FINAL version

Building Object Oriented Programs out of Pieces

Richard A. O'Keefe
Computer Science, Otāgo
ok@cs.otago.ac.nz

Abstract

This paper presents a technique for assembling Smalltalk programs out of pieces using propositional Horn clauses. The technique allows the dependencies and restrictions of a piece to be stated inside the piece or outside, allowing components from other dialects to be used. The technique is applicable to any OO language allowing class extensions.

Categories and Subject Descriptors D.3 [*Programming Languages*]: Language Constructs and Features

General Terms Object-Oriented Programming, preprocessing, autoload, Horn clauses

Keywords OOP, preprocessing, autoload, Horn clauses, conditional requirements

1. Background

In 2002 a student wanted to do a PhD on native code generation for Smalltalk [8]. We needed a baseline for comparisons, so this author began work on a straightforward Smalltalk to C compiler called 'astc". The student was unable to get funding, so went elsewhere. The compiler project continued, now with the aim of conforming to the ANSI Smalltalk standard [1] as well. It soon became clear that the speed of generated code was competitive with commercial Smalltalk systems (thanks to clever C compilers), and that this was a pleasant tool for writing actual programs, or would be if only compiling were not so slow.

The compiler does whole-program compilation via C in the manner of Colnet's SmartEiffel [4]. Compilation to C is very fast, but compilation of the resulting C is slow, so including the minimum necessary code is important for speed. It also helps with reliability: you cannot call the wrong method if it is not there. A simple way to select a minimal set of components was needed.

The following table shows how big the system is. The library files are Smalltalk code written for reuse, analoguous to other Smalltalks' initial image. The test files are independent test cases. The example files are independent smallish programs, some written for demonstration purposes and some written for actual use. The RosettaCode files are solutions to RosettaCode.org problems. The test files, example files, and RosettaCode files use the library, but none uses all of it. The lower half of the table shows the compiler and runtime support library; roughly half of all the C code is the BDW garbage collector and its concurrency support. The "req" column shows the number of unconditional includes and "req-if" the number of conditional ones.

#files	lines	SLOC	req	req-if	contents
719	147030	97493	2284	360	Library files.
219	22357	17288	530	0	Test files.
142	23370	15309	438	0	Example files.
655	28600	18884	1028	0	RosettaCode files.
22	27827	18061	94	0	Compiler (.c .h) files.
21	13623	8850	69	26	Runtime (.c .h) files
54	33994	17532	67	235	BDW GC
68	16014	8673	16	53	libatomic_ops

For much of its history, astc was developed on a SunBlade 100. Compiling the whole library to C takes 5 seconds; compiling the C code without optimisation takes another 8 minutes. Compiling *with* optimisation crashes the C compiler, which was not designed to handle 971,000 line source files. To get acceptable compile times, it was vital to compile less C, and that meant compiling just enough Smalltalk.

On a 2GHz Macintosh with OSX 10.11 and Clang 7.3, compilation is faster. The following table shows four test cases: the minimum required for ANSI compatibility, that plus 4410 SLOC of extra methods for collections, that plus an XML library and date and time classes, and the full library plus some test files. The table shows the number of lines of C code generated, the time for astc to generate that C code, and the time for Xcode's C compiler to compile and link the program, without and with optimisation. All times are real time in seconds.

text source	ST SLOC	C lines	astc time	clang time	clang -O2
ansi.st	9625	77396	0.09	6.6	17.0
collections.st	14080	127111	0.12	10.1	28.2
core+XML+time	31011	286202	0.12	21.1	56.9
full library	90209	967150	0.61	57.3	193.8

Example

The library includes random number generation and geometric objects. If you request random.st and geometry.st, then you probably expect a reasonable set of random selection from and generation of geometric objects, which random-geometry.st provides. Conversely, if you request random-geometry.st, random.st and geometry.st are automatically included. This sketches the code before the improvements of this paper. The last section shows the improved version.

```
"geometry.st defines Point Rectangle Circle.."
...
require: 'random-geometry.st' if: 'random.st'

"random.st, defines Random etc."
...
require: 'random-geometry.st' if: 'geometry.st'

"random-geometry.st"
require: 'geometry.st'
require: 'random.st'
...
Circle
  methods:
    atRandom
      ^self atRandom: Random default
    atRandom: aStream
      "Return a random Point in the receiver"
...
```

2. Requirements

- The key ideas of the technique should apply to several languages so that others can benefit from it. It must be straightforward to implement in a language other than the language it is used in. The astc compiler is written in C and does not use the Smalltalk runtime system. The facility described here was prototyped in AWK.
- It should be declarative to make it easier to reason about and use correctly.
- Since the goal is to make programs no larger than they need to be, it should be obvious that the technique selects a minimal set of components.
- In order to test and debug components written for this system, it should be possible to develop them in other Smalltalks and bring them back. That means that it should be possible to express the dependencies of such components in some other file, because other Smalltalks do not support this technique.
- Each included component should be processed just once. This is for simplicity of semantics: the result of building should be equivalent to a single file containing no inclusions or duplicates.
- Cyclic dependencies that do not force a textual cycle should be harmless. The random-geometry example shows why this is useful: requesting random.st and geometry.st is equivalent to requesting random-geometry.st, and that involves a cycle.
- It should cope with a manifestation of the expression problem, where you need method-m-in-class-C if and only if you have both method-m and class-C. This was the key problem that led to the present design. We don't want random numbers just because we've got points, and we don't want points just because we've got random numbers.
- It should support some form of conditional compilation. The system is developed under Solaris and Mac OS X and ported to OpenBSD, Linux, and Cygwin. They are not as compatible as they should be. Some of that can be hidden in C support code, but not all of it.

3. Target programming language characteristics

The technique was designed for Smalltalk in the first instance, but should be applicable to other languages which resemble Smalltalk in relevant aspects. The relevant aspects are

1. a program is a set of classes and/or functions;
2. a class has a *structure* and zero or more *extensions* providing methods;
3. the extensions of a class must follow its structure;
4. the structure of a class must follow the structures of its superclasses;
5. the order of structures, extensions, and functions is constrained only by points 2 and 3;
6. any changes to the syntax of the language (such as macros or user-defined operators precedence) are or may be confined to a single compilation unit.

Suitable programming languages include Smalltalk [1, 8], Common Lisp [11, 19], C♯ (counting partial classes as extensions) [5, sections 8.7.13 and 17.1.4] [16, section 10.2], and Swift (counting partial classes as extensions) [2].

4. Model

- A *library* is a finite map from *file names* to files.
- A *file* is a sequence of chunks.

- A *chunk* is a class chunk, a method chunk, or a require chunk.
- A *class chunk* defines the name, structure, and relationships of a class. For example,

```
AbstractRationalNumber subclass: #Fraction
  instanceConstantNames: 'num den'
```

- A *method chunk* defines one or more methods belonging to a class. For example,

```
Fraction
  methods:
    fractionPart
      ^self pvtClone
        pvtNumerator: (num rem: den)
         denominator: den
```

- A *require chunk* states a dependency between files, analogous to an include directive. For example,

```
require: 'collections.st'
```

There are three kinds of relationships between class (name)s, method (names), class chunks, and method chunks.
Definition: a class chunk defines a class, and a method chunk defines one or more methods.
Use: a class chunk may use one or more classes (as superclass(es), as a pool dictionary, or as an initialisation dependency), and a method chunk uses one class as its container and may use others as global variables.
Ordering: a class must be defined before it can be used as a superclass or pool dictionary, or have a method added to it.

5. Dead code elimination does not help

Since astc takes nearly two orders of magnitude less time than the C compiler, an obvious approach is to compile all the Smalltalk and generate code for only the live classes and methods.

The ⟨classDescription⟩ protocol [1, section 5.3.8] includes #allSubclasses and #name, meaning that any class can be looked up by name at run time. So no classes are dead. The ⟨Object⟩ protocol [1, section 5.3.1] includes #perform: and several other methods for calling a method by name. So no methods are dead. A clever dead code eliminator could still work if it determined that these methods were not used. Unfortunately, they *are* used in the library itself.

6. The C Preprocessor is not declarative

The C [13] preprocessor is almost adequate. The well known technique of using

```
#ifndef  FOO_INCLUDED_
#define  FOO_INCLUDED_
#include "foo"
#endif
```

handles multiple inclusion and cycles, albeit clumsily.

There are three problems. The first is that Smalltalk and C are lexically different. For example, // /* and */ in Smalltalk have nothing to do with comments. That simply suggests using something *like* cpp with different lexical rules. The second is that it is too powerful: the aim for this project was something that would simply aggregate text, not change it. Again, this arguments for something *like* cpp but weaker. The main problem is that the C processor was influenced by the PL/I preprocessor [9, Chapter 21] and the *m4* macro processor [12, 21], and like them it remains an imperative programming language. Macros act as variables, with #define and #undef as assignment statements. A file included in different contexts mat yield completely different contents each time.

At times in the history of astc, m4 [12, 21] and a cpp-like program have been used for experimental purposes. The cpp-like program was used to generate OpenSSL interface code for message digests, before the portability burden of relying on system installations of OpenSSL became too high. The outputs of template instantiation still have to be selected and combined. The cpp-like program is still used for generating some data files.

7. Autoloading is not enough

Autoloading is an old technique [17, section 12.4.4]. It works by constructing an inverted index from class names to file names. When processing a chunk,

- if it has an *ordered* use of a class that is not defined yet, look the class up in the index and compile the corresponding file now;
- if it has an *unordered* use of a class that is not defined yet, look the class up in the index and add the file name to a set of files to be processed later.
- Whenever there is nothing else to compile, remove any file name from that set of files and compile it.

Determining what the dependencies are is language-dependent, but the existence of such dependencies is common to the languages listed in the introduction. The ideal would be a technique that does not *require* autoloading but is compatible with it.

There are two problems with autoloading. The first is that it does not generalise to methods. There are two reasons for that.

- A method might be defined in an ancestral class, but need overriding in a descendant class if some condition is met. The fact that a method is available does not mean there is nothing more to process.
- A library might contain more than one definition of method-m for class-C (such as MacOSX *vs* Solaris).

The second problem is that autoloading does not work well when there are multiple files defining the same class. Again, operating system dependencies are the obvious case.

Having said this, autoloading works very well most of the time. Java compilers do something similar, but the search path mechanism is complex and means that you cannot tell simply by inspecting a file what other files it will use.

8. Ruby autoloading is imperative

Ruby has an autoload feature [20]. It is useful but not innovative. It is an executable statement, meaning that both the name of what is to be loaded and the file name it is to be loaded from can be computed by arbitrary expressions, and that an autoload statement in a file might or might not be executed. In effect, a Ruby autoload command is a delayed executable statement to load a file, conditional on whether a name has been defined.

This makes it difficult to analyse Ruby autoload commands *reliably* other than by having Ruby interpret them.

9. Steps to a solution

Like #include but different

File inclusion seemed like the simplest thing that could work, but the repeated inclusion of #include is unwanted. The chosen syntax is "require: ⟨file-name⟩"; the meaning is that the file will be processed at this point unless it has already been processed.

IF appears

ANSI Smalltalk contains a method for iterating over two sequences at the same time. The file `with.st`[1] defines generalisations of these, including for example folding over two sequences. It is similar in spirit to the ListPair module in the Standard ML Basis Library [6, section 11.21]. The file `lists.st` defines a linked-list class similar to ML's List module [6, section 11.20]. But while the method implementations in `with.st` work well for all the ANSI sequence classes [1, section 5.7], some of them have to be overridden for linked lists, so `with-lists.st` does that.

We want the method (re-)definitions in `with-lists.st` to be processed only if the program needs both the two-sequence iteration methods and linked lists. The C preprocessor would handle this by having you define "feature" macros to test whether a feature was wanted/had been provided. But it suffices to test whether a file has been processed.

The new syntax is "require: ⟨file-name-1⟩ if: ⟨file-name-2⟩", meaning

> if processing of file-name-2 has already begun, and processing of file-name-1 has not begun, process file-name-1 now".

This worked, and proved adequate for managing a library of some thousands of files, but it requires duplication:
`require: 'with-lists.st' if: 'lists.st'`
in `with.st` and
`require: 'with-lists.st' if: 'with.st'`
in `lists.st` so that `with.st` and `lists.st` can be processed in either order. This also introduces coupling between the files.

The basic problem was that if a requirement wasn't triggered right now, it was completely forgotten. The notation was not declarative yet.

Multiple conditions

An unconditional "require: G" occurring in file F is in fact conditional: G is to be required only if F is. That means that "require: G if: H" actually means that G is required if H and F are. So it turned out that non-trivial Horn clauses (explained in the next section) are useful.

The syntax is "require: F_0 if: $F_1 \& \ldots \& F_n$". The meaning is now "if the containing file and F_1 to F_n are all required, so is F_0." If the condition is not satisfied now, the rule must be put aside for later processing.

The other kind of Horn clause

A positive Horn clause says when something should be true. A negative Horn clause says that several things should *not* be true together. This turns out to be useful.

"`forbid:` $F_1 \& \ldots \& F_n$" says that it is an error for the containing file and F_1 to F_n to all be included in the same program.

In some programming languages, like normal Smalltalks, you can replace or redefine a class or method, so simply processing two definitions of the same class would not necessarily be an error. But

```
forbid: 'files-windows.st' & 'files-posix.st'
```

says that the two files must not be used together, and says this even if the files would apparently be compatible (they might use incompatible foreign code, for example).

10. Horn clauses

A propositional formula is false, true, a variable that could be true or false, or formulas composed using not, and, or, implies, and equivalent.

A *model* of a propositional formula is a mapping from its variables to {false,true} such that the formula evaluates to true.

A *minimal model* of a propositional formula is a model such if any variable mapped to true in that model is mapped to false instead, the result is not a model. For example, $\{x \mapsto true, y \mapsto true\}$ is not a minimal model of $(x \vee \neg x) \wedge y$

[1] The files mentioned in this section can be found through `http://www.cs.otago.ac.nz/staffpriv/ok/software.htm`, which links to an earlier snapshot of astc.

but $\{x \mapsto false, y \mapsto true\}$ is. Minimal models are not necessarily unique: $a \wedge b$ is satisfied by both $\{a, \neg b\}$ and $\{b, \neg a\}$.

Any propositional formula can be put into clausal form [14, pages 197–200], which is a set of clauses, each clause having the form

$$h_1, \ldots, h_m \leftarrow b_1, \ldots, b_m$$

where the h_i and b_j are variables, signifying "if all of $b_1, \ldots, b_m$ are true, then at least one of $h_1, \ldots, h_n$ must be true".

A formula and its clausal form have exactly the same meaning. In particular, they have the same models and the same minimal models.

A Horn clause is one in which $m \leq 1$. Conversion to clausal form works for first-order logic, not just propositional formulas, and the notion of a Horn clause applies there as well. The programming language Prolog [3, 10] is based on first-order Horn clauses.

A set of Horn clauses has a *unique* minimal model. In logic programming, the meaning of a pure Prolog program is defined to be this model [15]. Not only that, minimal models for Horn clause sets are *monotonic*: if we have concluded from a set of clauses that something must be true (a file must be included), adding another rule cannot cause us to change our minds.

For our purposes, we only need propositional Horn clauses, and the unique minimal model of a set of propositional Horn clauses can be found in linear time [18].

11. Final Design

A rule "require: F_0 [if: $F_1 \& \ldots \& F_n$]" appearing in file G stands for the Horn clause

$$F_0 \leftarrow G \wedge F_1 \wedge \ldots \wedge F_n$$

A rule "forbid: F_1 & ... & F_n" appearing in file G stands for the Horn clause

$$\leftarrow G \wedge F_1 \wedge \ldots \wedge F_n$$

When one of these rules is encountered, every F_j whose processing has already begun is crossed off. If the right hand side is now empty, the left hand side is processed, otherwise the rule is added to a set of rules. (Multiple occurrences of the same rule are harmless.)

When processing reaches the end of a file, each rule in the set of pending rules is checked, and the file just ended crossed off. As soon as a rule with an empty left hand side has everything in the right crossed off, the error must be reported and processing should stop. When a rule with a non-empty left hand side has everything on the right crossed off, the left hand side should be queued for processing, because more than one rule might be activated at the same time. The set of ready files mentioned in the section on autoloading can be used for this purpose. Indeed, autoloading as described above can be implemented as implicitly generated rules; autoloading and Horn clause requires work well together.

A rule in file F can be moved out and put into another file by adding F as a condition. This is perfectly safe as long as the rule does not express an *ordered* dependency. All the ordered dependences met so far have been unconditional, so the practice of writing all ordered dependencies — mostly avoided by autoloading — then a class thunk then method chunks, then unordered dependencies works well.

12. Does this satisfy the requirements?

The basic approach is language independent. Combining it with (compile-time) autoloading requires limited coupling to the language processor.

The notation is very nearly declarative. The major theoretical weakness is the handling of ordered dependencies. Perhaps something like the Immediate Dominance/Linear Precedence of Generalized Phrase Structure Grammar [7] could be used: "order: $F_1 < F_2$" would mean that *if* F_1 and F_2 are both required, F_1 must be processed first. However, it would be easy to process F_2 before encountering an order rule, which would be unsatisfactory. More work is needed here. At the moment, ordered dependencies are mainly handled by the compile-time autoload technique, but that requires manual insertion of rules when a dependency has more than one provider.

Conditional compilation can be handled by inverting the C preprocessor. Cpp uses feature macros because it cannot ask whether a file has been included. We can use empty files instead of feature macros. Rules like

```
require: 'posix-filename.st' if: 'Posix'
require: 'windows-filename.st' if: 'Windows'
```

work well. A marked difference from Cpp is that there is no #ifndef. All conditional alternatives must be stated positively.

With these improvements, the example in section 1 looks like this:

```
"geometry.st defines Point Rectangle Circle.."
...
require: 'random-geometry.st' if: 'random.st'

"random.st, defines Random etc."
...
"random.st not coupled to the other files "
require: 'random-geometry.st' if: 'geometry.st'

"random-geometry.st"
"Circle extended : autoload geometry.st"
"Random mentioned : autoload random.st"
...
Circle
  methods:
    atRandom
```

```
      ^self atRandom: Random default
    atRandom: aStream
      "Return a random Point in the receiver"
...
```

Because Circle is extended, geometry.st is autoloaded immediately. Because Random is mentioned, random.st is autoloaded after other processing is complete. This much Ruby can do, although the ordering would be different.

Treating require:if: as a logical rule means it only has to be stated in one file. This means that random.st is no longer coupled to the other files, nevertheless, if geometry.st is compiled before random.st, random-geometry.st will automatically be scheduled for compiling when random.st is finished.

Furthermore, treating require:if: as a logical rule means that the conditional rule can be moved out of geometry.st as

```
require: 'random-geometry.st' if:
  'random.st' & 'geometry.st'
```

decoupling geometry.st.

In a collection of over 1700 files, there were originally over 4600 require: lines of which 360 were conditional. Adding autoloading cut the average number of require: lines from 2.5 per file to 1.2 per file. Treating requires as logical rules rather than imperative commands reduced the number of conditional requires by half. of conditional requires in half, and moreover allowed coupling to be reduced: it is OK for a specialised class to know that it might have to plug into a general service (if that's wanted); it's not good for the general service to have to know about all the specialised classes.

This simple approach satisfies the requirements and has proven straightforward to use.

References

[1] *Programming Languages — Smalltalk*. American National Standards Institute, May 1998.

[2] *The Swift Programming Language (Swift 2.2)*. Apple Inc., June 2014. `https://swift.org/documentation/`.

[3] William F. Clocksin and Christopher S. Mellish. *Programming in Prolog*. Springer-Verlag, 5th edition, 2003.

[4] Dominique Colnet and Olivier Zendra. Optimizations of eiffel programs: Smalleiffel, the gnu eiffel compiler gruop. In *ACM SIGPLAN International Symposium on Memory Management (ISMM'98)*, pages 154–165, October 1998.

[5] *C♯ Language Specification*. European Computer Manufacturers' Association, June 2006.

[6] Emden R. Gansner and John H. Reppy, editors. *The Standard ML Basis Library*. Cambridge University Press, 2004.

[7] G. Gazdar, E. Klein, G. K. Pullum, and I. A. Sag. *Generalized Phrase Structure Grammar*. Basil Blackwell and Harvard University Press, 1985.

[8] Adele Goldberg and David Robson. *Smalltalk-80: The Language and its Implementation*. Addison-Wesley, 1983.

[9] *Enterprise PL/I for z/OS and OS/390 Language Reference Version 3 Release 2.0*. IBM, September 2002.

[10] *Information technology — Programming languages — Prolog – Part 1: General core*. ISO/IEC, June 1995.

[11] Guy L. Steele Jr. *Common Lisp: The Language*. Digital Press, 1990.

[12] Brian W. Kernighan and Dennis M. Ritchie. The *m4* macro processor. Technical report, Bell Laboratories, 1977.

[13] Brian W. Kernighan and Dennis M. Ritchie. *The C Programming Language*. Software Series. Prentice-Hall, 2nd edition, 1988.

[14] Robert Kowalski. *Logic for Problem Solving*. Artificial Intelligence Series. Elsevier North Holland, 1979.

[15] John W. Lloyd. *Foundations of Logic Programming*. Symbolic Computation. Springer-Verlag, 2nd edition, 1987.

[16] *C♯ Language Specification 5.0*. Microsoft, 2012. `https://www.microsoft.com/en-us/download/details.aspx?id=7029`.

[17] David A. Moon. *MacLisp Reference Manual*. MIT, April 1974. `http://www.softwarepreservation.org/projects/LISP/MIT/Moon-MACLISP_Reference_Manual-Apr_08_1974.pdf`.

[18] Richard A. O'Keefe. Finite fixed-point problems. In *Proceedings of the 4th International Conference on Logic Programming*, 1987.

[19] Kent Pitman. American national standard for programming language common lisp (x3j13). Technical Report ANSI INCITS 226-1994 (R2004), American National Standards Institute, 2004.

[20] *Ruby Core 2.0.0: Module*. Ruby documehtation project, February 2016. `http://ruby-doc.org/core-2.0.0/Module.html`.

[21] Kenneth J. Turner. Exploiting the *m4* macro language. Technical Report CSM-126, Department of Computer Science and Mathematics, The University of Stirling, September 1994. `http://www.cs.stir.ac.uk/~kjt/research/pdf/expl-m4.pdf`.