START: A Linguistic Reflection Tool Using Hyper-Program Technology

G.N.C. Kirby, R.C.H. Connor and R. Morrison

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

Abstract

The mechanism of linguistic reflection allows a programming system to generate new program fragments and incorporate them into the system. Although this ability has important applications in persistent systems, its use has been limited by the difficulty of writing reflective programs. This paper analyses the reasons for this difficulty and describes *START*, a hyper-text based tool for reflection which has been implemented in the Napier88 hyper-programming environment.

START supports the definition of structured program generators which may contain embedded direct links to other generators and to values, locations and types extant in the persistent environment. The benefits are greater ease of understanding through a clean separation of generator components, and a safer and more efficient mechanism for communication between generator and generated code.

1 Introduction

Linguistic reflection gives a programming system the ability to generate new program fragments and incorporate them into the ongoing computation. This has several applications in a persistent environment including supporting safe evolution of long-lived programs and data, and specifying highly generic programs that may be reused in many contexts. In strongly typed systems the linguistic reflection process includes checking of the generated program fragments to ensure type safety.

The details of linguistic reflection have been described by a number of authors [1-11]. A short overview of the technique and its applications in persistent systems will be given in order to motivate the main part of the paper, which analyses the problems involved in writing reflective generator programs and describes *START* (St Andrews Reflection Tool), a tool which aids in this process. One feature of START is its hyper-text user interface which allows conceptually different parts of a generator definition to be distinguished easily. The other main feature is the use of hyper-program representations [8, 12] to provide an additional binding mechanism. This gives increased safety and efficiency, and avoids the need to flatten type representations to textual form.

Forms of linguistic reflection are provided in Lisp [13], Scheme [14] and POP-2 [15]. This paper, however, will consider only strongly typed languages, relevant examples of which include PS-algol [16], Napier88 [17] and TRPL [18]. Linguistic reflection in these languages involves the execution of generator procedures which produce as their results representations of program fragments in the corresponding language. These program fragments are incorporated into the application after the appropriate validity checks.

Two varieties of type-safe linguistic reflection can be identified; these vary as to the time at which generator execution takes place. With compile time linguistic reflection, supported in TRPL, the generators are evaluated during the course of program compilation and the new code produced is incorporated into the program being compiled. This technique could be viewed as a sophisticated form of macro expansion, where the language used to evaluate the macro is the same as the programming language itself. With run time linguistic reflection, supported in PS-algol and Napier88, the generators are evaluated during program execution and the new code produced is compiled and executed in the same context. Both forms of linguistic reflection have the effect of blurring the distinction between compile time and run time.

One application of linguistic reflection is in supporting evolution in strongly typed persistent systems. The inevitable changes to meta-data in long-lived systems give rise to the problem of consistently changing all the affected programs and data. Given some mechanism for locating the relevant programs and data, linguistic reflection can be used to introduce transformed versions in a controlled manner. One approach to limited automation of this process is described in [19].

Another application is in providing highly generic programs. The reuse of existing software reduces development and maintenance costs; the more generic, or widely applicable, a software component is, the more likely it will be reused. Polymorphism [20, 21] is one powerful mechanism for genericity. There exist, however, some generic computations which are hard or impossible to express using parametric or inclusion polymorphism. The difficulty lies in the fact that the course of such a computation depends on details of the types of the input parameters, while parametric and inclusion polymorphism by their nature abstract such details away. A well used example is that of a strongly typed natural join function [22, 23], where the algorithm and the type of the result relation depend on the types of the input relations. This can be implemented using linguistic reflection, by defining a generator procedure which accepts representations of the types of the relations to be joined and generates the representation of a procedure to perform the join for those types.

Thus linguistic reflection provides a rich form of ad-hoc polymorphism [24, 25], with which generic yet type-dependent operators can be defined over wide ranges of types [1, 3, 23]. This mechanism has some similarities with that of 4GL systems, where generic application descriptions are automatically tailored to specific instances. In this case the structure of the type system and the language defines the primitives over which the generators are written.. It is somewhat ironic that the type system itself, which is often seen as restricting the expressibility of the language, can be used as the basis of very flexible and high level generation mechanisms. The technique is particularly suited to persistent systems since the persistent environment may be used as a cache to store executable versions of generated procedures.

This means that the generator need not be executed more than once for given parameters.

The next section discusses the problems encountered in writing reflective generator procedures for uses such as those outlined above. The remainder of the paper then introduces START, a generator editing tool which combines a hyper-text interface with hyper-program technology in an effort to tackle some of these problems.

2 Reflective Generators

This section examines the structure and role of the generator in the reflection process.

2.1 The Reflection Process

The process of linguistic reflection has been characterised in [10] as follows:

Programs in a language L manipulate a domain of values Val. This domain differs between 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. 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. In the known implementations of linguistic reflection Val_L is the set of character strings containing syntactically correct expressions in the cases of PS-algol and Napier88, and the set of valid parse trees in the case of TRPL.

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. An evaluation of an expression in L_R invokes a generator, a program that produces another program. Generators are written in a subset of the language L denoted by L_{Gen}. L_{Gen} may include all of L but the programs written in L_{Gen} must produce results in Val_L. Linguistic reflection thus involves the following steps during evaluation of a program in L:

- a construct in L_R is encountered;
- this causes a generator, written in L_{Gen}, to be evaluated;
- the result, a value in Val_L, represents a new construct in L;
- the new construct is checked and, if valid, executed.

Note that this description does not specify whether the generator evaluation takes place during compilation or execution, and is valid for either case.

2.2 Generator Structure

Each generator contains a result expression that when evaluated produces the generated program fragment. The code in this expression itself represents code, thus it belongs to the subset of L containing sentences that, when evaluated, produce values in Val_L. This subset will be denoted by L_L. The set L_L can be partitioned into two subsets, $L_{L_{Const}}$ and $L_{L_{Var}}$. The former, $L_{L_{Const}}$, contains those sentences that produce the same values in Val_L for all executions, while the latter, $L_{L_{Var}}$, contains sentences that may evaluate to different values on different executions. This is illustrated in Figure 1:

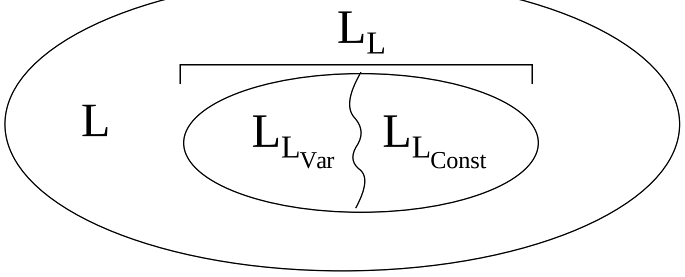


Figure 1: Subset relationships between code categories

Some examples of sentences in these sets are shown below for a language in which members of Val_L are strings, *makeCode()* denotes a call to a generator *makeCode*, and ++ denotes concatenation:

`a := a + 1`	$\in$	L
`"a := a + 1"`	$\in$	$L_{L_{Const}}$
`makeCode()`	$\in$	$L_{L_{Var}}$
`"a := " ++ makeCode()`	$\in$	$L_{L_{Var}}$

Note that the last example is itself a composition of two code fragments, one a member of $L_{L_{Const}}$ and the other a member of $L_{L_{Var}}$. In general a generator body contains a section of code in L, here termed the *prelude*, followed by a section of code in $L_{L_{Var}}$ that defines the resulting generated fragment, here termed the *result definition*. This is illustrated in Figure 2. The purpose of the prelude is to set up an environment in which the result definition is evaluated.

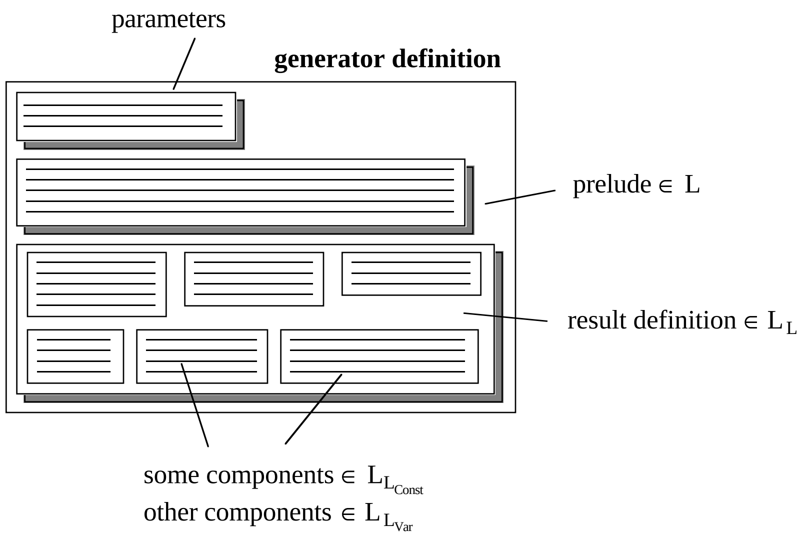

parameters
generator definition
prelude ∈ L
result definition ∈ L_L
some components ∈ $L_{L_{Const}}$
other components ∈ $L_{L_{Var}}$

Figure 2: Structure of a generator

In simple cases the generator body may contain only the result definition, and that code may lie in $L_{L_{Const}}$ rather than $L_{L_{Var}}$. In the general case the execution of a generator involves the evaluation of the prelude and those parts of the result definition that lie in $L_{L_{Var}}$, i.e., the variable parts. The parts in $L_{L_{Const}}$ do not need to be evaluated as they are constant over all executions of the generator. Typically the evaluation of the prelude affects the program fragments produced in the result definition. The result of the generator is obtained by composing the newly created fragments with the constant parts of the result definition.

2.3 An Example of a Generator

The next section will identify some of the problems involved in programming generators. This section concentrates on an example written in Napier88. Lack of space prevents comparison with other languages and notations. Rather than use a complete but trivial example, the main part of a generator for the (non-trivial) natural join problem is shown in Figure 3. To save space, various auxiliary procedure definitions are omitted. The form of the generator is a procedure which takes as parameters the representations of two structure types. These define the types of the relations to be joined: each relation is modelled as a set of structures. The output of the generator is the representation of a procedure to perform natural join over pairs of relations of these types.

The generator procedure, *defineJoin*, takes as parameters the type representations *type1* and *type2*, and returns a code representation as a string. Before the procedure declaration two strings are defined. *defineTypes* represents the definition of the type constructors *Comparison* and *Set* used within the generated code.

getPredefined represents code to bind to the existing procedures *mkEmptySet* and *mkComparison* in the persistent environment.

Within the body of *defineJoin*, a representation of the type of the join result, *resultType*, is computed from the input type representations by an auxiliary procedure *joinResultType*. Details of error reporting at this and other stages have been omitted for simplicity—for example the input types might not have compatible common fields on which to join, in which case the execution of the generator would fail. The generated code is then obtained by concatenating the type definitions and binding code with the definition of the join procedure itself. The general form of the join procedure is fixed by the string literals which are concatenated with the type dependent code fragments obtained by computing over the type representations.

The definitions of the auxiliary procedures *writeType*, *defineOneJoin* and *defineCompareResult* are omitted, as are the definitions of the types against which the generator itself is compiled and parts of the type definition string *defineTypes*. Details of how an executable version of the generated code is obtained are also not shown.

```
let defineTypes =
     "type Comparison[ T ] is structure( equal, lessThan :
                                    proc( T, T -> bool )

     rec type Set[ T ] is structure(
          insert :       proc( T -> Set[ T ] );
          difference :   proc( Set[ T ] -> Set[ T ] ) );
          …
          size :         proc( -> int ) )"

let getPredefined =
     "use PS() with
          mkEmptySet :   proc[ T ]( Comparison[ T ] -> Set[ T ] );
          mkComparison : proc[ T ]( proc( T, T -> bool ) ->
                               Comparison[ T ] ) in"

let defineJoin = proc( type1, type2 : TypeRep -> string )
begin
     let resultType = joinResultType( type1, type2 )

     defineTypes ++ "'n" ++ getPredefined ++ "'n" ++

     "begin
          rec let join = proc(
               rel1 : Set[ " ++ writeType( type1 ) ++" ] ;
               rel2 : Set[ " ++ writeType( type2 ) ++ " ] ->
               Set[ " ++ writeType( resultType ) ++ " ] )

          project rel1 as first onto
          populated :
          begin
               let joinOne = " ++
                    defineOneJoin( type1, type2, resultType ) ++
                    "( first( choose )(), rel2 )
               let joinOthers = join( first( rest )(), rel2 )

               project joinOne as firstJoined onto
               populated : firstJoined( union )( joinOthers )
               default :   joinOthers
          end
          default : mkEmptySet[ " ++ writeType( resultType ) ++
                    " ]( " ++
                    defineCompareResult( resultType ) ++ " )

          join
     end"
end
```

Figure 3: Example generator definition in Napier88

2.4 Why Programming Generators is Hard

Programmers writing generators in various languages have reported that generators are considerably more difficult to write and understand than conventional programs [3, 5, 7]. Some possible reasons for this are:

- A generator may describe a large class of programs rather than a single one. Although a conventional program may have many different possible execution paths, its structure is fixed. The structure of different programs produced by a single generator may differ widely. To understand a generator the reader must determine the features common to all programs produced by it, and understand how the parts that vary among the resulting programs relate to the input parameters to the generator. In the example the overall structure of the *join* procedure is constant while the details of the result type and the *joinOne* procedure are dependent on the input types.

- The programmer must perform a mental mapping between sentences in L and their representations in L_L. This is trivial for the example of `"rec let join = …"` but less so for `writeType( type1 )`.

- The constant and variable parts of the result definition appear different even though they both represent parts of the resulting program fragment. By the end of the generator execution they are integrated seamlessly but this is not apparent from inspection of the generator source code. This is seen in the same line as the previous example: …`Set[ " ++ writeType( type1 ) ++ " ]` …

- Code in different parts of the generator is evaluated at different times. During the execution of the generator, the prelude and those parts of the result definition in $L_{L_{Var}}$ are evaluated. Later during the reflection process the new code produced by the generator, comprising the $L_{L_{Const}}$ parts of the result definition composed with the fragments produced by the evaluation of the $L_{L_{Var}}$ parts, is evaluated. Thus adjacent parts of the result definition may be evaluated at different times and in different environments. In the example, execution of the generator involves evaluation of `joinResultType( type1,type2 )` in the prelude and `writeType( type1 )` in $L_{L_{Var}}$. Later the code `rel1 : Set[`, composed with the result of `writeType( type1 )`, is evaluated.

- Communication of data between evaluation environments is unwieldy. It may be that a value computed during the execution of a generator prelude is required in the generated code. This involves either generating code to create a copy of the value, or placing a reference to the value in some storage accessible from both evaluation environments and then generating code to retrieve it. The first option involves an execution overhead and precludes communication of the identity of the value. The latter option also involves an execution overhead, and in addition there may be a risk of the value being removed from the environment before the generated code is executed. This second mechanism is used in the example: the generator produces code to retrieve the procedures *mkEmptySet* and *mkComparison* from the persistent store.

- There may be a lot of syntactic noise in the code of a generator, particularly where constant and variable parts of the result definition are composed together. The example contains many concatenation and quote symbols.

- In languages where Val_L comprises string expressions, manipulation of program representations is unwieldy. One example of such a manipulation is determining the result type of a procedure from its representation. This is non-trivial when the representation is a string, since it involves parsing the string. A parse tree representation presents less of a problem, since the representation of the result type may be a component which can be accessed directly. This problem does not arise in the example since the generator is parameterised only by type representations.

These factors suggest several desirable features for any generator notation:

- Syntactic noise should be cut to a minimum.

- It should be easy to identify which parts of the result definition are constant, in $L_{L_{Const}}$, and which parts are variable, in $L_{L_{Var}}$.

- It should be possible to use different code representation forms in the constant and variable parts of the result definition. A textual form, such as strings, is easy to read in the constant parts since it gives a simple mapping between $L_{L_{Const}}$ and L. An abstract syntax form may be more suitable for the variable parts as it facilitates the expression of code representation manipulations.

- There should be a simple mechanism enabling the generated code to refer to values in scope in the generator prelude.

- Supplementary tools to aid understanding of generators should be provided. For example, a tool could display the resulting code produced by a generator for given inputs. This could help in understanding the relationships between generator parameters and the code fragments produced by $L_{L_{Var}}$ code.

3 START: The Generator Tool

3.1 Design Criteria

This section describes START, a tool designed to aid writing generators in Napier88. It is implemented within the Napier88 hyper-programming environment [26]. The principal ideas are:

1) to display the variable parts within a generator result definition as hyper-text links; and

2) to allow generated code to contain hyper-program links.

The first point addresses the problems of excessive syntactic noise within result definitions and distinguishing between constant and variable parts, by presenting a clean user interface. The second addresses the problem of communicating between the separate evaluation environments of generator and generated code, by allowing generators to produce *hyper-program* representations. A hyper-program is a program which contains both text and embedded direct links to existing values in the persistent environment [27, 8, 12]. A generated hyper-program may thus contain a direct link to a value in scope within the generator.

A window-based generator editor is used to allow the programmer to view a generator at various degrees of detail. At the most abstract level the programmer sees only the prelude code and the fixed parts of the result definition. The positions of the variable parts are indicated by light-buttons embedded in the code. This level of detail shows the programmer the main structure of the generated result, while abstracting over the variations that depend on the particular specialisation. To examine the details of the variations the programmer may press a button and view the corresponding code in a separate window. This use of windows allows much of the noisy syntax involved in combining parts of the result definition to be omitted, making it easier to read.

The usefulness of this ability to separate constant and variable parts of the result definition depends on the style in which generators are written. It is always possible to write generators in such a way that the entire result definition is variable; however the assumption is made that programmers will choose to write constant definitions for the generated code fragments that are common to all inputs.

Two additional design criteria were to give uniformity between generators and the variable $L_{L_{Var}}$ parts of result definitions—which may themselves be regarded as generators—and to allow arbitrary nesting of generators.

3.2 Generator Model Details

START supports a model in which each generator has two separate components: a *prelude* and a *result definition*. The prelude is a procedure that processes the parameters input to the generator, while the result definition is a variant that may be either a fragment of hyper-program source code or a procedure that produces such a fragment. These source code fragments may contain place-holders corresponding to further generators. Thus each $L_{L_{Var}}$ part of the result definition is itself represented by another generator.

To evaluate a generator its prelude is executed with the generator parameters passed to it. If the result definition is a procedure then it is executed in turn, with the results produced by the prelude passed to it. The result of this procedure, or the result definition itself in the other case, is a source code fragment which may contain place-holders for other generators. If so these generators are themselves evaluated and the resulting code fragments incorporated into the result. This process is continued until a source code representation without generator place-holders is obtained. A generator could be recursive, containing a place-holder to itself within its result definition, although the practical usefulness of this capability is questionable.

The ability of a generator to produce hyper-program source code containing links to data items means that generated code can refer directly to values constructed by the generator, or to values in the persistent store at the time of generator execution.

Figure 4 shows some of the Napier88 type definitions describing this model. Type *HyperText*, omitted to save space, represents fragments of hyper-text and is parameterised by the type of the embedded links. An instance of *HyperText* contains a text string and a list of <*link value, text position*> pairs. The definition of *Binding* is also omitted: this is used to denote hyper-program values, and instances may represent values, locations or types.

Generators are represented by instances of the structure type *Generator*. The first component is a procedure, *prelude*, that takes a Napier88 environment as its parameter and returns another environment. These environments contain the generator parameters and prelude results respectively.

The second component of a generator, the result definition, is an instance of the variant type *GeneratorResult*. Its value may be either an instance of type *GeneratorSource* or a procedure that takes an environment and produces a *GeneratorSource*. In the first case the result definition is a literal code fragment while in the second case it is a procedure that must be executed to produce a code fragment. In both cases the code fragment may contain hyper-program links and links to sub-generators, both represented by instances of the variant type *BindingOrGenerator*.

```
! Hyper-text with embedded references to instances of type Link
type HyperText[ Link ] is …

! Value, location or type
type Binding is …

! Generator containing a prelude and a result definition
rec type Generator is structure( prelude : proc( env -> env ) ;
                                 resultDefn : GeneratorResult )

! Result definition can be fixed or dependent on inputs
& GeneratorResult is variant(
           literal : GeneratorSource ;
           expression : proc( env -> GeneratorSource ) )

! Generated code is text with links to Bindings and
! other Generators
& GeneratorSource is HyperText[ BindingOrGenerator ]

& BindingOrGenerator is variant( binding : Binding ;
                                 generator : Generator )
```

Figure 4: Napier88 description of generator model

Note that the *literal* branch of *GeneratorResult* is redundant so far as expressiveness is concerned: a literal result definition could be expressed as a procedure which ignored its parameters and always produced the same result. However, the presence of this branch enables the generator editor to display a meaningful representation of the result definition.

The generator construction system provides a number of pre-defined types and procedures that may be linked into generators and generated code. The procedures provide set operations and analysis and synthesis of both type representations and source representations. Procedures and types may be linked directly into generator definitions using hyper-program construction tools [8].

3.3 The START Generator Editor

3.3.1 User Interface

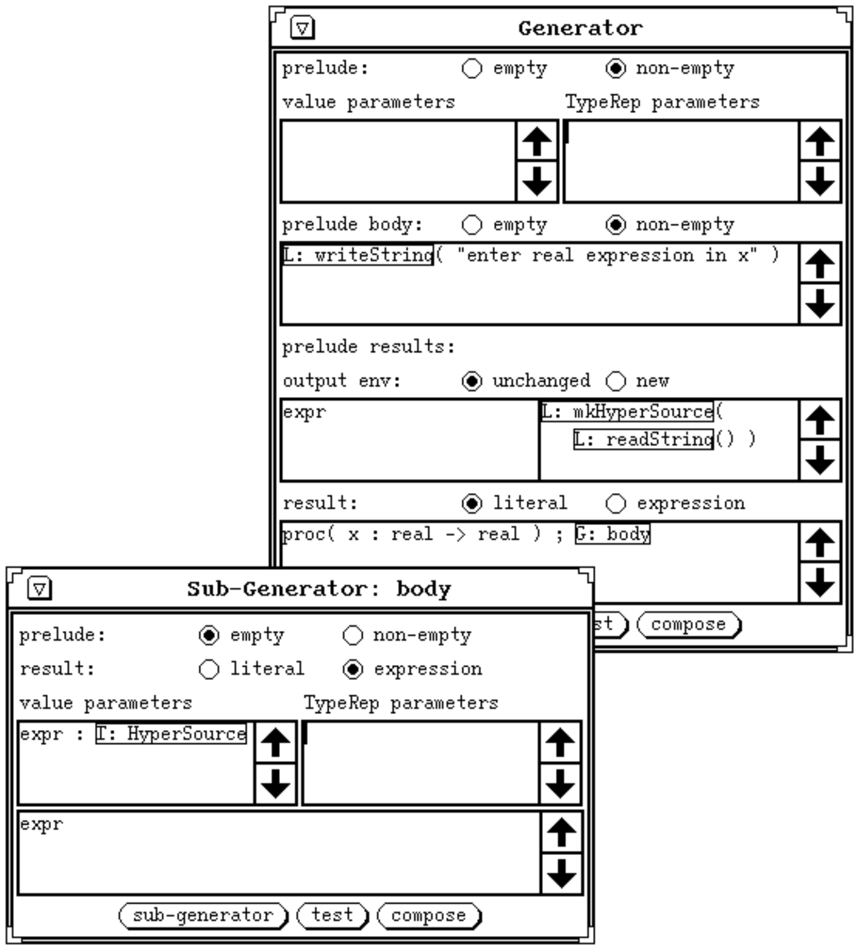
The generator editor provides a form window with a number of fields corresponding to the various components of a generator. When a particular component is not present, or empty, the field is not displayed. To create a generator the programmer fills in the fields as appropriate. All fields may contain hyper-program links to values, locations and types identified using an external browsing tool [26]. These are distinguished by the prefixes *V:*, *L:* and *T:* respectively shown on the link labels.

The first set of fields contains the prelude definition. One field contains the names and types of the prelude parameters. As a short cut for the programmer a separate field is provided for type representation parameters, since these are expected to be particularly common. Here the programmer need only specify the names of the parameters, since they are assumed to be of type *TypeRep*.

Other fields contain the body of the prelude code and the outputs from the prelude which will be passed on to the variable parts of the result definition. The result definition itself may be an expression, in which case the result varies between evaluations, or a literal, in which case the result is always the same. For an expression the editor provides fields for both general parameters and type representations, as for the prelude parameters. The result body then contains code which when executed will generate a code representation.

For a literal a single field contains the result code. In addition to the normal hyper-program links, a literal result definition may also contain links to sub-generators, corresponding to embedded variable parts of the result. These links are distinguished by the prefix *G:* on their labels. The programmer creates a sub-generator at the current insertion point by pressing the **sub-generator** button. This inserts a link and invokes a new generator editing window. The nesting of generators may be continued to any depth. If the insertion point lies within an expression definition when the sub-generator button is pressed a new generator is created but the link to it is a hyper-program link and has a *V:* prefix. Thus a result definition is represented as a form of hyper-text comprising a graph of linked generators and sub-generators.

Figure 5 shows an example of a generator which prompts the user for a string, converts it to a source representation, and incorporates it into the generated representation of a procedure mapping reals to reals. The names on the buttons denoting the links are present to aid understanding but do not affect the semantics of the code fragments: the programmer could change the button names if desired. The initial names are supplied by the browsing tool with which the links are created.



Generator

prelude: empty [✓ non-empty]

value parameters

TypeRep parameters

prelude body: empty [✓ non-empty]

```
L: writeString( "enter real expression in x" )
```

prelude results:

output env: [✓ unchanged] new

```
expr          L: mkHyperSource(
                 L: readString() )
```

result: [✓ literal] expression

```
proc( x : real -> real ) ; G: body
```

Sub-Generator: body

prelude: [✓ empty] non-empty

result: literal [✓ expression]

value parameters

```
expr : T: HyperSource
```

TypeRep parameters

```
expr
```

Figure 5: A simple generator

When all details have been filled in the programmer can create an instance of type *Generator* by pressing the **compose** button. The generator is passed to the browsing system where it may be evaluated for particular parameters, stored or manipulated in any other manner.

3.3.2 Example

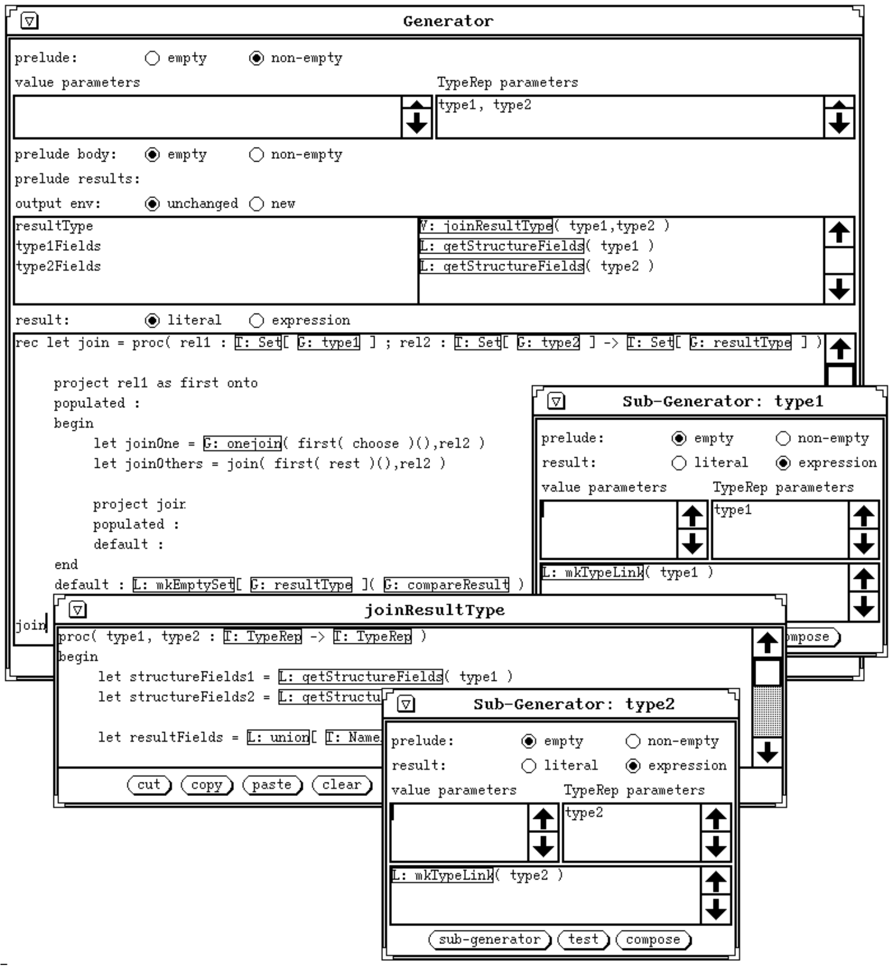
Figure 6 shows a generator editor used for the natural join example. The large window contains the generator for *join* itself, while the others contain sub-generators *type1* and *type2* and an auxiliary procedure *joinResultType*.

The prelude takes two type representation parameters of type *TypeRep*, representing the tuple types of the input relations. For brevity the checks to ensure that they represent structure types are not shown. The prelude enriches the input environment with three new values: *resultType*, which represents the tuple type of the result relation, and *type1Fields* and *type2Fields* which are sets containing the field information for the two input types. The value *resultType* is obtained by calling a procedure *joinResultType*, a direct link to which is contained in the prelude code. The structure field information is obtained using the pre-defined procedure *getStructureFields* which returns a set of <*name, type representation*> pairs.

The result definition is a literal and contains the definition of the generated *join* procedure. The code contains a number of links to sub-generators. These are used to define, in order of appearance, the first input type, the second input type, the result type (twice), a procedure to perform a join between a single tuple and a relation, and a procedure to compare instances of the result type.

The result definition also contains a direct link to the pre-defined procedure *mkEmptySet*. This contrasts with the Napier88 solution in which the result definition contains code to link to the procedure in the persistent store. The direct link notation is both more concise and more secure, as there is no danger of access to the procedure being removed between the times of evaluation of the generator and execution of the generated result.

Figure 6 also shows the definition of the procedure *joinResultType* which computes a representation of the result type. This is achieved by constructing the union of the two sets containing the names and types of the fields of the input types and using the pre-defined procedure *mkStructureType* to create a type representation. The other windows show the sub-generators *type1* and *type2*. These generators use the pre-defined procedure *mkTypeLink* to obtain links to the types represented by the input type representations.



prelude: empty [✓ non-empty]
value parameters
TypeRep parameters: type1, type2
prelude body: [✓ empty] non-empty
prelude results:
output env: [✓ unchanged] new

resultType	V: joinResultType(type1,type2)
type1Fields	L: getStructureFields(type1)
type2Fields	L: getStructureFields(type2)

result: [✓ literal] expression

```
rec let join = proc( rel1 : T: Set[ G: type1 ] ; rel2 : T: Set[ G: type2 ] -> T: Set[ G: resultType ] )

    project rel1 as first onto
    populated :
    begin
        let joinOne = G: onejoin( first( choose )(),rel2 )
        let joinOthers = join( first( rest )(),rel2 )

        project joir
        populated :
        default :
    end
    default : L: mkEmptySet[ G: resultType ]( G: compareResult )

join
```

prelude: [✓ empty] non-empty
result: literal [✓ expression]
value parameters
TypeRep parameters: type1

```
L: mkTypeLink( type1 )
```

```
proc( type1, type2 : T: TypeRep -> T: TypeRep )
begin
    let structureFields1 = L: getStructureFields( type1 )
    let structureFields2 = L: getStructu

    let resultFields = L: union[ T: Name
```

prelude: [✓ empty] non-empty
result: literal [✓ expression]
value parameters
TypeRep parameters: type2

```
L: mkTypeLink( type2 )
```

Figure 6: Generators *join*, *type1* and *type2*

The point of having a distinguished literal branch in the result definition type, rather than representing all result definitions as procedures, should now be apparent. It means that the fixed parts of the result definition can be displayed in the generator editor without having to evaluate the generator against any particular parameters.

The combination of the START tool with the hyper-programming environment gives the following advantages over the pure Napier88 solution of Figure 3:

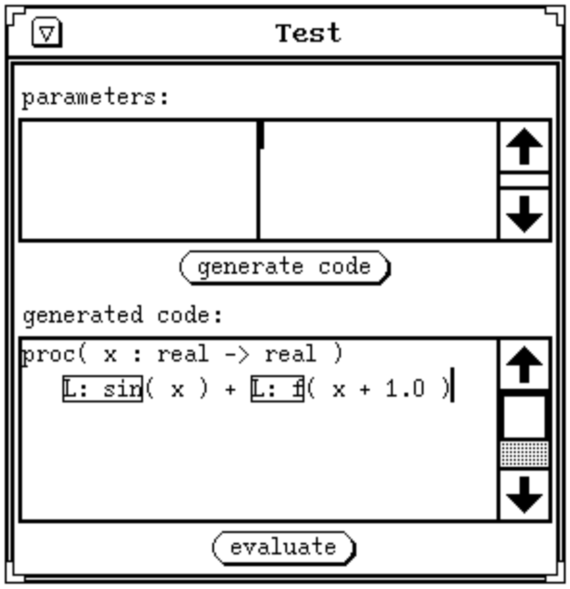
- It may be easier to understand the general form of the generated code, that of a procedure which takes two sets as parameters and returns a third.
- Less code is generated since type definitions and specifications of how to access values in the persistent environment are replaced by hyper-program links.

- There is no need to flatten a structured type representation to a textual form for inclusion in the generated code, as performed by the procedure *writeType* in Figure 3.

3.4 Testing Generators

The testing facility allows the programmer to test a generator with various inputs. When the **test** button is pressed a new window is displayed, containing a sub-window in which values for the generator parameters may be entered. The programmer can then press the **generate code** button to evaluate the generator with those parameters. If the generator executes successfully the resulting code representation is displayed in the lower sub-window. One possible reason for failure of the generator is that the parameters supplied are not compatible with those expected by the generator: in this case a message to that effect is displayed. When generated successfully, the code may itself be evaluated by pressing the **evaluate** button. If the generated code is well formed it is executed and any resulting value displayed by the browsing system; otherwise messages indicating the errors are displayed.

Figure 7 shows a test window for a generator which takes no parameters. The **generate code** button has been pressed and a procedure definition has been generated. The generated code contains hyper-program links to the existing procedure locations *sin* and *f*. The programmer could now create an instance of the new procedure by pressing **evaluate**.



```
proc( x : real -> real )
   L: sin( x ) + L: f( x + 1.0 )
```

Figure 7: Generator test window

4 Open Problems and Future Work

4.1 Errors

Many problems with this technique remain. One of the principal problems is dealing with errors, which may arise at any of the following points:

- during the execution of a generator;
- during compilation of the generated code produced by a generator; or
- during execution of the generated code.

An exception mechanism supported by the underlying language could be used to handle errors occurring during the execution phases. In the absence of such a mechanism the generator model could be refined to allow a generator to return errors instead of a code fragment. For example, a natural join generator could return a *join not defined* error when the input types had no common fields.

Errors during compilation and execution of the generated code are more problematic for the user, who is unlikely to know or care about the details of the generator. In the absence of such errors the user may even be unaware of the existence of the generated code: the generator could be hidden by an encapsulating procedure which calls the generator and then calls the generated code to produce the required result. However, errors in the generator code itself may result in invalid code being generated. While the resulting compilation errors could be reported to the user these are unlikely to be useful to anyone other than the author of the generator. Similarly errors may arise during the execution of the generated code; the challenge is to be able to report these errors to the user in terms of the user problem domain rather than the domain of execution of the generated code.

It is not clear how far the static checking of generators can be developed. An ideal system would be able to determine from static analysis of a generator whether the generated code would always be syntactically correct. Intuitively this appears undecidable for any non-trivial generator language. It is conceivable however that it would be possible to design a generator language which was sufficiently restricted to allow static checking while retaining enough expressibility to be useful.

4.2 Language Issues

The approach taken with the START tool has been to design a generator model as an add-on to the unchanged Napier88 language. Alternatively, *generator* could be provided as a construct within a language. This would be a more elegant solution but it is not clear what other benefits, if any, would result. Two other possibilities for future work are to allow pattern matching on the structure of type and program representations within generators, and to allow more highly structured program representations to be used in conjunction with string representations. These features are both found in TRPL [18], and in the persistent context might allow manipulation of program representations to be expressed more clearly and succinctly.

4.3 User Interface

Plans for further development in the user interface area include provision of enhanced tools for testing generators, and the use of colour in the generator editor to distinguish the varieties of embedded links.

5 Conclusions

Linguistic reflection has a number of applications in persistent systems, but the difficulty of writing and understanding generators has limited its use. This paper has identified some of the reasons for this and described START, a hyper-text based generator editor tool designed to ease the process. The tool relies heavily on the existing Napier88 hyper-programming environment [26].

One feature of START is its hyper-text user interface which allows conceptually different parts of a generator definition to be distinguished easily. It enables the programmer to concentrate either on the form of the generated code common to all uses or details of type dependent parts as required. The other main feature is the use of hyper-program representations to provide an additional binding mechanism. Where appropriate the code generated may contain embedded direct hyper-program links to values, locations and types in the persistent environment. This results in shorter generated code and increased safety and efficiency. It also avoids the need to flatten type representations to textual form.

The generator editor has been implemented in the Napier88 hyper-programming environment. Although the current editor only supports the development of Napier88 generators, it is believed that the concepts could be applied to any self-supporting persistent programming system.

6 Acknowledgements

We thank Dave Stemple for many stimulating discussions on the topics discussed here. This work was supported by ESPRIT III Basic Research Action 6309 FIDE$_2$ and SERC grant GR/F 02953.

References

1.* Dearle A, Brown AL. Safe Browsing in a Strongly Typed Persistent Environment. Comp. J. 1988; 31,6:540-544

2. Alagic S. Persistent Metaobjects. In: A. Dearle, G. M. Shaw and S. B. Zdonik (ed) Implementing Persistent Object Bases. Morgan Kaufmann, 1990, pp 27-38

3. Cooper RL. On The Utilisation of Persistent Programming Environments. Ph.D. thesis, University of Glasgow, 1990

4. Philbrow PC. 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

5. Sheard T. Automatic Generation and Use of Abstract Structure Operators. ACM ToPLaS 1991; 19,4:531-557

6. Hook J, Kieburtz RB, Sheard T. Generating Programs by Reflection. Oregon Graduate Institute of Science & Technology Report CS/E 92-015, 1992

7.* Kirby GNC. Persistent Programming with Strongly Typed Linguistic Reflection. In: Proc. 25th International Conference on Systems Sciences, Hawaii, 1992, pp 820-831

8.* Kirby GNC. Reflection and Hyper-Programming in Persistent Programming Systems. Ph.D. thesis, University of St Andrews, 1992

9. 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

10.* Stemple D, Stanton RB, Sheard T et al. Type-Safe Linguistic Reflection: A Generator Technology. ESPRIT BRA Project 3070 FIDE Report FIDE/92/49, 1992

11.* Stemple D, Morrison R, Kirby GNC, Connor RCH. Integrating Reflection, Strong Typing and Static Checking. In: Proc. 16th Australian Computer Science Conference, Brisbane, Australia, 1993, pp 83-92

12.* Kirby GNC, Connor RCH, Cutts QI, Dearle A, Farkas AM, Morrison R. Persistent Hyper-Programs. In: A. Albano and R. Morrison (ed) Persistent Object Systems, Proc. 5th International Workshop on Persistent Object Systems, San Miniato, Italy. Springer-Verlag, 1992, pp 86-106

13. McCarthy J, Abrahams PW, Edwards DJ, Hart TP, Levin MI. The Lisp Programmers' Manual. M.I.T. Press, Cambridge, Massachusetts, 1962

14. Rees J, Clinger W. Revised Report on the Algorithmic Language Scheme. ACM SIGPLAN Notices 1986; 21,12:37-43

15. Burstall RM, Collins JS, Popplestone RJ. Programming in POP-2. Edinburgh University Press, Edinburgh, Scotland, 1971

16. PS-algol Reference Manual, 4th edition. Universities of Glasgow and St Andrews Report PPRR-12-88, 1988

17.* Morrison R, Brown AL, Connor RCH et al. The Napier88 Reference Manual (Release 2.0). University of St Andrews Report CS/93/15, 1993

18. Sheard T. A user's Guide to TRPL: A Compile-time Reflective Programming Language. COINS, University of Massachusetts Report 90-109, 1990

19.* Connor RCH, Cutts QI, Kirby GNC, Morrison R. Using Persistence Technology to Control Schema Evolution. In: Proc. 9th ACM Symposium on Applied Computing, Phoenix, Arizona, 1994, pp 441-446

20. Strachey C. Fundamental Concepts in Programming Languages. Oxford University Press, Oxford, 1967

21. Milner R. A Theory of Type Polymorphism in Programming. Journal of Computer and System Sciences 1978; 17,3:348-375

22. Codd EF. Extending the relational model to capture more meaning. ACM ToDS 1979; 4,4:397-434

23. Stemple D, Fegaras L, Sheard T, Socorro A. Exceeding the Limits of Polymorphism in Database Programming Languages. In: F. Bancilhon, C. Thanos and D. Tsichritzis (ed) Lecture Notes in Computer Science 416. Springer-Verlag, 1990, pp 269-285

24. Kaes S. Parametric Overloading in Polymorphic Programming languages. In: Lecture Notes in Computer Science 300. Springer-Verlag, 1988, pp 131-144

25. Wadler P, Blott S. How to Make ad-hoc Polymorphism Less ad-hoc. In: Proc. 16th ACM Symposium on Principles of Programming Languages, Austin, Texas, 1989

26.* Kirby GNC, Brown AL, Connor RCH et al. The Napier88 Standard Library Reference Manual Version 2.2. University of St Andrews Report CS/94/7, 1994

27.* Farkas AM, Dearle A, Kirby GNC, Cutts QI, Morrison R, Connor RCH. Persistent Program Construction through Browsing and User Gesture with some Typing. In: A. Albano and R. Morrison (ed) Persistent Object Systems, Proc. 5th International Workshop on Persistent Object Systems, San Miniato, Italy. Springer-Verlag, 1992, pp 376-393

*Available via *ftp* from
`ftp-fide.dcs.st-andrews.ac.uk/pub/persistence.papers`

or via *WWW* from
`http://www-fide.dcs.st-andrews.ac.uk:8080/Publications.html`