Programmatic Building of Models Just for Pretty Printing

Tero Hasu
Helsinki Institute for Information Technology
PO Box 9800, FI–02015 TKK, Finland
tero.hasu@hiit.fi

Abstract

In-memory object models of programs are commonly built by tools to facilitate program analysis and manipulation. We claim that for some applications it makes sense to construct such models for the sole purpose of pretty printing, and explain the reasoning behind our claim in this paper. We also describe a tool we have created to support this approach to pretty printing; the tool derives, from an annotated grammar, both an object-oriented API for model building, as well as corresponding routines for pretty-printing built models.

KEYWORDS: code generation, grammarware, object-oriented programming, pretty printing, program representation

1 Introduction

Pretty-printing capability is required in tools intended to produce readable source code. There are a number of ways one might choose to implement pretty printing. In many tools one requires an abstract syntax tree (AST) of each processed program for analysis and/or manipulation, and in those cases it is natural to write a routine that traverses the AST to emit textual code. However, when implementing a tool that does not transform programs, but rather reads in some input and generates a *new* program based on the input, it is far less clear how pretty printing would best be implemented.

When an AST is not required for code analysis or manipulation purposes, one may choose from a number of alternative approaches to pretty printing. In this paper we explore the idea of constructing an AST-like model of the program anyway, solely for pretty printing purposes. We talk about *model*s, as in object models specifying what is to be printed. We avoid talking about ASTs, so as not to imply that we are interested in the abstract syntax of the target language; we want to know how to print an object, but not necessarily what specific target language construct it represents. We focus on model construction that is done imperatively and incrementally, by writing statements that instantiate and add objects into a model, in any desired order.

The rest of this paper is organized as follows. In Section 2, we discuss our pretty-printing approach in more detail, and consider potential applications. In Section 3 we introduce a specific implementation of the approach, by describing a tool we created to facilitate the implementation of pretty-printable model builders. We look at related work in Section 4, and conclude in Section 5.

2 Constructive Pretty Printing

We refer to our pretty printing approach as *constructive pretty printing* (CPP); with this term we try to emphasize that the defining characteristic of the approach is that one explicitly constructs, bit by bit, a model to be printed. The model objects may be of different native types, and to support incremental model building, they are likely to contain named fields into which to insert more objects in any desired order.

An alternative way to "construct" a model is to essentially just list its contents, in a declarative fashion, allowing for variability by letting lists contain non-literal expressions. This approach is widely used by Lisp programmers at least, and is likely to result in somewhat shorter model building code than in CPP. The *convenience* of writing such code depends largely on how conveniently one can express a named list in the language syntax.

A common code generation approach not involving model building is to use a *template engine* (e.g., Smarty, Velocity, Cheetah). In *template-based code generation* [6], one specifies what to generate using a textual template, but may embed directives within the text to account for variability. The directives—usually expressed in the engine implementation language—are expanded by the engine. The concept of template engines is easy to understand, and most implementations are straightforward to use. These are very desirable properties, but we still argue that alternative solutions—such as CPP—are more suitable for some applications. We believe CPP to be a good approach at least in cases where:

- One prefers to program imperatively. It is natural for people to think in terms of objects and actions.
- One wants to concentrate on abstract syntax, without worrying about delimiters and other notation that can be automatically emitted as required.
- The data to be printed is highly variable. For example, a template engine is of little assistance in printing arithmetic expressions of variable length and content.

- One wants all formatting rules in one place. A major problem with template engines is that formatting information is spread around all the templates being used, and this can easily lead to inconsistencies. In CPP, code specifying *what* to print and *how* to print it is kept separate.
- One requires indentation with variable levels of nesting. With template engines, one must be very careful with whitespace and line breaks to get the formatting right, and even then, producing variable levels of nesting gets difficult. In CPP, the semantics to decide when to indent can be in the model.
- One wants conditional line breaking. If a line is getting too long, one must know *where* it is okay to break it; again, in CPP, there can be sufficient semantics in the model.
- One simply does not want to work with strings. Code with a lot of string manipulation tends to be tedious to write and hard to read. In CPP, such code can be isolated in the printing routines.

One solution that also suits the above cases, but does not quite meet our definition of CPP, is Builder [3]. It is similar to template engines, but in the Builder case, a template is specified as Ruby code that programmatically builds an XML document for pretty printing. The formatting of the output text is left to Builder. Sample building code and the resulting output is shown below.

Listing 1: Printing XML with Builder. [3]

```
Builder::XmlMarkup.new(:target=>STDOUT, :indent=>2).
  person { |b| b.name("Jim"); b.phone("555-1234") }
```

Listing 2: Builder output.

```
<person>
  <name>Jim</name>
  <phone>555-1234</phone>
</person>
```

The Builder approach differs from CPP in that each XML element builder method, by the time it returns, will have caused the printing of the entire element—no model gets built[1]. As a result, one has to specify the entire document at once, in the order in which XML elements are to appear in the document. CPP is more flexible, but that flexibility comes with overhead in constructing and traversing models.

3 qretty

To support the use of CPP, we developed a tool called qretty. It is a Ruby library that makes it possible to dynamically derive, based on an annotated grammar of a language, an object-

[1]At time of writing, support for generating DOM-like structures with Builder is planned.

oriented API for building models representing expressions in the language. qretty also produces code for pretty printing the models according to hints in the grammar.

3.1 Specifying a Grammar

qretty requires a grammar specification as input. The grammar is specified in Ruby, using a provided API, and may be annotated with layout-related information. Some tools try to keep different grammar concerns such as base syntax and layout separate; GPP (see Section 4), for example, does this by having separate grammar and formatting rules for each non-terminal. We chose not to do this in qretty to avoid the extra work involved in maintaining multiple rules per non-terminal.

Below is an example grammar specification, extracted from an as-yet-unreleased tool in which qretty is used for pretty printing C++ type specifiers; we are using the tool to convert GCC-XML generated C++ interface descriptions into a different format.

Listing 3: A grammar specified in Ruby, using the qretty API.

```
crule(:type_spec,
  seq(basic(:typename),
    opt(" ", :declarator)))
crule(:ptr_declarator,
  seq("*", opt(:declarator)))
crule(:ref_declarator,
  seq("&", opt(:declarator)))
crule(:array_declarator,
  seq(opt(:declarator),
    "[", opt(ident(:num)), "]"))
crule(:func_declarator,
  seq("(", opt(:declarator), ")",
    "(", opt(:funcargs), ")"))
arule(:funcargs,
  commalist(:type_spec))
crule(:cv_declarator,
  seq(choice(namlit(:const),
        namlit(:volatile)),
    opt(" ", :declarator)))
crule(:name_declarator,
  ident(:name))
arule(:declarator,
  basic(:declarator))
```

Listing 4: An approximate EBNF translation.

```
type_spec ::=
  TYPENAME
  (" " declarator)?
ptr_declarator ::=
  "*" declarator?
ref_declarator ::=
  "&" declarator?
array_declarator ::=
  declarator?
  "[" NUM? "]"
func_declarator ::=
  "(" declarator? ")"
  "(" funcargs? ")"
funcargs ::=
  type_spec (", " type_spec)*
cv_declarator ::=
  ("const" |
   "volatile")
  (" " declarator)?
name_declarator ::=
  NAME
declarator ::=
  ptr_declarator | ...
```

qretty includes an API for dynamically generating a set of classes corresponding to a grammar specification. Each `crule` gets its own class, whose instances get *field*s (for adding model objects) based on the named terms appearing on the right-hand side of the rule. `arules` do not get a class; instead, their fields are folded into their containing rules. This is an important feature, as many "off-the-shelf" grammars result in deep grammar trees; one can achieve a shallower class hierarchy merely by judiciously using `arule` declarations instead of `crule` declarations.

Task	CPU time (seconds)
Grammar specification analysis (C++ grammar)	1.64
Class hierarchy generation (C++ grammar)	0.17
Model building (C++ declaration)	0.00
Pretty printing (C++ declaration, 10 times)	0.16

Table 1: qretty performance measurements. Times listed are the average of 10 rounds, run on a PC with a 2.80 GHz Pentium 4 processor and 1 GB of memory. The measured program analyzed 210 grammar rules, generated 134 Ruby classes based on the rules, built a model of a short C++ class declaration (2 superclasses, two members), and printed the declaration 10 times. The analysis time does not include parsing performed by the Ruby runtime at program startup.

qretty has a weakness in that it does not scale well to handle large grammars. For one thing, given a complex grammar it can be difficult to create a corresponding class hierarchy that—despite the complexity—provides a usable model building API. Also, qretty is slow in analyzing large grammars, as we noticed trying to use a fairly complete C++ grammar. For related performance figures, look at Table 1.

3.2 Building a Model

Immediately after a class hierarchy has been generated, instances of the classes can be used to form tree structures constituting models for pretty printing. qretty-generated classes have accessor methods for getting and setting child nodes, as one would expect.

Also, as described in Section 3.1, qretty knows the concept of a field, and each field has what we call a *builder setter* method, intended to make model building convenient. Depending on the receiving field, a builder setter decides whether to create a new node object. If so, it determines the type of object to create, passes its arguments to the constructor of the object, and then assigns the resulting object to the appropriate instance variable. If not, it simply uses its argument as the value to assign. The method returns the assigned object, and, if a Ruby block is passed, also passes the object to the block.

Below we give an example of model building, emulating the Builder example of Section 2. In addition to the model building code, both the used grammar specification and the produced output are shown. Two alternative syntaxes for defining a `person` are included to demonstrate how the use of Ruby blocks makes the tree structure of the model clearer.

Listing 5: Grammar specification.

```
crule(:xml_markup, opt(seplist(:person, nl)))
crule(:person, choice(seq("<person>", nl, indent(one_or_more(
  seq(choice(:name, :phone), nl))), "</person>"), "<person/>"))
mfield [:name, :phone], :pname => :@list
crule(:name, seq("<name>", ident(:name), "</name>"))
```

```
cfield :name
crule(:phone, seq("<phone>", ident(:phone), "</phone>"))
cfield :phone
```

Listing 6: Model building code and a pretty printing request.

```
model = ast::XmlMarkup.new
model.person { |b| b.name "Jim"; b.phone "555-1234" }
b = model.person; b.name "Tim"; b.phone "555-4321"
CodePrinter::pp(model)
```

Listing 7: The pretty printed output.

```
<person>
  <name>Jim</name>
  <phone>555-1234</phone>
</person>
<person>
  <name>Tim</name>
  <phone>555-4321</phone>
</person>
```

An obvious problem with qretty is that the produced model building API has no visible interface definition, forcing programmers to deduce it from the grammar specification. qretty uses runtime reflection for code generation, and there presently is no option to generate API documentation either.

At no point during or after model building does qretty validate tree structure [12], nor is there static typing support in Ruby that could be used to prevent builder code from mistakenly placing a node into a context where the grammar does not allow it. We do not perceive this as a big problem, since the preferred way for building models is via builder setters, which automatically create nodes of the correct type.

3.3 Pretty Printing a Model

qretty provides an API via which a model subtree may be pretty printed. Parameters can be passed to choose an output stream, or to specify maximum line width, for instance. The implementation makes use of a *printer method* named `qretty_print` that qretty includes in all the classes it generates. When invoked, a generated printer method matches the receiver's instance data to the corresponding grammar rule to determine what to print.

During printing, an object we call a *printer visitor* essentially walks the model depth-first, passing itself to the printer method of each node; the printer methods are expected to print themselves using the API provided by the visitor. For purposes of flexibility, qretty allows a hand-coded class to be included in a model class hierarchy, as long as it implements pretty printing in a compatible manner; in this case the right-hand side of the corresponding grammar rule need not be given in the grammar specification.

There is no support for having generated printer methods pass or make use of any context information, which makes it somewhat inconvenient to deal with language constructs that print differently depending on context. Should context information be required, one can attempt to encode it in the grammar, or implement select printer methods manually.

4 Related Solutions

There are many tools [1, 7] capable of generating APIs for operating on grammatically structured data, but we do not know of any tool apart from qretty designed to generate classes for the specific purpose of pretty printing. With such specialization, semantics not relevant in the context of pretty printing may be omitted, leading to shorter model building code. One just requires enough object semantics for correct coercion to strings for printing, and enough structural information to enable formatting.

Some grammar-based tools [10, 11] restrict themselves to so-called *structured context-free grammars* [11], which can—in generating classes—be mapped to a class inheritance hierarchy such that the presence of certain kind of non-terminals is implicit in the inheritance relationships, without concrete nodes for those non-terminals needing to appear in ASTs. For similar shallowing of models, a qretty user must enhance the grammar with sufficient annotations. qretty accepts all context-free grammars—the classes it generates do not inherit from each other, nor do they have a statically typed interface; they only form a hierarchy through builder setters' knowledge of what classes should be instantiated for which field.

GPP [8, 9] is one of the most powerful and generally applicable pretty-printing solutions around, but it does not generate a language-specific API for programmatic building of models. The GPP toolset can handle a parse tree, an AST, or—indirectly—source-code text as input, but if one has none of these, a solution similar to qretty might be helpful for building suitable input. GPP is part of the Stratego/XT [14] program transformation toolkit. There are a number of others, such as DMS [2] and CodeWorker [4], and while these all are capable of pretty printing, they are rather large systems, and might be overkill to use just for that purpose. qretty is not a reengineering tool, but it integrates easily within Ruby applications that need to generate *new* code.

CodeDOM [5] provides a fixed API for building models of programs translatable to any of multiple supported languages; qretty does not support multiple target languages for a single model. While CodeDOM has better multi-language rendering support, its weaknesses are that it does not provide elements to represent all language features of all target languages, and that CodeDOM model building code gets quite verbose. qretty avoids these problems by generating target language specific APIs designed for convenient model building.

5 Conclusion

In this paper, we have explored the idea of programmatically constructing models just for pretty printing. We listed a number of situations where applying CPP might be warranted, but any benefits must naturally be weighed against implementation effort. qretty is a tool that can help reduce the effort required, as it is capable of producing grammar-specific class hierarchies and associated pretty-printing routines. Unlike most grammar-dependent software, it even supports languages defined at runtime; a new grammar specification can be created and processed at any time, and the resulting classes can be put into an anonymous module to allow unneeded definitions to be discarded.

Aside from implementation effort, one must also consider whether it is possible to achieve convenient model building in a given case. Either the model building language or the printed language might make it hard to do so. qretty models are built in Ruby, whose syntax seemed quite acceptable for the task, but we would have liked a language feature similar to the JavaScript `with` statement for specifying the default receiver for a set of method calls.

For a friendly model building API, one probably requires memorable naming and a class hierarchy of reasonable depth. qretty's grammar specification language can assist in making class hierarchies shallower than grammar trees. Naming comes fairly naturally for some languages; in our XML example, method names directly map to element names in the XML document schema. There is no immediately obvious way to map C++ language constructs to method names, however, as we found in trying to define a C++ program model building API.

qretty is available for download [13], along with another library called `codeprint`. The latter provides functionality for printing and formatting text, offering control over indentation and line breaking, for instance, and qretty depends on it for low-level formatting tasks. The reader should note that the present bad performance of qretty excludes its use from many applications. It would be possible to drastically improve performance, at least by switching to compile-time code generation, but this is left for future work.

Acknowledgements

We thank Ken Rimey and the anonymous referees for constructive criticism on earlier revisions of this paper, and Pekka Kanerva for feedback on the qretty documentation. We gratefully acknowledge that this work has been funded under E!2023 ITEA S4ALL (Services for All).

References

[1] ANTLR. `http://www.antlr.org/`.

[2] I. Baxter, P. Pidgeon, and M. Mehlich. DMS: Program transformations for practical scalable software evolution. In *Proceedings of the International Conference on Software Engineering*. IEEE Press, 2004.

[3] Builder for Markup. `http://builder.rubyforge.org/`.

[4] CodeWorker. `http://codeworker.free.fr/`.

[5] .NET framework developer's guide: Generating and compiling source code dynamically in multiple languages. `http://msdn.microsoft.com/`.

[6] Jack Herrington. *Code Generation in Action*. Manning, 2003.

[7] H. A. de Jong and P. A. Olivier. Generation of abstract programming interfaces from syntax definitions. *Journal of Logic and Algebraic Programming (JLAP)*, 59:35–61, April-May 2004. Issues 1–2.

[8] Merijn de Jonge. A pretty-printer for every occasion. In *Proceedings of the 2nd International Symposium on Constructing Software Engineering Tools*, Wollongong, Australia, 2000.

[9] Merijn de Jonge. Pretty-printing for software reengineering. In *Proceedings of International Conference on Software Maintenance (ICSM 2002)*, pages 550–559. IEEE Computer Society Press, October 2002.

[10] maketea theory. `http://www.phpcompiler.org/doc/maketeatheory.html`.

[11] The metaprogramming system – reference manual. Technical Report MIA 91-14, Mjølner Informatics, February 2002.

[12] Terence Parr. Translators should use tree grammars. `http://www.antlr.org/article/1100569809276/use.tree.grammars.tml`, November 2004.

[13] qretty. `http://pdis.hiit.fi/s4all/download/qretty/`.

[14] Eelco Visser. Program transformation with Stratego/XT: Rules, strategies, tools, and systems in StrategoXT-0.9. In *Domain-Specific Program Generation*, volume 3016 of *Lecture Notes in Computer Science*, pages 216–238. Springer-Verlag, June 2004.