Tools for Automatic Interface Generation in Scheme

Augustin Lux
Project Orion
INRIA - Sophia Antipolis, France *
Email: Augustin.Lux@sophia.inria.fr

Abstract

We first present present two basic tools for the formal manipulation of C/C++ programs: a general syntax analyzer for C++, producing Abstract Syntax Trees, and an "unparser" translating ASTs to program text. These two Scheme programs are the basis of a third tool providing the automatic integration of C and C++ code into Scheme. The hard part of this last problem is wrapper generation which is solved entirely on the AST level. The first two of these tools are completely general, and should be useful for numerous applications. The wrapper generator, currently used in our local Scheme implementation, can easily be adapted to other implementations. The strong points of our tools are: handling of (almost) complete C++ including overloading, operators, templates; use of ASTs for C++ programs; and the availability as Scheme code.

1 Automatic Interface Generation

Scheme, inasmuch as it is an incarnation of λ-calculus, is a completely open-ended language. From an abstract point of view, there is no reason why procedures and methods written in C or C++ cannot just be loaded into a Scheme environment. However, technically this is a hairy problem; Scheme systems usually propose a foreign function interface requiring quite a bit of manual work. Some seven years ago, in their paper on "Sweet Harmony", Davis et al. suggested how easy and transparent this connection should be (and also listed some of the inherent difficulties). In particular, they proposed that the "foreign function interface" procedures necessary to link C-code into a Lisp interpreter be produced automatically, from the information given by C++ header files. This idea has been taken up in several Scheme systems, as well as for other languages, like Perl, Tcl, Python. The home page of the "Simplified Wrapper and Interface Generator" SWIG gives a useful overwiew. However, all these program generators impose some limitations on the complexity of the C++ code that can be automatically handled.

When we wanted to add to our Scheme system [1] the facility of loading C++-code, we quickly gave up the idea to reuse some existing program for interface generation, and decided to implement a new solution based on a simple principle: the use of abstract tree structures to analyze and synthesize programs. In this perspective, the programs to convert between source text and abstract syntax trees are just utilitarians; indeed they are. However, realizing such programs as general tools, independent of any application, implies a large amount of work.

[1] our local Scheme platform is intended for use in Robotics, Ai, and computer VIsion, whence the acronym Ravi.

2 Parsing and Unparsing C++-code

Abstract syntax should eliminate syntactic sugar from a program, and make program structure evident as a tree. Our abstract syntax is based on 4 fundamental concepts (which are by no means specific to C): expression, type, declaration, and instruction.

We present ASTs through 2 example declarations, the complete definition is given in an internal paper. The general form of a declaration is

```
(c-declare storage-class name type [opt-fields])
```

Example 1: `float (*tab[5])();`
has the abstract form

```
(c-declare #f tab (c-array (* (-> () float))) 5)
```

Example 2. The declaration

```
static float x=sin(pi/2), *px=&x;
```

has the abstract form

```
((c-declare static x float (sin (/ pi 2)))
 (c-declare static px (* float) (& x)))
```

One particular property of the parser is that it can parse program fragments; it doesn't care about undeclared identifiers, as long as it knows all type identifiers (it can even guess on these). This means preprocessor directives, especially #include-directives, may be handled or not, according to several program switches. Flexibility concerning these points is very important for a parsing tool, because program manipulation/synthesis applications have particular needs which are very different from those of a compiler.

The unparser tool translates abstract tree structures into C++ files, that can then be compiled and linked with Scheme code (or used in any other way).

3 Program Synthesis for Interface Generation

Using the tools described in the preceding section, the interface generator works with abstract trees only, which makes the problem much more manageable. Generally speaking, we have to generate one "interface-function" for each C-function or C++-method, and to introduce C++-type information in the Scheme environment. Program generation is

facilitated by special Scheme syntax for C++ code-patterns, very much like the backquote facility for Scheme.

As an example, here is the interface function generated for the constructor of a class *Point*, defined as follows:

```
class Point{
public: Point (void);
        Point (int, int);
        Point (Point&);
   ...
```

The constructor shows the handling of overloading, using systematic type tests to guarantee correct argument types:

```
void Sc_Point_Sconstructor()
{
  ScVal * FB = & VS_ElemQ(- GetPar());
  int nbarg = ScMV::CI;
  if(nbarg == 0)
    SetResult(TypeC(TypeC_no_1,(void *)(new Point())));
  else if(IsFixNum(FB[0]))
    {
      int param_0 = GetFixVal(FB[0]);
      if(nbarg == 2)
        if(IsFixNum(FB[1]))
          {
            int param_1 = GetFixVal(FB[1]);
            SetResult(TypeC(TypeC_no_1,
                      (void *)(new Point(param_0,
                                         param_1))));
          }
        else goto bad;
      else goto bad;
    }
  else if(CheckTypeC(FB[0],TypeC_no_1))
    {
      Point * param_0 = (Point *)GetCPtr(FB[0]);
      if(nbarg == 1)
        SetResult(TypeC(TypeC_no_1,
                  (void *)(new Point(* param_0))));
      else goto bad;
    }
  else goto bad;
  return;
  bad:
Errorf("bad args for function %s",
       "Point::constructor");}
```

At runtime, C++-data are connected via handles to Scheme's heap; these handles are of the Scheme-type *c-object*, which gives access to information for dynamic type-checking, garbage collection, and for printing. These informations are given to the generator in a separate file. In particular, code can be generated for reference counting, or to delete objects when handles are garbage collected.

4 Conclusion

Because we wanted tight integration between Scheme and C++ in our Scheme platform we have developed a set of new tools, all written in Scheme, for program synthesis in general and for wrapper generation in particular.

All these programs tackle (almost) the full generality of real world programs: overloading, operator definitions, templates, preprocessor directives. Important information that cannot be deduced from header files may be supplied in a separate file, or in the form of special comments.

The interface generator is now used on a routine basis, especially for large programs in image processing containing several thousand methods.

The abstract tree structure is completely independent of any application, we have also used it, for instance, to generate XML-interfaces, and to facilitate the access to our programs for people who dont't know Scheme. Indeed, by using the C++-parser instead of the Scheme read function, we are able to mimick something like a C++ interpreter, blurring the syntactic differences between Scheme and C++.

For the source code of the programs, see http://www-prima.inrialpes.fr/Ravi

5 References

[1] www.swig.org/index.html
Simplified Wrapper and Interface Generator
[2] http://www-prima.imag.fr/Ravi
[3] H.Davis et al.: Sweet harmony: the Talk/C++ connection. 1994 ACM Lisp Conference, p. 121