The following paper was originally published in the
Proceedings of the Conference on Domain-Specific Languages
Santa Barbara, California, October 1997

ASTLOG: A Language for Examining Abstract Syntax Trees

Roger F. Crew
Microsoft Research

For more information about USENIX Association contact:

1. Phone: 510 528-8649
2. FAX: 510 548-5738
3. Email: office@usenix.org
4. WWW URL: http://www.usenix.org

ASTLOG: A Language for Examining Abstract Syntax Trees

Roger F. Crew

Microsoft Research
Microsoft Corporation
Redmond, WA 98052
rfc@microsoft.com

Abstract

We desired a facility for locating/analyzing syntactic artifacts in abstract syntax trees of C/C++ programs, similar to the facility `grep` or `awk` provides for locating artifacts at the lexical level. Prolog, with its implicit pattern-matching and backtracking capabilities, is a natural choice for such an application. We have developed a Prolog variant that avoids the overhead of translating the source syntactic structures into the form of a Prolog database; this is crucial to obtaining acceptable performance on large programs. An interpreter for this language has been implemented and used to find various kinds of syntactic bugs and other questionable constructs in real programs like Microsoft SQL server (450Klines) and Microsoft Word (2Mlines) in time comparable to the runtime of the actual compiler.

The model in which terms are matched against an implicit current object, rather than simply proven against a database of facts, leads to a distinct "inside-out functional" programming style that is quite unlike typical Prolog, but one that is, in fact, well-suited to the examination of trees. Also, various second-order Prolog set-predicates may be implemented via manipulation of the current object, thus retaining an important feature without entailing that the database be dynamically extensible as the usual implementation does.

1 Introduction

Tools like `grep` and `awk` are useful for finding and analyzing lexical artifacts; e. g., a one-line command locates all occurences of a particular string. Unfortunately, many simple facts about programs are less accessible at the character/token level, such as the locations of assignments to a particular C++ class member. In general, reliably extracting such syntactic constructs requires writing a parser or some fragment thereof. And after writing one's twenty-seventh parser fragment, one might begin to yearn for a more general tool capable of operating at the syntax-tree level.

Even given a compiler front-end that exposes the abstract syntax tree (AST) representation for a given program, there remains the question of what exactly to do with it. To be sure, supplying a C programmer with a sufficiently complete interface to this representation generally solves any problem one might care to pose about it. One may just as easily say that all problems at the lexical level may be solved via proper use of the UNIX standard IO library `<stdio.h>`, a true, but utterly trivial and unsatisfying statement. The question is rather that of building a simpler, more useful and flexible interface: one that is less error-prone, more concise than writing in C, and more directly suited to the task of exploring ASTs. We first consider a couple of prior approaches.

1.1 The `awk` Approach

One of the more popular approaches is to extend the `awk` [AKW86] paradigm. An `awk` script is a list of pairs, each being a regular-expression with an accompanying statement in a C-like imperative language. For each line in the input file, we consider each pair of the script in turn; if the regular-expression matches the line, the corresponding statement is executed.

Extending this to the AST domain is straightforward, though with numerous variations. One defines a regular-expression-like language in which to express tree patterns and an `awk`-like imperative language for statements. The tree nodes of the input program are traversed in some order (e.g., preorder), and for each node the various pairs of the script are considered as before.

We have two objections to this approach, the first having to do with the hardwired framework that usually implicit. In some cases (e. g., TAWK [GA96]), the traversal order for the AST nodes is essentially fixed; using a different order would be analogous to attempting to use plain `awk` to scan the lines of a text file in reverse order. In A* [LR95], while the user may define a

general traversal order, only one traversal method may be defined/active at any given time, making difficult any structure comparisons between subtrees or other applications that require multiple concurrent traversals. Since the imperative language is quite general in both cases, little is definitively impossible, however for some applications one may be little better off than when programming in straight C.

The second objection has to do with the kinds of pattern-abstraction available. Inevitably there exist simply-described patterns that are a poor fit to a regular-expression-like syntax. This tends to happen when said simple descriptions are in terms of the idioms of a particular programming language; most of the various tree-`awk` pattern languages tend to be designed with the intent of being language independent.

Suppose one wishes to find all consecutive occurrences of one statement immediately preceding another, e. g., places where a given system call `syscall();` is followed immediately by an `assert();` (on the theory that testing of outcomes of system calls should be done in production code rather than just debugging code). A tree-regular-expression pattern of the form

⟨`syscall()` pattern⟩; ⟨`assert()` pattern⟩

(where ; is the regular-expression sequence operator) finds all instances of the two calls occurring consecutively within a single block, but it misses instances like

```
syscall();
{
  assert();
  ...
}
```

and

```
if (...) {
  syscall();
}
else {
  ...
}
assert();
```

While the tree-`awk` languages allow one to write patterns to match each of these cases, without a pattern-abstraction facility, we may be back at square one when it comes time to look for some *different* pair of consecutive function calls. We prefer to write a single consecutive-statement pattern-constructor *once* and then be able to use it for a variety of cases where we need to find pairs of consecutive statements satisfying certain criteria, invoking it as

`follow_stmt`(⟨`syscall()` pattern⟩, ⟨`assert()` pattern⟩)

for the above problem, or, if we instead want to be finding all of the places where a C switch-case falls through, as

`follow_stmt`(`not`(⟨unconditional-jump pattern⟩), ⟨case-labeled stmt pattern⟩)

One solution, used by TAWK, is to use `cpp`, the C preprocessor, to preprocess the script, allowing for pattern-abstractions to be expressed as `#define` macros whose invocations are then expanded as needed. This is unsatisfactory in a number of ways, whether one wants to consider the problem of recursively-defined patterns, macros with large bodies that result in a corresponding blow-up in the size of the script, or the difficulty of tracing script errors that resulted from complex macro-expansions.

Another way out is to fall back on the procedural abstraction available in the imperative language that the patterns invoke. One essentially uses a degenerate pattern that always matches and then allows the imperative code to test whether the given node is in fact the desired match, defining functions to test for particular patterns. Once again, it seems we are back to programming in straight C and not deriving as much benefit from having a pattern language available as we could be.

In general, the philosophical underpinning of the `awk` approach is that the designer has already determined the kinds of searches the user will want to do; the effort is put towards making those particular searches run efficiently. There is also an assumption that the underlying imperative language for the actions has all the abstraction facilities one will ever need, so that if the pattern language is lacking in various ways, this is not deemed a serious problem. While this is not an unreasonable approach, we have less confidence of having identified all of the reasonable search possibilities, and thus would prefer instead to make the pattern language more flexible and extensible, being willing to sacrifice some efficiency to do so.

1.2 The Logic Programming Approach

Another common approach is to run an inference engine over a database of program syntactic structures [BCD+88, BGV90, CMR92]. Prolog [SS86] is a convenient language for this sort of application. Backtracking and a form of pattern matching are built in, the abstraction mechanisms to build up complex predicates exist at a fundamental level, and finally, Prolog allows for a more declarative programming style.

script	::= *named-clause**	script file syntax
query	::= *imports?* `(` *varname** `)` *clause-body* `;`	query syntax
imports	::= `{` *varname+* `}`	
named-clause	::= *opname anon-clause*	
anon-clause	::= `(` *term** `)` *clause-body?* `;`	
clause-body	::= `<-` *term+*	
	Essential Term Syntax	
term	::= *literal*	reference to denotable object
	::= *varname*	
	::= *opname* `(` *term** `)`	compound term
	::= `FN` *imports?* `(` *anon-clause+* `)`	anonymous predicate-operator-valued ("lambda") term
	::= `'` *opname arity-spec?*	named predicate-operator-valued ("function quote") term
	::= `(` *term* `)(` *term** `)`	"application" term
	Gratuitous Term Syntax	
	::= `#` *constname*	named constant (≡ corresponding literal number)
	::= `[` *term** `]`	`[ ]` ≡ `nil()`, `[`*term*`]` ≡ `cons(`*term*`,nil())`, etc...
	::= `[` *term+* `|` *term* `]`	`[` *term*1 `|` *term*2 `]` ≡ `cons(`*term*1`,`*term*2`)`, etc...
arity-spec	::= `/` *integer*	

Figure 1: Complete Syntax of ASTLOG

The problems with using Prolog are two-fold. First there is the issue of efficiency. Second, we must represent the AST for our source program in the Prolog database. Large programs ($10^5 - 10^6$ lines) will result in correspondingly large Prolog databases, most likely with a significant performance penalty.

We finesse the second problem by not attempting to import the source program's AST at all, instead opting to modify the interpretation of the predicates and queries of Prolog so as to be applicable to external objects rather than just facts provable in the existing database. Removing reasons that require the database to grow beyond the initial script creates significant opportunities for optimization. This, however, requires removing primitives like `assert`() and `retract`() that allow for the dynamic (re)definition or removal of predicates, which in turn removes many higher-order logical features that are defined in terms of them. Fortunately, some of the more essential ones can be restored at relatively little cost.

2 Elements of ASTLOG

Figure 1 gives the complete syntax for our language, ASTLOG. The ASTLOG interpreter reads a script of user-defined predicate operator definitions and then runs one or more queries.

As in Prolog, the definition of a user-defined predicate operator is composed of one or more *clauses*. A compound term *opname*(*term*,...) appearing at top level in a clause body is interpreted as a predicate, whether *opname* be primitive or user-defined. In the latter case, the script is searched for a defining clause whose head terms successfully unify with the respective operand terms of the given compound term, variables are bound accordingly, and the terms of the clause body are likewise interpreted. The clause *succeeds* (i. e., is found to be true) if all of its body terms succeed. Whenever a clause head fails to unify, or a clause body term *fails* (i. e., is found to be false), or any primitive term fails by the rules of evaluation of that primitive, we backtrack to the last point where there was a choice (e. g., of clauses to try for a given compound term) and continue.

A *query* is a clause whose head terms are all variables. Ultimately, whenever all terms of a query body succeed, the bindings of any variables listed in the query head (*qhead*) are reported. Otherwise, we report failure. Thus far, this is all exactly like Prolog.

2.1 Objects

ASTLOG refers to external objects. Given a C/C++ compiler front end that provides a (C++) interface to the syntactic/semantic data structures built during the parse of a given program, it is simple to graft this onto the core of ASTLOG so that it may recognize object references corresponding to

- whole C/C++ programs,
- single files,
- symbols,
- AST nodes (including statements, expressions, and declarations), and

- C/C++ type descriptions.

For the purposes of ASTLOG, an *object* is simply some external entity that is significant for its identity and for the primitive predicates that it may satisfy. To simplify the language we regard the traditional constants (integers, floats, and strings) to be references to "external" objects as well, though one could just as easily take the converse view in which the universe of object references is just a (very large) pool of constants ("atoms" in the usual Prolog terminology).

In any case, object references are terms in ASTLOG. Only references to equal objects can unify, equality meaning numeric equality for numbers, same-sequence-of-characters for strings, and identity for all other classes of objects. Only objects that have denotations (numbers, strings and the unique null object `*`) can find their way into scripts.

2.2 The Current Object

The first significant departure from the Prolog model is that a query or predicate term always evaluates under an ambient *current object*. Every query and every term being evaluated as a predicate is not so much a standalone statement that may or may not be intrinsically true (i. e., provable from the "facts" in the script) as it is a specification that may or may not be satisfied by the current object, or, alternatively, a *pattern* that may or may not *match* the current object. For example, in Prolog

`odd`(3)

always succeeds by virtue of 3 being odd or because the "fact" `odd`(3) exists in the script somewhere. By contrast, in ASTLOG

`odd`()

succeeds if the current object happens to be the integer 3, fails if the current object is 4, and raises an error if the current object is the string `"Hi mom"`. Another way to view this is that every predicate term takes an extra, hidden current-object operand.

While one normally only expects to see compound (and application) terms in predicate position, ASTLOG allows variables and object references there as well. The rules for matching are as follows:

- An object reference matches the current object iff it references an equal object.
- A bound variable matches according as whatever term it is bound to.
- An unbound variable gets bound to reference the current object (and thus automatically matches it).
- A compound term whose operator is defined via clauses matches iff there exists a clause whose head operands unify with the term operands and whose body terms themselves all match the current object.

Section 3.1 describes the operator-valued and application terms.

The evaluation rules for compound terms having primitive operators are widely varied, however the operands are usually treated one of two ways:

1. (*foo-pred*) requiring the operand to be match some object (which becomes the current object for that evaluation), not necessarily the same current object as that which the full term is being matched against. For example, the operand of `strlen` (see Figure 2) and the second operand of `with` are treated this way.
2. (*foo*) requiring the operand be an object reference, whether this be a literal or an object-reference-bound variable. The operands of `re`, `gt`, and the first operand of `with` are treated this way.

Most primitives also expect a current object to be of a particular kind and raise an error if confronted with something different.

The use of an implicit current object is not by itself an increase in expressivity. If we had, in a Prolog database, terms representing the various AST nodes, there would be a fairly straightforward translation of ASTLOG terms into Prolog terms, one in which we simply modify all terms to make the current object an explicit operand.

Nevertheless, ASTLOG programs exhibit a distinct style of programming. Consider as an example that we might, in a typical functional language, write a function call

`strlen`(*string*)

to find the length of the string returned by the expression *string*. Here the length result is implicitly returned to the context of the call. In Prolog, the natural style would be to express this as a relation

`strlen`(`string`, `length`)

which stipulates that `length` is in fact the length of `string`. In ASTLOG, we would write

`strlen`(*length-pred*)

where now it is the string argument that is implicitly supplied (as the current object) *by* the context while the length result is returned *to* the subterm *length-pred*, which in turn can be some arbitrary term expecting a numeric current object as its implicit argument. For example, given an `odd`() predicate as

- `and(`*object-pred*, ...`)`
  The current object satisfies every *object-pred* operand.
- `or(`*object-pred*, ...`)`
  The current object satisfies some *object-pred* operand.
- `if(`*object-pred*, *then-pred*, *else-pred*`)`
  The current object satisfies *then-pred* or *else-pred* according as it satisfies or fails to satisfy *object-pred* (once; if *object-pred* matches but *then-pred* does not, we do not retry *object-pred*).
- `not(`*object-pred*`)`
  = `if(`*object-pred*, `or()`, `and()`)
- `with(`*object*, *object-pred*`)`
  object satisfies *object-pred* (outer current object is ignored).
- `strlen(`*integer-pred*`)`
  The current string object has length satisfying *integer-pred*.
- `re(`*string*`)`
  The regular expression *string* matches the current string.
- `gt(`*integer*`)`
  The current integer is greater than *integer*.
- `minus(`*integer-pred*, *integer*`)`
  integer-pred matches the current integer + *integer*.
- `minus(`*integer*, *integer-pred*`)`
  integer-pred matches *integer* − the current integer. (An error is raised if neither operand of a `minus` term is an integer object reference.)
- `plus(`*integer-pred*, *integer*`)`
  integer-pred matches the current integer − *integer*.

Figure 2: Some core ASTLOG primitives

above, the term `strlen(odd())` would match any string consisting of an odd number of characters. It is this "inside-out functional" evaluation strategy that makes ASTLOG well-suited to constructing anchored patterns to match tree-like structures.

2.3 Examples

Given the set of AST node primitives in Figure 3, we could write

```
and(op(#=), kid(#LEFT, asym(sname("foo"))))
```

which would be satisified by any AST node that is an assignment (=) expression whose left-hand side is itself a symbol expression where the symbol name is `"foo"`. Here, `#=` and `#LEFT` are numeric literals for the assignment node opcode and the assignment target's child-index, respectively.

To define a predicate `assignment/2` to match assignment nodes, a script could include the clause

```
assignment(target, value)
      <-  op(#=),
          kid(#LEFT, target),
          kid(#RIGHT, value);
```

which would then allow writing the previous term as

```
assignment(asym(sname("foo")), _)
```

As in Prolog, the underscore (`_`) is "wild-card" variable, i. e., one that is internally given a distinct identity so as not to be conflated with any other instances of `_`. Such a variable, being guaranteed to be unbound, will match any object or unify with any term.

Defining a general purpose node-traversal predicate is also straightforward

```
somenode(pred)
      <-  or(pred, kid(_, somenode(pred)));
```

Given this definition, an attempt to match `somenode(`*test*`)` to a given node will create an instance of the defining clause of `somenode/1` above with `pred` bound to *test*. Satisfying the clause body requires that either `pred` match the current node, or, if (when) that fails, that `kid(_,somenode(pred))` match the current node. The latter in turn will attempt to match the variable `_` with 0 (easy) and the term `somenode(pred)` with the first child, and, when that fails, `_` with 1 and `somenode(pred)` with the second child, etc... Making the interpreter fail and backtrack after each hit (in the usual manner of Prolog) eventually causes *test* to be matched with the original node and all of its descendants.

- `parent`(*ast-pred*)
  This AST node is not a root node and its parent satisfies *ast-pred*.
- `kid`(*integer-pred*, *ast-pred*)
  This AST node has a child satisfying *ast-pred* whose (0-based) index satisfies *integer-pred*.
- `kidcount`(*integer-pred*)
  The number of children of this AST node satisfies *integer-pred*.
- `op`(*integer-pred*)
  The opcode of this AST node satisfies *integer-pred*.
- `atype`(*type-pred*)
  This AST node has a return type satisfying *type-pred*.
- `asym`(*symbol-pred*)
  This AST node is a symbol satisfying *symbol-pred*.
- `aconst`(*const-pred*)
  This AST node is a constant (integer, float or string) satisfying *const-pred*.
- `sname`(*string-pred*)
  This symbol's name satisfies *string-pred*.

There are named constants available for designating the opcodes of various kinds of nodes for use in `op()` terms, and the indices of particular children for use in `kid()` terms.

Figure 3: Some primitive node and symbol predicates

So, if we issue the query

```
(v) <- somenode(
    assignment(asym(sname("foo")), v)
                    );
```

on the root node of some function's AST, we obtain, via the successive bindings reported for `v` on each hit, all of the expressions assigned to variables named `"foo"` within that function.

As an example that makes less trivial use of backtracking, consider the problem of whether two trees have the same structure (i. e.., root nodes have the same opcode and all corresponding children have the same structure).

```
sametree(node)
    <-  op(nodeop),
        with(node, op(nodeop)),
        not(and(with(node, kid(n, nkid)),
                kid(n, not(sametree(nkid))))));
```

This defines a predicate `sametree`(*node*) that holds iff *node* is a reference to an AST node with the same structure as the current object. The first line of the clause body binds the current node's opcode to `nodeop`, the second line compares that to the opcode of `node`, while the remaining lines search for children whose subtrees have distinct structure. The term `kid(n,nkid)` will match each child of `node`, since both variables are initially unbound. If `sametree(nkid)` happens to be true of the corresponding child of the current node, the inner `not` fails and we go back and try another child of `node`. If `sametree(nkid)` happens to be true of *every* corresponding child of the current node, then the enclosing `not` and thus the outer `sametree(node)` invocation succeeds.

The preceding version of `sametree/1` is a purely structural comparison; there is no attempt to take account of the commutativity/associativity of the various operators, e. g., `a+b` and `b+a` are not considered the same. If, say, we *did* want to consider commutativity, we could define

```
csametree(node)
  <- op(nodeop),
     with(node,op(nodeop)),
     kidcount(if(with(nodeop,commutes()),
                 any_perm(perm),
                 id_perm(perm))),
     not(and(with(node,kid(corresp(perm,n),
                           nkid)),
             kid(n,not(csametree(nkid))))));
```

along with suitable definitions of

`commutes`()
: the current integer is the opcode of a commutative operator,

`any_perm`(*perm*)
: *perm* is any permutation of the sequence $0, \ldots,$ $(\langle\text{current-object}\rangle - 1)$,

`id_perm`(*perm*)
: *perm* is the identity permutation of the sequence $0, \ldots, (\langle\text{current-object}\rangle - 1)$,

`corresp`(*perm*, *n*)
: permutation *perm* takes the current integer to something matching *n*.

Here, permutations can be represented by list terms. Note that since all of the commutative C/C++ operators are, in fact, binary, this all simplifies significantly.

It should, incidentally, be clear that there is nothing about the core language that is specifically tailored for the examination of compiler-produced ASTs, let alone ASTs for a given language. The language in fact lends itself to the examination of a wide variety of external structures, e. g., hierarchical file systems, or collections of web pages. All that is needed is a suitable collection of primitive ASTLOG predicates for querying said structures.

```
// FOLLOW_STMT(P1 P2)
//   <=> P1 and P2 are true of consecutive statements in this AST

follow_stmt(p1, p2)
  <- if(op(#FUNCTION),
        kid(#FUNCTION/BODY, follow_stmt(p1,p2,*)),
        follow_stmt(p1,p2,*));

follow_stmt(p1, p2, after)
  <- cond(op(#BLOCK),     follow_block_stmt(p1, p2, after),

          op(#IF),        kid(not(#IF/PRED),follow_stmt(p1, p2, after)),
          op(#SWITCH),    kid(#SWITCH/BODY, follow_stmt(p1, p2, after)),

          op(#WHILE),     follow_iter_stmt(#WHILE/BODY,p1, p2, after),
          op(#DO),        follow_iter_stmt(#DO/BODY,   p1, p2, after),
          op(#FOR),       follow_iter_stmt(#FOR/BODY,  p1, p2, after),

          or(op(#LABEL),op(#CASE),op(#DEFAULT)),
           kid(#LABELSTMT/STMT, follow_stmt(p1, p2, after)),

          follow_simple_stmt(p1, p2, after));

follow_simple_stmt(p1, p2, after)
  <- with(after, not(*)), p1, with(after, first_stmt(p2));

follow_iter_stmt(nbody,p1,p2,after)
  <- or(follow_simple_stmt(p1, p2, after),
        and(this, kid(nbody, follow_stmt(p1, p2, this))));

follow_block_stmt(p1, p2, after)
  <- and(kid(minus(next,1), first),
         if(kid(next, second),
            with(first, follow_stmt(p1, p2, second)),
            with(first, follow_stmt(p1, p2, after))));

first_stmt(p)
  <- if(op(#BLOCK),
        kid(0,first_stmt(p)),
        stmt);

// CASEFALL()
//   emits all locations of switch-case fallthroughs in this AST tree
casefall()
  <- follow_stmt(and(not(op(or(#BREAK,#CONTINUE,#GOTO,#RETURN))),first),
                 op(#CASE)),
     with(first,sfa(emit("Fall through to next case.")));
```

Figure 4: Actual ASTLOG code for `follow_stmt` and how one uses it to find case statement fallthroughs. The `cond` operator is an if-then-elseif- construct, that is, `cond`$(p_1, e_1, p_2, e_2, \ldots, e)$ is equivalent to `if`$(p_1, e_1, $`if`$(p_2, e_2, \ldots e))$. `sfa(emit(`*string*`))` always succeeds and, as a side-effect, emits the source location of the current AST node in `grep`-output form.

```
flatten(test, lst)
  <- flatten(test, lst, []);

flatten(test, head, tail)
  <- if(test,
        first(head, hrest),
        unify(head, hrest)),
     flattenkids(test, 0, hrest, tail);

flattenkids(test, n, head, tail)
  <- if(kid(n, flatten(test, head, mid)),
        and(with(n, minus(nplus1,1)),
            flattenkids(test, nplus1,
                        mid, tail)),
        unify(head, tail));

first([o|rest],rest) <- o;
unify(x,x);
```

Figure 5: Definition of `flatten`

```
flatten2(test, lst)
  <- flatten2(test, lst, []);

flatten2(test, head, tail)
  <- if((test)(value),
        unify(head, [value|hrest]),
        unify(head, hrest)),
     flatten2kids(test, 0, hrest, tail);

flatten2kids(test, n, head, tail)
  <- if(kid(n, flatten2(test, head, mid)),
        and(with(n, minus(nplus1,1)),
            flatten2kids(test, nplus1,
                         mid, tail)),
        unify(head, tail));

unify(x,x);
```

Figure 6: Parameterized version, `flatten2`

3 Higher order features

We have already included some of the non-1st-order features of Prolog, notably "cut" (in the form of `if()`) and the corresponding notion of negation, `not()`. There are others that turn out to be essential as well.

3.1 Lambdas and Applications

One may observe that, in `somenode`(*test*), because this is an existential query, it does not matter that we are matching the same term *test* to every node of the tree. If variables in *test* get bound as a result of matching a given node, those bindings will be undone prior to advancing to the next node.

If one instead wants to write a conjunctive predicate over all tree nodes, say

`flatten`(*test*, `list`)

which holds if `list` is a list of *all* descendant nodes satisfying *test*, — we give a definition in Figure 5 — this will not work correctly if *test* contains any variables that are bound during the course of matching any node; said variables will *stay* bound for the duration of the `flatten` evaluation.

Even in an existential query, there is the possibility that the *test* being passed in will itself need to take a parameter. For example, one might imagine defining a version of `sametree` that also requires an additional user-specified *test* to hold at each corresponding pair of nodes. If *test* is a mere compound term, it can be matched against one of the nodes, but not both.

Thus we introduce "application" terms and operator-valued terms ("lambdas"). For an application (*fterm*)(*term*, ...) to match the current object, the term *fterm* must be (or be a variable bound to) a predicate-operator-valued term, which will either be

- a reference, `'foo/3` to a named predicate operator, in which case the application evaluates exactly as the corresponding compound term would, or
- an anonymous predicate operator `FN{`*importvars*...`}(`*anon-clauses*...`)`, in which case the application evaluates *almost* exactly as if there were a named predicate-operator defined by the given clauses and this were a compound term on that operator. The difference is that any variables of those clauses that are also on the `{`*importvars*...`}` list are identified with the correspondingly-named variables in the clause where the `FN` term occurs lexically.

An `FN` term with imports can be thought of as a kind of *closure*.

The parameterized version of `flatten`, namely

`flatten2`(*test*, `list`)

which holds iff `list` is a list of all x corresponding to descendants that (*test*)(x) matches, is defined in Figure 6.

The parameterized version of `sametree` is invoked as

`sametree`(*node*, *equiv*)

which holds iff *node* is a reference to an AST node with the same tree structure as the current node *and*, for every descendant n of *node*, the corresponding node in the current tree satisfies *equiv*(n); this predicate is defined in Figure 7. This definition demonstrates the use

```
sametree(node,equiv)
 <- unify(same,
     FN{same,equiv}
      ((node)
        <- op(nodeop),
           with(node,op(nodeop)),
           (equiv)(node),
           not(and(with(node,kid(n,nkid)),
                   kid(n,not((same)(nkid)))))),
    (same)(node);
```

Figure 7: Parameterized version of `sametree`

of import lists, both to define a recursive anonymous predicate, and to make *equiv* available at once to all evaluations of that predicate. Given that definition, the following

```
sametree(node,
    FN((n) <- if(aconst(c),
                 with(n, aconst(c)),
                 and()); ))
```

would then test whether the current tree has the same structure as underneath node *and* such that all corresponding constants are the same.

3.2 Queries as Objects

Sometimes one wishes to build a collection or some other kind of aggregate of all objects found by a query. Unfortunately, when backtracking to get to the next hit, information about the previous hit will generally be lost. One solution is to rewrite the query into a conjunctive form, as we did in the previous section converting writing `flatten` as a conjunctive version of `somenode` (see Figure 5). We can already see that even in simple cases this process can be non-trivial and is not readily generalized.

It may also be the case for some conjunctive queries that they require memory proportional to the size of the data structure being searched, instead of merely memory proportional to the *depth* of the data structure. Judicious use of `if()` — ASTLOG's moral equivalent of the cut operator — can avoid this, but this is sometimes cumbersome to get right.

As it happens, Prolog provides a number of *set-predicates* for accumulating query results. For example,

`bagof(x,` *term*`, list)`

binds `list` to a list of the bindings of `x` corresponding to each instance where *term* holds true. Unfortunately, this is usually implemented in terms of `assert` and `retract`, meaning we would have to abandon the idea

- `query(`*fterm*, *query-pred*`)`
  The embedded query state object created from *fterm* satisfies *query-pred*.
- `qnext(`*pred*, *thisquery-pred*, *nextquery-pred*`)`
  If the current embedded query state is a failure, *pred* is true, otherwise the current object satisfies *thisquery-pred* and, after the embedded query is advanced to the next hit or to failure, the resulting query state satisfies *nextquery-pred*.
- `qget(`*object-pred*, ...`)`
  Each *object-pred* matches the object bound to the corresponding variable in the head of the embedded query corresponding to the current query state object. An error will be raised if the embedded query has failed or if any head variable is not bound to an object.

Figure 8: Embedded Query State Primitives

of keeping our script small and fixed. Even just adding this as a new primitive is dubious if we have to add, say, another new primitive to merely count query hits, and yet more new primitives for each accumulation method anyone dreams up.

The key observation is that the execution model of ASTLOG allows for the possibility of treating some subset of its own internal structures as "external" objects which can then serve as the current object of various kinds of queries. To be sure, some care needs to be exercised, since the internal structures of ASTLOG are not static the way the program ASTs are. We can however, take a query whose hits we wish to accumulate, and encapsulate its state after a given hit as an ASTLOG object. Such an *embedded* query in a given state can now be the current object for the evaluation of some other predicate term. We thus only need to provide suitable primitive predicates applicable to query-state objects that may be used in such a term. Figure 8 lists these primitives.

Using this mechanism, it is then possible to define a wide variety of accumulators of query results. Given an AST node, and a query to see if there exists a descendant satisfying `test(x)`

```
() <- somenode(test(x));
```

the corresponding query to count the number of descendants satisfying `test(x)` would be

```
(n)  <-  query(FN(() <- somenode(test(x)); ),
                qcount(n));
```

where `qcount/1` is defined as in Figure 9. Evaluating the `query()` term starts an embedded query corresponding to the first operand and builds a query state object representing the resulting first state (first hit

```
qcount(n) <- qcount(0, n);

qcount(sofar, return)
  <- qnext(unify(sofar, return),
           with(sofar, minus(sofarp1,1)),
           qcount(sofarp1, return));

qlist(lst)
  <- qnext(unify(lst, []),
           qget(first(lst,rest)),
           qlist(rest));

// utilities
first([o|rest],rest) <- o;
unify(x,x);
```

Figure 9: Query Accumulators `qcount` and `qlist`

or failure). This object then becomes the current object to which we try to match `qcount(n)`. It is the `qnext()` term therein that does the actual work. If the query-state is a success state, we increment the count of hits thus far (`sofar`), advance the embedded query, and recursively try to match a `qcount` term to the new state. If the query-state is a failure, we unify the count of hits thus far with the return variable.

To build a list of bindings for `x` corresponding to the query hits, we can do

```
(list)  <-  query(FN((x) <- somenode(test(x)); ),
                  qlist(list));
```

which is essentially the same as before except that now `qlist(list)` uses `qget` to examine the query state. Since the embedded query has only one head variable `x`, the `qget` term must likewise have at most one operand.

Some care is required when using embedded queries to phrase them so that the head variables will always be bound to objects. `qget()` will in fact raise an error if a head variable is not bound to an object. This requirement is crucial since, with a non-object term, there is no guarantee that said term will remain intact when the embedded query backtracks to the next state. Better to keep terms constructed by an embedded query from polluting the outer world.

The mechanism is also somewhat impure in that evaluating a `qnext` on a given query state object essentially destroys that object. Subsequent attempts to match additional terms against that query state will raise an error since the state of a query is lost once we advance it.

4 Implementation

ASTLOG has been implemented as an interpreter in roughly 11,000 lines of C++ for the core ASTLOG interpreter and supporting utilities. Another 1100 lines define the roughly 60 primitives and supporting structures to invoke the various functions of the AST library. Coverage of the library API is in not entirely complete, but it is sufficient to perform various interesting tasks:

- Finding all instances of a simple assignment expression (`=`) occurring in *any* boolean context, for example,

  ```
  if ((major == SORTM)
      || (major == MEMORYM)
      || ((major == BUFFERM)
          && (minor = B_NOIO)))
  ```

- Finding all instances of an equality-test (`==`) or dereference expression occurring in *any* void context (i. e., where results are discarded); the converse to the previous problem.

- Finding all `case` statement fall-throughs, i. e., where the preceding statement is not a break.

- Finding various patterns of irreducible control-flow in functions.

- Obtaining all static call-graph edges.

- Computing the McCabe cyclomatic complexity [McC76] of a function. Our code to do so looks like

  ```
  mccabe(n) <- query(
     FN(()<- somenode(
                op(or(#IF,#FOR,#DO,
                      #WHILE,#CASE,
                      #?,#||,#&&)));)
     qcount(minus(n,1))
  );
  ```

  which might be compared with the 17-line version in Aria [DR96]. Admittedly, fairness would probably entail including the definitions of `somenode` and `qcount` as well.

- Finding gaps (unused space due to alignment rules) in structure definitions; this is a matter of traversing C type structures rather than ASTs.

A typical running time (on a 200MHz Pentium P6 with 64meg of RAM) for a one-pass search that evaluates a simple predicate on every AST node in Microsoft SQLserver (roughly 450,000 lines, 4300 functions) is roughly 10 minutes, of which 7.5 minutes are taken up

by the AST library building the actual trees. For Microsoft Word (roughly 2,000,000 lines) the corresponding times are 45-60 minutes of which about 30 minutes is taken up by the tree builder.

Though this dreadfully slow in comparison with `grep`, these times are arguably acceptable in comparison with the times taken by the actual compiler — what one might expect for a tool that requires the use of compiler's data structures. One is, of course, free to write arbitrarily non-linear programs in ASTLOG, so there are no guarantees. In any case we would doubtless see a certain amount of speedup if we actually were to attempt some kind of compilation of the ASTLOG code.

5 Conclusions and Future Work

We have described a language for doing syntax-level analysis for C/C++ programs, though the core language is, in fact, adaptable to many other kinds of structures. As with previous such tools, the utility to users who are thus no longer required to write their own parse/semantic-analysis phase is apparent. The contribution here is a pattern language sufficiently powerful to provide traversal possibilites beyond what is naturally available in prior `awk`-like frameworks while avoiding some of the inefficiencies of importing the entire program structure into a logical inference engine. The Pan work [BGV90] stressed the need to partition code and data; this we have done in a rather straightforward way. The surprise is that the Prolog-with-an-ambient-current-object model turns out to be so well suited to analyzing treelike structures.

To be sure, there are various rough edges:

1. As already noted, embedded queries are slightly unsafe; there may exist a more robust set of primitives to use. Some form of type inference to detect unsafe uses of `qnext` may also be worth considering. More generally, there is the issue of typing of ASTLOG expressions to reduce the incidence of unbound variables or objects of the wrong type appearing as operands where object-references of a particular type are required.

2. Occasionally, we run up against the generally cumbersome nature of arithmetic in Prolog, which is arguably *worse* in ASTLOG. The "inside-out functional" nature of ASTLOG may be good for AST patterns, but it can make arithmetic operations like

   $$\texttt{with}(\texttt{n}, \texttt{divide}(\texttt{minus}(\texttt{x}, 1), 2))$$

   downright unreadable. Algebraic syntax could help, e. g.,

   $$\texttt{with}(\texttt{n}, (\texttt{x} - 1)/2)$$

   but even so, one must stare at this pretty hard to realize that `n` is being multiplied by 2 and then incremented by 1.

   One possibility is to complicate the language by introducing actual "forward" functional operator definitions. For example, with such forward operators for addition and multiplication, one could then write

   $$\texttt{with}(2 * \texttt{n} + 1, \texttt{x})$$

   where the appearance of the + (`plus`) term in a slot normally requiring an object reference invokes the forward return-value-to-context definition of the operator + to sum its operands rather than the usual "backward" return-value-to-operand definition (see Figure 2) in which one operand is treated as a predicate.

3. Though there is a surprising amount of mileage to be had via instantiating terms with unbound variables in them, there are those occasions when a genuinely mutable data structure is required. Fortunately, given the strong partition between the script/database and the objects, having mutable objects exist and primitives that side-effect them when they match would not disrupt ASTLOG's execution model.

4. Currently, new primitives need to be manually written. Given the current collection of macros available, this is not actually an arduous task. Still, while language-independence was not one of our priorities, given that the core language is rather language-independent anyway, one would hope for a more automatic means of adapting ASTLOG to work with other language parsers, perhaps by adapting GENII [Dev92] or some similar tool to generate code for the basic primitive predicate operators for a fresh language.

6 Acknowledgements

ASTLOG would not have been possible without the existence of an AST library for C/C++ implemented by the members of Program Analysis group at Microsoft Research, particularly Linda O'Gara, David Gay, Erik Ruf and Bjarne Steensgaarde. I would also like to thank Bruce Duba, Michael Ernst, Chris Ramming, and the conference reviewers for much useful commentary and discussion.

References

[AKW86] A. V. Aho, B. W. Kernighan, and P. J. Weinberger. *The AWK Programming Language.* Addison Wesley, Reading, MA, 1986.

[BCD+88] P. Borras, D. Clement, Th. Despeyroux, J. Incerpi, G. Kahn, B. Lang, and V. Pascual. Centaur: The system. In *Proceedings of the SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments*, Boston, MA, 1988.

[BGV90] Robert A. Ballance, Susan L. Graham, and Michael L. Van De Vanter. The pan language-based editing system for integrated development environments. In *Proceedings of the 4th ACM SIGSOFT Symposium on Software Development Environments*, pages 77–93, Irvine, CA, 1990.

[CMR92] Mariano Consens, Alberto Mendelzon, and Arthur Ryman. Visualizing and querying software structures. In *Proceedings of the Fourteenth International ACM Conference on Software Engineering*, pages 138–156, 1992.

[Dev92] Premkumar T. Devanbu. Genoa - a customizable, language-and-front-end independent code analyzer. In *Proceedings of the Fourteenth International ACM Conference on Software Engineering*, pages 307–319. ACM Press, 1992.

[DR96] Premkumar T. Devanbu and David S. Rosenblum. Generating testing and analysis tools with aria. *ACM Transactions on Software Engineering and Methodology*, 5(1):42–62, January 1996.

[GA96] William G. Griswold and Darren C. Atkinson. Fast, flexible syntactic pattern matching and processing. In *Proceedings of the IEEE Workshop on Program Comprehension*. ACM Press, 1996.

[LR95] David A. Ladd and J. Christopher Ramming. A*: A language for implementing language processors. *IEEE Transactions on Software Engineering*, 21(11):894–901, November 1995.

[McC76] T. McCabe. A complexity measure. *IEEE Transactions on Software Engineering*, 2(4):308–320, December 1976.

[SS86] Leon Sterling and Ehud Shapiro. *The Art of Prolog: Advanced Programming Techniques*. MIT Press series in logic programming. The MIT Press, Cambridge, MA, 1986.

Appendix

For those who would prefer to see a slightly more formal description, we include a brief outline of an operational semantics for ASTLOG in Figure 10, one that bears some resemblance to the actual implementation.

For any given term that is not an object reference, one may imagine there being numerous instances of that term in existence at any given time. We differentiate the various instances by assigning each a unique *frame identifier* (f) which is only significant for its identity. A variable v occurring within a given term t may, for a particular instance $\langle f, [\![t]\!]\rangle$ of that term, be bound to some object o or other term instance $\langle f', [\![t']\!]\rangle$, this being indicated by having a *binding*, i.e., one of $\langle f, [\![\mathtt{v}]\!]\rangle \sim o$ or $\langle f, [\![\mathtt{v}]\!]\rangle \sim \langle f', [\![t']\!]\rangle$ present in the current *binding stack*, which in turn is nothing more than a list of bindings. The semantic function vlookup$(B, \langle f, [\![t]\!]\rangle)$ returns

- $\langle f, [\![t]\!]\rangle$ itself if t is not a variable.
- $\perp$ if the variable t is not bound in B.
- o if $\langle f, [\![t]\!]\rangle \sim o$ is in B
- vlookup$(B, \langle f', [\![t']\!]\rangle)$ if $\langle f, [\![t]\!]\rangle \sim \langle f', [\![t']\!]\rangle$ is in B.

At any given time, the full state of our abstract machine is described by a *failure* of the form $B \vdash C :: F$ which consists of

- the current binding stack B,
- the current *continuation* $C = (o, f, g, C')$, which in turn consists of a current object o, a current frame identifier f, a current *goal*, usually a term, but this can also be one of the auxiliary goals "apply(...)" or "cut(...)," and finally another continuation C' to which we advance if the goal succeeds
- the next failure F, to which we advance if the current goal fails.

Note that unlike the case where the goal succeeds, failure may involve undoing one or more bindings; thus, a failure (F) contains its own binding stack (a subset of B) whereas the continuations (C, C') do not.

The bottom half of Figure 10 (partially) defines a transition relation between states of the abstract machine. Given an initial current object o and a query $[\![query]\!]$ with n head variables, we take the initial state to be

$$F_0 = [] \vdash (o, f_0, \text{apply}(f_0, [\![query]\!], [\![\mathtt{v}_1, \ldots, \mathtt{v}_n]\!]), \text{yes}) :: \text{no}$$

If there is a sequence of transitions

$$F_0 \longrightarrow^* B_1 \vdash \text{yes} :: F_1$$

then we have a hit and the various query head bindings are available as vlookup$(B_1, \langle f_0, [\![\mathtt{v}_i]\!]\rangle)$ for $i = 1 \ldots n$. Likewise, if

$$F_k \longrightarrow^* B_k \vdash \text{yes} :: F_{k+1}$$

then we have a $(k+1)^{\text{th}}$ hit.

The semantic function

$$\text{mgu}(B, f, [\![t_1, \ldots, t_n]\!], f', [\![t'_1, \ldots, t'_n]\!])$$

returns an augmented binding stack that includes B together with those additional bindings that make up the most general unifier of the respective term instances $\langle f, [\![t_1]\!]\rangle$ with $\langle f', [\![t'_1]\!]\rangle$, etc.... If there is no most general unifier, mgu() returns ufail.

In the actual implementation, because the script is fixed, we may precompute at load time mgus of all pairs of same-operator-and-arity compound terms occurring in the script, making clause invocation no more expensive than a function call in many cases. We also omit the "occurs check" [SS86] for the run-time portion of unification (i.e., where we're transitively following variable bindings), with the usual increase in speed and infinite-loop risk. Thus far, unification has played a somewhat smaller role in ASTLOG scripts than expected, so there's some question whether we need to be doing even this much.

As noted above objects only unify with equal objects. The idea of allowing an object to unify with a compound predicate term that matches it has been considered, but rejected due to the significant complications it would introduce. Also, once one has subgoals being attempted during the course of unification, the user's control over evaluation order is drastically reduced, something to be avoided if one is interested in having users being able to write efficient scripts.

$$
\begin{array}{rll}
\mathit{CompTerms} = & \mathit{OpTerms} + \mathit{LambdaTerms} + \mathit{AppTerms} & [\![\texttt{op}(t_1, \ldots)]\!], [\![\texttt{FN}(\mathit{clauses})]\!], [\![(\mathit{fterm})(t_1, \ldots)]\!] \\
\mathit{NonObjTerms} = & \mathit{Vars} + \mathit{CompTerms} & [\![\texttt{var}]\!] \\
\mathit{Terms} = & \mathit{NonObjTerms} + \mathit{Objects} & [\![o]\!] \\
\mathit{Goals} = & \mathit{Terms} & [\![t]\!] \\
& + \mathit{FrameIds} \times \mathit{LambdaTerms} \times \mathit{Terms}^* & \mathrm{apply}(f, [\![\mathit{fterm}]\!], [\![t_1, \ldots]\!]) \\
& + \mathit{Failures} \times \mathit{Terms} & \mathrm{cut}(F, [\![t]\!]) \\
 & & \\
\mathit{Objects} & & o \\
\mathit{FrameIds} & & f \\
\mathit{Bindings} = & (\mathit{FrameIds} \times \mathit{Vars}) & \langle f, [\![\texttt{v}]\!] \rangle \sim o \\
& \times (\mathit{Objects} + (\mathit{FrameIds} \times \mathit{NonObjTerms}))) & \langle f, [\![\texttt{v}]\!] \rangle \sim \langle f', [\![t]\!] \rangle \\
\mathit{BindingStacks} = & \mathit{Bindings}^* & B \\
\mathit{Conts} = & (\mathit{Objects} \times \mathit{FrameIds} \times \mathit{Goals} \times \mathit{Conts}) + \{\mathrm{yes}\} & (o, f, [\![t]\!], C) \\
\mathit{Failures} = & ((\mathit{BindingStacks} + \{\mathrm{ufail}\}) \times \mathit{Conts} \times \mathit{Failures}) & B \vdash C :: F \\
& + \{\mathrm{no}\} &
\end{array}
$$

$$
\begin{array}{rl}
\mathrm{vlookup} : & \mathit{BindingStacks} \times \mathit{FrameIds} \times \mathit{NonObjTerms} \to \{\bot\} + \mathit{Objects} + (\mathit{FrameIds} \times \mathit{NonObjTerms}) \\
\mathrm{flookup} : & \mathit{OpIds} \times N \to \{\bot\} + \mathit{LambdaTerms} \\
\mathrm{frames} : & \mathit{BindingStacks} \to \mathcal{P}(\mathit{FrameIds}) \\
\mathrm{mgu} : & \mathit{BindingStacks} \times \mathit{FrameIds} \times \mathit{Terms}^* \times \mathit{FrameIds} \times \mathit{Terms}^* \to \mathit{BindingStacks} + \{\mathrm{ufail}\}
\end{array}
$$

$$
B \vdash (o, f, [\![o]\!], C) :: F \longrightarrow B \vdash C :: F
$$

$$
\frac{o \neq o'}{B \vdash (o, f, [\![o']\!], C) :: F \longrightarrow F}
$$

$$
\frac{\mathrm{vlookup}(B, \langle f, [\![\texttt{var}]\!] \rangle) = \langle f', [\![\mathit{term}]\!] \rangle}{B \vdash (o, f, [\![\texttt{var}]\!], C) :: F \longrightarrow B \vdash (o, f', [\![\mathit{term}]\!], C) :: F}
$$

$$
\frac{\mathrm{vlookup}(B, \langle f, [\![\texttt{var}]\!] \rangle) = \bot}{B \vdash (o, f, [\![\texttt{var}]\!], C) :: F \longrightarrow [@B, \langle f, [\![\texttt{var}]\!] \rangle \sim o] \vdash C :: F}
$$

$$
\frac{\mathrm{vlookup}(B, \langle f, [\![t_1]\!] \rangle) = o'}{B \vdash (o, f, [\![\texttt{with}(t_1, t_2)]\!], C) :: F \longrightarrow B \vdash (o', f, [\![t_2]\!], C) :: F}
$$

$$
\frac{\mathrm{flookup}([\![\texttt{op}]\!], n) = \mathit{fterm}, \mathit{fterm} \neq \bot}{B \vdash (o, f, [\![\texttt{op}(a_1, \ldots, a_n)]\!], C) :: F \longrightarrow B \vdash (o, f, \mathrm{apply}(f, [\![\mathit{fterm}]\!], [\![a_1, \ldots, a_n]\!]), C) :: F}
$$

$$
\frac{\mathrm{vlookup}(B, \langle f, [\![\mathit{fterm}]\!] \rangle) = \langle f', [\![\mathit{fterm}']\!] \rangle}{B \vdash (o, f, [\![(\mathit{fterm})(a_1, \ldots, a_n)]\!], C) :: F \longrightarrow B \vdash (o, f, \mathrm{apply}(f', [\![\mathit{fterm}']\!], [\![a_1, \ldots, a_n]\!]), C) :: F}
$$

$$
B \vdash (o, f, \mathrm{apply}(f', [\![\texttt{FN}\{\texttt{i}_1, \ldots\}()]\!], [\![a_1, \ldots, a_n]\!]), C) :: F \longrightarrow F
$$

$$
\begin{array}{c}
f'' \notin \mathrm{frames}(B), B' = \mathrm{mgu}([@B, \langle f'', [\![\texttt{i}_1]\!] \rangle \sim \langle f', [\![\texttt{i}_1]\!] \rangle, \ldots], f, [\![a_1, \ldots, a_n]\!], f'', [\![t_1, \ldots, t_n]\!]) \\
\hline
\begin{array}{l}
B \vdash (o, f, \mathrm{apply}(f', [\![\texttt{FN}\{\texttt{i}_1, \ldots\}((t_1, \ldots, t_n)\mathit{body}_1 \ldots ; \mathit{clause}_2 \ldots)]\!], [\![a_1, \ldots, a_n]\!]), C) :: F \\
\quad \longrightarrow B' \vdash (o, f'', [\![\texttt{and}(\mathit{body}_1 \ldots)]\!], C) \\
\qquad :: (B \vdash (o, f, \mathrm{apply}(f', [\![\texttt{FN}\{\texttt{i}_1, \ldots\}(\mathit{clause}_2 \ldots)]\!], [\![a_1, \ldots, a_n]\!]), C) :: F)
\end{array}
\end{array}
$$

$$
\mathrm{ufail} \vdash (o, f, [\![t]\!], C) :: F \longrightarrow F
$$

$$
\begin{array}{l}
B \vdash (o, f, [\![\texttt{and}(t_1, t_2)]\!], C) :: F \longrightarrow B \vdash (o, f, [\![t_1]\!], (o, f, [\![t_2]\!], C)) :: F \\
B \vdash (o, f, [\![\texttt{or}(t_1, t_2)]\!], C) :: F \longrightarrow B \vdash (o, f, [\![t_1]\!], C) :: (B \vdash (o, f, [\![t_2]\!], C) :: F) \\
B \vdash (o, f, [\![\texttt{if}(t_1, t_2, t_3)]\!], C) :: F \longrightarrow B \vdash (o, f, [\![t_1]\!], (o, f, \mathrm{cut}(F, [\![t_2]\!]), C)) :: (B \vdash (o, f, [\![t_3]\!], C) :: F) \\
B \vdash (o, f, \mathrm{cut}(F', [\![t]\!]), C) :: F \longrightarrow B \vdash (o, f, [\![t]\!], C) :: F'
\end{array}
$$

Figure 10: Outline of ASTLOG Operational Semantics