Practical Metaprogramming

James R. Cordy
Medha Shukla

Software Technology Laboratory
Department of Computing and Information Science
Queen's University
Kingston, Canada K7L 3N6

External Technical Report 92-342
October 1992

Telephone: (613) 545 6054   Email: cordy@qucis.queensu.ca

Abstract

Metaprogramming is the process of specifying generic software source templates from which classes of software components, or parts thereof, can be automatically instantiated to produce new software components. Metaprograms are specified in an annotated by-example style accessible to ordinary programmers of the source language. Annotations are in the form of Prolog-like predicates that specify the conditions under which different parts of the source template are to be instantiated. Instantiation of a source component is done by specifying facts about the new application in a database, from which the appropriate instance of the metaprogram is automatically inferred using Prolog-style deduction.

This paper describes a practical metaprogramming system being developed as part of the ITRC Software Life Cycle Technology project, which utilizes source transformation to implement all phases of the metaprogramming process. Metaprograms are automatically transformed into TXL (Tree Transformation Language) source transformation tasks that automatically implement the instantiation process using TXL. Examples are shown of the use of metaprogramming in automatically deriving C language glue routines that allow Prolog programs to access the GL graphics library.

1 Background

The potential advantages of metaprogramming, the process of writing programs whose purpose is to generate new source programs, have been described many times in the software reuse literature [Levy 86] [Basset 87] [Cleaveland 88]. In its simplest form, parameterized macros and conditional compilation, metaprogramming is used to enhance productivity in virtually every software project. The stronger form of metaprogramming, deriving large parts of a system's source code directly from the system's design, has not yet been widely accepted, although it offers the potential of even greater gains in productivity and reliability of software products. This has been partly because until recently software system designs were not typically stated using formal languages, and hence were difficult to process automatically; partly because of a lack of a general approach to metaprogramming; and partly because of the large learning curve associated with source manipulation tools that had the potential to help solve the problem.

The first of these problems, the lack of a formal encoding of software design, is rapidly becoming a problem of the past as more and more systems are designed using formal design tools such as VDM [Jones 86], Z [Spivey 88], and GraphLog [Consens, Mendelzon & Ryman 91] to aid in understanding properties of the design.

The second problem, lack of a general approach, is largely solved by the recent availability of practical term rewriting systems such as REFINE [Kotik & Markosian 90], Gentle [Schröer 89] [Vollmer 91] and TXL [Cordy, Halpern & Promislow 88, 91]. The third problem, lack of a metaprogramming notation that is at once powerful, general, and accessible to ordinary programmers, is still largely unsolved.

1.1 An Example Problem

Recognizing that the time was ripe for solving the metaprogramming notation problem in an accessible and practical way, Arthur Ryman posed the problem as a concrete challenge at the 24th meeting of IFIP Working Group 2.4 in 1990 [Ryman 90]. The problem was posed using an example: assume that the entire interface design of a small (about 200 entry procedures) software system, the Silicon Graphics Inc. GL graphics library, is encoded as a knowledge base of Prolog facts (Figure 1). Using a by-example style accessible to programmers of the C and Prolog programming languages, specify metaprograms to automatically generate the source code for the glue necessary to make GL available to Prolog programmers.

```
function (zbuffer).
returns (zbuffer, int).

function (color).
parameter (color, index, int, in).

function (winopen).
parameter (winopen,name,"char*”,in).
returns (winopen, gid, int).
```

Figure 1. Partial Prolog Database for the GL Library Interface (from [Ryman 90]).

Running these metaprograms should yield four programming "artifacts": a set of Prolog external predicate declarations for the GL library routines; a set of C external routine declarations for the C glue routines to attach the GL library routines to Prolog; the set of C glue routines themselves; and a C entry point array to map Prolog predicates to the corresponding C glue routines (Figure 2).

Ryman exhibited examples of what he considered to be ideal metaprograms for this task (Figure 3). The examples exhibited two critical properties: (1) the metaprograms should use the target source language directly by example in order to be writable by ordinary programmers of the language; (2) the metafeatures added to the source language should be as simple and unobtrusive as possible, while allowing great expressiveness and power.

1.2 Existing Tools

Many existing tools can be adapted to attack the by-example metaprogramming task, although none can yet meet all the criteria perfectly.

Many powerful preprocessor systems are described in the literature, and some of them, for example AWK [Aho, Weinberger and Kernighan 79], are powerful enough to handle the generation of the various artifacts from a Prolog database. The difficulty in using these as metaprogramming notations is that: firstly, learning and writing of AWK and similar processor programs is a difficult and error-prone process far removed from the simple specifications envisaged by Ryman; secondly, AWK programs are certainly not in the by-example style; and thirdly, AWK deals only with text-level patterns, making metaprograms that manipulate whole program parts difficult to write.

Prolog [Kowalski 74] can be used to program metaprograms for some languages, including Prolog itself. The search paradigm of Prolog can automatically search the database for facts required to generate instances of the metaprogram, and because Prolog deals with a term space, it does not share the text-only limitations of AWK. However, Prolog has a limited range of syntax that can be conveniently dealt with (certainly it cannot handle C syntax directly as terms) and metaprograms cannot be encoded in a by-example style.

REFINE [Kotik & Markosian 90] and Gentle [Schröer 89] [Vollmer 91] are program transformation systems based on compiler technology. Each of these systems uses a grammar for the language to be manipulated and a set of transformations to apply to the parse tree. Either of them can be used to implement metaprograms by encoding them as transformation functions to be applied to the database parse tree to transform it into a parse tree for the required results. These systems have the advantage that metaprograms can easily manipulate high-level language concepts directly as subtrees, so very complex metaprograms can be written. The disadvantages of these systems are the weakness of their LALR or LL parsing methods, which makes it a tedious and difficult job to create a grammar that can handle both the target language and the database; the necessity of understanding the details of LALR or LL parsing technology in order to use the tools; and the necessity of programming the search of the database parse tree by hand, using a specialized transformation function notation that must be mastered before metaprograms can be written.

TXL [Cordy, Halpern & Promislow 88, 91] is a program transformation system based on Prolog-like transformation rules. Like REFINE and Gentle, TXL uses a context-free grammar for the language to be manipulated. However, unlike the compiler technology-based tools, TXL uses a fully general context free parser that allows the user's manual reference grammar for the target language to be used more or less directly, without conversion to meet LALR or LL grammar restrictions. This avoids the *external C routine declarations for C glue routines*

```
extern int mpro_zbuffer();
extern void mpro_color();
extern gid mpro_winopen();
  . . .
```

C glue routines

```
int mpro_zbuffer(p)
    struct {
        Boolean *b;
    } *p;
{
    Boolean b;
    b = (Boolean) *p -> b;
    zbuffer(b);
    return(0);
}
    . . .
```

C entry point array

```
struct
    {
        char name[];
        int (*addr)();
    } func[] =
{
    {"arc", mpro_arc},
    {"arcf", mpro_arcf},
        . . .
    {"zclear", mpro_zclear},
    {"", 0}
};
```

external Prolog predicate decl’ns

```
external (gl_arc/5, "c$/ARC(+f,+f,+f,+n,+n)$/").
external (gl_arcf/5, "c$/ARCF(+f,+f,+f,+n,+n)$/").
  . . .
external (gl_zclear/0, "c$/ZCLEAR$/").
```

Figure 2. Portions of the Required Source Artifacts of the Metaprograms for the Prolog GL Library Interface (from [Ryman 90]).

The entire set of required artifacts includes approximately 200 routines in each form.

```
struct
    {
        char name[];
        int (*addr);
    } func[] =
{
    {"X", Y},       function (F), upper_lower (X,F),
                        concat("mpro_",F,Y).
    {"", 0}
}
```

Figure 3. Example Metaprogram to Generate an Entry Point Array for the GL Library Interface (from [Ryman 90]).

The left-hand side is a literal example of the C code template for the result; the right-hand side (in italics) specifies the pattern of database facts necessary to instantiate the lines on the left. In this simple case, the line {"X",Y}, is to be instantiated once for every function in the database.

difficulties in creating a grammar, and allows programmers to use the familiar terms of their language user's guide to refer to target language concepts. Like Prolog, TXL can automatically search the database for facts that trigger generation of instances of the metaprogram. TXL metaprograms must however be written in a specialized TXL rule notation which is not directly by-example and takes time to master (see example, Figure 4).

While none of these systems directly support by-example metaprogramming, the last three, REFINE, Gentle and TXL, certainly have transformation engines capable of performing the instantiation task. In the next section, we explore the possibility of implementing by-example metaprogramming using the TXL transformation engine.

2 μ* : A Family of Metalanguages

μ* (pronounced "mew-star") is a family of by-example metaprogramming languages that share a common metanotation and implementation. The philosophy of the family is exactly the ideal: the metaprogramming language for each target language consists of the language itself, augmented with meta-annotations specifying conditions on the database. For example, μC, the metalanguage for C, consists of C program syntax, optionally annotated with meta-annotations. The syntax of meta-annotations is the same across all target languages. In each case, the syntax of the basic metalanguage is the syntax of the language itself, and the syntax of the meta-annotations is the syntax of μ*. The target language can be any programming or specification language with a formal syntax.

```
function createCEParray
   replace [repeat PrologFact_or_C]
       DataBase [repeat PrologFact_or_C]
   construct EmptyEParray
                 [list initializer]
       % empty
   construct Null [initializer]
       {"", 0}
   by
       struct
           {
                char name[];
               int (*addr);
           } func[] =
       {
           EmptyEParray
                [addEntry each DataBase]
                [, Null]
       };
end function

function addEntry FS [PrologFact_or_C]
   deconstruct FS
       'function ( F [id] ).
       RPS [repeat parameterSpec]
       ORS [opt returnsSpec]
       OFS [opt failsSpec]
   construct MPRO [id]
       'mpro_
   construct SF [stringlit]
       ""
   construct NewEntry [initializer]
       { SF [" F] , MPRO [concatId F] }
   replace [list initializer]
       EPlist [list initializer]
   by
       EPlist [, NewEntry]
end function
```

Figure 4. TXL rules for the example metaprogram of Figure 3.

2.1 By-Example Metaprogramming

In μ*, every program written in a target language is a metaprogram unconditionally generating itself. Thus every C progra is automatically a μC program, and every Prolog program is a μProlog program. Syntactically contained program fragments (for example, declarations, statements, and so on) are also in general metaprograms for themselves.

2.2 Notation

The addition of meta-annotations to a metaprogram attaches the metaprogram to the database and makes generation of the annotated parts conditional on the facts in the database. A difficulty in Ryman's suggested metanotation is the problem of specifying the exact section of source to be affected by an annotation. When the annotated section is only one line long, as in Figure 3, there is no problem, but when it is more extensive, there can be an ambiguity. μ* uses explicit bracketing of the affected area by enclosing it in backslashes, followed by the meta-annotation and a double backslash to mark its end, as shown in Figure 5. The backslash is the only symbol reserved by μ*; when used with target languages where an unadorned backslash already has meaning, it can be replaced with any other single symbol.

Because in many cases the intended role of the affected area in the target source is ambiguous (particularly when parts are intentionally underspecified, see 2.4 Refinement and Nested Metaprograms), the role must be given explicitly following the bracketed area, as shown in Figure 5. The role is the name of the intended part of speech in the target language reference syntax (that is, the common name of the entity in the target language, for example *statement* or *declaration* in C) enclosed in square brackets [ ].

2.3 Generative Metaprograms

The μ* annotation language provides two basic operations: *when*, which includes a section of target source conditionally on the provability of a predicate on the database, and *each*, which generates one copy of the section of target source for every solution to a predicate on the database (Figure 6). These two operations can be nested to give complex combinations of conditional generation.

```
const char *strsignal(int n)
{
  static char buf[sizeof("Signal ") + 1 + INT_DIGITS];
  \ if (n >= 0 && n < NSIG && sys_siglist[n] != 0)
      return sys_siglist[n];
  \ [statement]
      when listing
  \\
  sprintf(buf, "Signal %d", n);
  return buf;
}
```

Figure 5. Trivial Example μC Metaprogram.

The `if` *statement enclosed in backslashes is conditionally included in instances of the metaprogram only if* listing *is a fact in the database. The annotation* [statement] *describes the role of the optional ection in the program.*

The predicate following *when* or *each* can be one of the following:

a. a simple Prolog predicate (e.g. `parameter(F,P,T)`)

b. *predicate* **and** *predicate*

c. *predicate* **`or`** *predicate*

d. **`not`** *predicate*

The database is searched for solutions to each annotation predicate. When a solution is found, the metavariables in the predicate are bound to the terms found in the solution in the database. The metavariables can then be instantiated in the target source generated for that solution. Repeated instances of a metavariable in a predicate specify unification in the usual Prolog way, so the predicate *`function(F[id])`* ***`and`*** *`returns(F,int)`* specifies only those identifiers that are functions that return the type *int*.

Since solutions to the predicates are used in the generated target source, they have the same ambiguity of role that source sections can have, and so they are also required to be labelled with their syntactic role in the target language. So, for example, the Prolog predicate *`parameter(F,P,T)`* must appear as *`parameter(F[id],P[id],T[type])`* in a μC annotation, where *`[id]`* specifies the role of C identifier, and *`[type]`* the role of C type. The second and subsequent uses of the same metavariable need not be labelled because their role is already specified.

Figure 6 shows an example using these features to specify a μC metaprogram to generate the external C routine declarations artifact of the GL example. Note that the 'mpro_' prefix is intentionally not yet appended to the generated routine names in this solution in order to demonstrate metaprogram refinement in the next example.

2.4 Refinement and Nested Metaprograms

When programmers write code templates, they often use a pseudo-code style in which descriptive identifiers take the place of sections to be filled in later, as shown in Figure 7. μ* provides this same feature by allowing metavariable identifiers to take the place of any part of a target source fragment enclosed in backslashes, and by allowing later refinement of the role and source text of the metavariable, either as part of the solution to a predicate, or by using a *where* clause.

```
\ extern FType F (); \ [declaration*]
  each function (F [id]) and returns (F, FType [type])
\\
```

Figure 6. Example μC Metaprogram to Generate the External C Routine Declarations Artifact of the GL Example.

The interpretation is that a sequence of declarations is to be generated, one for each solution to the predicate in the database. The predicate specifies that we want each identifier which is both a function identifier and returns some type (i.e., is not void). The second reference to `F` is not labelled with a role because it already has a role (`[id]`, C identifier) specified in its first use.

A *where* clause has the following form:

```
where ident
    \
        source
    \ [role]
        predicate
    \\
```

The *ident* is the name of the metavariable and *role* is its part of speech in the target language. The final `\\` is required only when the *where* clause is embedded in another source fragment.

The *where* clause is a nested metaprogram that generates a target source fragment and binds it to a metavariable for use in other parts of the metaprogram, for example, the main source text. The backslash-bracketed source fragment part of a *where* clause is optional when there is a predicate. For example, the following *where* specifies a search of the database for a solution to the predicate `returns(F,FType)` where *F* is an already bound metavariable:

```
where FType [type]
        returns (F,Ftype)
```

A *where* clause can also use ***each*** in its predicate to generate source fragments from the entire set of solutions to the predicate. For example, the following *where* clause binds *Funcs* to a sequence of all the function identifiers in the database.

```
where Funcs
    \ F \ [id*]
        each function (F [id])
```

```
\ extern FType MproF ( ParameterComment ); \ [declaration*]
   each function (F [id])
      where MproF [id]
         'mpro_ [concat F]
      where FType [type]
         returns (F,Ftype) or 'void
      where ParameterComment [comment]
         /* \ ParmType ParmName; \ [declaration*]
               each parameter (F,InOut[id],ParmType[type],ParmName[id])
            \\
         */
\\
```

Figure 7. Refined μC Metaprogram for the External Routine Declarations Artifact of the GL Example.

This version uses where clauses to refine the result using nested metaprograms, and improves over the simple version of Figure 6 in that it correctly renames the functions starting with 'mpro_' and inserts a comment showing the C parameter declarations of the functions' parameters.

Figure 7 shows a more sophisticated version of the μC metaprogram for the external C routine declarations artifact of the GL database. In this version, metaprogram refinement and nested metaprograms are used to generate a version of the external declarations that are annotated with C comments giving the expected parameter types of the glue routines .

Figure 8 compares the outputs of the two versions.

3 Implementation of μ* Using TXL

μ* is being implemented in prototype using the TXL tree transformation engine [Cordy, Halpern & Promislow 88, 91] by translating μL metaprograms for each target language L to corresponding TXL metaprograms. The TXL metaprogram is then combined with reference grammars for the target language L and Prolog to create a TXL program that implements the instantiation of the μL metaprogram from a Prolog database, as shown in Figure 9. The translation of μL metaprograms to corresponding TXL metaprograms is also achieved using a source transformation specified and implemented in TXL. A similar approach could be used to implement μ* using REFINE or Gentle.

4 Experience with μ*

μ* is still in the developmental stages, and thus far has been prototyped only to the extent of supporting μC for the purpose of implementing the metaprograms of the GL example. μC metaprograms to derive all of the required source artifacts from the GL database have been written and run as described in Section 3 using hand translation to TXL. All four of them produce correct and complete C output in a pleasing form directly from the original Prolog database for the GL interface. As shown in Figure 10, μC metaprograms can become very sophisticated indeed, although the basic paradigm of using an example C code template for the desired result remains, and μC metaprograms are easily read by C programmers.

output of the metaprogram of Figure 6

```
extern int zbuffer();
extern gid winopen();
```

output of the metaprogram of Figure 7

```
extern int mpro_zbuffer( /* */ );
extern void mpro_color( /*int index;*/ );
extern gid mpro_winopen(/*char* name;*/);
```

Figure 8. Comparison of the Output of the μC Metaprograms in Figures 6 and 7 Given the Database of Figure 1.

Note that in addition to not giving the parameter types in the result, the simple metaprogram of Figure 6 actually had a bug in that its search predicate did not find functions in the database with no explicitly specified return type. Hence the 'color' function is missing from its output.

The major difference in μ* from Ryman's "ideal" metaprograms is the use of explicit syntactic roles for metavariables, and the use of nested metaprograms to increase expressive power. While explicit roles necessitate some knowledge of target language syntax on the part of the programmer, the use of the user's guide reference syntax for the target language rather than a compiler-oriented LALR or LL grammar makes this relatively painless. In addition, the added redundancy of roles for metavariables increases precision in the specification of metaprograms and greatly decreases the chances of error, and the manipulation of large sections of code is easier and more convenient using syntactic handles than it could be in a text-based implementation.

The automatic implementation of the general μ* processor using TXL is complete and is currently in the testing stages [Shukla 92]. In addition to the original goal of automatically deriving code sections from design databases, there are plans to use metaprogramming as a code reuse technique. By annotating existing source modules to turn them into generic versions and instantiating them using a database of facts about new applications, we can use metaprogramming to implement retroactive reuse in the style originally proposed by Baker [Baker 88]. No doubt experience with the technique with other languages and applications will improve and refine our knowledge of metaprogramming and its practical uses in software development and reuse.

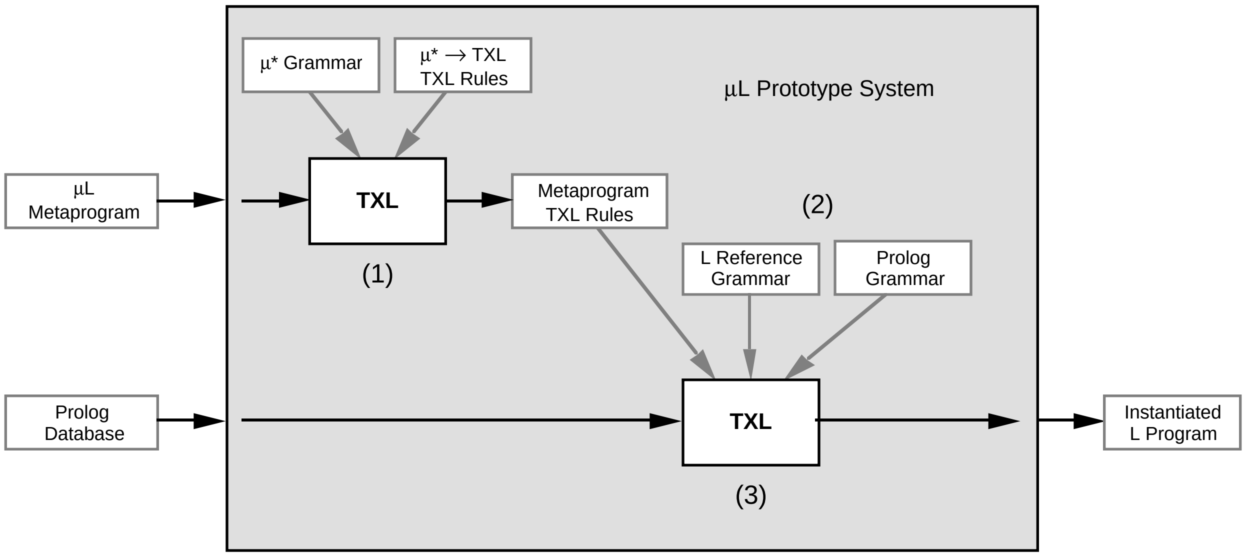

Figure 9. Prototype Implementation of μ* Using TXL.

A TXL source transformation is used to ranslate μL metaprograms to TXL metaprograms for target language L (1). The result is combined with standard reference grammars for L and Prolog to give a complete TXL program (2), which is then run with a Prolog database as input. The database is transformed by the TXL program to a target language L instantiation of the original μL metaprogram (3). The entire process is very efficient, running in the order of seconds on practical examples on an RS6000 workstation.

```
\    int MproF (p);
         struct {
              ParameterPointerFields
              ResultPointerField
              DummyField
         } *p;
     {
         ParameterDeclarations
         ResultDeclaration
         InputParameterBindings
         ResultAssignment
         OutputParameterBindings
         FailCondition
     }
\ [declaration*]

     each function (F [id])

       where MproF [id]
         'mpro_ [concat F]

       where ParameterPointerFields
         \ ParmType *ParmName; \ [declaration*]
              each parameter (F, ParmType [type], ParmName [id], InOut [id])

       where ResultPointerField
         \ ResultType *ResultName; \ [declaration]
              returns (F, ResultType [type], ResultName [id])

       where DummyField
         \ int *dummy; \ [declaration]
              not returns (F, ResultType [type], ResultName [id])
                and not parameter (F, ParmType [type], ParmName [id], InOut [id])

       where ParameterDeclaration
         \ ParmType ParmName; \ [declaration*]
         each parameter (F, ParmType [type], ParmName [id], InOut [id])

       where ResultDeclaration
         \ ResultType ResultName; \ [declaration]
              returns (F, ResultType [type], ResultName [id])

       where InputParameterBindings
         \ ParmName = (ParmType) *p -> ParmName; \ [statement*]
              each parameter (F, ParmType [type], ParmName [id], in)

       where ResultAssignment
         \ ResultName = F (Parameters); \ [statement]
              returns (F, ResultType [type], ResultName [id])

       where Parameters
         \ ParmName \ [id,]
              each parameter (F,ParmType [type], ParmName [id],InOut [id])

       where OutputParameterBindings
         \ (ParmType) *p -> ParmName = ParmName; \ [statement*]
              each parameter (F, ParmType [type], ParmName [id], out)

       where FailCondition
         \ return Condition; \ [statement]

              fail (F, Condition [expression])
\\
```

Figure 10. Complete μC Metaprogram to Generate the C Glue Routines Artifact of the GL example.

5 Acknowledgements

This work is supported by the Information Technology Research Centre as part of the Software Life Cycle Technology project (a joint project of Queen's University and the University of Toronto in association with the Advanced Software Design Technology project of the Centre for Advanced Studies, IBM Canada Limited) and in part by the Natural Sciences and Engineering Research Council of Canada as part of the Programming Language Technology project at Queen's University.

References

[Aho, Weinberger and Kernighan 79] Al Aho, Peter Weinberger, and Brian Kernighan,
AWK - a pattern scanning and processing language. *Software Practice and Experience* 9 (1979).

[Baker 88] Thomas D.J. Baker,
Retroactive reusability of existing code. M.Sc. thesis, Department of Computing and Information Science, Queen's University (1988).

[Basset 87] Paul G. Basset,
Frame-based software engineering. *IEEE Software* 4(4) (1987): 9-16.

[Cleaveland 88] J.Craig Cleaveland,
Building application generators. *IEEE Software* 5(4) (1988): 25-33.

[Consens, Mendelzon & Ryman 91] Mariano Consens, Alberto Mendelzon, and Arthur Ryman,
Visualizing and querying software structures. *Proc. 1991 CAS Conference* (1991): 17-35.

[Cordy, Halpern & Promislow 88] James R. Cordy, Charles D. Halpern, and Eric Promislow,
TXL: a rapid prototyping system for programming language dialects.
Proc. IEEE 1988 Int. Conf. on Computer Languages (1988): 280-285.

[Cordy, Halpern & Promislow 91] James R. Cordy, Charles D. Halpern-Hamu, and Eric Promislow,
TXL: a rapid prototyping system for programming language dialects.
Computer Languages 16(1) (1991): 97-107.

[Jones 86] C.B. Jones,
Systematic Software Development Using VDM. Prentice-Hall International (1986).

[Kotik & Markosian 90] Gordon B. Kotik and Lawrence Z. Markosian,
Program transformation: the key to automating software maintenance and re-engineering.
IEEE Trans. Software Eng. 16(9) (1990): 1024-1043.

[Kowalski 74] R. Kowalski,
Predicate logic as a programming language. *Proc. IFIP 74 Congress* (1974): 569-574.

[Levy 86] L.S. Levy,
A metaprogramming method and its economic justification. *IEEE Trans. Software Eng.* 12(2) (1986): 100-111.

[Ryman 90]   Arthur Ryman,
Requirements for a metaprogramming language. *Presentation at the 24th meeting of IFIP Working Group 2.4, Kingston, Canada* (1990).

[Schröer 89]   F.W. Schröer,
Klauselbasierte übersetzerbeschreibungen (Clause-based compiler specification).
Studienpapiere der GMD Forschungsstelle an der Universität Karlsruhe, Karlsruhe, Germany (1989).

[Shukla 92]   Medha Shukla,
A practical metaprogramming language and its implementation.   M.Sc. thesis,
Department of Computing and Information Science, Queen's University (expected 1992).

[Spivey 88]   J.M. Spivey,
Introducing Z: A Specification Language and its Formal Semantics.   Cambridge University Press (1988).

[Vollmer 91]   Jürgen Vollmer,
Experiences with Gentle: efficient compiler construction based on logic programming.
Proc. 3rd International Symposium on Programming Language Implementation and Logic Programming (PLILP 91), Springer Verlag Lecture Notes in Computer Science 528 (1991): 425-426.