the interpreter could compromise type safety at the application level;

- making reflective techniques easier to use by the programmer: examples of reflective programs have been deliberately omitted from the paper as such programs tend to be very hard to understand.

Acknowledgements

We are grateful for useful discussions with Robin Stanton, Tim Sheard, Leo Fegaras, Richard Cooper, Malcolm Atkinson and Suad Alagic. The work was supported by ESPRIT III Basic Research Action 6309 — FIDE2 and EPSRC Grant GR/J67611. Richard Connor is supported by EPSRC Advanced Fellowship B/94/AF/1921.

References

[ABC+83] Atkinson, M.P., Bailey, P.J., Chisholm, K.J., Cockshott, W.P. & Morrison, R. "An Approach to Persistent Programming". Computer Journal 26, 4 (1983) pp 360-365.

[ACP+91] Abadi, M., Cardelli, L., Pierce, B.C. & Plotkin, G. "Dynamic Typing in a Statically Typed Language". ACM Transactions on Programming Languages and Systems 13, 2 (1991) pp 237-268.

[Ala90] Alagic, S. "Persistent Metaobjects". In **Implementing Persistent Object Bases**, Dearle, A., Shaw, G.M. & Zdonik, S.B. (ed), Morgan Kaufmann (1990) pp 27-38.

[AM85] Atkinson, M.P. & Morrison, R. "Procedures as Persistent Data Objects". ACM Transactions on Programming Languages and Systems 7, 4 (1985) pp 539-559.

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

[BO91] Buneman, P. & Ohori, A. "A Type System that Reconciles Classes and Extents". In **Database Programming Languages: Bulk Types and Persistent Data**, Kanelakis, P. & Schmidt, J.W. (ed), Morgan Kaufmann, Proc. 3rd International Workshop on Database Programming Languages, Nafplion, Greece (1991) pp 191-202.

[CCK+94] Connor, R.C.H., Cutts, Q.I., Kirby, G.N.C. & Morrison, R. "Using Persistence Technology to Control Schema Evolution". In Proc. 9th ACM Symposium on Applied Computing, Phoenix, Arizona (1994) pp 441-446.

[CM88] Cardelli, L. & MacQueen, D.B. "Persistence and Type Abstraction". In **Data Types and Persistence**, Atkinson, M.P., Buneman, O.P. & Morrison, R. (ed), Springer-Verlag (1988) pp 31-42.

[Con90] Connor, R.C.H. "Types and Polymorphism in Persistent Programming Systems". Ph.D. Thesis, University of St Andrews (1990).

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

[DM88] Danvy, O. & Malmkjaer, K. "Intensions and Extensions in a Reflective Tower". In Proc. ACM Symposium on LISP and Functional Programming, New York (1988) pp 327-341.

[FD93] Farkas, A. & Dearle, A. "Octopus: A Reflective Language Mechanism for Object Manipulation". In **Database Programming Languages**, Beeri, C., Ohori, A. & Shasha, D.E. (ed), Springer-Verlag, Proc. 4th International Conference on Database Programming Languages, New York City (1993) pp 50-64.

[FW84] Friedman, D.P. & Wand, M. “Reification: Reflection Without Metaphysics”. In Proc. ACM Symposium on Lisp and Functional Programming (1984) pp 348-355.

[FW86] Friedman, D.P. & Wand, M. “The Mystery of the Tower Revealed: A Non-reflective Description of the Reflective Tower”. In Proc. ACM Symposium on LISP and Functional Programming, New York (1986) pp 298-307.

[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.

[GR83] Goldberg, A. & Robson, D. **Smalltalk-80: The Language and its Implementation**. Addison Wesley, Reading, Massachusetts (1983).

[HS94] Hook, J. & Sheard, T. “A Semantics of Compile-time Reflection”. Oregon Graduate Institute of Science & Technology (1994).

[KCM94] Kirby, G.N.C., Connor, R.C.H. & Morrison, R. “START: A Linguistic Reflection Tool Using Hyper-Program Technology”. In **Persistent Object Systems, Tarascon 1994**, Atkinson, M.P., Maier, D. & Benzaken, V. (ed), Springer-Verlag, Proc. 6th International Workshop on Persistent Object Systems, Tarascon, France (1994) pp 355-373.

[LRN86] Laird, J., Rosenbloom, P. & Newell, A. “Chunking in SOAR: The Anatomy of a General Learning Mechanism”. Machine Intelligence 1, 1 (1986).

[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).

[Mae87] Maes, P. “Concepts and Experiments in Computational Reflection”. In Proc. OOPSLA’87, Orlando, Florida (1987) pp 147-155.

[MBC+94] Morrison, R., Brown, A.L., Connor, R.C.H., Cutts, Q.I., Dearle, A., Kirby, G.N.C. & Munro, D.S. “The Napier88 Reference Manual (Release 2.0)”. University of St Andrews Technical Report CS/94/8 (1994).

[Mul92] Muller, R. “M-LISP: A Representation-Independent Dialect of LISP with Reduction Semantics”. ACM Transactions on Programming Languages and Systems 14, 4 (1992) pp 589-616.

[PL89] Pfenning, F. & Lee, P. “LEAP: A Language with Eval and Polymorphism”. In **Lecture Notes in Computer Science 359**, Díaz, J. & Orejas, F. (ed), Springer-Verlag, Proc. 3rd International Joint Conference on Theory and Practice of Software Development, Barcelona, Spain (1989) pp 345-359.

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

[SH94] Sheard, T. & Hook, J. “Type Safe Meta-Programming”. Oregon Graduate Institute (1994).

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

[Smi82] Smith, B.C. “Reflection and Semantics in a Procedural Language”. Ph.D. Thesis, MIT (1982).

[Smi84] Smith, B. “Reflection and Semantics in LISP”. In Proc. 11th ACM Symposium on Principles of Programming Languages, New York (1984) pp 23-35.

[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).

Appendix: Anatomy of Reflection

A.1 Terminology

Let a program be written in a language *L* which defines a domain of values *Val*. Execution is performed by a meta-function *eval* which takes a program and produces a resulting value:

$$eval : L \rightarrow Val$$

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. Let there also 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, or the set of well-formed syntax trees. 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*.

The language constructs which cause linguistic and behavioural reflective computation are denoted by L_{LR} and L_{BR}, subsets of *L* together comprising the reflective sub-language:

$$(L_{LR} \cup L_{BR}) \subset L$$
$$Val_L \subseteq Val$$

The raising operation which transforms a program representation in the value domain Val_L into a program in *L* is represented by another meta-function, *raise*:

$$raise : Val_L \rightarrow L$$

A.2 Linguistic Reflection

Linguistic reflection involves a program generating new program fragments which are integrated into its own execution. So the executing program produces a value in Val_L which *represents* a new program fragment. That value is then transformed into an actual program fragment which is incorporated into the executing program. This pattern of computation can be represented by the part of the *eval* function shown below:

```
meta-fun eval (e : L) → Val      ! e is a program fragment in L which produces a result in Val

        case e of
        ...
        ConstructOfL_LR => eval (raise (eval (e)))
        ...
```

where the ellipses cover all the non-reflective evaluations. The construct *eval (raise (eval (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 inner application of *eval* causes the generation of a program fragment representation in Val_L, which is then transformed into a real program fragment by *raise* and then evaluated by a second application of *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 $Val_{L\text{-}L_{LR}}$, the set of values that represent non-reflective program constructs. Syntactic correctness requires that the result of *eval (e)* is in Val_L for all reflective expressions. A generated expression must also be internally type consistent and typed correctly for its context.

A.3 Behavioural Reflection

With behavioural reflection a reflective program may change the definition of *eval*. This is harder to describe in the framework outlined above but it still involves a double evaluation.

In order for an executing program to change *eval*, the meta-function *change_eval* is required. This takes an existing evaluation function and a representation in the value domain of the required change, and returns a new evaluation function.

```
change_eval : ((L → Val) × Val) → (L → Val)
```

The double evaluation can be seen when considering the evaluation of a behavioural reflective construct together with the remainder of the program. Evaluation of the remainder takes place under control of the modified evaluation function. This is shown in the part of the eval function shown below:

```
meta-fun eval (e : L) → Val

        case e of
        ...
        (e1; e2) =>
              case e1 of
              ...
              ConstructOfLBR =>  let eval' = change_eval (eval, eval (e1))
                                 in eval' (e2)
              ...
        ...
```

The first evaluation takes place when a construct in L_{BR} is encountered, and evaluated to produce a value representing the required change in the interpreter. A new evaluation function *eval'* is then formed, based on the existing *eval* and the change representation. The second evaluation then involves the execution of the remainder of the program using the new evaluation function.

Note that the interpreter change representation generated by *eval* (e_1) may comprise updated interpreter data structures, new interpreter code or both, depending on the definition of *change_eval*. In the case of generating new interpreter code essentially the same process as for linguistic reflection is taking place, except that the generated code may be in a different language from the code which generates it.

A.4 Inputs

Section 3.4 described several operators which are used to obtain information about the current state of the system, on which system changes may be based. In the case of behavioural reflection a meta-function *reify* takes an existing evaluation function and returns a representation of its state in the value domain. Information may thus pass between program and interpreter in both directions.

```
reify : (L → Val) → Val
```