Retrofitting the AutoBayes Program Synthesis System with Concrete Syntax

Bernd Fischer
Eelco Visser

Technical Report UU-CS-2004-012
Institute of Information and Computing Sciences
Utrecht University

February 2004

To appear as
B. Fischer and E. Visser. Retrofitting the AutoBayes Program Synthesis System with Concrete Syntax In Lengauer et al., editors, Domain-Specific Program Generation, Lecture Notes in Computer Science. Spinger-Verlag, 2004.

See also
`http://www.stratego-language.org/Stratego/PrologTools`

ISSN 0924-3275

Address:
Bernd Fischer
RIACS / NASA Ames Research Center
Moffett Field, CA 94035, USA
`fisch@email.arc.nasa.gov`

Eelco Visser
Institute of Information and Computing Sciences
Utrecht University
P.O.Box 80089
3508 TB Utrecht
`visser@acm.org`
`http://www.cs.uu.nl/people/visser`

Retrofitting the AutoBayes Program Synthesis System with Concrete Syntax

Bernd Fischer[1] and Eelco Visser[2]

[1] RIACS / NASA Ames Research Center, Moffett Field, CA 94035, USA
fisch@email.arc.nasa.gov
[2] Institute of Information and Computing Sciences, Universiteit Utrecht
3508 TB Utrecht, The Netherlands, visser@acm.org

Abstract. AUTOBAYES is a fully automatic, schema-based program synthesis system for statistical data analysis applications. Its core component is a schema library, i.e., a collection of generic code templates with associated applicability constraints which are instantiated in a problem-specific way during synthesis. Currently, AUTOBAYES is implemented in Prolog; the schemas thus use abstract syntax (i.e., Prolog terms) to formulate the templates. However, the conceptual distance between this abstract representation and the concrete syntax of the generated programs makes the schemas hard to create and maintain.

In this paper we describe how AUTOBAYES is retrofitted with concrete syntax. We show how it is integrated into Prolog and describe how the seamless interaction of concrete syntax fragments with AUTOBAYES's remaining "legacy" meta-programming kernel based on abstract syntax is achieved. We apply the approach to gradually migrate individual schemas without forcing a disruptive migration of the entire system to a different meta-programming language. First experiences show that a smooth migration can be achieved. Moreover, it can result in a considerable reduction of the code size and improved readability of the code. In particular, abstracting out fresh-variable generation and second-order term construction allows the formulation of larger continuous fragments.

1 Introduction

Program synthesis and transformation systems work on two language levels, the object-level (i.e., the language of the manipulated programs), and the meta-level (i.e., the implementation language of the system itself). Conceptually, these two levels are unrelated but in practice they have to be interfaced with each other. Often, the object-language is simply embedded within the meta-language, using a data type to represent the abstract syntax trees of the object-language. Although the actual implementation mechanisms (e.g., records, objects, or algebraic data types) may vary, embeddings can be used with essentially all meta-languages, making their full programming capabilities immediately available for program manipulations. Meta-level representations of object-level program fragments are then constructed in an essentially syntax-free fashion, i.e., not using the notation of the object-language, but using the operations provided by the data type.

However, syntax matters. The conceptual distance between the concrete programs that we understand and the meta-level representations that we need to use grows with the complexity of the object-language syntax and the size of the represented program fragments, and the use of abstract syntax becomes less and less satisfactory. Languages like Prolog and Haskell allow a rudimentary integration of concrete syntax via user-defined operators. However, this is usually restricted to simple precedence grammars, entailing that realistic object-languages cannot be represented well if at all. Traditionally, a quotation/anti-quotation mechanism is thus used to interface languages: a quotation denotes an object-level fragment, an anti-quotation denotes the result of a meta-level computation which is spliced into the object-level fragment. If object-language and meta-language coincide, the distinction between the language levels is purely conceptual, and switching between the levels is easy; a single compiler can be used to process both levels. If the object-language is user-definable, the mechanism becomes more complicated to implement and usually requires specialized meta-languages such as ASF+SDF [5], Maude [3], or TXL [4] which support syntax definition and reflection.

AUTOBAYES [9,8] is a program synthesis system for the statistical data analysis domain. It is a large and complex software system implemented in Prolog and its complexity is comparable to a compiler. The synthesis process is based on schemas which are written in Prolog and use abstract syntax representations of object-program fragments. The introduction of concrete syntax would simplify the creation and maintenance of these schemas. However, a complete migration of the implementation of AUTOBAYES to a different meta-programming language requires a substantial effort and disrupts the ongoing system development. To avoid this problem, we have chosen a different path.

In this chapter, we thus describe the first experiences with our ongoing work on adding support for user-definable concrete syntax to AUTOBAYES. We follow the general approach outlined in [15], which allows the extension of an arbitrary meta-language with concrete object-language syntax by combining the syntax definitions of both languages. We show how the approach is instantiated for Prolog and describe the processing steps required for a seamless interaction of concrete syntax fragments with the remaining "legacy" meta-programming system based on abstract syntax—despite all its idiosyncrasies. With this work we show that the approach of [15] can indeed be applied to meta-languages other than Stratego. To reduce the effort of making such instantiations we have constructed a generic tool encapsulating the process of parsing a program using concrete object-syntax. Furthermore, we have extended the approach with object-level comments, and object-language specific transformations for integrating object-level abstract syntax in the meta-language.

The original motivation for this specific path was purely pragmatic. We wanted to realize the benefits of concrete syntax without forcing the disruptive migration of the entire system to a different meta-programming language. Retrofitting Prolog with support for concrete syntax allows a gradual migration. Our long-term goal, however, is more ambitious: we want to support domain experts in creating and maintaining schemas. We expect that the use of concrete syntax makes it easier to gradually "schematize" existing domain programs. We also plan to use different grammars to describe programs on different levels of abstraction and thus to support domain engineering.

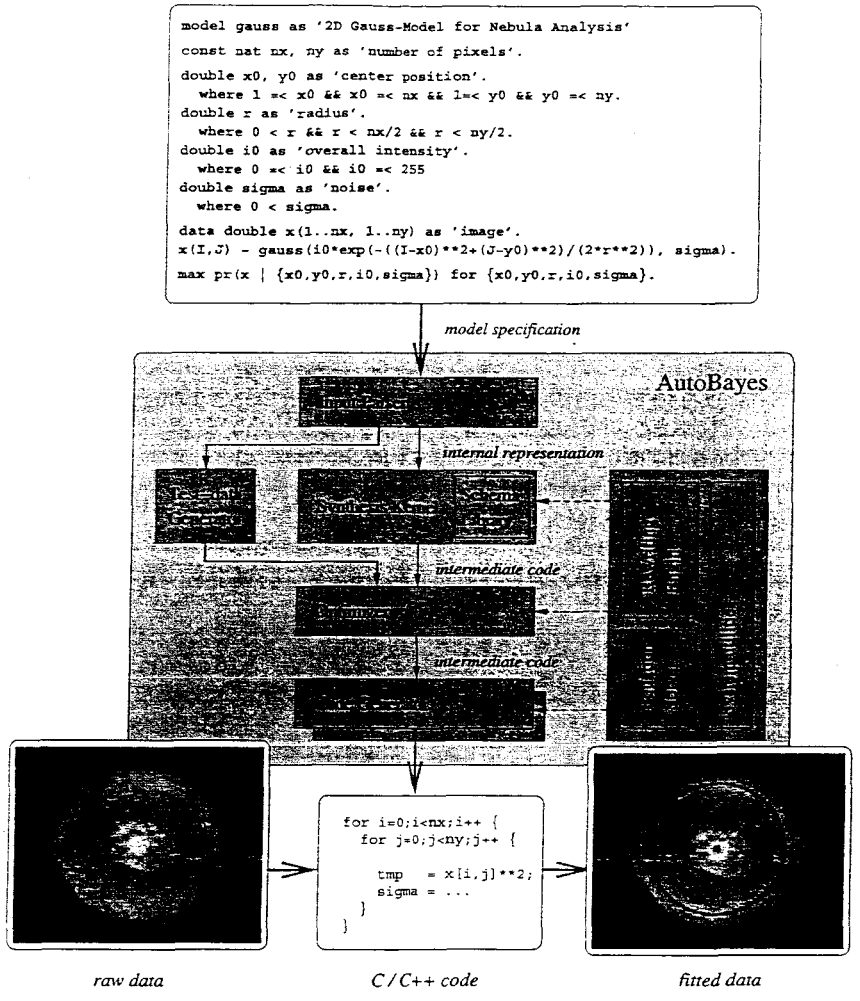

Fig. 1. AUTOBAYES system architecture.

2 Overview of the AutoBayes System

AUTOBAYES is a fully automatic program synthesis system for data analysis problems. It has been used to derive programs for applications like the analysis of planetary nebulae images taken by the Hubble space telescope [7,6] as well as research-level machine learning algorithms [1]. It is implemented in SWI-Prolog [18] and currently comprises about 75,000 lines of documented code; Figure 1 shows the system architecture.

AUTOBAYES derives code from a *statistical model* which describes the expected properties of the data in a fully declarative fashion: for each problem variable (i.e., observation or parameter), properties and dependencies are specified via probability

distributions and constraints. The top box in Figure 1 shows the specification of a nebulae analysis model. The last two clauses are the core of this specification; the remaining clauses declare the model constants and variables, and impose constraints on them. The distribution clause

```
x(I,J) ~ gauss(i0 * exp(-((I-x0)**2+(J-y0)**2)/(2*r**2)), sigma).
```

states that, with an expected error *sigma*, the expected value of the observation x at a given position (i, j) is a function of this position and the nebula's unknown center position (x_0, y_0), radius r, and overall intensity i_0,. The task clause

```
max pr(x|{i0,x0,y0,r,sigma}) for {i0,x0,y0,r,sigma}.
```

specifies the analysis task, which the synthesized program has to solve, i.e., to estimate the parameter values which maximize the probability of actually observing the given data and thus under the given model best explain the observations. In this case, the task can be solved by a mean square error minimization due to the gaussian distribution of the data and the specific form of the probability. Note, however, that (i) this is not immediately clear from the model, (ii) the function to be minimized is not explicitly given in the model, and (iii) even small modifications of the model may require completely different algorithms.

AUTOBAYES derives the code following a schema-based approach. A *program schema* consists of a parameterized code fragment (i.e., template) and a set of constraints. Code fragments are written in ABIR (AUTOBAYES Intermediate Representation), which is essentially a "sanitized" variant of C (e.g., neither pointers nor side effects in expressions) but also contains a number of domain-specific constructs (e.g., vector/matrix operations, finite sums, and convergence-loops). The fragments also contain parameterized object-level comments which eventually become the documentation of the synthesized programs. The parameters are instantiated either directly by the schema or by AUTOBAYES calling itself recursively with a modified problem. The constraints determine whether a schema is applicable and how the parameters can be instantiated. They are formulated as conditions on the model, either directly on the specification, or indirectly on a Bayesian network [2] extracted from the specification. Such networks are directed, acyclic graphs whose nodes represent the variables specified in the model and whose edges represent the probabilistic dependencies between them, as specified by the distribution clauses: the variable on the left-hand side depends on all model variables occuring on the right-hand side. In the example, each x_{ij} thus depends on i_0, x_0, y_0, r and *sigma*.

The schemas are organized hierarchically into a schema library. Its top layers contain decomposition schemas based on independence theorems for Bayesian networks which try to break down the problem into independent sub-problems. These are domain-specific divide-and-conquer schemas: the emerging sub-problems are fed back into the synthesis process and the resulting programs are composed to achieve a solution for the original problem. Guided by the network structure, AUTOBAYES is thus able to synthesize larger programs by composition of different schemas. The core layer of the library contains statistical algorithm schemas as for example *expectation maximization* (EM) [10] and *nearest neighbor clustering* (k-Means); usually, these generate the skeleton of

the program. The final layer contains standard numeric optimization methods as for example the simplex method or different conjugate gradient methods. These are applied after the statistical problem has been transformed into an ordinary numeric optimization problem and AUTOBAYES failed to find a symbolic solution for that problem. The schemas in the upper layers of the library are very similar to the underlying theorems and thus contain only relatively small code fragments while the schemas in the lower layers closely resemble "traditional" generic algorithm templates. Their code fragments are much larger and make full use of ABIR's language constructs. These schemas are the focus of our migration approach.

The schemas are applied exhaustively until all maximization tasks are rewritten into ABIR code. The schemas can explicitly trigger large-scale optimizations which take into account information from the synthesis process. For example, all numeric optimization routines restructure the goal expression using code motion, common subexpression elimination, and memoization. In a final step, AUTOBAYES translates the ABIR code into code tailored for a specific run-time environment. Currently, it provides code generators for the Octave and Matlab environments; it can also produce standalone C and Modula-2 code. The entire synthesis process is supported by a large meta-programming kernel which includes the graphical reasoning routines, a symbolic-algebraic subsystem based on a rewrite engine, and a symbolic equation solver.

3 Program Generation in Prolog

In the rest of this chapter we will describe how AUTOBAYES is retrofitted with concrete object syntax for the specification of program schemas. We start in this section with a description of program generation with abstract syntax in Prolog. In Section 4 we describe the replacement of abstract syntax with concrete syntax. In Sections 5 to 7 we then discuss the techniques used to implement this embedding.

Figure 2 shows an excerpt of a schema that implements (i.e., generates code for) the Nelder-Mead simplex method for numerically optimizing a function with respect to a set of variables [11]. The complete schema comprises 508 lines of documented Prolog-code, and is fairly typical in most aspects, e.g., the size of the overall schema and of the fragment, respectively, the amount of meta-programming, or the ratio between the code constructed directly (e.g., `Code`) and recursively (e.g., `Reflection`). This schema is also used to generate the algorithm core for the nebula specification example.

3.1 Abstract Syntax in Prolog

The simplex schema is implemented as a single Prolog clause which takes as arguments an expression `Formula` representing the function to be optimized, a set `Vars` of target variables, and an expression `Constraint` representing the constraints on all variables. It returns as result `Code` an ABIR code fragment which contains the appropriately instantiated Nelder-Mead simplex algorithm. This code is represented by means of a Prolog term. In general, a Prolog term is either an atom, a variable,[3] a *functor* application $f(t_1, ..., t_n)$, applying a functor f to Prolog terms t_i, or a list $[t_1, \ldots, t_n]$

[3] Prolog uses capitalization to distinguish a variable `X` from an atom `x`.

```
schema(Formula, Vars, Constraint, Code) :-
  ...
  model_gensym(simplex, Simplex),
  SDim  = [dim(A_BASE, Size1), dim(A_BASE, Size0)],
  SDecl = matrix(Simplex, double, SDim,
                 [comment(['Simplex data structure: (', Size, '+1) ',
                           'points in the ', Size,
                           '-dimensional space'])]),
  ...
  var_fresh(I),
  var_fresh(J),
  index_make([I, dim(A_BASE, Size0)], Index_i),
  index_make([J, dim(A_BASE, Size1)], Index_j),
  Center_i   =.. [Center, I],
  Simplex_ji =.. [Simplex, J, I],
  Centroid =
    for([Index_i],
        assign(Center_i, sum([Index_j], Simplex_ji), []),
        [comment(['Calculate the center of gravity in the simplex'])]),
  ...
  simplex_try(Formula, Simplex, ...,
              -1, 'Reflect the simplex from the worst point (F = -1)',
              Reflection),
  ...
  Loop = while(converging([...]),
               series([Centroid, Reflection, ...], []),
               [comment('Convergence loop')]),
  ...
  Code = block(local([SDecl, ...]),
               series([Init, Loop, Copy], []),
               [label(SLabel), comment(XP)]).
```

Fig. 2. AUTOBAYES-schema for the Nelder-Mead simplex method (excerpt)

of terms. An ABIR program is then represented as a term by using a functor for each construct in the language, for example:

```
assign : lvalue * expression * list(comment) -> statement
for    : list(index) * statement * list(comment) -> statement
series : list(statement) * list(comment) -> statement
sum    : list(index) * expression -> expression
```

Thus, if i represents an index, s a statement, and c a comment, the term `for([`i`],`s`,[`c`])` represents a for-statement.

Each goal of the form $X = t$ binds a Prolog term t to a Prolog variable X. However, in the schema, the program for `Code` is not constructed as a single large term, but rather assembled from smaller fragments by including the terms bound to these variables. In Figure 2, the terms corresponding to ABIR fragments are distinguished typographically using italics, but this is a conceptual distinction only.

3.2 Meta-Programming Kernel

In addition to goals composing program fragments by direct term formation, the schema contains recursive schema invocations such as `simplex_try`, which produces code for the `Reflection` fragment from a more constrained version of `Formula`. Furthermore, the schema calls a number of meta-programming predicates. For example, the `var_fresh(X)` predicate generates a fresh object variable and binds it to its argument, which is a meta-level variable. This prevents variable clashes in the generated program. Similarly, the `index_make` predicate constructs an index expression.

The schema in Figure 2 also uses second-order terms to represent array accesses or function calls where the names are either given as parameters or automatically renamed apart from a meaningful root (cf. the `model_gensym(simplex, Simplex)` goal). A fully abstract syntax would use additional functors for these constructs and represent for example an access to the array `simplex` with subscripts `pv0` and `pv1` by `arraysub(simplex, [var(pv0), var(pv1)])`. However, this makes the abstract syntax rather unwieldy and much harder to read. Therefore, such constructs are abbreviated by means of simple functor applications, e.g., `simplex(pv0, pv1)`. Unfortunately, Prolog does not allow second-order term formation, i.e., terms with variables in the functor-position. Instead, it is necessary to use the built-in `=..`-operator, which constructs a functor application from a list where the head element is used as functor name and the rest of the list contains the arguments of the application. Hence, the schemas generate array access expressions such as the one above by goals such as `Simplex_ji =.. [Simplex, J, I]`, where the meta-variables `Simplex`, `J`, and `I` are bound to concrete names.

4 Migrating from Abstract Syntax to Concrete Syntax

The excerpt shows why the simple abstract syntax approach quickly becomes cumbersome as the schemas become larger. The code fragment is built up from many smaller fragments by the introduction of new meta-variables (e.g., `Loop`) because the abstract syntax would become unreadable otherwise. However, this makes it harder to follow and understand the overall structure of the algorithm. The schema is sprinkled with a large number of calls to small meta-programming predicates, which makes it harder to write schemas because one needs to know not only the abstract syntax, but also a large part of the meta-programming base. In our experience, these peculiarities make the learning curve much steeper than it ought to be, which in turn makes it difficult for a domain expert to gradually extend the system's capabilities by adding a single schema.

In the following, we illustrate how this schema is migrated and refactored to make use of concrete syntax, using the `Centroid` fragment as running example.

4.1 Concrete Syntax

The first step of the migration is to replace terms representing program fragments in abstract syntax by the equivalent fragments in the concrete syntax of ABIR. Thus, the `Centroid` fragment becomes:

```
Centroid = |[
  /* Calculate the center of gravity in the simplex */
  for( Index_i:idx )
    Center_i := sum( Index_j:idx ) Simplex_ji:exp
]|
```

Here, we use `|[ ... ]|` to quote a piece of ABIR code within a Prolog program.

4.2 Meta-variables

In the translation to concrete syntax, Prolog variables in a term are *meta-variables*, i.e., variables ranging *over* ABIR code, rather than variables *in* ABIR code. In the fragment `|[ x := 3 + j ]|`, `x` and `j` are ABIR variables, whereas in the fragment `|[ x := 3 + J:exp ]|`, `x` is an ABIR variable, but `J:exp` is a meta-variable ranging over expressions. For the embedding of ABIR in Prolog we use the convention that meta-variables are distinguished by capitalization and can thus be used directly in the concrete syntax without tags. In a few places, the meta-variables are tagged with their syntactic category, e.g., `Index_i:idx`. This allows the parser to resolve ambiguities and to introduce the injection functions necessary to build well-formed syntax trees.

4.3 Abstracting from Meta-programming Operations

The next migration step eliminates calls to meta-programming predicates and replaces them by appropriate abstractions in ABIR. First, we remove the creation of index expressions such as `index_make([I, dim(A_BASE, Size0)], Index_i)` and replace the corresponding `Index` variables directly with the more natural loop index notation:

```
Centroid = |[
  /* Calculate the center of gravity in the simplex */
  for( I := A_BASE .. Size0 )
    Center_i := sum( J := A_BASE .. Size1 ) Simplex_ji:exp
]|
```

Incidentally, this also eliminates the need for the `idx`-tags because the syntactic category is now determined by the source text.

Next, array-reference creation with the `=..` operator is replaced with array access notation in the program fragment:

```
Centroid = |[
  /* Calculate the center of gravity in the simplex */
  for( I := A_BASE .. Size0 )
    Center[I] := sum( J := A_BASE .. Size1 ) Simplex[J, I]
]|
```

Finally, the explicit generation of fresh object-variables using `var_fresh` is expressed *in* the code by tagging the corresponding meta-variable with `@new`, a special anti-quotation operator which constructs fresh object-level variable names: of

```
Centroid = |[
  /* Calculate the center of gravity in the simplex */
  for( I@new := A_BASE .. Size0 )
    Center[I] := sum( J@new := A_BASE .. Size1 ) Simplex[J, I]
]|
```

Thus 10 lines of code have been reduced to 5 lines, which are more readable.

4.4 Fragment Inlining

The final step of the migration consists of refactoring the schema by inlining program fragments; the fragments are self-descriptive, do not depend on separate calls to meta-programming predicates, and can be read as pieces of code. For example, the fragment for `Centroid` above can be inlined in the fragment for Loop, which itself can be inlined in the final `Code` fragment. After this refactoring, a schema consists of one, or a few, large program patterns.

In the example schema the use of concrete syntax, `@new`, and inlining reduces the overall size by approximately 30% and eliminates the need for explicit meta-programming. The reduction ratio is more or less maintained over the entire schema. After migration along the lines above, the schema size is reduced from 508 lines to 366 lines. After white space removal, the original schema contains 7779 characters and the resulting schema with concrete syntax 5538, confirming a reduction of 30% in actual code size. At the same time, the resulting fewer but larger code fragments give a better insight into the structure of the generated code.

5 Embedding Concrete Syntax into Prolog

The extension of Prolog with concrete syntax as sketched in the previous section is achieved using the syntax definition formalism SDF2 [14,12] and the transformation language Stratego [16,13] following the approach described in [15]. SDF is used to specify the syntax of ABIR and Prolog as well as the embedding of ABIR into Prolog. Stratego is used to transform syntax trees over this combined language into a pure Prolog program. In this section we explain the syntactical embedding, and in the next two sections we outline the transformations mapping Prolog with concrete syntax to pure Prolog.

5.1 Syntax of Prolog and ABIR

The extension of a meta-language with concrete object syntax requires an embedding of the syntax of object code fragments as expressions in the meta-language. We thus created syntax definitions for Prolog and ABIR using SDF. An SDF production $A_1 ... A_n \text{->} A_0$ is a context-free grammar rule that declares that the concatenation of strings of sorts A_1 to A_n is a string of sort A_0. The following is a fragment from the Prolog syntax definition with productions for clauses and terms. Note that the SDF construct `{S l}+` denotes one or more `S`s *separated* by `l`s.

```
module Prolog
exports
  context-free syntax
    Head ":-" Body "."          -> Clause {cons("nonunitclause")}
    Goal                        -> Body {cons("bodygoal")}
    Term                        -> Goal
    Functor "(" {Term ","}+ ")" -> Term {cons("func")}
    Term Op Term                -> Term {cons("infix")}
    Variable                    -> Term {cons("var")}
    Atom                        -> Term {cons("atom")}
    Name                        -> Functor {cons("functor")}
    Name                        -> Op {cons("op")}
```

The {cons(c)} annotations in the productions declare the constructors to be used in abstract syntax trees corresponding to the parse trees over the syntax definition. Similarly, the following is a fragment from the syntax definition of ABIR:

```
module ABIR
exports
  context-free syntax
    LValue ":=" Exp              -> Stat {cons("assign")}
    "for" "(" IndexList ")" Stat -> Stat {cons("for")}
    {Index ","}*                 -> IndexList {cons("indexlist")}
    Id ":=" Exp ".." Exp         -> Index {cons("index")}
```

5.2 Combining Syntax Definitions

Since SDF is a modular syntax definition formalism, combining languages is simply a matter of importing the appropriate modules. In addition, object-language expressions should be embedded in meta-language expressions. The following module defines such an embedding of ABIR into Prolog:

```
module PrologABIR
imports Prolog ABIR
exports
  context-free syntax
    "|[" Exp  "]|" -> Term {cons("toterm")}
    "|[" Stat "]|" -> Term {cons("toterm")}
  variables
    [A-Z][A-Za-z0-9_]*        -> Id {prefer}
    [A-Z][A-Za-z0-9_]* ":exp" -> Exp
```

The module declares that ABIR Expressions and Statemements can be used as Prolog terms by quoting them with the |[and]| delimiters, as we have seen in the previous section. The variables section declares schemas for *meta-variables*. Thus, a capitalized identifier can be used as a meta-variable for identifiers, and a capitalized identifier tagged with :exp can be used as a meta-variable for expressions.

6 Exploding Embedded Abstract Syntax

6.1 Embedded Abstract Syntax

After parsing a schema with the combined syntax definition the resulting abstract syntax tree is a mixture of Prolog and ABIR abstract syntax. For example, the Prolog-goal

```
Code = |[ X := Y:exp + z ]|
```

is parsed into the abstract syntax term

```
bodygoal(infix(var("Code"), op(symbol("=")),
               toterm(assign(var(meta-var("X")),
                             plus(meta-var("Y:exp"),var("z"))))))
```

The language transitions are characterized by the `toterm`-constructor, and meta-variables are indicated by the `meta-var`-constructor. Thus, `bodygoal` and `infix` belong to Prolog abstract syntax, while `assign`, `var` and `plus` belong to ABIR abstract syntax.

6.2 Exploding

A mixed syntax tree can be translated to a pure Prolog tree by "exploding" embedded tree constructors to functor applications:[4]

```
bodygoal(infix(var("Code"),op(symbol("=")),
               func(functor(word("assign")),
                    [func(functor(word("var")),[var("X")]),
                     func(functor(word("plus")),
                          [var("Y:exp"),
                           func(functor(word("var")),
                                [atom(quotedname("'z'"))])])])))
```

After pretty-printing this tree we get the pure Prolog-goal

```
Code = assign(var(X), plus(Y, var('z')))
```

Note how the meta-variables `X` and `Y` have become Prolog variables representing a variable name and an expression, respectively, while the object variable `z` has become a character literal. Also note that `X` is a a meta-variable for an *object-level identifier* and will eventually be instantiated with a character literal, while `Y` is a variable for an *expression.*

[4] Note that `functor` is just a term label and different from the built-in predicate `functor/3`.

```
strategies
  explode = alltd(?toterm(<trm-explode>))
  trm-explode = trm-metavar <+ trm-op
rules
  trm-metavar : meta-var(X) -> var(X)
  trm-op : Op#([]) -> atom(word(<lower-case>Op))
  trm-op : Op#([T | Ts]) -> func(functor(word(<lower-case>Op)),
                                 <map(trm-explode)>[T | Ts])
```

Fig. 3. Rules and strategy for exploding embedded abstract syntax.

6.3 Implementing Explosion in Stratego

Explosion is defined generically using transformations on mixed syntax trees, i.e., it is independent from the object language. The complete explosion transformation takes about 35 lines of Stratego and deals with special cases such as strings and lists, but the essence of the transformation is shown in Figure 3. A detailed explanation of the specification is beyond the scope of this chapter. For an introduction to Stratego see [13].

The `explode` strategy uses the generic traversal strategy `alltd` to descend into the abstract syntax tree of the Prolog program. When encountering a term constructed with `toterm`, its argument is exploded using the `trm-explode` transformation, which either applies one of the rules `trm-op` or the rule `trm-metavar`. The latter rule turns a meta-variable encountered in an embedded term into a Prolog variable. The `trm-op` rules transform constructor applications. The left-hand side of the rules have the form `Op#(Ts)`, thus generically decomposing a constructor application into its constructor (or operator) `Op`, and the list of arguments `Ts`. If the list of arguments is empty, an atom is produced. Otherwise a functor application is produced, where the arguments of the functor are recursively exploded by mapping the `trm-explode` strategy over the list of arguments.

7 Custom Abstract Syntax

Parsing and then exploding the final `Centroid`-fragment on page 9 then produces the pure Prolog-goal

```
Centroid =
  commented(
   comment(['Calculate the center of gravity in the simplex ']),
    for(indexlist([index(newvar(I),var(A_BASE),var(Size0))]),
        assign(arraysub(Center,[var(I)]),
               sum(indexlist([index(newvar(J),
                                    var(A_BASE),var(Size1))]),
                   call(Simplex,[var(J),var(I)])))))
```

Comparing the generated `Centroid`-goal above with the original in Figure 2 shows that the abstract syntax underlying the concrete syntax fragments does not correspond

exactly to the original abstract syntax used in AutoBayes. That is, two different abstract syntax formats are used for the ABIR language. The format used in AUTOBAYES (e.g., Figure 2) is less explicit since it uses Prolog functor applications to represent array references and function calls, instead of the more verbose representation underlying the concrete syntax fragments.

In order to interface schemas written in concrete syntax with legacy components of the synthesis system, additional transformations are applied to the Prolog code, which translate between the two versions of the abstract syntax. For the `Centroid`-fragment this produces:

```
Centroid =
  for([idx(newvar(I),A_BASE,Size0)],
      assign(arraysub(Center,[I]),
             sum([idx(newvar(J),A_BASE,Size1)],call(Simplex,[J,I]))),
      [comment(['Calculate the center of gravity in the simplex '])])
```

7.1 Lifting Predicates

In AutoBayes, array accesses are represented by means of functor applications and object variable names are generated by gensym-predicates. This cannot be expressed in a plain Prolog term. Thus `arraysubs` and `calls` are hoisted out of abstract syntax terms and turned into term constructors and fresh variable generators as follows:

```
var_fresh(I), _a =.. [Center,I], var_fresh(J), _b =.. [Simplex,J,I],
Centroid =
  for([idx(I, A_BASE, Size0)],
      assign(_a, sum([idx(J, A_BASE, Size1)], _b)),
      [comment(['Calculate the center of gravity in the simplex '])])
```

Hence, the embedded concrete syntax is transformed exactly into the form needed to interface it with the legacy system.

8 Conclusions

Program generation and transformation systems manipulate large, parameterized object language fragments. Operating on such fragments using abstract-syntax trees or string-based concrete syntax is possible, but has severe limitations in maintainability and expressive power. Any serious program generator should thus provide support for concrete object syntax together with the underlying abstract syntax.

In this chapter we have shown that the approach of [15] can indeed be generalized to meta-languages other than Stratego and that it is thus possible to add such support to systems implemented in a variety of meta-languages. We have applied this approach to AutoBayes, a large program synthesis system that uses a simple embedding of its object-language (ABIR) into its meta-language (Prolog). The introduction of concrete syntax results in a considerable reduction of the schema size ($\approx$ 30%), but even more importantly, in an improved readability of the schemas. In particular, abstracting out fresh-variable generation and second-order term construction allows the formulation of

larger continuous fragments and improves the locality in the schemas. Moreover, meta-programming with concrete syntax is cheap: using Stratego and SDF, the overall effort to develop all supporting tools was less than three weeks. Once the tools were in place, the migration of a schema was a matter of a few hours. Finally, the experiment has also demonstrated that it is possible to introduce concrete syntax support gradually, without forcing a disruptive migration of the entire system to the extended meta-language. The seamless integration with the "legacy" meta-programming kernel is achieved with a few additional transformations, which can be implemented quickly in Stratego.

8.1 Contributions

The work described in this chapter makes three main contributions to domain-specific program generation. First, we described an extension of Prolog with concrete object syntax, which is a useful tool for all meta-programming systems using Prolog. The tools that implement the mapping back into pure Prolog are available for embedding arbitrary object languages into Prolog.[5] Second, we demonstrated that the approach of [15] can indeed be applied to meta-languages other than Stratego. We extended the approach by incorporating concrete syntax for object-level comments and annotations, which are required for documentation and certification of the generated code [17]. Third, we also extended the approach with object-language-specific transformations to achieve a seamless integration with the legacy meta-programming kernel. This allows a gradual migration of existing systems, even if they were originally designed without support for concrete syntax in mind. These transformations also lift meta-computations from object code into the surrounding meta-code. This allows us to introduce abstractions for fresh variable generation and second-order variables to Prolog.

8.2 Future Work

In future work, we will migrate more schemas to concrete syntax to make the maintenance of the AUTOBAYES easier. We expect that these changes will confirm our estimate of 30% reduction in the size of the schemas.

We also plan to investigate the usefulness of concrete syntax in a gradual "schematization" of existing domain programs. The basic idea here is to use the existing program initially unmodified as code fragment in a very specialized schema, and then to abstract it incrementally, e.g., by parameterizing out names or entire computations which can then be re-instantiated differently during synthesis. Finally, we plan to use grammars as types to enforce that the fragments are not only syntactically well-formed but actually contain code of the right form. We hope that we can support domain engineering by using grammars on different levels of abstraction.

Acknowledgements We would like to thank the anonymous referees for their comments on a previous version of this paper.

[5] `http://www.stratego-language.org/Stratego/PrologTools`

References

1. W. Buntine, B. Fischer, and A. G. Gray. Automatic derivation of the multinomial PCA algorithm. Technical report, NASA/Ames, 2003. Available at `http://ase.arc.nasa.gov/people/fischer/`.
2. W. L. Buntine. Operations for learning with graphical models. *JAIR*, 2:159–225, 1994.
3. M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and J. F. Quesada. Maude: specification and programming in rewriting logic. *Theoretical Computer Science*, 285(2):187–243, 2002.
4. J. R. Cordy, I. H. Carmichael, and R. Halliday. *The TXL Programming Language, Version 8*, Apr. 1995.
5. A. van Deursen, J. Heering, and P. Klint, editors. *Language Prototyping. An Algebraic Specification Approach*, volume 5 of *AMAST Series in Computing*. World Scientific, Singapore, September 1996.
6. B. Fischer, A. Hajian, K. Knuth, and J. Schumann. Automatic derivation of statistical data analysis algorithms: Planetary nebulae and beyond. Technical report, NASA/Ames, 2003. Available at `http://ase.arc.nasa.gov/people/fischer/`.
7. B. Fischer and J. Schumann. Applying autobayes to the analysis of planetary nebulae images. In J. Grundy and J. Penix, editors, *Proc. 18th ASE*, pages 337–342, Montreal, Canada, October 6–10 2003. IEEE Comp. Soc. Press.
8. B. Fischer and J. Schumann. AutoBayes: A system for generating data analysis programs from statistical models. *JFP*, 13(3):483–508, May 2003.
9. A. G. Gray, B. Fischer, J. Schumann, and W. Buntine. Automatic derivation of statistical algorithms: The EM family and beyond. In S. Becker, S. Thrun, and K. Obermayer, editors, *NIPS 15*, pages 689–696. MIT Press, 2003.
10. G. McLachlan and T. Krishnan. *The EM Algorithm and Extensions*. Wiley Series in Probability and Statistics. John Wiley & Sons, New York, 1997.
11. W. H. Press, B. P. Flannery, S. A. Teukolsky, and W. T. Vetterling. *Numerical Recipes in C*. Cambridge Univ. Press, Cambridge, UK, 2nd. edition, 1992.
12. M. G. J. van den Brand, J. Scheerder, J. Vinju, and E. Visser. Disambiguation filters for scannerless generalized LR parsers. In N. Horspool, editor, *Compiler Construction (CC'02)*, volume 2304 of *Lecture Notes in Computer Science*, pages 143–158, Grenoble, France, April 2002. Springer-Verlag.
13. E. Visser. Program transformation with Stratego/XT. Rules, strategies, tools, and systems in Stratego/XT 0.9. In this volume.
14. E. Visser. *Syntax Definition for Language Prototyping*. PhD thesis, University of Amsterdam, September 1997.
15. E. Visser. Meta-programming with concrete object syntax. In D. Batory, C. Consel, and W. Taha, editors, *Generative Programming and Component Engineering (GPCE'02)*, volume 2487 of *Lecture Notes in Computer Science*, pages 299–315, Pittsburgh, PA, USA, October 2002. Springer-Verlag.
16. E. Visser, Z.-e.-A. Benaissa, and A. Tolmach. Building program optimizers with rewriting strategies. In *Proceedings of the third ACM SIGPLAN International Conference on Functional Programming (ICFP'98)*, pages 13–26. ACM Press, September 1998.
17. M. Whalen, J. Schumann, and B. Fischer. Synthesizing certified code. In L.-H. Eriksson and P. A. Lindsay, editors, *Proc. FME 2002: Formal Methods—Getting IT Right*, volume 2391 of *LNCS*, pages 431–450, Copenhagen, Denmark, July 2002. Springer.
18. J. Wielemaker. *SWI-Prolog 5.2.9 Reference Manual*. Amsterdam, 2003.