Adding Concrete Syntax to a Prolog-Based Program Synthesis System

(Extended Abstract)

Bernd Fischer and Eelco Visser

[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. Program generation and transformation systems manipulate large, parameterized object language fragments. Support for user-definable concrete syntax makes this easier but is typically restricted to certain object and meta languages. We show how Prolog can be retrofitted with concrete syntax and describe how a seamless interaction of concrete syntax fragments with an existing "legacy" meta-programming system based on abstract syntax is achieved. We apply the approach to gradually migrate the schemas of the AUTOBAYES program synthesis system to concrete syntax. First experiences show that this can result in a considerable reduction of the code size and an improved readability of the code. 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.

1 Introduction

Program generation 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 an abstract data type to represent the abstract syntax trees of the object language. The actual implementation mechanisms (e.g., records, objects, or algebraic data types) may vary but embeddings can be used with arbitrary meta languages and make their full programming capabilities immediately available for program manipulations. Meta-level representations of object-level program fragments are then built in an essentially asyntactic fashion using the operations provided by the data type.

However, syntax matters. The conceptual distance between the concrete programs that we understand and their 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

so 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 switch between the then purely conceptual language levels is easy and a single compiler can be used to process them both. If the object language is user-definable, the mechanism becomes more complicated to implement and usually requires specialized meta languages such as ASF+SDF [6], Maude [5], or TXL [4] which support syntax definition and reflection.

In this paper, we follow a slightly different path. We describe the first experiences with our ongoing work on adding support for user-definable concrete syntax to AUTOBAYES [9,7], a large, schema-based program synthesis system implemented in Prolog. We follow the general approach outlined in [15], which allows the extension of an arbitrary meta language with concrete object language syntax by merging 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.

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.

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 [8] as well as research-level machine learning algorithms [1]. It is implemented in SWI-Prolog [16] and currently comprises about 64,000 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 just 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 center position (x_0, y_0), radius r, and overall intensity i_0,. The task clause

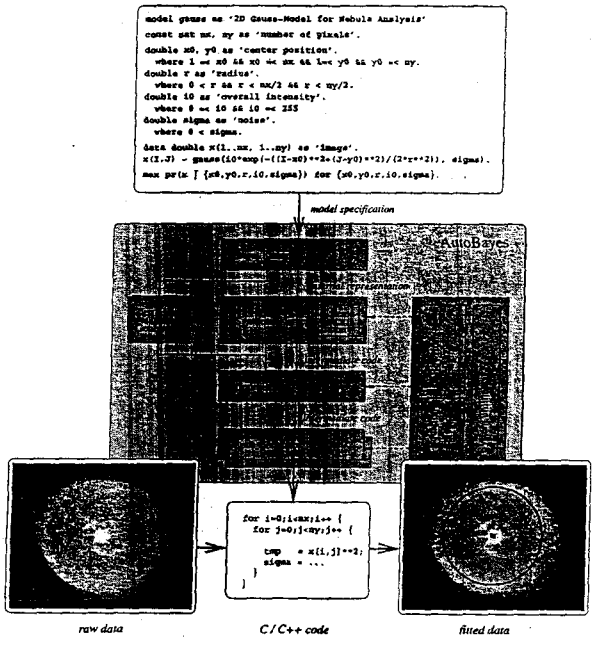

Fig. 1. AUTOBAYES system architecture.

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

specifies the analysis task 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 thus 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., no 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 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 [3] extracted from the specification.

The schemas are organized hiearchically 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*; 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 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 Migrating from Abstract Syntax to Concrete Syntax

In the existing AUTOBAYES-implementation, schemas are simply Prolog-clauses and code fragments are simply Prolog-terms. The excerpt in Figure 2 shows 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.

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 (e.g., `model_gensym`, or `index_make`); this 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. The use of Prolog's term builder `=..` (which is required as Prolog does not support second-order patterns) is particularly pervasive because schemas tend to be parameterized with the names of object-level data structures. In our experience, these peculiarities make the

```
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)

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 show how this schema is migrated and refactored, making it easier to understand and maintain. The first step is to replace terms representing abstract syntax by concrete syntax literals, e.g.,

```
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 `|[ ... ]|` as quotation operator but leave anti-quotation implicit. Prolog (meta-) variables are distinguished by capitalization and can thus be used directly in the

concrete syntax. A general anti-quotation mechanism is not required since Prolog is a relational language and the result of a meta-computation is not uniquely determined—any number of variables can be instantiated as a result. 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.

The next step inlines the indexes, which eliminates the calls of the `index_make` meta-predicate shown in Figure 2.

```
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 tags because the syntactic category is now determined by the source text. Next, the array-references are inlined, thus eliminating the `=..`-constructors.

```
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 object variables are tagged with `@new`; this is a special anti-quotation operator which constructs fresh object-level variable names.[1]

```
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]
]|
```

Here, the use of concrete syntax and the `@new` reduces the overall size by approximately 30% and eliminates the need for any explicit meta-programming. The reduction ratio is more or less maintained over the entire schema. After migration along the lines above (i.e., replacing the individual code fragments by concrete syntax and inlining the results at their use sites), the schema size is reduced from 508 lines to 366 lines.[2] At the same time, the resulting fewer but larger code fragments give a better insight into the structure of the generated code.

---

[1] In ABIR, index variables are declared implicitly, so that the construction of an explicit declaration is not required.

[2] Comparing lines of code is a rather imprecise measurement. After white space removal, the original schema has 7779 characters and the resulting schema with concrete syntax 5538, confirming a reduction of 30% in actual code size.

4 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 [13,2] and the transformation language Stratego [12,14] 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. Apart from the application to Prolog, this work extends [15] with additional transformations on the embedded object code in order to produce code compatible with the legacy AutoBayes system and to support abstractions for second-order variables and fresh variable generation. In this section we give an overview of the components of `concrete-pl`, the transformation system mapping Prolog with concrete syntax to pure Prolog.

4.1 Combining Syntax Definitions

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 of Prolog and ABIR using SDF. Since SDF is a modular syntax definition formalism, combining languages is simply a matter of importing the appropriate modules, as illustrated by the following excerpt from the embedding of ABIR into Prolog:

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

This module allows us to use ABIR Expressions and `Statemements` as Prolog terms, by quoting them with the `|[` and `]|` delimiters. 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. As mentioned above, a general antiquotation mechanism other than the inclusion of meta-variables is not required since Prolog is a relational language and the result of a meta-computation is not uniquely determined.

4.2 Exploding 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 + z ]|
```

is parsed into the abstract syntax tree

```
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. A mixed syntax tree can be translated to a pure Prolog tree by "exploding" embedded tree constructors to functor applications:

```
bodygoal(infix(var("code"),op(symbol("=")),
               func(functor(word("assign")),
                    [func(functor(word("var")),[var("X")]),
                     func(functor(word("plus")),
                          [var("y"),
                           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.

Explosion is defined generically using transformations on mixed syntax trees, i.e., *it* is independent from the object language. The basic transformation is rewriting functor applications in ABIR abstract syntax to functor applications in Prolog abstract syntax. This is expressed by the following Stratego transformation rule:

```
TrmOp : Op#(Ts1) -> func(functor(word(<lower-case>op)), Ts2)
        where <map(trm-explode)> Ts1 => Ts2
```

Several other rules deal with special constructs such as meta-variables and lists. Rewriting the final `Centroid`-fragment on page 6 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)])))))
```

4.3 Custom Abstract Syntax

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. The latter version is less precise, but more compact, since it was designed for direct use. In order to interface schemas written in concrete syntax with legacy components of the synthesis system, additional transformations are thus 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 '])])
```

4.4 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.

5 Conclusions

Syntax matters. 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. Our work on extending the AutoBayes synthesis system with concrete syntax shows that this can result in a considerable reduction in the code size, but more importantly, an improved readability of the code. 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. The goal of having domain experts write meta-programs with concrete syntax depends on more than just concrete syntax. There are more aspects that make meta-programming hard. However, concrete syntax should make it easier rather than more difficult, we expect. Further evaluation of the approach involving domain experts should make clear whether this goal is achievable.

The contribution of this work is twofold. In the first place, we describe an extension of Prolog with concrete object syntax, which is a useful tool for all meta-programming systems using Prolog. The `concrete-pl` tool that implements the mapping back into pure Prolog is available as a general tool for embedding object languages in Prolog.[3] In the second place, we demonstrate that the approach of [15] can indeed be applied to other meta languages than Stratego, and we extend it with object-language-specific transformations to achieve the integration with a legacy systems. This allows a gradual migration of existing systems, even if they were originally designed without support for concrete syntax in mind.

The `concrete-pl` tool is independent of the embedded object language. However, the transformations applied to the Prolog-code after explosion are specific to the implementation of the ABIR embedding. This aspect should be generalized in order to support arbitrary object languages. This mainly requires factoring out the postprocessing transformations on the Prolog-code and make these a parameter of the tool.

References

1. W. Buntine, B. Fischer, and A. Gray. Automatic Derivation of the Multinomial PCA Algorithm. Technical report, 2003. Available at `http://ase.arc.nasa.gov/people/fischer/`.
2. M. van den Brand, J. Scheerder, J. Vinju, and E. Visser. "Disambiguation Filters for Scannerless Generalized LR Parsers". In *Compiler Construction, LNCS 2304*, pp. 143–158. Springer, 2002.
3. W. Buntine. "Operations for learning with graphical models". *JAIR 2*:159–225, 1994.
4. J. Cordy, I. Carmichael, and R. Halliday. *The TXL Programming Language, Version 8*, April 1995.
5. M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and J. Quesada. "The Maude System". In *RTA-10, LNCS 1631*, pp. 240–243. Springer, 1999.
6. A. van Deursen, J. Heering, and P. Klint, (eds.). *Language Prototyping. An Algebraic Specification Approach, AMAST Series in Computing 5*. World Scientific, 1996.
7. B. Fischer and J. Schumann. "AutoBayes: A System for Generating Data Analysis Programs from Statistical Models". *JFP*, 2003. To appear. Available at `http://ase.arc.nasa.gov/people/fischer/`.
8. B. Fischer and J. Schumann. Automating the Analysis of Planetary Nebulae Images, 2003. Submitted. Available at `http://ase.arc.nasa.gov/people/fischer/`.
9. A. Gray, B. Fischer, J. Schumann, and W. Buntine. "Automatic Derivation of Statistical Algorithms: The EM Family and Beyond". In *NIPS 15*, MIT Press, 2002.
10. G. McLachlan and T. Krishnan. *The EM Algorithm and Extensions*. Wiley, 1997.
11. W. Press, B. Flannery, S. Teukolsky, and W. Vetterling. *Numerical Recipes in C*. Cambridge Univ. Press, Cambridge, 2nd edition, 1992.
12. E. Visser, Z.-e.-A. Benaissa, and A. Tolmach. "Building Program Optimizers with Rewriting Strategies". In *ICFP-3*, pp. 13–26. ACM, 1998.
13. E. Visser. *Syntax Definition for Language Prototyping*. PhD thesis, University of Amsterdam, September 1997.
14. E. Visser. "Stratego: A Language for Program Transformation based on Rewriting Strategies. System Description of Stratego 0.5". In *RTA-12, LNCS 2051*, pp. 357–361. Springer, 2001.
15. E. Visser. "Meta-Programming with Concrete Object Syntax". In *Generative Programming and Component Engineering, LNCS 2487*, pp. 299–315. Springer, 2002.

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

16. J. Wielemaker. *SWI-Prolog 3.1 Reference Manual, Updated for Version 3.1.0 July, 1998.* Available at `www.swi-prolog.org`.