Olmar: manipulating C and C++ abstract syntax trees in Ocaml

Hendrik Tews*
Radboud Universiteit Nijmegen, The Netherlands
http://www.cs.ru.nl/∼tews

June 4, 2007

Olmar is a branch of the Elsa C++ parser that adds the ability to marshal the abstract syntax tree of any C or C++ input as Ocaml variant type to the disk. The marshaled abstract syntax tree can later be processed with a pure Ocaml program. Alternatively additional Ocaml code could also be linked directly into the elsa parser. This paper describes Olmar, the technology used inside it and the first Olmar application: `ast_graph` for visualizing C and C++ abstract syntax trees.

1 Introduction

For applying formal methods to C++ programs one needs sooner or later access to the abstract syntax tree (Ast) of C++ programs. For applications in formal methods one mainly needs to inspect the Ast and walk over it. It is my strong believe that this sort of programming is best done with pattern matching over an ML-like variant type. However, freely available C++ parsers are written in C or C++ and there is no pattern matching facility for the internally used data structures.

C++ is terribly difficult to parse. The grammar is not LALR(1), but even with general LR-parsing techniques there remain ambiguities in the grammar that can only be resolved with some semantic analysis. Developing a new C++ parser in a functional programming language is therefore no practical option.

In order to solve the need for a pattern-matchable Ast of C++ programs I decided to mount an Ocaml back-end to an existing, free C++ parser. The Ocaml back-end would rebuild the internal data structures of the C++ parser as an Ocaml variant type.

*The author has been supported by the European Union through PASR grant 104600.

I decided to use Ocaml as pattern-matching language because I am very familiar with Ocaml and its foreign language interface.

In the search for a free C++ parser I came across Elsa [McP], which is now part of the Oink static analysis tools [WCM]. Elsa is a C++ parser that includes type-checking, semantic disambiguation and even template instantiation. Scott McPeak, the author of Elsa, already noticed that ML-style variant types are the right data structures for abstract syntax trees. Unfortunately this knowledge did not let him abandon his most beloved C++. Instead he wrote a preprocessor, called Astgen, that translates ML-style variant type descriptions into a C++ class hierarchy. Elsas internal abstract syntax tree is to a large extend described in the Astgen input language. One can therefore easily add functionality in the Elsa Ast by doing some Astgen meta programming.

The rich functionality and the possibility of saving work with Astgen meta-programming led to the decision to reuse Elsa as C++ parser. The Elsa and Oink maintainers are very suspicious of Ocaml. Additionally, they were very ineffectively organised before the resignation of Scott McPeak in March 2007. I was therefore unable to contribute any of the new functionality back to the Elsa distribution. The new Ocaml back-end of Elsa is therefore available as a branch of the Elsa development under the name Olmar at http://www.cs.ru.nl/∼tews/olmar.

The functionality of Olmar is mostly complete: All Ast data that I am aware of is translated into Ocaml. There exist one real Olmar application: ast_graph for visualising Asts, see Figure 1 for an example. Additionally, there is check_oast for consistency checking of the generated Ocaml data and count-ast, a simple demo application. In the Robin project [Tew07] we plan to use Olmar to develop a semantics compiler, that translates C++ into its semantics in Pvs.

In the following I talk a lot about abstract syntax trees of C++ programs. They will appear as C++ data structures inside the Elsa parser or as Ocaml data structures. I will refer to these different incarnations as the *Elsa Ast* and the *Ocaml Ast*, respectively. I will refer to the process of translating an Elsa Ast into an Ocaml Ast as *Ast reflection into Ocaml.*

Section 2 describes the internals of Olmar, Section 3 tells how to use Olmar, Section 4 is about ast_graph, the visualisation program and Section 5 describes the Olmar API.

2 Architecture and Technology

The original Elsa distribution is split into 4 subdirectories.

smbase Library with basic data and container types. Although Elsa is of course written in C++, it does not use the standard template library. It only relies on the stuff provided from smbase.

ast The Astgen tool, which generates a C++ class hierarchy from an ML-style variant type description. The code generated by Astgen includes many utility functions and various visitors (if requested in the Astgen source). The data type for the abstract syntax tree of Astgen input files is also generated by Astgen.

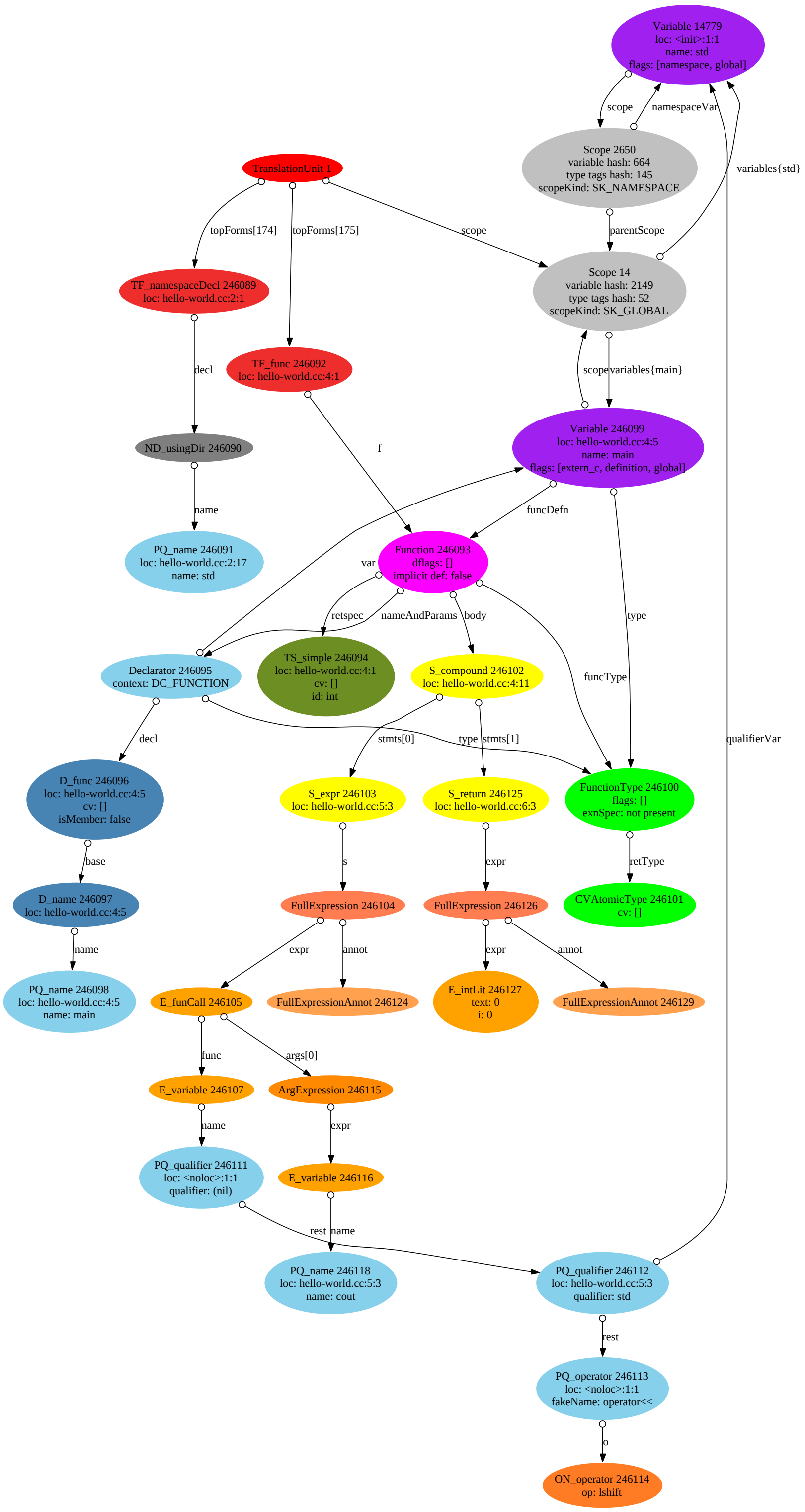

Figure 1: Part of the abstract syntax tree of the hello world program from page 9.

elkhound A parser generator for generalised LR parsing. The Elsa parser is generated by Elkhound. Elkhound can generate parsers in C++ or Ocaml.

elsa The Elsa parser and type checker. Here are the Astgen sources for the Elsa syntax tree and the C++ grammar for Elkhound. Further there are more than 1000 test cases in the subdirectory in.

Olmar contains a fifth subdirectory.

asttools Ocaml sources for ast_graph, check_oast and the count-ast example. There are some general purpose Olmar modules that probably every Olmar application will reuse. The Ocaml code for the Ast reflection into Ocaml mostly resides in the elsa and in the ast subdirectories.

I considered three approaches for obtaining abstract syntax trees of C++ programs in Ocaml. The first two have important deficiencies. Olmar uses the third approach.

Ocaml back-end of Elkhound With the Ocaml back-end of Elkhound one could use the Elsa C++ grammar in order to let Elkhound generate a C++ parser in Ocaml. One would however loose Elsas disambiguation, the type checking and the template instantiation.

XML Elsa can dump the internal Ast in XML format. A promising idea was to let Ocaml read such XML files. However, PXP, the Ocaml XML library from Gerd Stolpmann [Sto], generates an Ocaml object hierarchy from the XML input. Transforming such an object hierarchy into a variant type requires, precisely like an event based PXP pull parser, some kind of high-level reparsing of the Ast, something that I wanted to avoid by using an XML library. What is missing is a tool that reads an XML document type definition and generates from it both a suitable variant type and a XML parser that reads conforming XML files into the variant type.

Ocaml reflection capabilities for Ast nodes The Elsa Ast class structure is equipped with an additional Ast traversal function that translates every Elsa Ast object into a corresponding Ocaml value. The construction of Ocaml values works bottom-up. Astgen generates various visitors, but the visitor methods have always return type `void`. Therefore these visitors cannot be used because the Ocaml values of the child nodes must be returned to the parent.

The data in the Elsa Ast can be divided into three parts:

Syntax nodes The syntax nodes directly represent the input file. The corresponding C++ classes are almost completely generated by Astgen.[1] There are altogether 32 different types of syntax nodes.

[1] The exception is `D_attribute` that is manually defined as a special case of `D_grouping` for the `IDeclarator` nodes.

Type nodes Type nodes are built by the type checker. They do not represent concrete syntax. The class hierarchy for type nodes is defined manually (there exist plans to refactor the the data of the type nodes into Astgen controlled classes). There are 10 different type nodes.

Plain data The remaining primitive data in the Ast, mostly enumerations, but also strings and integers. There are 16 enumeration types with together 219 enum constants.

The types of the syntax and type nodes can again be distinguished into

Structured node types where the node type is split into several subtypes. For instance the node type of statements has 22 subclasses, for every statement one. The 20 structured node types split into 142 variants.

The structure of the syntax nodes has always height two, that is, there is one general node type, such as statement, that is split into several node subtypes, such as `S_if`, `S_switch` and so on. However, these node subtypes are not again split into different variants. These types are best mapped to an Ocaml variant type, where every node subtype gives rise to one constructor.

The type nodes are handwritten, not bound to Astgen limitations, and have sometimes a more complicated structure. I nevertheless map every node super type to an Ocaml variant. I thereby loose some structure. For instance `AtomicType` has a `NamedAtomicType` variant, which is split again into five variants. In the Ocaml Ast the type `NamedAtomicType` does not exist, it corresponds to five of the six constructors of `AtomicType`.

Flat node types such as the node type for whole translation units (consisting of just a list of `TopForm`'s and optionally a scope) that are not split into different cases. There are altogether 22 flat node types. In the Ocaml Ast, flat node types are represented as tuples or records.

As said before, reflection into Ocaml works in principle bottom-up, combining the Ocaml values of the child nodes into a new Ocaml value for the current node. However, there are the following complications.

Null pointers In C++ a pointer to a child node can of course be `NULL`. In comments in the source code the child links of the Elsa Ast are relatively well classified into *nullable* and *non-nullable* ones. In the Ocaml Ast nullable child node links are wrapped in an option type. To minimise the number of options I started with all child nodes non-nullable and added option wrappers for all the segmentation faults that the Elsa regression tests produced. Currently there are 118 option wrappers among the 329 links to child nodes in the Ocaml Ast type.

Sharing To save memory and for resolving cycles (see next point) Ocaml reflection must preserve sharing of nodes in the Elsa Ast. Therefore every Ast node holds an Ocaml value that is initialised with the corresponding Ocaml Ast node when it has been computed. This value is returned when the node is visited a second time.

These pointers from Elsa to Ocaml Ast nodes must be registered with the Ocaml garbage collector as *global roots*. For big Asts they are the source of a serious performance problem. On every minor collection *all* global roots are scanned by the garbage collector, even if the pointed value is already in the old generation. To improve performance Olmar changes the minor heap size from 32KB to 8MB.

Cycles The syntax nodes alone form a real tree, that is, there are no cycles in the Elsa Ast that consist of syntax nodes only. The type nodes, however, add lots of cycles. Strictly speaking the Elsa Ast is only a connected directed graph.

There are simple cycles between the type nodes themselves, for instance every declared entity contains a pointer back to its scope and the scope, of course contains a pointer to all its declarations. There are also cycles between type and syntax nodes, because certain syntax nodes have a link to their type and certain type nodes have a link to the syntax from which they stem.

One could have avoided the problem with cycles by only reflecting the syntax nodes to Ocaml. Then, at least for some applications, one would have to derive type information a second time on the Ocaml side. Alternatively one could try to reflect the type nodes to Ocaml but *somehow* without the cycles. However, the fields in the Elsa Ast cannot be clearly divided into cyclic and non-cyclic ones. A field that closes a cycle in one situation does not participate in any cycle in different situations. To avoid cycles it is necessary to at least partly revise the type nodes. This is far beyond my current level of understanding of the Elsa Ast.

When constructing the Ocaml Ast cycles are resolved as follows: From every possible cycle I select one edge as the *cycle closing edge*. Those cycle closing edges are wrapped in a `option ref` type in the Ocaml Ast. The tree traversal that constructs the Ocaml Ast never follows a cycle closing edge. Instead, the corresponding Ocaml field is initialised with `ref None`. Additionally this reference cell and the target node of the cycle closing edge are registered in the list of *broken cycles*. When the tree traversal finishes I pick a reference cell and an Elsa Ast node from the list of broken cycles. I restart the traversal on the node (which might add more to the list of broken cycles) and update the reference cell after it has been finished. The Ocaml Ast generation is complete when the list of broken cycles becomes empty.

To determine the cycle closing edges I also used the Elsa regression tests. I added code for cycle detection and started with no cycle closing edges. Every time a cycle was detected by the regression tests I investigated the situation and determined a new cycle closing edge.

Correctness and type safety of the Ocaml Ast Although an Ocaml call-back is used for value construction there are plenty possibilities to construct an incorrect Ocaml value. The Ocaml marshalling functions that are used to read and write the Ast to disk are not safe. An incorrect Ocaml Ast value could therefore lead to crashes of pure Ocaml programs.

I first used a branch of the Ocaml system for type safe unmarshalling [HMC] for testing. However, this code suffers from serious performance problems. It takes more than 4 hours to check 7MB of Ocaml Ast data. I therefore eventually wrote my own Memcheck module [Tew06], which only needs 50 seconds to check the 7MB. The check_oast utility distributed with Olmar relies on Memcheck to type check Ocaml Asts. The functionality can easily be integrated in any Olmar application.

The tree traversal for the reflection into Ocaml performs the following operations:

1. Register local variables with the Ocaml garbage collector.
2. Return immediately if the Ocaml Ast value has been constructed before.
3. If not done yet, obtain a handle to the Ocaml value construction function for the type or constructor corresponding to this node.
4. Cycle check: test that the current node is not on the stack of currently visited nodes and add it to the stack.
5. Compute the Ocaml Ast nodes for all children. Children that are on a cycle closing edge are treated as described above.
6. Construct the Ocaml value for this node with the function from point 3.
7. Remove this node from the stack of currently visited nodes and return.

Using an Ocaml call-back to allocate and fill the value in Point 6 has the advantage that the Ocaml reflection code does not need to worry about the encoding of variant type constructors in Ocaml (which depends on the order in the Ocaml type declaration). The treatment of cycles could be simplified a lot if the Ocaml value would be allocated before visiting the children and filled afterwards without an Ocaml call-back. A future version of Olmar might adopt this approach. However, initially I was too much afraid of incorrect Ocaml Asts.

3 Using Olmar

Olmar is available with a BSD license. For download and installation see the Olmar website http://www.cs.ru.nl/ tews/olmar/.

To use Olmar there are two options:

1. Let the Elsa parser ccparse save the Ocaml Ast into a file and have a stand-alone application that later reads it. The latter application can be a pure Ocaml program, enjoying all the nice properties of statically checked type-safe programs.

2. Link the application into the Elsa parser ccparse and call it from ccparse's main function.

I prefer Option 1 because of the easier development, the type-safety bonus and because the additional overhead is negligible. Therefore I will only provide some hints for Option 2 at the end of this section.

The following steps are necessary to use the abstract syntax tree of a file input.cc in a stand-alone Ocaml application:

1. The Elsa parser does not contain a pre-processor. Therefore one needs to run the preprocessor from a standard C++ compiler, for instance for GCC:

   ```
   g++ -E -o input.ii input.cc
   ```

   This saves the pre-processed code in file input.ii.

2. Run the Elsa parser with Ocaml Ast generation enabled. Ocaml Ast generation can be enabled either via the option `-oc file`, which expects a file name for saving the Ocaml Ast. Alternatively one can supply the tracing Option `-tr marshalToOcaml`. Then the Ocaml Ast filename is derived from the input by appending ".oast". For instance

   ```
   ccparse -oc input.oast input.ii
   ```

   will save the Ocaml Ast in input.oast.

3. Optionally type-check the generated Ocaml Ast with check_oast. This will check if the Ocaml Ast is a valid image of the expected type of Ocaml Asts.

   ```
   check_oast input.oast
   ```

4. Load the Ocaml Ast into the target application. This can be done with the function Oast_header.unmarshal_oast. Ocaml Ast files contain a string with version information on the first line. It follows data written by the marshalling function from the Ocaml library. First the number of syntax tree nodes as an Ocaml integer and then the real Ast as marshaled value of type annotated translationUnit_type.

Stand-alone applications with integrated Elsa parser Technically the ccparse from Olmar is a combined C++/Ocaml application with C++ main function, where Ocaml code is only called as call back from C++. The easiest way to add more Ocaml code is to add the relevant source files to the CCPARSE_ML makefile variable in the elsa subdirectory. One only has to register the new entry points with Ocaml's call back facility during module initialisation (or put a hook into the caml_callbacks module) and add code to elsa/main.cc to call the new facilities. The Ocaml Ast is build in function marshal_to_ocaml in main.cc.

4 ast_graph: visualising C++ syntax trees

ast_graph is currently the only real Olmar application. To use it one follows steps 1 and 2 of the preceding section to generate the Ocaml Ast file and applies ast_graph to the Ocaml Ast file. ast_graph will pretty print the syntax tree in the dot file format, which can then be further processed with the graphviz utilities [Gra].

Visualisations of complete Asts are unwieldy huge for the following reasons:

- For typical C++ files only a fraction of the code belongs to the file itself. The overwhelming part comes from included files. For instance with gcc 4.1.2 including iostream adds about 30,000 lines of which about one half is real code containing more than just white space or a single brace.

- There are about 400 built-in functions. Therefore even the empty input file contains more than 1000 nodes for built-in functions.

I have not yet found suitable software to display graphs in the order of 100,000 nodes. All of zgrviewer, ghostview, xfig fail for huge graphs or do not provide efficient functionality to scroll and zoom in such large displays. ast_graph provides therefore options to select the nodes that are pretty printed.

As example consider the following simple hello world program.

```
#include <iostream>
using namespace std;

int main(){
  cout << "Hello World!\n";
  return 0;
}
```

It abstract syntax tree contains 306,768 nodes. Figure 1 was generated with the following commands.

```
g++ -E -o hello-world.ii hello-world.cc
ccparse -oc hello.oast hello-world.ii
ast_graph -loc hello-world.cc -dia 2 hello.oast
dot -Tps nodes.dot -o hello.eps
```

The two options for ast_graph have the following effect: First all nodes with location information from the file "hello-world.cc" are selected. Then all nodes reachable with two steps (either both forward or both backward) are added. Using -dia 3 gives already a huge graph with about 700 nodes, because it adds the successor nodes of the node "Scope 2650". There is one such successor node for every element in the name space std.

5 Olmar programming interface

The Ocaml Ast has type `annotated translationUnit_type`. The Ocaml type definition is distributed over the files elsa/ast_annotation.ml (type `annotated`), elsa/cc_ml_types.ml (enumerations for syntax nodes), ml_ctypes.ml (enumerations for type nodes) and cc_ast_gen_type.ml (syntax and type nodes). All Ocaml node types are polymorphic in order to simplify extensions of the Ast: One simply has to replace the type `annotated` with something more complex. The type `annotated` currently contains an integer that uniquely identifies the Ast node and (the pretty much useless) memory address of the corresponding Elsa Ast node.

The unique node integers are guaranteed to be in the interval $[1 \ldots \mathit{node\_count}]$, which implies further, that all integers in this interval are actually identifying some node. The dense numbering of Ast nodes can be used to store the Ast nodes in an array. The module asttools/superast.ml provides an Ast node super-type and functions to store a syntax tree into an array.

Olmar applications can either be written as recursive function over the syntax tree or as iteration over the Ast array. The count-ast demo application is available in both version: as recursive function (asttools/count-ast.ml) and as iteration (asttools/count-ast-new.ml).

The recursive function approach is complicated because of the cycles in the Ast. All children links of an `option ref` type might lead into a cycle. However, some parts of the Ast might only be accessible via such an `option ref` link. The module asttools/dense_set.ml provides the type of sets of positive integers, based on bitmap implementation. It is used in count-ast and ast_graph to avoid cycles.

Last but not least, the module asttools/ast_util provides utility functions to extract the node annotation and location information (when present).

6 Conclusion

This paper presents the internal and the programming interface of Olmar, a branch of the Elsa C++ parser that adds functionality to export a C++ abstract syntax tree into Ocaml as a variant type.

An interesting idea to enhance Olmar is to apply the here described Ocaml reflection to the abstract syntax tree of Astgen files and to use it to export the abstract syntax tree of the Elsa Astgen sources to Ocaml. This would allow to write Ocaml programs that generate most of the tedious Ast traversal code inside ast_graph and count-ast.

Regrettably, a merge of Olmar back into the official Elsa distribution seems not very likely at the moment because of technical but also social issues. See the oink-devel mailing list for lengthy discussions.

References

[Gra] Graphviz – graph visualization software. available from www.graphviz.org.

[HMC] G. Henry, M. Mauny, and E. Chailloux. Type-safe unmarshalling for objective caml. Web site http://www.pps.jussieu.fr/~henry/marshal/.

[McP] Scott McPeak. Elsa: The Elkhound-based C/C++ Parser. http://www.cs.berkeley.edu/~smcpeak/elkhound/sources/elsa/.

[Sto] G. Stolpmann. The xml parser for o'caml: Pxp. Web site http://www.ocaml-programming.de/programming/pxp.html.

[Tew06] H. Tews. Memcheck: runtime typechecking for ocaml. Web site http://www.cs.ru.nl/~tews/memcheck/, December 2006.

[Tew07] H. Tews. Micro hypervisor verification: Possible approaches and relevant properties. accepted at the NLUUG Voorjaarsconferentie 2007, February 2007. Available from `www.cs.ru.nl/~tews/science.html`.

[WCM] Daniel S. Wilkerson, Karl Chen, and Scott McPeak. Oink: a Collaboration of C++ Static Analysis Tools. http://www.cubewano.org/oink/.