The Saturn Program Analysis System

Alex Aiken Suhabe Bugrara Isil Dillig Thomas Dillig
Brian Hackett Peter Hawkins

Stanford University

December 1, 2006

Contents

Chapter 1

Introduction

Saturn is a system for the static analysis of programs. Saturn aims to be both highly scalable and precise, with the goal of eventually being able to verify the absence of certain kinds of bugs in real systems. Saturn is based on three main ideas:

- Saturn is summary-based: each function f is analyzed separately, producing a summary of f's behavior. At call sites for f, only f's summary is used. Summary information may also be attached to types, global variables and other values.

- Saturn is also constraint-based: analysis is expressed as a system of constraints describing how the state at one program point is related to the state at adjacent program points. The primary constraint language used in Saturn is boolean satisfiability, with each bit accessed by a procedure or loop represented by a distinct boolean variable.

- Program analyses in Saturn are expressed in a logic programming language with support for constructing constraints and accessing summaries.

In combination, these ideas give Saturn the ability to succinctly express precise analyses while also providing the ability to scale to very large programs. The use of constraints and logic programs allows succinct analyses, which are easier to understand and verify correct than analyses written at a lower level of abstraction. Bit-level path-sensitive analysis gives precision, while analyzing a single function at a time and summarization give scalability—Saturn is routinely used to run whole-program analyses on the entire Linux kernel (with more than 6MLOC) and other large open source projects. Collectively these analyses have found thousands of previously unknown bugs in such projects.

Another important Saturn feature is a parallel backend, allowing multiple functions to be analyzed at the same time; clusters of up to 100 processors can be utilized effectively, depending on program size and the computational intensity of the analysis. All Saturn analyses are currently for C programs, though the ideas could be applied to other languages.

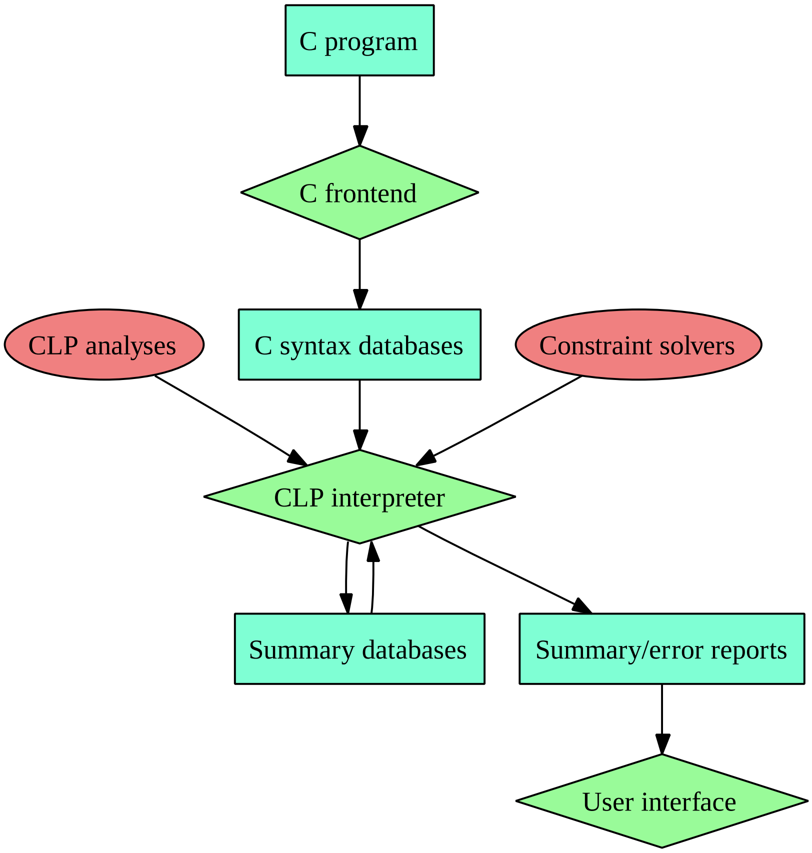

Figure 1.1: Saturn toolchain structure

This document provides an overview of the logic programming language, various Saturn analyses, as well as the tools that, while not part of any static analysis, are needed to run most or all analyses. The Saturn project is very much a work in progress, and it is possible (even likely) that this document has errors, omissions, and inconsistencies; the authors would appreciate notification of any such problems the reader may find.

1.1 Saturn Design

Figure 1.1 shows at a high level the Saturn toolchain. The C frontend encodes the full CIL syntax for a program as sets of predicates, storing them all in a few syntax databases. A program analysis, written in the Saturn-specific CLP (**C**ompositional **L**ogic **P**rogram) programming language, is then run over each function in the syntax databases by the CLP interpreter, constructing constraints and querying constraint

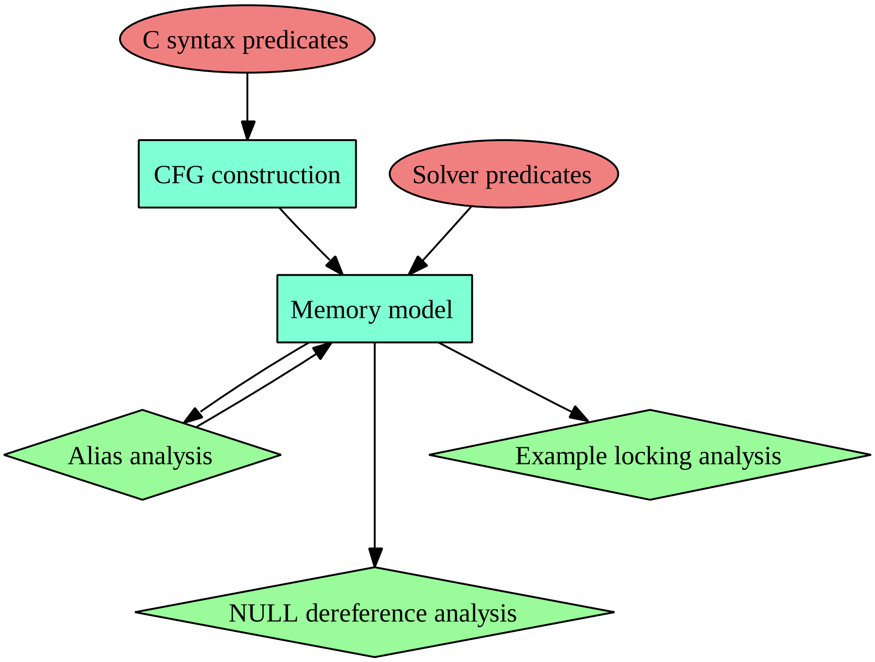

Figure 1.2: Saturn analyses structure

solvers plugged into the interpreter, constructing summary information and producing error reports. These reports can then be viewed either as plaintext or via an HTML-based UI, depending on the analysis.

Each CLP analysis is thus responsible for generating summary information and reports using only the source syntax. To ease this, core CLP components are used to construct higher level representations of the source, including control flow graph(s) for each function and a path-sensitive memory model of the points-to and integer value relationships at each program point. Most Saturn analyses build on top of one or both of these core components; Figure 1.2 shows these components and their relation to the included analyses.

Chapter 2

Quickstart

This chapter gives a quick example of how to run a complete analysis and examine the results. These instructions assume you are using a Linux system, that you have installed Saturn as per the INSTALL file instructions, and that you have added the `clpa/bin` directory to your path.

The `bftpd` package is an implementation of a simple FTP server. Download a copy of `bftpd-1.6.tar.gz` from `http://saturn.stanford.edu/misc`, extract the files, and run configure:

```
tar xvf bftpd-1.6.tar.gz
cd bftpd-1.6
./configure
```

The next step is to gather the source files for analysis. Because build processes can be complex, and are complex for large projects, the only reliable way to find the source files actually used to build the system is to monitor the build process itself. Saturn has a tool `clpamake.pl` just for this purpose; see Section 7.3.2 for more information. From the same directory, run

```
clpamake.pl -sources=. -root=.
```

This command will finish quickly and add a sub-directory `logic` to the current directory. Change into the logic directory and examine its contents:

```
cd logic
ls
```

You will see a number of `.db` files:

```
cil_body.db  cil_enum.db  cil_init.db  process.db
cil_comp.db  cil_glob.db  ppfile.db
```

These are Saturn *databases* containing sets of facts to be used in subsequent analyses of this program. The initial set of databases generated by `clpamake.pl` encode the program's abstract syntax tree. The following command run's Saturn's *null dereference* analysis, which warns of possible `NULL` pointer dereferences. This analysis will likely take a few minutes to execute. In the `logic` directory, run

```
clpa --no-fixpoint --timeout 60  PATH_TO_CLPA/analysis/null/null.clp
```

The `--no-fixpoint` flag analyzes each function once and `--timeout 60` instructs the analyzer to spend no more than 60 seconds on each function. The null dereference analysis prints out the names of functions as it analyzes them, along with any errors it detects. The first report is on line 121 of `bftpd-1.6/options.c`:

```
@121 red Possible NULL dereference of endp . . .
```

The source code around this line is:

```
120:   if (grp->users)
121:      endp = endp->next = malloc(sizeof(struct list_of_struct_passwd));
122:   else
123:      grp->users = endp = malloc(sizeof(struct list_of_struct_passwd));
```

The questionable expression is the dereference of `endp` in `endp->next`. This line appears in a loop and a quick perusal of the code reveals that `endp` is always `NULL` on entry to the loop. If the dereference is safe, then, it must be because `grp->users` is `NULL` if `endp` is `NULL`. But there is no relationship between `grp->users` and `endp` in this code; if they are correlated in some way it is not enforced in `options.c`. While we cannot state for certain that this report is a bug without more detailed knowledge of the program, it is at least suspicious.

The tool reports a possible error at one other line (line 102) in the same file`options.c`. This error report is quite similar to the first one discussed above.

Chapter 3

Tutorial

This chapter gives a tutorial overview of a Saturn analysis. The tutorial is intended to be relatively self-contained; it should be possible to get a general idea of how Saturn analyses are written and used by reading only the tutorial if the reader has some background in either program analysis or logic programming. Details omitted or only alluded to in this presentation are discussed in subsequent chapters.

We present the design of a *locking analysis*, a safety property that has become a standard example in the software verification literature. Consider a thread that manipulates a lock l. The thread should never lock l twice without an intervening unlock, or else the single thread may deadlock the whole system.[1] Similarly, a thread should should not unlock l twice without an intervening lock acquire. The goal of the analysis is to identify individual locking/unlocking operations in a program that may violate these specifications. Note that this is not a concurrency property, as we are only focusing on the behavior of a single thread.

In the directory 'analysis/locking' there are two versions of the locking analysis 'locking.clp' and 'simplelocking.clp'. In the same directory there are also several instructive example programs in the regression test directories 'baseXXX'. We present the programs in an order natural for discussion, not in the line order of the programs themselves.

We begin with 'simplelocking.clp'; a copy of the code is included in Appendix A. The first few lines

```
% Path-insensitive interprocedural locking analysis.

import "../memory/scalar_sat.clp".

analyze session_name("cil_body").
```

show several features of CLP, the Compositional Logic Programming language in which Saturn program analyses are written. The first line is a comment, indicated

[1] We assume that locks are not reentrant—i.e., a thread cannot acquire again a lock it already holds.

by the leading `%`. The `import` directive includes the file 'memory/scalar_sat.clp' in the current program. It is typical (and encouraged) for Saturn analyses to be built out of hierarchy of modules, each of which `import`s the other modules that it needs. In this case, while it is not obvious from the line above, the locking analysis is importing Saturn's *memory model* (Section 5.3), which provides a great deal of functionality and is itself built from several other modules. Most importantly, the memory model describes, for each function, all the locations accessed by the function's body, aliasing information, and the path-sensitive conditions under which each node in the function's control-flow graph is reached and each location is written. The memory model is parameterized by the representation of integers and other scalar values, and there are several choices for how to model such values and evaluate constraints over them. In this case, 'scalar_sat.clp' specifies that integer values and formulas should be converted to bit vectors and constraints evaluated using a SAT solver (Section 5.3.4).

The `analyze` directive tells the CLP interpreter what *sessions* to analyze. For scalability, Saturn analyses do not analyze the entire program at once, nor is a representation of the entire program ever constructed in main memory. Instead, parts (sessions) of the program are analyzed separately and the analysis computes *summaries* for each part. The `cil_body` sessions are complete function bodies. An alternative, which is easier to use in most cases, is the `cil_sum_body` session, which splits out every loop into a separate control-flow graph; thus a `cil_sum_body` is guaranteed to be loop free, with loops being modeled as tail-recursive function calls (Section 5.2.3).

3.1 The Locking Property

The locking property can be encoded by considering every lock to be in one of three states: *unlocked*, *locked*, or *error*. Consider two primitive locking operations, each of which takes a single lock argument l:

- The function `lock`. If l is in the *unlocked* state then the function terminates with l in the *locked* state. Otherwise the function terminates with l in the *error* state.
- The function `unlock`. If l is in the *locked* state then the function terminates with l in the *unlocked* state. Otherwise the function terminates with l in the *error* state.

Note that the functions described above are not the actual names of the primitive locking functions in any real system; we have simply chosen these names for clarity. The same signatures would apply to the primitive lock and unlock functions in, say, the kernel of an operating system.

The first part of the locking analysis implements the function signatures of the locking primitives. The declaration

```
type lockstate ::= locked | unlocked | error.
```

declares a new type `lockstate` with three constructors for the three lock states. The analysis also declares several predicates (relations) that use lock states. Every fact that can be reasoned about by the program analysis is an instance of some predicate.

```
predicate state(P:pp,T:t_trace,S:lockstate,G:g_guard).
```

The predicate `state` takes four typed arguments; CLP is strongly and statically typed and the types of all predicate arguments must be declared by the programmer. The notation `X:t` declares an argument of type `t` with mnemonic name `X` (the name is optional documentation). The `state` predicate's arguments include three frequently used types:

- A `pp` is a *program point*, an identifier that uniquely names a point in a program. Program points are created in the conversion of the initial abstract syntax trees (produced directly by the front-end parser) to a control-flow graph (Section 5.2.1).
- A `t_trace` is a *trace*, which names a memory location or a set of memory locations (e.g., arrays are treated as sets of memory locations). Analyses mostly deal with traces as abstract locations and nothing more, though as we shall see traces do have structure and in most analyses at least a few rules will be concerned with the trace structure. Traces are defined by the memory model (Section 5.3.1).
- A `g_guard` is a boolean formula that captures a program condition, usually the net effect of the predicates in `if` statements on some set of paths. Guards are used to encode path sensitivity in an analysis. Guards are also defined by the memory model (Section 5.3.4).

The interpretation of the `state` predicate is that at program point `P`, the lock at memory location `T` is in state `S` whenever guard `G` is true.

The simple locking analysis is interprocedural and flow-sensitive. The full explanation of the analysis can be divided into three parts:

- how function calls are modeled,
- how information is merged at control join points,
- how primitive instructions are modeled.

The first item is the interprocedural analysis, while the second and third together constitute the intraprocedural part of the analysis.

3.2 Interprocedural Analysis

We begin with the interprocedural part of the analysis. Another predicate defined by the locking analysis is

```
predicate cedge(I:c_instr,T:t_trace,SIN:lockstate,SOUT:lockstate).
```

The type `c_instr` is our first example of a type defined in a *package*, in this case the `translatecil` package (see Section 8.11). A package is a collection of types, predicates, and rules implemented in another language, such as C or OCaml (the two languages that are currently supported); package `translatecil` describes all the types and predicates that encode the CIL syntax for a C program. In the predicate `cedge`, while the type `c_instr` is any CIL instruction (assignments, assembly, and function calls), in fact this predicate is only used to record information about function calls—the name `cedge` stands for *call edge*. The meaning of the predicate is that function call `I` maps the lock with trace `T` from initial lock state `SIN` to final lock state `SOUT`. Information for specific call sites is computed via CLP *rules* that add new facts, instances of the `cedge` predicate. For example:

```
dircall(I,"lock"),
    +cedge(I,drf{root{arg{0}}},locked,error),
    +cedge(I,drf{root{arg{0}}},unlocked,locked).
```

All CLP rules (and, indeed, rules in any logic programming language) describe how to infer new facts. Together they give a declarative specification for inferring the full set of facts within a function; the CLP interpreter simply applies the analysis rules in all possible places to infer new facts, until there are no more rules which can be applied.

Most rules have a simple form such as the above, a comma-separated list of predicates. The rule is evaluated by walking down it left to right. Predicates with a `+` in front of them are *additions*, and represent new facts which should be inferred, added to the set of known facts. Predicates with no `+` in front are *finds*, indicating that for any known fact matching the predicate, any variables such as `I` should be instantiated and the rest of the rule evaluated. This rule introduces two other new things:

- The predicate `dircall(I,FN)` is true if the instruction `I` is a *direct call* of the function named `FN` (Section 5.2.1). A direct call means that the function name `FN` appears explicitly in the program, as in `lock(l)`, in constrast to an *indirect call* through a pointer such as `(*f)(l)`.

- The trace arguments to the `cedge` predicates indicate specific memory locations. Because we are specifying the locking semantics of the *lock* primitive, we must say explicitly which memory location holds the affected lock. The term *trace* is meant to suggest a trace or sequence of operations needed to reach the location from some *root*, which is a location with a program name such as a local or global variable or, in this case, a function argument. Thus, `drf{root{arg{0}}}` is the dereference of the root `arg{0}`, the first argument of the function.

With all of this in mind we can now explain what the rule means: For any instruction `I` which is a direct call to the primitive function `lock`, then two facts are added. If the lock pointed to by the first argument of the call is `locked` on entry to `I`, then the lock is in the `error` state on exit; i.e., there is a double-locking bug caused by this call to `lock`. Similarly, if the lock pointed to by the first argument of the call is `unlocked` on entry, then the lock is in the `locked` state on exit. The rule for direct calls to `unlock` is similar:

```
dircall(I,"unlock"),
    +cedge(I,drf{root{arg{0}}},locked,unlocked),
    +cedge(I,drf{root{arg{0}}},unlocked,error).
```

Note that the trace `drf{root{arg{0}}}` is the location of the formal parameter in the called function, not the location of the actual parameter that is passed from the caller. We return to this point in Section 3.4.

For calls to functions other than the primitive locking operations we must communicate information between the caller and the callee. Each function computes *summary edges* that describe its net locking behavior:

```
predicate sedge(T:t_trace,SIN:lockstate,SOUT:lockstate).
```

The interpretation of an `sedge` is that some function other than `lock` or `unlock` may map a lock at location `T` that is in the `SIN` state on function entry to the `SOUT` state on function exit. Note that there is no argument indicating the function the predicate is talking about. We add this information using a *session* definition which will be used to store the

```
session sum_locking(FN:string) containing [sedge].
```

Sessions are the mechanism used to store summaries for interprocedural analysis. While predicate instances represent individual facts, sessions represent sets of such facts (in this case, only the `sedge` predicate) which can be directly updated or queried by the analysis. When a function is being analyzed, it will update its own summary (see below), and when calls to it are encountered, its summary will be queried:

```
dircall(I,F), sum_locking(F)->sedge(T,SIN,SOUT), +cedge(I,T,SIN,SOUT).
```

This rule means that for any direct call to any function `F`, query `F`'s summary information to determine what `sedges` it has, and then add new `cedge` facts in the same manner as `cedge` facts were added for calls to `lock` and `unlock`.

So far we have discussed how summary information for a function `F` is propagated to a call site for `F`. The other half of interprocedural analysis is how summary information is computed for a function in the first place. The locking analysis introduces another predicate

```
predicate trace_trans(T:t_trace,G:g_guard,SIN:lockstate,SOUT:lockstate).
```

which holds if the entire body for the currently analyzed function transitions a lock at location `T` from an initial lock state `SIN` to a final lock state `SOUT` if guard `G` holds. Computing the `trace_trans` facts using the memory model and `cedge` facts is the goal of the intraprocedural analysis, discussed starting in Section 3.3. The rule for computing locking summaries is then:

```
cil_curfn(F), trace_trans(T,SG,SIN,SOUT), guard_sat(SG),
    +fedge(F,T,SIN,SOUT), +sum_locking(F)->sedge(T,SIN,SOUT).
```

The predicate `cil_curfn` names the currently analyzed function; it is provided by the `translatecil` package (Section 8.11). The predicate `guard_sat` holds if its boolean formula argument is satisfiable; it is provided by the memory model (Section 5.3.4) as a wrapper for the boolean constraint solver interface in the `solve_sat` package (Section 8.8). Thus, this rule says that if function `F` maps trace `T` from state `SIN` to `SOUT` under condition `SG` and there is some execution in which that condition can be satisfied, then the transition is added to the function summary for `F`. This rule is a typical example of the use of constraints in Saturn: we compute constraints (normally boolean constraints but interfaces to other constraint theories are provided by other packages) and then at some point ask whether the constraints are satisfiable. Unsatisfiable constraints represent only infeasible computations; in this case we do not add a transition to the function summary if its associated guard is unsatisfiable. Note also that this rule performs abstraction: we discard the guard `SG` in the locking summary, effectively saying that the transition can always happen and not just when `SG` is satisfied. Computing a very precise intraprocedural analysis which is abstracted in procedure summaries for scalability and termination is also typical of Saturn analyses.

Finally, the rule above uses a new predicate `fedge` with the same signature as `sedge` except the function name. This is not a summary predicate, but is just used in a *query* to print as output all matching facts that were added to the summary:

```
predicate fedge(FN:string,T:t_trace,SIN:lockstate,SOUT:lockstate).

?- fedge(F,A,SIN,SOUT).
```

3.3 Join Points

We turn now to the rules for the intraprocedural analysis—the analysis within a single procedure. Like dataflow analyses, the Saturn locking analysis can be described as a combination of *transfer functions* for each kind of statement (Section 3.4) and how information at join points (places where multiple control paths merge) is computed:

```
predicate smerge(P:pp,T:t_trace,S:lockstate,G:g_guard).
```

The predicate `smerge` has exactly the same signature as the `state` predicate: the guard `G` under which a lock `T` has a certain lock state `S` at a program point `P`. In fact, `smerge` is used to compute `state` at a program point. The invariant that the analysis maintains is that while there may be many `smerge` facts for a program point, one for each distinct control path reaching that program point, there will be only one `state` facts per program point, representing the merge of all `smerge` facts. The rule that uses `smerge` facts is

```
smerge(P,T,S,_), pp_order(P,DEP,DIST), $order(DEP,DIST,0),
    \/smerge(P,T,S,G):#or_all(G,MG), +state(P,T,S,MG).
```

This rule introduces two more new features. The first is the *collection* over `smerge(P,T,S,G)`. Normally a find on a predicate succeeds once for every fact that matches it, separately instantiating the remainder of the rule with the values of the variables in the find. A collection succeeds exactly once for the set of all facts matching the predicate. Collections are consumed by special *collection predicates*, which perform some reduction on the set. In the rule above, the collection predicate `#or_all` (provided by the package `biteval`, Section 8.7) binds the merged guard `MG` to the disjunction of all the guards `G` in the collection. We then add the single `state` fact that at point `P` lock `T` is in lock state `S` if the single guard `MG` is satisfied.

Merging the guards in this way does not lose information, but allows the internal representation for the merged guards to be simplified substantially, avoiding the exponential blowup in the size of formulas and number of separate `state` facts that could otherwise be encountered.

The first line of the rule serves two distinct and important roles:

- The initial find `smerge(P,T,S,_)` identifies any `smerge` fact and binds `P`, `T`, and `S` to ground terms, but it does not bind `G` (the '`_`' is a *wildcard* matching any value but introducing no bindings). Thus, the collection, which is evaluated with `P`, `T`, and `S` bound to particular values, only has one unbound variable `G` over which it ranges. This behavior is exactly what we want: for each separate `P`, `T`, and `S` we want to compute the disjunction of all the possible guards `G`. The initial find will succeed multiple times for different combinations of program points, locks, and lock states, and one `state` fact will be computed for each.

- There is a problem with collections in a logic programming language. The collection requires that *all* facts that match the collection be present when the rule is evaluated—the collection can only be safely evaluated when there is no possibly that another rule will add another fact belonging to the collection. The CLP interpreter uses *stratification* (Section 4.11) to determine statically an order of evaluation for the rules so that evaluation of collections are evaluated at a point where it is known that no future fact additions could *invalidate* the collection. However, stratification cannot always analyze rule sets precisely enough, and so CLP provides a mechanism for the programmer to specify the dynamic execution order of rules. In the rule above, the `$order(DEP,DIST,0)` says the collection evaluations should be ordered lexicographically by *depth* `DEP` and *distance* `DEP`, which are properties of program points given by the predicate `pp_order`. Depth refers to loop nesting depth and distance to a program points position in a depth-first numbering of the control-flow graph. The order is such that all the `smerge` facts for a program point are computed before the rule above executes.

3.4 Statements

At the start of a function body we initialize the lock memory locations to some initial state. Functions are analyzed separately and in a bottom up call graph order, so when a function is analyzed we do not know what states the locks may be in at call sites of the function. Thus, we analyze the function under the assumption that the lock may be either locked or unlocked. However, we also know that the lock cannot be both locked and unlocked simultaneously. We use entry_locked, defined below, to capture this last restriction:

```
predicate entry_locked(in T:t_trace,LKG:g_guard,UKG:g_guard).

?entry_locked(T,_,_), #id_g(br_abit{ar_extra{t_locked{T}}},LKG),
    #not(LKG,UKG), +entry_locked(T,LKG,UKG).

entry(PIN), icall(P0,_,I), cedge(I,CT,_,_),
    inst_trace(s_call{I},P0,CT,trace{T},_), entry_locked(T,LKG,UKG),
    +state(PIN,T,locked,LKG), +state(PIN,T,unlocked,UKG).
```

The predicate entry_locked records a trace (lock) T and the initial conditions on entry to the function under which T is locked LKG and unlocked UKG. Consider now the first rule. Ignoring for the moment the predicate begining with ?, the second predicate creates a new unconstrained boolean variable using #id_g and binds it to LKG. The #not(LKG,UKG) negates LKG and binds the result to UKG. (Both #id_g and #not are defined in the package biteval, Section 8.7) So, if LKG is the boolean variable β, then UKG is $\neg\beta$, and the rule adds the fact entry_locked(T, β, $\neg$beta) and we see that the initial state of the lock is either locked or unlocked, but not both. The br_abit{ar_extra{t_locked{T}}} identifier used in #id_g is a unique identifier for the new unconstrained variable; it ensures we associate each different lock trace T with a different boolean variable.

The first predicate of the rule above is our first example of a *wait* predicate. This wait only succeeds for traces T where some other rule tries to evaluate a find entry_locked(T,...). Waits are used to express demand-driven computations in what is otherwise an eager evaluation model. Without waits, a rule is evaluated exhaustively for all combinations of values that succeed. In logic programming terminology, CLP uses a *bottom up* evaluation strategy. Without the wait in the rule above, then, an entry_locked fact would be added for every memory location being modeled, not just the locations that hold locks. The wait ensures that the rule applies only to locations that some part of the analysis is interested in—i.e., the locks. Thus, waits express a demand-driven or *top down* evaluation strategy which can be freely mixed with the base bottom up strategy (the Prolog-style notation 'head :- body' for rules is also supported, and converted internally into waits).

The second rule infers the locations of locks accessed by the current function, and adds their initial state at the CFG entry point. Since locks are only manipulated by primitive functions or by other functions that themselves call the locking primitives, we discover that a lock location is accessed by finding the called function which has a lock state transition in its summary for that location. Writing the rule above for this idea requires three new predicates:

- The predicate `entry(P)` identifies the unique program point `P` that is the entry point of the current function body (Section 5.2.1).
- The predicate `icall(P0,P1,I)` identifies each function call `I` of any form (direct or indirect) where `P0` and `P1` are the program points before and after the call, respectively (Section 5.2.1).
- The predicate `inst_trace`, which stands for *instruction trace* is central to interprocedural analysis (Section 5.3.6). Recall that call edges (`cedge` facts) record the location (trace) of a lock in the scope of the callee (i.e. if `cedge` holds for `drf{root{arg{0}}}` then it is the first argument to the call, not the current function). We need the corresponding location (trace) in the caller, and `inst_trace` provides this mapping: the first and second arguments identify the call site (the call instruction and program point), the third argument is the trace in the callee, and the fourth argument is the corresponding trace in the caller. The fifth argument is a guard under which the mapping holds, which for this rule is not used because we just need the locks that **could** be accessed within the current function.

We can now explain in detail how the rule discovers which locks are accessed by the current function. Consider a function call at a program point `P0` (the find on `icall`) that has a call edge on location `CT` (the find on `cedge`), which means that `CT` has a lock state transition and is therefore a lock. Now, if `CT` corresponds to caller location `T` (the find on `inst_trace`), then we look up the initial conditions `LKG` and `UKG` under which `T` is locked and unlocked respectively (the find on `entry_locked`, which triggers the wait in the previous rule) and add appropriate `state` facts for the entry program point of the function.

Now that the lock states at the function entry point have been set up, they must be propagated forward through the CFG to find out their new states at exit. The locking analysis models three kinds of CFG transitions: assignments, conditional branches, and function calls.

The rule for assignment statements `iset` is:

```
iset(P0,P1,_), state(P0,T,S,G), eguard(P0,P1,G,EG), +smerge(P1,T,S,EG).
```

Analogous to `icall`, the `iset` predicate has three arguments: the program point before the assignment `P0`, the program point after the assignment `P1`, and the assignment instruction itself. Assignments statements cannot affect the states of locks, so we do not care what the assignment actually is. We are not even concerned with whether pointers to locks are created and copied by assignmens, because all of that is covered by the underlying memory model. Recall that traces represent memory locations that hold locks, not program variables; the memory model infers what variables refer to which locations. The locking analysis never needs to explicitly refer to that information, it simply keeps track of the memory locations that are locks.

Returning the assignment rule, it is read as follows. Consider an assignment between program points P0 and P1. If the state of lock T is S at point P0 under guard G, then the state of T at P1 will also be S but with a possibly different guard EG (the add of smerge). The guard EG is computed using the eguard predicate (Section 5.3.5), which combines the guard G tracked by the locking analysis with whatever guard the underlying memory model has inferred controls the transfer of control from P0 to P1.

The next two rules deal with conditional branches:

```
branch(P,P0,_,_), state(P,T,S,G), eguard(P,P0,G,EG0), +smerge(P0,T,S,EG0).
branch(P,_,P1,_), state(P,T,S,G), eguard(P,P1,G,EG1), +smerge(P1,T,S,EG1).
```

These rules are similar to the iset rule, except that a branch has two possible successors (P0 and P1 above).

The remaining statement form is function calls. Since function calls are the only statements that actually affect lock states, there are two different cases to handle, depending on whether there is a transition for the lock. Any call which does not have a transition on a lock does not affect its state. First we consider the case where a lock has an explicit transition:

```
icall(P0,P1,I), cedge(I,CT,SIN,SOUT),
    inst_trace(s_call{I},P0,CT,trace{T},BG),
    state(P0,T,SIN,SG), #and(BG,SG,G),
    eguard(P0,P1,G,EG), +smerge(P1,T,SOUT,EG).
```

Consider a a call I between program points P0 and P1. If there is a call edge cedge for this call where callee trace CT makes a lock state transition from SIN to SOUT, and the corresponding caller trace is T if guard BG is true and lock T is in state SIN if guard SG holds before the call, then after the call T is in state SOUT if the conjunction of BG and SG is true.

The second case is where there is no transition on the lock. We must be careful because the same lock may be accessed in multiple ways and under different conditions, so we need to compute the condition under which the lock is not accessed by *any* callee transition.

```
predicate edge_negate(P:pp,T:t_trace,NG:g_guard).
icall(P,_,I), cedge(I,CT,_,_),
    inst_trace(s_call{I},P,CT,trace{T},G),
    #not(G,NG), +edge_negate(P,T,NG).
```

The edge negate predicate records the complement of the condition under which trace T is affected by a lock state transition at a call site.

```
icall(P0,P1,_), state(P0,T,S,SG),
    pp_order(P0,DEP,DIST), $order(DEP,DIST,1),
    \/edge_negate(P0,T,NG):#and_all(NG,MNG), #and(MNG,SG,G),
    eguard(P0,P1,G,EG), +smerge(P1,T,S,EG).
```

Consider a lock T in at the program point P0 before a function call. We can get the condition MNG under which T is not affected by any transition on the call using a collection. This is then conjoined with the condition SG under which T was in state S before the call to get a state where T is still in state S after the call.

Once again, because of the collection, the stratification algorithm needs ordering information to guarantee that all elements of the collection are available when the collection is evaluated. Note that the above rule handles the situation where a lock is completely unaffected by a call (i.e., there are no transitions on the lock at all), in which case the conjunction of the negation of all the transition guards is simply *true*.

The following rule specifies that any lock in the error state is still in the error state after a function call:

```
icall(P0,P1,_), state(P0,T,error,G),
    eguard(P0,P1,G,EG), +smerge(P1,T,error,EG).
```

Recall that trace_trans is used to capture the net effect of an entire function body on a lock. The final rule generates these trace_trans facts:

```
exit(P), state(P,T,S,SG), entry_locked(T,LKG,UKG),
    #and(SG,LKG,LKGG), +trace_trans(T,LKGG,locked,S),
    #and(SG,UKG,UKGG), +trace_trans(T,UKGG,unlocked,S).
```

This rule first finds a lock T in state S under condition SG at the exit point P of the function body. If the condition under which T was locked on entry to the function body is LKG, then under the conjunction of conditions SG and LKG the function body maps T from state locked to state S. The case where T is unlocked on entry is similar.

3.5 Examples

Change into the directory analysis/locking/base04 and examine the file run.clp:

```
rm *.dot *.db 2> /dev/null
cilcc test.c
clpa --quiet ../simplelocking.clp $*
```

The first line cleans out any .dot and .db files from previous runs. If a CLP program gives unexpected results, one of the first things to check is whether it is reading from stale databases. Issuing the command ./run while in this directory runs the simple locking analysis on the file test.c. We explain the output in conjunction with the functions in test.c. The function foo is

```
void foo(spinlock *a)
{
  lock(a);
  unlock(a);
}
```

and the inferred fedges for foo in the output are

```
fedge("foo",drf{root{arg{0}}},locked,error).
fedge("foo",drf{root{arg{0}}},unlocked,unlocked).
```

Thus, the analysis correctly infers that foo locks its argument if it is initially unlocked and has a double locking error if the argument is initially locked. The function bar is

```
void bar(spinlock *a, int b)
{
  if (b)
    lock(a);
  if (b)
    unlock(a);
}
```

and the analysis results for bar are

```
fedge("bar",drf{root{arg{0}}},locked,error).
fedge("bar",drf{root{arg{0}}},locked,locked).
fedge("bar",drf{root{arg{0}}},unlocked,unlocked).
```

Here we can see the path-sensitivity of the analysis, which has discovered that an unlocked lock is always left in the same state by a call to bar. Recall that in writing the analysis we were not concerned with path sensitivity at all. The necessary analysis and correlation of the branch conditions has been performed by the imported memory analysis. A locked lock can either cause a double locking error (if b is true) or be left in the same state by bar (if b is false). The code for sip is

```
int sip(spinlock *a) {
  if (g) {
    lock(a);
    return 1;
  } else {
    return 0;
  }
}
```

and the results of the analysis are

```
fedge("sip",drf{root{arg{0}}},unlocked,locked).
fedge("sip",drf{root{arg{0}}},locked,locked).
fedge("sip",drf{root{arg{0}}},unlocked,unlocked).
fedge("sip",drf{root{arg{0}}},locked,error).
```

In this example we can see that the analysis has also done as well as it can; without further information, all and only the possible state transitions of the code are summarized by the fedges. The function sip is used by sap:

```
void sap(spinlock *a)
{
  int i;
  i = sip(a);
  if (i) {
    unlock(a);
  }
}
```

for which the simple locking analysis gives the following results:

```
fedge("sap",drf{root{arg{0}}},unlocked,locked).
fedge("sap",drf{root{arg{0}}},locked,locked).
fedge("sap",drf{root{arg{0}}},unlocked,unlocked).
fedge("sap",drf{root{arg{0}}},locked,unlocked).
fedge("sap",drf{root{arg{0}}},locked,error).
fedge("sap",drf{root{arg{0}}},unlocked,error).
```

Now we can see a shortcoming of the simple locking analysis, which is that in practice it is common for return values to indicate the state of locks manipulated by a function. For example, primitive *trylock/tryunlock* functions return a boolean indicating whether or not the lock/unlock operation succeeded. Functions like `sap` then conditionally apply locking operations based on this return value. Because the simple locking analysis does not track the correlation between the lock state and the return value, the analysis of `sap` yields no information at all: the lock can transition from any possible input state to any possible output state. We discuss extensions that do a better job with return values in Section 3.7. The final example is the function `wrong`

```
void wrong(spinlock *a, int b)
{
  while (b) {
    lock(a);
  }
}
```

which has a double locking error assuming that `b` can ever be true. The analysis produces the following rather surprising results:

```
fedge("wrong",drf{root{arg{0}}},unlocked,unlocked).
fedge("wrong",drf{root{arg{0}}},locked,locked).
```

No error is reported, so this summary is clearly not conservative—the simple locking analysis is unsound. The issue is that the analysis does not handle loops; there is a separate instruction form for loops, and the analysis we have presented simply ignores it. While this approach is useful for bug finding, it is not adequate for any verification application. We discuss the preferred method of handling loops in sound analyses in Section 3.6.

3.6 Handling Loops

From the point of view of program analysis handling loops is in most respects a special case of analyzing recursive functions. Saturn provides a standardized form of control-flow construction that literally converts loops into tail recursive functions, making it possible to treat all recursion uniformly.

The file `locking/simplelocking2.clp` is a modified version of `simplelocking.clp` that handles loops correctly. There is a test case for this modified version of the simple locking analysis in `analysis/locking/base07`; we begin by examining the `run` script in this directory:

```
rm *.dot *.db 2> /dev/null
cilcc test.c
clpa --quiet ../../base/sumbody.clp
clpa --quiet ../simplelocking2.clp $*
```

The new third line runs the stand-alone analysis `sumbody.clp`, which encapsulates each loop in its own control flow graph. This file gives an example of how CLP programs can use the databases other CLP programs compute, rather than importing the other programs directly in the same CLP execution.

The databases produced by `sumbody.clp` present an interface that differs from the base control-flow graphs. In particular, functions must be identified not just by name, but also by the place in the function where the control-flow graph came from in the initial code. The main differences between `simplelocking2.clp` and `simplelocking.clp` are:

- The analyze directive at the beginning of the program is

  ```
  analyze session_name("cil_sum_body").
  ```

  This directive is needed in any program that is analyzing the function bodies produced by `sumbody.clp`.

- In several places throughout the analysis, the type `c_instr` is replaced by `sum`, which is a union of all the types of instructions that can count as functions (i.e., the function itself, function calls, loops, and assembly directives). The name `sum` is meant to suggest that these are the instructions that require function summaries. Because `sum` includes loops, this change generalizes rules that previously worked only for function calls to also work for loops.

- The `sum_locking` session predicate is changed to take two arguments, the function name and a particular `sum` within the function.

- Matches on direct calls `dircall` and call instructions `icall` are replaced by the predicate `isum(P0,P1,I)`, which matches every instruction `I` of `sum` type, and, if needed, `isum_target(I,F,S)`, which gives the function name `F` and `sum S` of the target of the call `I`.

- Uses of the predicates `entry` and `exit`, which match the entry and exit program points of a function body respectively, are replaced by a combination of `cil_cursum(S)`, which matches the current `sum` being analyzed, and `sum_bound(S,PIN,POUT)`, which gives the entry and exit points `PIN` and `POUT` of the `sum`, respectively.
- The predicate `fedge` is given a new `sum` argument, so that the output shows both the function name and the part of the control flow graph (the main function body or a loop) of the locking facts.

In the directory `analysis/locking/base07` execute `./run`. The output of the modified locking analysis differs only in the information for the function `wrong`. Notice that output is generated both for the loop and the function several times during the analysis—the implementation repeats the analyses as many times as needed to reach a fixed point where no more facts are added to any of the session predicates. The final `fedge` facts inferred for `wrong` are

```
fedge("wrong",s_func,drf{root{arg{0}}},unlocked,unlocked).
fedge("wrong",s_func,drf{root{arg{0}}},locked,locked).
fedge("wrong",s_func,drf{root{arg{0}}},unlocked,locked).
fedge("wrong",s_func,drf{root{arg{0}}},locked,error).
fedge("wrong",s_func,drf{root{arg{0}}},unlocked,error).

fedge("wrong",s_loop{"s#0"},drf{root{arg{0}}},locked,locked).
fedge("wrong",s_loop{"s#0"},drf{root{arg{0}}},unlocked,unlocked).
fedge("wrong",s_loop{"s#0"},drf{root{arg{0}}},unlocked,locked).
fedge("wrong",s_loop{"s#0"},drf{root{arg{0}}},locked,error).
fedge("wrong",s_loop{"s#0"},drf{root{arg{0}}},unlocked,error).
```

The second group of facts gives the facts for the loop, which say that an error (specifically, a double locking error) is possible whether the lock is initially locked or unlocked. It is also possible that the lock will be untouched, or that the lock will be acquired (if the lock is not initially held and the loop executes exactly once).

3.7 Interprocedural Path Sensitivity

The memory analysis provides full path sensitivity within a single function body, but accurate analysis sometimes requires tracking predicates across function calls as well. In the case of analyzing locks, the most important interprocedural predicates correlate a function's return value with the state of locks on exit from the function. The file `analysis/locking.clp` generalizes `simplelocking.clp` to incorporate return values. The fundamental change is to the definition of call edges:

```
predicate cedge(I:c_instr,T:t_trace,SIN:lockstate,SOUT:lockstate,NEZ:bool).
```

The new definition adds the boolean argument `NEZ`, for "not equal to zero", indicating the state transition both when the return value is non-zero (`NEZ` is true) and when the return value is zero (NEZ is false). A void function that does not return a value is by convention treated as if it always returns zero.

This change allows the analysis to model *trylocks*, which, as discussed above, are functions with a return value indicating whether the lock/unlock operation was successful. So, for example, the trylock function (which attempts to acquire a trylock), is modeled as

```
dircall(I,"trylock"),
    +cedge(I,drf{root{arg{0}}},locked,locked,false),
    +cedge(I,drf{root{arg{0}}},unlocked,unlocked,false),
    +cedge(I,drf{root{arg{0}}},locked,error,true),
    +cedge(I,drf{root{arg{0}}},unlocked,locked,true).
```

If the locking operation fails (return value is zero) the state of the trylock is unaffected. If the operation succeeds (return value non-zero) trylock is acquired if it was not previously held; otherwise, there is a double-locking error.

Other significant differences from the simple locking analysis are:

- The signature of sedge is also extended to include an NEZ argument.
- A few rules use negation, a CLP feature we have not yet discussed. For example, in the rule

  ```
  call_merge(I,P0,P1,false,T,S,G), ~callret(I,_),
       eguard(P0,P1,G,EG), +smerge(P1,T,S,EG).
  ```

  the second goal matches all calls I that do not return a value. Just as with collection predicates, we must take care that the negations are tested after no further facts that might match the goal can be added. In general it may be necessary to supply ordering information to help the system with the order of evaluation of rules with negations, but for this particular rule CLP's stratification algorithm finds an ordering with the other rules in the analysis without additional information.
- There is a new predicate

  ```
  predicate return_nez(NEZ:bool,G:g_guard).
  ```

  that tracks the guard under which the current function returns a zero or non-zero value.
- Because the analysis requires knowledge of return values to compute the summary predicates, there are new rules that manipulate return values. The following rule

  ```
  cil_var_return(X), cil_make_lval(X,"rlv",LV), cil_make_offset(X,"roff",OFF),
      +cil_lval_var(LV,X,OFF), +cil_off_none(OFF),
      cil_make_exp(X,"rexp",E), +cil_exp_lval(E,LV), exit(P), evals(P,E,V),
      #id_g(br_cmp{sc_eqz{V}},EQZG), +return_nez(false,EQZG),
      #not(EQZG,NEZG), +return_nez(true,NEZG).
  ```

> is perhaps the most involved rule in the locking analysis. The first part of the rule gets the return variable for the function, which is always in a variable `cil_var_return(X)` that the front-end parser guarantees exists for functions that return values. The next several goals create an expression `E` from `X` that can be dereferenced. The value of the return variable at the function exit point `P` is bound to `V` by the predicate `evals(P,E,V)` (the `evals` predicate is computed by the memory analysis). The final goals create a fresh guard capturing when `V` is zero or non-zero.

In the directory `locking/base08` execute `./run`. Note that the results for function `wrong` have reverted to the unsound results of `simplelocking.clp`; the program `locking.clp` does not include the modifications to analyze loops, which is suggested as an exercise for the interested reader. The analysis of functions `sip` is, however, much more refined than before:

```
fedge("sip",drf{root{arg{0}}},locked,locked,false).
fedge("sip",drf{root{arg{0}}},unlocked,unlocked,false).
fedge("sip",drf{root{arg{0}}},unlocked,locked,true).
fedge("sip",drf{root{arg{0}}},locked,error,true).
```

The first two facts say that if the return value is zero then the lock is unaffected by the call to `sip`. If the return value is `true` then either the lock is acquired or there is a double-locking error (i.e., if `sip` attempts to acquire an already held lock). The knowledge of the relationship between `sip`'s return values and its locking behavior greatly improves the analysis of `sap`:

```
fedge("sap",drf{root{arg{0}}},locked,error,false).
fedge("sap",drf{root{arg{0}}},unlocked,unlocked,false).
fedge("sap",drf{root{arg{0}}},locked,locked,false).
```

The system now correctly infers that either the lock ends in the same state it started in, or there is a double locking error if the lock is initially held and the call to `sip` attempts to acquire it again.

Chapter 4

Saturn CLP Language Reference

4.1 Overview

This chapter describes the syntax and semantics of the CLP logic programming language used within Saturn for writing program analyses. The full logic program grammar is shown in Figure 4.1. Each program consists of a series of rules, queries, definitions, and directives, each terminated with a period. Each of these is described in turn.

4.2 Facts

The fundamental unit of discourse in a logic program is the *fact*. Every fact is an instance of some *predicate* paired with an application of some *session function*. These encode, respectively, the fine-grained and coarse-grained aspects of the fact. For example, to encode the fact that some variable `X` may point to `Y` at program point `P` in the body of function `F`, we use the predicate `pointsto(P,X,Y)` and session function `body(F)`, representing the fact as `body(F)->pointsto(P,X,Y)`.

Using session functions allows us to analyze each function in a program separately and yet allow for information dependencies between functions. Rules are always evaluated in the context of a particular session (typically a function body), referring to predicates *pred* in that session as simply *pred*, and predicates *pred* in other sessions *sess* as $sess{-}{>}pred$. The name of the session being used for evaluation can be accessed by using the session name as a regular predicate (i.e. the session `body(F)` always contains the fact `body(F)`).

$$\begin{array}{rcl} pred & ::= & \textbf{symbol}\ \text{‘(’}\ val, ..., val\ \text{‘)’} \\ sess & ::= & \textbf{symbol}\ \text{‘(’}\ val, ..., val\ \text{‘)’} \\ spred & ::= & pred \mid sess\ \text{‘}{-}{>}\text{’}\ pred \end{array}$$

program ::= ϵ | *toplevel* ‘.’ *program*
toplevel ::= *rule* | *query* | *typedef* | *preddef* | *sessdef* | *analyze* | *using* | *import*

pred ::= **symbol** ‘(’ *val, ..., val* ‘)’
sess ::= **symbol** ‘(’ *val, ..., val* ‘)’
spred ::= *pred* | *sess* ‘−>’ *pred*
val ::= ‘_’ | **var** | **string** | **int** | **float** | *listval* | *sumval*
listval ::= ‘[’ *val, ..., val* ‘]’ | ‘[’ *val, ..., val* ‘|’ **var** ‘]’
sumval ::= **symbol** | **symbol** ‘{’ *val, ..., val* ‘}’

rule ::= *goal* | *spred* ‘:-’ *goal*
goal ::= ‘+’ *spred* | *spred* | ‘?’ *pred* | *val* ‘=’ *val* | *val* ‘\=’ *val*
| *goal* ‘,’ *goal* | *goal* ‘;’ *goal* | ‘(’ *goal* ‘)’ | ‘∼’ *goal* | ‘\/’ *goal* ‘:’ *pred*
query ::= ‘?-’ *spred*

type ::= **symbol** | **symbol** ‘[’ *tval, ..., tval* ‘]’
tval ::= [**var** ‘:’] **var** | [**var** ‘:’] *type*

typedef ::= ‘type’ *type* | ‘type’ *type* ‘=’ *type* | ‘type’ *type* ‘::=’ *tsumval*|...|*tsumval*
tsumval ::= **symbol** | **symbol** ‘{’ *tval, ..., tval* ‘}’

preddef ::= ‘predicate’ **symbol** ‘(’ *mtval, ..., mtval* ‘)’
mtval ::= *tval* | ‘in’ *tval* | ‘out’ *tval*

sessdef ::= ‘session’ **symbol** ‘(’ *tval, ..., tval* ‘)’ [‘containing’ ‘[’ **symbol**, ..., **symbol** ‘]’]

analyze ::= ‘analyze’*pred*
using ::= ‘using’ **symbol**
import ::= ‘import’ **string**

Terminals **symbol** are alphanumeric strings beginning with a lowercase letter, **var** are alphanumeric strings beginning with a capital letter, **string** are double quoted escaped strings, **int** are integer constants, and **float** are floating point constants. Lists of values *val, ..., val* are comma separated, while sums of values *tsumval*|...|*tsumval* are separated by ‘|’.

Figure 4.1: Logic program grammar

Arguments to predicate instances *pred* and session function applications *sess* may be the wildcard value '_', named variables, constant strings, integers, floats, lists, or user-defined *composite sum-values.* Lists may be expressed as either a fixed length list of zero or more elements `[E1,E2,...,En]` or as a variable length list with one or more head elements and a tail `[E1,E2,...,En|TAIL]`. For more information on sum-values, see Section 4.5.

$$\begin{array}{rcl} \mathit{val} & ::= & \text{`\_'} \mid \textbf{var} \mid \textbf{string} \mid \textbf{int} \mid \textbf{float} \mid \mathit{listval} \mid \mathit{sumval} \\ \mathit{listval} & ::= & \text{`['}\ \mathit{val}, ..., \mathit{val}\ \text{`]'} \mid \text{`['}\ \mathit{val}, ..., \mathit{val}\ \text{`|'}\ \textbf{var}\ \text{`]'} \\ \mathit{sumval} & ::= & \textbf{symbol} \mid \textbf{symbol}\ \text{`\{'}\ \mathit{val}, ..., \mathit{val}\ \text{`\}'} \end{array}$$

4.3 Rules

Derivation rules infer and manipulate facts about the program of interest. When a rule is executed, it queries various facts of interest, instantiating any free variables it has, and may or may not add new facts in response. All rules in the program execute independently from one another, and communicate via adding and querying new facts. Any particular rule may be executed many times, once for each set of predicates that match the queries it makes.

Two kinds of rules are supported. The first kind has the form `Goal`, where `Goal` is a valid goal, as described in the next section. The last action performed by `Goal` must be an add operation. The second kind of rule is a Prolog-style horn clause, and has the form `Head :- Body`, where `Head` is a valid predicate term, and `Body` is a valid goal. Whenever `Body` successfully executes, the fact given by `Head` will be added.

$$\mathit{rule} \quad ::= \quad \mathit{goal} \mid \mathit{spred}\ \text{`:-'}\ \mathit{goal}$$

Note that a horn-clause style rule `Head :- Body` is equivalent to the rule `?Head, Body, +Head`. Conversely, the first kind of rule can be written as a sequence of horn-clause style rules, using one rule per add goal. As both styles of rule are logically equivalent, the choice of which to use is merely a stylistic preference.

A goal is the unit of execution. Each goal represents an operation, and may either succeed or fail. Whenever a goal succeeds, it binds zero or more variables to particular ground values, which can be used to instantiate the remainder of the rule. A goal may succeed in multiple different ways, binding the same variables to different ground values. A goal has one of the following forms:

$$\begin{array}{rcl} \mathit{goal} & ::= & \text{`+'}\ \mathit{spred} \mid \mathit{spred} \mid \text{`?'}\ \mathit{pred} \mid \mathit{val}\ \text{`='}\ \mathit{val} \mid \mathit{val}\ \text{`\textbackslash='}\ \mathit{val} \\ & \mid & \mathit{goal}\ \text{`,'}\ \mathit{goal} \mid \mathit{goal}\ \text{`;'}\ \mathit{goal} \mid \text{`('}\ \mathit{goal}\ \text{`)'} \mid \text{`}\sim\text{'}\ \mathit{goal} \mid \text{`\textbackslash/'}\ \mathit{goal}\ \text{`:'}\ \mathit{pred} \end{array}$$

4.3.1 Conjunction

A conjunction goal $goal_1, ..., goal_n$ consists of two or more goals separated by the conjunction operator ','. Each subgoal $goal_i$ is executed in turn, from left to right. If

$goal_i$ succeeds, the remainder of the conjunction is instantiated using the variables bound by $goal_i$ (the remainder may thus be instantiated many different times if $goal_i$ succeeds in multiple ways) and then executed. The entire conjunction succeeds if and only if all subgoals succeed, and binds all variables bound by any of its subgoals.

Most rules can be written as a single conjunction goal, where each subgoal is either an add or find (see below).

4.3.2 Disjunction

A disjunction goal $goal_1; ...; goal_n$ consists of two or more goals separated by the disjunction operator ';'. Each subgoal $goal_i$ is executed independently from one another. The entire disjunction succeeds if and only if any subgoal succeeds, and binds the variables bound by the subgoal that succeeded.

Nested conjunctions and disjunctions can be grouped using parentheses if necessary. Note that conjunction goals bind more tightly than disjunction goals, so that `G0, G1; G2, G3` is equivalent to `(G0, G1); (G2, G3)`.

4.3.3 Add

The add goal $+spred$ indicates that *spred* should be added to the set of known facts. An add goal always succeeds, and binds no variables. Rules consisting only of add goals encode axioms in the program.

Example 4.1 Consider the problem of encoding the finite state machine for a spin lock. We use a single predicate `trans(S0,SLK,SUK,S1)`. `S0` is the state of the lock before the transition, and `S1` is the state after. `SLK` and `SUK` indicate what `S1` is if `S0` is locked or unlocked, respectively. If `S0` is in the error state, `S1` will also be in the error state.

We encode this definition of `trans` using add rules, making one rule each for whether `S0` is locked, unlocked, or in the error state:

```
+trans("locked",SLK,_,SLK).
+trans("unlocked",_,SUK,SUK).
+trans("error",_,_,"error").
```

We could also have placed all three add operations in the same rule using a conjunction goal.

4.3.4 Find

A find goal *spred* succeeds if any (present or future) known fact matches *spred*, binding all free variables in *spred* to the corresponding values in any such fact.

Example 4.2 Continuing the example with `trans`, now we want rules for evaluating the 'lock' and 'unlock' functions on values passed into them. We use two new predicates: `state(P,X,S)` indicates that at program point `P`, lock `X` is in state `S`.

`call(P0,P1,F,X)` indicates that there is a call to `F` with argument `X`, where `P0` and `P1` are the program points directly before and after the call, respectively.

We encode the operation of 'lock' and 'unlock' as follows:

```
call(P0,P1,"lock",X), state(P0,X,S0),
    trans(S0,"error","locked",S1), +state(P1,X,S1).

call(P0,P1,"unlock",X), state(P0,X,S0),}
    trans(S0,"unlocked","error",S1), +state(P1,X,S1).
```

4.3.5 Wait

The wait goal ?*pred* succeeds whenever a find operation is performed by another rule in the program on a predicate matching *pred*, binding any free variables in *pred* to the corresponding values in the find operation.

Example 4.3 Consider the problem of evaluating dereference expressions under a flow-sensitive points-to analysis; particular dereferences may have many different targets depending on the point that they are evaluated at. We use three predicates: `pointsto(P,X,Y)` indicates that at program point `P`, `X` may point to `Y`. `exp_deref(E,ED)` indicates that `E` is the dereference expression of `ED` (i.e. `E` = ∗`ED`). `eval(P,E,X)` indicates that at program point `P`, `E` may evaluate to `X`.

We can encode the dereference operation as follows:

```
exp_deref(E,ED), eval(P,ED,X), pointsto(P,X,Y), +eval(P,E,Y).
```

However, this rule will be exhaustively applied, evaluating the targets of `E` at every point in the function. We are generally interested in the value of `E` at only a few program points of interest and can delay evaluation of the rule to only these points using the wait operation as follows:

```
exp_deref(E,ED), ?eval(P,E,_),
    eval(P,ED,X), pointsto(P,X,Y), +eval(P,E,Y).
```

In effect, the wait operation is an explicit name for the magic version of a predicate produced by the magic sets transformation, which is a means of obtaining demand-driven evaluation in the otherwise bottom-up evaluation. The magic sets transformation is applied only to those predicates with an associated wait operation.

4.3.6 Equality/Disequality

The equality goal $v0 = v1$ succeeds if and only if $v0$ and $v1$ are unifiable, binding any free variables in $v0$ or $v1$. The disequality goal $v0 \backslash= v1$ succeeds if and only if $v0$ and $v1$ are **not** unifiable, and does not bind any variables. In both cases either $v0$ or $v1$ must be ground.

4.3.7 Negation

The negation ∼*goal* succeeds if and only if *goal* fails, and binds no variables.

Example 4.4 In the following program, the print operation will not be executed because the fact `item(1)` will be derived from the second rule.

```
~item(1), +print(1).
+item(1).
```

However in the following program if `item(1)` does not initially hold, it will be added and invalidate the negation. Cycles like this are detected and reported during parsing by the rule stratification algorithm (see Section 4.11).

```
~item(1), +item(1).
```

4.3.8 Collection

The collection goal \/*goal* : *pred* quantifies over all variable bindings produced by *goal* when it succeeds, using a special collection predicate *pred*. A collection goal is similar to a find goal, except that rather than succeeding many times, once for each matching fact, the goal succeeds just once, binding some variable(s) to the result of the collection. Several collection predicates are available, with each specifying a different way to combine the set of facts and instantiate the remainder of the rule.

Example 4.5 In the following program, the print operation is executed three times, once for each item. The result is `1`, `2`, and `3` on separate lines.

```
+item(1), +item(2), +item(3).
item(X), +print(X).
```

In contrast, in the following program, the print operation is executed once for the list of items. The result is a single line with the list `[1,2,3]`.

```
+item(1), +item(2), +item(3).
\/item(X):list_all(X,LIST), +print(LIST).
```

4.4 Queries

Queries supply a predicate, printing to the output all known matching facts.

$$query \quad ::= \quad \text{'?-'}\ spred$$

4.5 Type Definitions

All predicates and sessions in the language are well-typed, and all program rules are statically type-checked during parsing. Each language type is a symbol, which may take any number of typed arguments. The arguments may be type variables (in the case of polymorphic types) or concrete types, prefixed by an optional name (this name is unused by the checker and is provided to aid documentation). Types for builtin constants are `string`, `int`, and `float`. Lists have polymorphic type `list[T]`, which may be instantiated with any type `T`.

$$\begin{aligned} \textit{type} &::= \textbf{symbol} \mid \textbf{symbol}\ \text{‘[’}\ \textit{tval}, ..., \textit{tval}\ \text{‘]’} \\ \textit{tval} &::= [\textbf{var}\ \text{‘:’}]\ \textbf{var} \mid [\textbf{var}\ \text{‘:’}]\ \textit{type} \end{aligned}$$

Programs are free to define their own types using the ‘type’ keyword at the program’s top level. Type definitions can be simple forward declarations (which must be filled in later), aliases for other types, or sets of possible composite sum-values, using a BNF-style notation. Once a type is defined as a set of sum-values, those sum-values (and any arguments) may be used in rules as predicate arguments or components of other sum-values.

$$\begin{aligned} \textit{typedef} &::= \text{‘type’}\ \textit{type} \mid \text{‘type’}\ \textit{type}\ \text{‘=’}\ \textit{type} \mid \text{‘type’}\ \textit{type}\ \text{‘::=’}\ \textit{tsumval}|...|\textit{tsumval} \\ \textit{tsumval} &::= \textbf{symbol} \mid \textbf{symbol}\ \text{‘\{’}\ \textit{tval}, ..., \textit{tval}\ \text{‘\}’} \end{aligned}$$

4.6 Predicate Definitions

Any predicate used in a program must be defined before it can be used. Predicate definitions supply types and modes for each argument, which are used during parsing to check the rules for errors but do not affect the runtime behavior of the rules. Modes are used exclusively in predicate definitions, and are either ‘in’ or ‘out’, with a default value of ‘out’ if omitted. When doing a find, negate, or collect operation on the predicate, ‘in’ arguments must be ground (can’t contain any wildcards or variables) and ‘out’ arguments may be anything. When doing an add operation on the predicate, ‘out’ arguments must be ground and ‘in’ parameters may be anything.

$$\begin{aligned} \textit{preddef} &::= \text{‘predicate’}\ \textbf{symbol}\ \text{‘(’}\ \textit{mtval}, ..., \textit{mtval}\ \text{‘)’} \\ \textit{mtval} &::= \textit{tval} \mid \text{‘in’}\ \textit{tval} \mid \text{‘out’}\ \textit{tval} \end{aligned}$$

4.7 Session Definitions

Any session function used in a program must be defined before it can be used, supplying the argument types. Optionally the definition can also specify the set of predicate names which may be contained within the session, which will be checked whenever an add or find goal is performed on the session.

$$\textit{sessdef} ::= \text{‘session’}\ \textbf{symbol}\ \text{‘(’}\ \textit{tval}, ..., \textit{tval}\ \text{‘)’}\ [\text{‘containing’}\ \text{‘[’}\ \textbf{symbol}, ..., \textbf{symbol}\ \text{‘]’}]$$

Whenever a session is used explicitly in an add or find goal, all the session arguments must be ground. Note that when predicates are added to a session during analysis, the effect of that add is only visible after the analysis of the current session terminates—operationally, new facts are only committed to the databases once the session completes. If the current session performs both add and find operations on the same summary session, then it has created a circular dependency and it will be continually reanalyzed by the interpreter until the summary session stops changing (i.e. a fixpoint has been reached).

Additionally, any facts that are added to the current session *without* using the session name are not committed to the databases. Thus, the new facts that are needed by other sessions must be added using the `+S->P` form.

Several special built-in predicates may be used in add goals to manipulate sessions at a broader level than basic adds and finds. These are listed below.

- `merge_preds()`

Add `merge_preds()` in combination with another session to merge all the predicates in that session into the currently analyzed one. For example, performing the add `+summary(FN)->merge_preds()` merges all predicates in `summary(FN)` into the current session. This is usually done to extend a function body session with information about other source program symbols (structures, globals, etc.) relevant to the analysis but not included in the function body session.

- `clear_old_preds()`

Add `clear_old_preds()` in combination with another session to remove all of the predicates previously existing in that session. Normally, predicates added into a session are merged with the predicates previously existing in that session once a function body has finished being analyzed. By performing a `clear_old_preds()` this behaviour is overridden, and only the newly added predicates will be present in the session on completion.

- `process_dependency(SNAME:string,...)`

When iterating over all the sessions in a program, the Saturn interpreter processes sessions in a topological sort according to a process order graph. For example when analyzing function bodies, a function is by default usually be analyzed before its callers. The frontend generating the initial session databases fills in this initial dependency information, however for sessions generated by scripts there is no default ordering. Adding the predicate `S0(ARGS0)->process_dependency(S1,ARGS1)` indicates that that session `S1(ARGS1)` should be placed before `S0(ARGS0)`. Note that this is evaluation order is not guaranteed (in case there are cycles in the graph, for example), and that the additional dependency does not take effect until the next execution of the interpreter.

4.8 Analyze Directives

Analyze directives specify properties that control analysis at an interprocedural level. Each directive takes the form of a predicate, and can be specified at most once.

$$\textit{analyze} \quad ::= \quad \text{`analyze'}\ \textit{pred}$$

The possible analyze directives are as follows:

- `session_name(SNAME:string)`

The session to iterate over, with `SNAME` given by an earlier 'session' definition. All known sessions mapped by the named session function are processed during interprocedural analysis.

- `analysis_name(ANAME:string)`

The name of this analysis, printed out with each processed session.

- `topdown()`

When analyzing sessions representing function bodies, order interprocedural analysis to process them in a top down order.

- `bottomup()`

When analyzing sessions representing function bodies, order interprocedural analysis to process them in a bottom up order. This is the default if neither `topdown()` nor `bottomup()` is specified.

- `eager()`

Always use the interprocedural analysis order (topdown or bottomup) to order sessions in this analysis. This overrides the default behavior, which is to use the interprocedural analysis order only if two sessions have already been analyzed the same number of times, and to prefer sessions that have been reanalyzed fewer times otherwise.

- `priority(PRI:int)`

When cofixpointing between multiple analyses, set the relative priority of this analysis. This analysis is always run before other analyses that have a higher value for `PRI`.

4.9 Using Directives

Built-in constraint solvers and other packages define their own set of predicates, which can be used by the rules after a 'using' directive is added for the package name. The list of available packages is provided in Section 8.

using ::= 'using' **symbol**

4.10 Import Directives

Import directives add the rules and directives in another logic program to the current one. The other program's file name is supplied, which may be absolute or relative to the directory containing the current program.

import ::= 'import' **string**

4.11 Stratification

The Saturn interpreter uses *stratification* for ordering the evaluation of rules in a CLP program. Stratification looks at the dependencies introduced between different rules by the adds, finds, and other operations performed, and uses these dependencies to prioritize which parts of which rules are executed first whenever the interpreter has the choice of executing multiple rules.

Stratification is only necessary in the presence of negation and collection goals. If a program does not contain any negations or collections, the result is always the same regardless of rule execution order.

Example 4.6 Consider the following program:

1. `foo(), +bar(1).`
2. `foo(), +bar(2).`
3. `+foo().`

If no facts are known, then the initial find operations performed by rules 1 and 2 will initially fail, so only rule 3 can be executed. This will add fact `foo()`, so that both rules 1 and 2 may now be executed. If rule 1 is evaluated first, `bar(1)` will be added followed by `bar(2)`. If rule 2 is evaluated first, `bar(2)` will be added followed by `bar(1)`. In both cases however, the final set of facts is the same, and unaffected by the ordering used.

Now, consider the use of a negation or collection goal. Whenever a negation is executed by the interpreter, the currently known set of facts is inspected, and if the subgoal cannot currently succeed, the negation goal succeeds. If a fact is subsequently added that causes the subgoal to succeed (and the negation to fail), the negation will have been invalidated and the interpreter result may be incorrect. Similarly, whenever a collection is executed, all currently known ways for the subgoal to succeed will be examined and used to generate the collection's output variable bindings. If an additional way is subsequently found for the subgoal to succeed, the collection will have been invalidated.

Stratification solves this problem by finding an ordering for the program's rules that avoids invalidation of any negation or collection goals. Conceptually, we find all rules that could lead to facts being added which affect a particular negation/collection, and make sure those rules are always evaluated before the negation/collection.

Example 4.7 Consider the following program:

```
1.  foo(), +bar().
2. +foo().
```

If no facts are known, then if the interpreter initially attempts to execute rule 1, the negation will succeed and `bar()` will be added. When rule 2 is subsequently executed, the added `foo()` will invalidate the earlier negation. The stratification algorithm will recognize that rule 1 depends on predicate `foo`, and rule 2 may generate `foo`, so rule 2 should be executed first. In this case then, `foo()` is added, the negation in rule 1 fails, and `bar()` is **not** added, the correct result.

Stratification is more difficult in the case of cyclic dependencies. What if two negations/collections in different rules depend on each other? In this case the stratification algorithm will not be able to decide what ordering to use, and will need assistance from the analysis writer.

Example 4.8 Consider the following program:

```
1. ~bar(), +foo().
2. ~foo(), +bar().
```

Either rule 1 or rule 2 may be executed first. If rule 1 is executed first, `foo()` will be added and rule 2 will fail. If rule 2 is executed first, `bar()` will be added and rule 1 will fail. Thus the analysis result is non-deterministic and extra information is needed to find the right evaluation order. The stratification algorithm generates a warning for this program.

The `$order` builtin predicate may be used as a find subgoal to order the evaluation of any negations or collections appearing later in the rule, relative to any additional negations or collections in the same stratification cycle. `$order` takes any number of integer-typed arguments (which may be variables whose value was bound by previous subgoals), though within a particular stratification cycle the actual number of arguments used should be fixed between the different uses of `$order`.

Example 4.9 Returning to the program from the earlier example, we can use `$order` to force a particular order of execution.

1. `$order(0), ~bar(), +foo().`
2. `$order(1), ~foo(), +bar().`

In this case rule 1 will always be executed first, according to its higher `$order(0)` priority, eliminating the non-determinism.

From the analysis-writing perspective, `$order` is most useful when performing collections or negations while traversing an acyclic structure, such as a control flow graph, expression tree, or loop nesting hierarchy. Using `$order` gives enough expressiveness that the correct rule ordering can be expressed. Note, however, that using `$order` does not eliminate the chance of invalidating negations/collections or of non-deterministic rules. In the former case an error message is printed for each invalidation that occurs, indicating an incorrect use of `$order`.

Chapter 5

Saturn CLP Analysis Implementations

5.1 Overview

This chapter describes the CLP analyses and analysis infrastructure included with Saturn. The analysis infrastructure includes general purpose CLP files that are not stand-alone analyses, but are imported and used by most Saturn stand-alone analyses:

- Section 5.2 describes the CFG construction infrastructure components, including various ways of generating CFGs as well as marking per-function points of summarization (loops, calls, etc.)

- Section 5.3 describes the memory model used by analyses. This includes the naming method for abstract locations and values, the path-sensitive intra-procedural model of points-to and value aliasing information, and the inter-procedural propagation of information about abstract locations.

The remaining sections document the following stand-alone analyses:

- Section 5.4 describes the alias analysis and how it interacts with the memory model.

- Section 5.5 describes the NULL pointer dereference analysis.

5.2 CFG and Summary Construction

To perform a flow- or path-sensitive analysis on a function, we first need a control flow graph describing the various instructions that may be executed and the jumps and conditional branches between them. The source CIL syntax databases generated do not contain any control flow information beyond the location of if statements, while statements, etc. The CLP files 'base/cilcfg.clp' and 'base/loops.clp' are responsible for generating full control flow graphs for the function and its loops based on these syntax predicates.

Each CFG is a graph whose nodes are program points and whose edges are program actions (Section 5.2.1). Loops in the function are usually handled by splitting them off from the rest of the function's body, generating a smaller CFG with a tail-recursive invocation of the loop itself (Section 5.2.2). This ensures that each function/loop CFG is an acyclic graph, a crucial property needed by the memory model described in Section 5.3.

Within a function the loops, calls, and outer function itself are all natural points for generating summary information. The common treatment of these summaries is described in Section 5.2.3.

Any analysis generating CFGs can import the CLP file 'base/cfgdot.clp', which generates, for each function 'foo', a DOT graph file 'foo_cfg.dot' which shows the various nodes and edges in the CFGs of 'foo,' and can be viewed using `dotty` or any other DOT viewer.

5.2.1 CFG program points and actions

Individual program points in the CFG have type `pp`. These indicate distinct points in the possible execution paths for a function. Type `pp` can be considered abstract (though it is defined in 'base/cilcfg.clp'), however there are a few important predicates for reasoning with points:

- `entry(P:pp)`: `P` is the unique entry point of the current function.
- `exit(P:pp)`: `P` is the unique exit point of the current function.
- `point_location(P:pp,FILE:string,LINE:int)`: `P` corresponds to a location on line `LINE` of source file `FILE`. Note that there may be multiple points on any given line.

Different program points are connected to one another by edges representing the different program actions which may be taken. The different edges are as follows:

- `iset(P0:pp,P1:pp,I:c_instr)`: Points `P0` and `P1` are connected by a CIL set instruction `I`, such that `cil_instr_set(I,LV,E)` for some `LV/E`.
- `iasm(P0:pp,P1:pp,I:c_instr)`: Points `P0` and `P1` are connected by a CIL set instruction `I`, such that `cil_instr_asm(I,TARGETS)` for some `TARGETS`. By default assembly is ignored when generating the CFG. To enable construction of `iasm` edges, add the predicate `cfg_translate_asm()`.
- `icall(P0:pp,P1:pp,I:c_instr)`: Points `P0` and `P1` are connected by a CIL call instruction `I`, such that `cil_instr_call(I,FNE)` for some `FNE`. Call instruction targets can be found with the `dircall(I:c_instr,FN:string)` predicate identifying direct calls only, or the `anycall(I:c_instr,FN:string,T:calltype)`

predicate identifying both `direct` and `indirect` calls depending on `T`. Computing possible targets of indirect calls requires the alias analysis to be run, see Section 5.4.4.

- `iloop(P0:pp,P1:pp,L:loop)`: Points `P0` and `P1` are connected by an inner loop `L` within this function. Generation of `iloop` edges is discussed in Section 5.2.2.
- `branch(P:pp,P0:pp,P1:pp,E:c_exp)`: Program execution will branch from `P` to `P0` if expression `E` holds at `P`, or to `P1` if `E` does not hold.

Example 5.1 Consider the following C program:

```
void bar();
void foo(int *x, int b, int c)
{
    if (b) *x = 0;
    if (c) *x = 1;
    else   bar();
}
```

Execute the following CLP program:

```
import "base/loops.clp".
import "base/cfgdot.clp".
analyze session_name("cil_body").
```

This yields the file 'foo_cfg.dot', which can be viewed with `dotty` and is shown in Figure 5.1. Each circle is a program point, with edges between them the different CFG edges; for each `branch(P,P0,P1,E)`, there is an `ebtrue` connecting `P` with `P0`, and an `ebfalse` connecting `P` with `P1`. The entry and exit points are labelled with boxes. Note that strings such as `"#5"` are the unique identifiers used for different CIL expressions, instructions etc. These can be converted to a non-unique human-readable representation using the `exp_string` etc. predicates defined in 'base/cilbase.clp'.

5.2.2 Handling loops

Directly translating the source function's syntax into a CFG will generate a loop wherever the source function contains a loop, either an explicit while or for loop, or an implicit loop created by goto back edges. Many analyses expect a loop-free CFG though in order to terminate, so there are several ways to treat the source function's loops during CFG construction.

- Preserving back edges. Importing 'base/loops.clp' and adding the predicate `preserve_loops()` will preserve any source function loops in the final CFG. This is incompatibe with the memory model as described in Section 5.3, but can be useful for writing a CFG-based analysis from scratch.

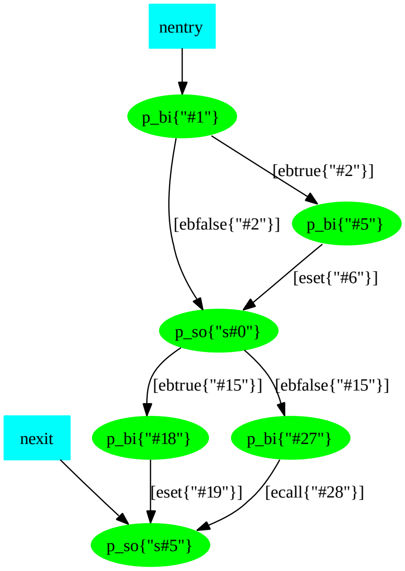

Figure 5.1: DOT graph for Example 5.1

- Breaking back edges. This is the default behavior resulting from just importing ‘base/loops.clp’. Any back edges for loops which introduce cycles will simply be omitted from the CFG, and there will not be any path in the CFG from most points within the loop body to points in the function after the loop executes.

- Splitting loops into tail recursive CFGs. Importing ‘base/loops.clp’ and adding the predicate `split_loops()` will split all loops apart, generating one CFG for the outer function and one CFG for each inner loop. The flow of control between these CFGs is encoded in `iloop(P0,P1,L)` edges, which indicate that the named loop is to be executed. For each loop, `iloop` edges will be generated at two places: at the initial invocation of the loop by the function (or an outer loop), and at the end of the loop’s own CFG where the back edge would normally be, encoding a tail recursive invocation of the loop. For propagating summary information between inner/outer loops and the function itself, see Section 5.2.3.

Example 5.2 Consider the following C program:

```
int foo(int b, int c)
{
    while (b) {
        if (c) return 0;
    }
```

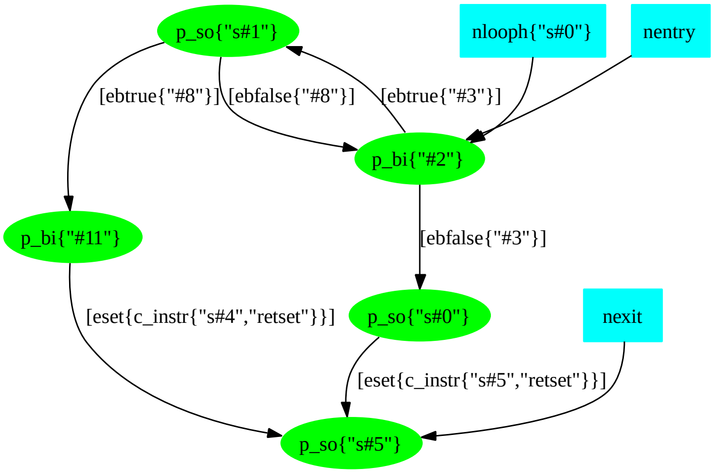

Figure 5.2: DOT graph for Example 5.2 preserving loops

```
    return 1;
}
```

Figure 5.2 shows the CFG for this function with back edges preserved, i.e. after adding `preserve_loops()`. Figure 5.3 shows the default CFG for this function with back edges broken. Note that this graph is identical to that in Figure 5.2, except that the back edge from `p_so{"s#1"}` to `p_bi{"#2"}` (the false branch of the if statement) has been removed and replaced with an edge to the point `p_lnext{"s#1"}`, which is not connected to anything.

Figure 5.4 shows the CFG for this function where the loop body is split off. Loop edges have been introduced and the loop head `p_bi{"#2"}` is now in a CFG separate from the main function.

Most analyses based on the memory model in Section 5.3 use loop splitting, as this preserves the full source program semantics of the loops. The default behavior of breaking back edges is useful though for quickly getting an analysis up and running (and easy to convert to split loops later), and in some cases works in general as well.

5.2.3 Summaries and CFGs

Summary-based program analyses usually want to attach summary information to multiple points within a function. These include calls, inline assembly, inner loops, as well as the function itself. The `sum` type describes these common summarization points and can be used to fold together properties shared by all of them into single predicates and small sets of rules, simplifying the analysis-writing process. The `sum` type is defined as follows:

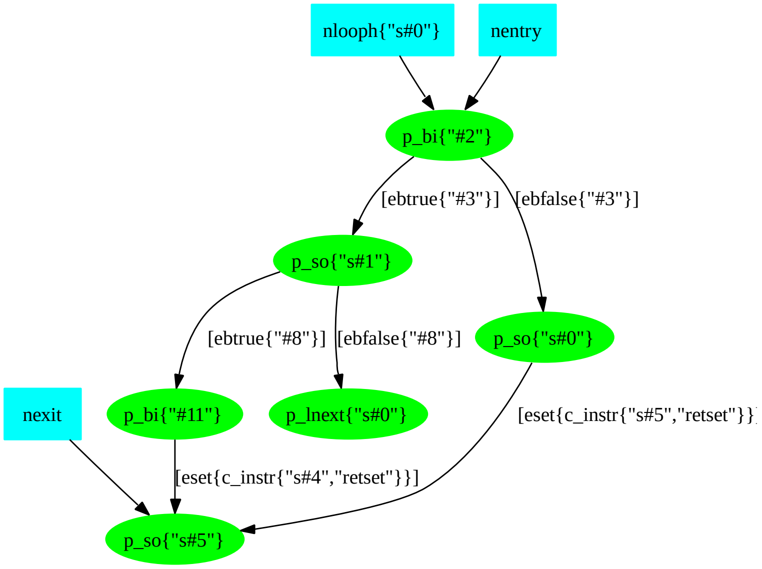

Figure 5.3: DOT graph for Example 5.2 breaking loop back edges

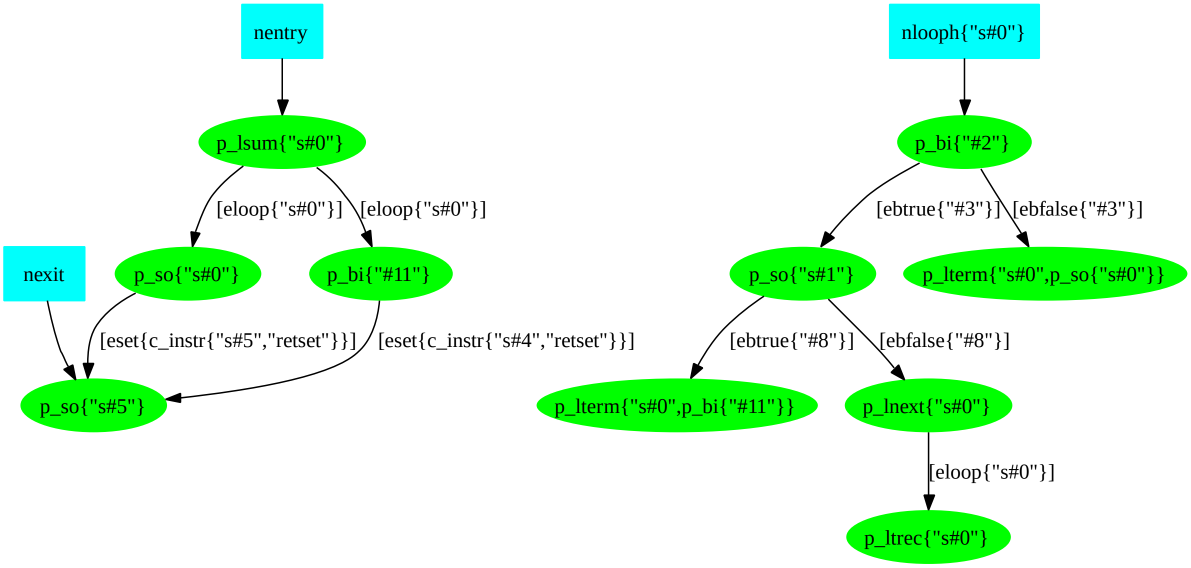

Figure 5.4: DOT graph for Example 5.2 splitting loop into separate CFGs

- s_func: The current function.
- s_loop{L:loop}: A loop within the current function.
- s_call{I:c_instr}: A call instruction within the current function.
- s_asm{I:c_instr}: An inline assembly instruction within the current function.

Conceptually, every sum value is defined in one place and invoked in another. The client program analysis attaches a summary to each sum to bridge this gap, generating the summary at the definition, and applying the summary at the invocation (or vice versa). Several predicates can be used to help this process along:

- sum_body(SUM:sum,P:pp): For s_func and s_loop, identifies all points in the CFG which defines the behavior of SUM.
- sum_bound(SUM:sum,PIN:pp,POUT:pp): For s_func and s_loop, identifies the entry/exit points of the CFG which defines the behavior of SUM.
- isum(P0:pp,P1:pp,SUM:sum): Folds icall, iloop, and iasm CFG edges into a single predicate. Summary SUM is executed at P0, and when it finishes control is transferred to P1.
- isum_target(SUM,CFN,CSUM): Gets the definition point corresponding to an s_call or s_loop summary. For calls this will be the s_func sum within any possible callee, and for loops this will just be the same loop within the current function.

Example 5.3 Consider the problem of identifying which functions may transitively call the exit function and prematurely terminate the program. We can compute this by finding direct calls to exit, and pushing this information bottom-up through direct and indirect calls and loop nestings, computing a summary session sum_may_exit with the desired property.

```
import "base/loops.clp".
analyze session_name("cil_body").

+split_loops().

session sum_may_exit(FN:string) containing [smayx].
predicate smayx().

% SUM within the current FN may exit. SUM may be s_func, s_loop or s_call
predicate may_exit(SUM:sum).

% get direct calls to exit
dircall(I,"exit"), +may_exit(s_call{I}).
```

```
% propagate bottom up through inner loops and calls
anycall(I,CFN,_), sum_may_exit(CFN)->smayx(), +may_exit(s_call{I}).
isum(P,_,CSUM), may_exit(CSUM), sum_body(SUM,P), +may_exit(SUM).

% update the current function's summary
may_exit(s_func), cil_curfn(FN), +sum_may_exit(FN)->smayx().
```

An additional option for writing analyses built around the `sum` type is to perform the analysis on the `cil_sum_body` session rather than the `cil_body` session. Instead of analyzing an entire function at a time which may contain additional CFGs for each loop, a single `sum` is analyzed at a time, identified by the `cil_cursum(SUM:sum)` predicate (`SUM` may be either the outer function `s_func` or some `s_loop`).

To populate the `cil_sum_body` sessions, run the 'base/sumbody.clp' stand-alone analysis.

Example 5.4 Consider again the problem of finding functions which may exit prematurely. After running `sumbody.clp` we can run a slightly terser analysis which computes this information at the granularity of individual functions/loops, not just individual functions.

```
import "base/loops.clp".
analyze session_name("cil_sum_body").

session sum_may_exit(FN:string,SUM:sum) containing [smayx].
predicate smayx().

% the current SUM may exit
predicate may_exit().

% get direct calls to exit
dircall(I,"exit"), +may_exit().

% propagate bottom up through inner loops and calls
isum(_,_,SUM), isum_target(SUM,CFN,CSUM),
    sum_may_exit(CFN,CSUM)->smayx(), +may_exit().

% update the current summary
may_exit(), cil_curfn(FN), cil_cursum(SUM), +sum_may_exit(FN,SUM)->smayx().
```

5.3 Memory Analysis

The Saturn memory model is used to construct a precise, path-sensitive model of all the points-to and integer-value relationships at each point in the current function and its loops. Most Saturn analyses build on top of this model, querying the state at different points to construct summary information and error reports. The CLP files 'memory/traces.clp', 'memory/paths.clp', 'memory/memory.clp', 'memory/scalar.clp' and 'memory/scalar_sat.clp' are collectively responsible for constructing the memory model. When writing an analysis, only 'memory/scalar_sat.clp' needs to be included.

The core predicate representing the path-sensitive points-to graph for the function is `val(P:pp,S:t_trace,T:t_trace_val,G:g_guard)`. This indicates that at point `P` in the function, the memory location represented by `S` has the value `T` (which may be the address of another memory location) when boolean condition `G` holds. Memory locations `t_trace` are described in Sections 5.3.1 and 5.3.2, location values `t_trace_val` are described in Section 5.3.3, and boolean formulas `g_guard` are described in Section 5.3.4. These are then tied together in describing the path-sensitive points-to graph (Section 5.3.5) and propagation of location information between procedures (Sections 5.3.6 and 5.3.7).

The memory model can be used either with or without the alias analysis described in Section 5.4. If the alias analysis is run, the potential aliases found will be stored in summary databases and used automatically by the memory model when constructing the function's initial points-to graph and when applying pointer side effects for function calls. If the alias analysis is **not** run, then the memory model will assume non-aliasing between different pointers when constructing the initial graph and applying side effects. Note that in both cases, the memory model will precisely model the effects of any intra-procedural aliasing, i.e. that introduced by assignments performed within the currently analyzed function (Section 5.3.5).

5.3.1 Memory location traces

Each concrete memory location that can be accessed by the current function is represented within the memory model as a value of type `t_trace`. Traces are represented as a series of zero or more memory operations (dereferences and field accesses) performed on a root variable with a fixed stack or static allocation. Traces have several important properties:

1. Any concrete location is represented by at most one trace. No two different traces can represent the same concrete location and thus be aliased.

2. Every concrete location reachable via memory operations from the root variables is represented by a trace. Unreachable data has no representation; this includes, for example, data local to any callers which did not escape to the callee.

3. Traces may represent more than one concrete memory location. These are termed *soft* traces; see Section 5.3.2.

4. The trace representation for a memory location is canonical within each `sum` (either the function or an inner loop; see Section 5.2.3). Traces are defined in terms of the program state at entry to the `sum`, and the trace used to represent a location is not affected by any assignments or other operations performed.

5. The trace representation for a memory location is **not** canonical across different `sum` values. The same memory location may be represented by different traces within a caller vs. its callee, or by an outer function vs. an inner split loop. Each trace must then be converted when crossing these boundaries (Section 5.3.6).

Type `t_root` describes the possible stack- or statically-allocated root variables for a trace, and is defined as follows:

- `arg{A:int}`: The stack location of argument number `A` (zero-indexed) to the current function.
- `glob{G:string}`: The statically allocated location of global variable `G`. Note that global variables and functions declared as `static` will be renamed to the form 'file:globname' during parsing to ensure uniqueness.
- `local{L:string}`: The stack location of local variable `L` to this function.
- `return`: The stack location storing the current function's return value.
- `temp{TMP:string,WHERE:string}`: The stack location of a temporary variable used by this function. `TMP` is a unique identifier for the temporary, while `WHERE` is a non-unique usage description for the temporary. Most temporary variables store call return values, in which case `WHERE` indicates the called function's name.
- `asm_in{A:int}`: Input argument to an inline assembly instruction.
- `asm_out{A:int}`: Output argument to an inline assembly instruction.
- `cstr{STR:string}`: The statically allocated location of a constant string, read-only global data.

Type `t_trace` is then defined as follows:

- `root{R:t_root}`: The stack/static location of root variable `R`.
- `drf{T:t_trace}`: The result of dereferencing the trace `T` *at entry to the current* `sum`. Any assignments to `T` performed later in the `sum` do **not** affect the location represented by `drf{T}`. Note that for any two different pointer traces `T1` and `T2`, since `drf{T1}` and `drf{T2}` are different and thus `T1` and `T2` are effectively non-aliased. If `T1` and `T2` are considered possibly aliased at entry to the `sum` (due to the alias analysis), one of `drf{T1}` and `drf{T2}` will be chosen by the memory model to represent the single target.
- `fld{T:t_trace,F:string,C:string}`: Field `F` of `T`, with respect to composite type `C`. Where `T` represents the base location of a structure, `fld{T,F,C}` is the location resulting from adding to `T` the offset into type `C` of field `F`.

- `rfld{T:t_trace,F:string,C:string}`: Reverse of `fld{T,F,C}`. Where `T` is an internal location for field `F` of a structure of type `C`, `rfld{T,F,C}` is the base location of the structure itself. This models the common C idiom of passing internal pointers to structures between functions, and then using pointer arithmetic to recover the base pointer of the structure.

The predicate `trace_root(in T:t_trace,R:t_root)` fetches the root variable associated with each trace. For example, the trace `fld{drf{root{arg{2}}},"f","str"}` has root `arg{2}`.

Traces sometimes need to be directly constructed or traversed by analyses, and several predicates can be used to perform such operations.

- `trace_simplify(in SUM:sum,in OT:t_trace,NT:t_trace)`: Used when applying memory operations `drf`, `fld`, and `rfld` to an existing trace to construct a new trace. Where `OT` is an existing trace with a single memory operation applied, applies any necessary simplifications to get a new trace `NT` which should be used subsequently. These simplifications are necessary to ensure each concrete memory location has a single trace representation, and include folding together opposing `fld` and `rfld` operations (`fld{rfld{T,F,C},F,C}` is equivalent to `T`), and folding recursive structure traversals (`drf{fld{drf{T},"next","list"}}` is equivalent to `drf{T}`).

- `trace_sub(in T:t_trace,TS:t_trace,TR:t_trace)`: Gets any subtrace `TS` of `T`, setting `TR` to the series of memory operations needed to be applied to `TS` to yield `T`. `TR` does not contain a root, but rather is a *relative* trace containing the special value `empty`. For example, for the trace `fld{drf{root{arg{0}}},"f","str"}` the following hold:

  ```
  trace_sub(*,fld{drf{root{arg{0}}},"f","str"},empty)
  trace_sub(*,drf{root{arg{0}}},fld{empty,"f","str"})
  trace_sub(*,root{arg{0}},fld{drf{empty},"f","str"})
  ```

- `trace_compose(in SUM:sum,in TS:t_trace,in TR:t_trace,T:t_trace)`: Reverses `trace_sub`, composing the subtrace `TS` with the relative trace `TR` to yield the original trace `T`.

- `trace_relative(in T:t_trace)`: `T` does not have a root, and instead contains the special value `empty` and represents a series of memory operations rather than an actual set of locations.

5.3.2 Softness and aggregate memory traces

Most traces manipulated within a function represent a single concrete memory location. For example, traces for the stack locations of arguments, local variables, and so forth always represent a single location. However, in order to capture the possible properties of arrays, recursive structures and other aggregates a single trace will be used which represents every corresponding memory location within that structure. These *soft* traces must be treated in special ways by many analyses.

Soft traces can be identified with the `trace_soft(in SUM:sum,in T:t_trace,ST:soft_type)` predicate, where `ST` indicates why the trace is soft (if it is so), chosen from these values:

- `array`: `T` is an array of known length either stack/statically allocated, or inline with some structure.

- `ptarray`: `T` is the target of some pointer passed in from outside `SUM` – `T` is `drf{PT}` for some `PT` – and `PT` was used in pointer arithmetic somewhere in `SUM` to walk down an array. Pointers passed in are soft only if multiple different elements may be accessed through operations `x[i]`, `*(x+i)` etc. Identifying these requires interprocedural analysis and is performed as part of the alias analysis (Section 5.4.3).

- `recurse`: `T` is a recursive structure whose links were traversed somewhere in `SUM` to walk down the structure. As with `ptarray`, recursive structures with multiple cells are soft only if multiple different cells were accessed, and again identifying these requires the alias analysis to be run first.

- `softsub`: `T` contains a soft subtrace, and can thus represent multiple different locations itself. If `T` is soft, `drf{T}` represents all the initial targets of the locations represented by `T`.

5.3.3 Integer/location trace values

Values that a trace can possess are represented with the type `t_trace_val`.

- `trace{T:t_trace}`: Address of the memory location(s) represented by `T`. If trace `T0` points to `T1`, `T0` has value `trace{T1}`. `trace{drf{T}}` is also used to model the initial value of an integer-typed location `T` passed into the `sum` being analyzed. In this case `drf{T}` can be viewed as the location resulting from casting `T` to a pointer and dereferencing it (legal to do in C); straight assignments and reads of pointer and integer values are handled in the same way within the memory model.

- `nrep{N:t_nrep}`: Particular intermediate integer value resulting from a computation performed within the current `sum`. These can represent integer constants, integer arithmetic operations, coercion results, etc.

Several different models can be used for intermediate integer values computed within a `sum`, each of which defines `t_nrep` differently. For most purposes the model used by ‘scalar.clp’ is best, which precisely models almost all integer operations using values of type `t_nrep = scalar`. Each `scalar` generated by the memory model can be treated as abstract, though predicate `scalar_string(in scalar,out string)` generates a fairly readable representation for each value.

Type `scalar` is defined as a set of sum values in 'scalar.clp', and primarily consists of unconstrained initial trace values and unary/binary operations over these. New `scalar` values can be constructed simply as instances of these sum values.

5.3.4 Guards and path-sensitivity

All conditions used to encode path-sensitive information and other constraints within the memory model (and most additional analyses) are represented with the type `g_guard = bval[g_bit]`, i.e. boolean formulas with arbitrary combinations of conjunction, disjunction and negation over variables of type `g_bit`. Each `g_guard` represents a specific condition which either holds or does not hold in any concrete execution, and can be freely combined with `#and`, `#or` and `#not`, or inspected with `guard_string(in g_guard,out string)`. Rather than using `#sat` to determine satisfiability, however, which just looks at the boolean connectives to determine satisfiability and not the meaning of any arithmetic operations within the `scalar` values in the guard, the following predicates should be used to test individual `g_guard` values. File 'scalar_sat.clp' implements these predicates, converting each scalar to a bitvector and applying bitwise operations over those vectors to determine precisely the satisfiability of the formula.

- `guard_sat(in G:g_guard)`: `G` is satisfiable, holding under at least one assignment of values to its unconstrained variables.

- `guard_valid(in G:g_guard)`: `G` is valid, holding for all possible assignments of values to its unconstrained variables.

- `guard_implies(in G0:g_guard,in G1:g_guard)`: Under any assignment in which `G0` holds, `G1` holds as well.

- `guard_equivalent(in G0:g_guard,in G1:g_guard)`: `G0` and `G1` hold for the same set of assignments.

Satisfying assignments of type `g_guard_asn` can be constructed for guards and used to instantiate other guards and trace values.

- `guard_sat_asn(in G:g_guard,ASN:g_guard_asn)`: `G` is satisfiable, and `ASN` is a satisfying assignment for it.

- `asn_guard(in ASN:g_guard_asn,in G:g_guard)`: `G` holds under the assignment `ASN`.

- `asn_value(in ASN:g_guard_asn,in V:t_trace_val,N:int)`: The value of `V` under the assignment `ASN` is `N`.

Type `g_bit` is defined in `scalar.clp`, and primarily consists of boolean comparisons over `scalar` values. New comparison guards can be constructed with `#id_g` applied to values of this type.

5.3.5 Path-sensitive points-to graphs

The path-sensitive points-to graph for each `sum` is comprised of two parts. First we compute the conditions under which each program point is reachable, then we refine with conditions for the per-program point value of traces, CIL expressions and lvalues.

- `guard(P:pp,in G:g_guard)`: Program point `P` is reached by the function when `G` holds. Of all the concrete executions for the current `sum`, those which pass through `P` are exactly those under which `G` holds.

- `val(in P:pp,in S:t_trace,T:t_trace_val,G:g_guard)`: At program point `P`, the memory location represented by `S` has the value `T` when `G` holds. Of all the concrete executions for the current `sum`, if `P` is indeed reached by that execution, then for the `T` given by the guard `G` which holds under the execution (multiple `T/G` pairs may exist for `val`), the value the value the location represented by `S` has at `P` is that given by `T`.

- `lval(in P:pp,in LV:c_lval,S:t_trace,G:g_guard)`: At program point `P`, the memory location referred to by source lvalue `LV` is `S` when `G` holds.

- `eval(in P:pp,in E:c_exp,T:t_trace_val,G:g_guard)`: At program point `P`, the value of source expression `E` is `T` when `G` holds.

- `access(P:pp,T:t_trace,AT:access_type)`: At program point `P`, trace `T` may be directly accessed, indicated by `AT` as either `read` or `write`.

- `eguard(in P0:pp,in P1:pp,in G:g_guard,EG:g_guard)`: For any edge in the CFG from `P0` to `P1` refine any condition `G` to `EG` according to the condition under which the CFG edge transition occurs. Where `guard(P0,G0)` and `guard(P1,G1)`, if `EG&G1` holds, then `G&G0` holds and the edge will be taken during function execution. This predicate is most useful for analyses which which define transfer functions for new path-sensitive properties, in which case `eguard` should be used to refine any guard being added at `P1`.

The file 'memory/guarddot.clp' generates, for each function 'foo', a DOT graph file 'foo_guard.dot' which shows the guard for each program point in the CFGs of 'foo'. The file 'memory/valdot.clp' generates, for each trace 'x' written within each function 'foo', a DOT graph file 'foo_x_val.dot' which shows the values of 'x' and associated guards for each program point in the CFGs of 'foo'.

Example 5.5 Consider the following C program:

```
void foo(int a, int b, int c)
{
  int x;
  if (a) x = 0;
  if (b) {
    if (c) x = 1;
```

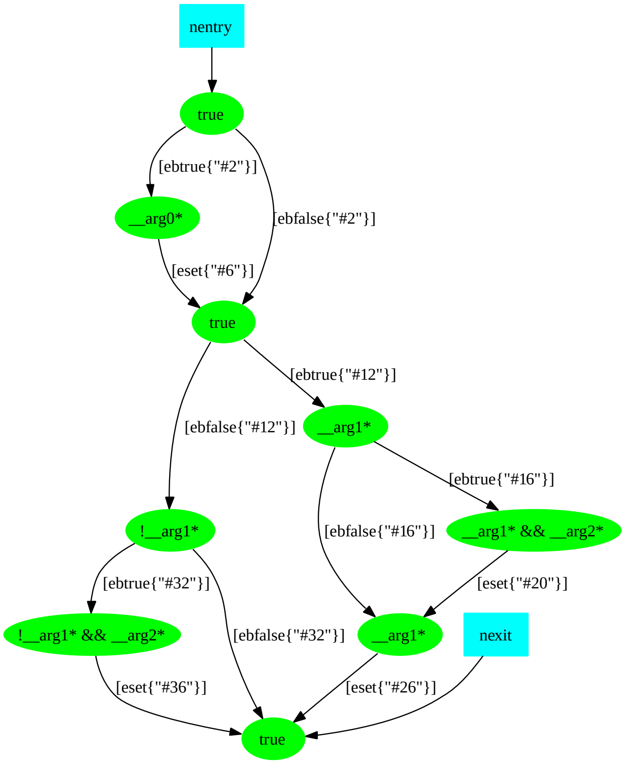

Figure 5.5: DOT graph for Example 5.5

```
    x = 2;
  }
  else {
    if (c) x = 3;
  }
}
```

Execute the 'memory/guarddot.clp' program. The resulting graph, in 'foo_guard.dot', is shown in Figure 5.5. Each node corresponds to a program point and is labelled with the guard at that point. Note how the guard changes with each branch and merge.

All propagation of `val` and generation of `lval` and `eval` is performed by `memory.clp`. There are two important properties maintained by `val`. The first property is that if `val(P,S,T,G0)` and `guard(P,G1)`, `G0` does **not** necessarily imply `G1`. To get the condition under which `S` has value `T` at `P` **and** `P` is itself reachable, compute the conjunction of `G0` and `G1`.

Example 5.6 Consider the following C program:

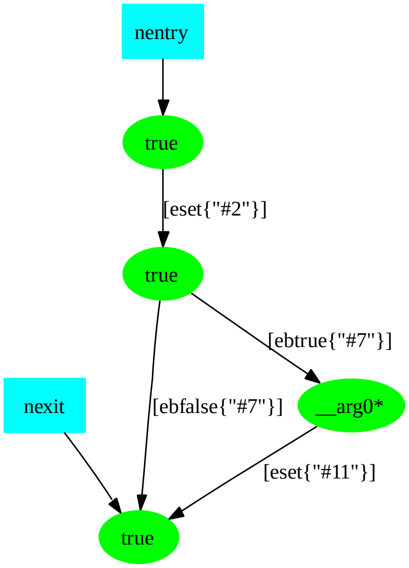

Figure 5.6: DOT graph for Example 5.6 showing program point guards

```
void foo(int a)
{
  int x = 0;
  if (a) x = 1;
}
```

Execute the 'memory/guarddot.clp' and 'memory/valdot.clp' programs. The resulting graphs, in 'foo_guard.dot' and 'foo_x_val.dot', are shown in Figures 5.6 and 5.7, respectively. Note that before the second assignment '`x = 1`' (`eset{"#11"}`), the program point guard is `__arg0*` indicating that the first argument must be non-zero, but that the condition under which `x` has value `0` is `true`. To get the condition where `x` has value `0` at this point and the point itself is reachable, compute the conjunction of these two guards.

The second property maintained by `val` is that when the same trace can have multiple different values at some point, the associated guards are disjoint **unless** the trace itself is soft. This means that the single location referenced by the trace can't have two different values at once, but that if the trace is soft then the locations it represents may have multiple different values.

Example 5.7 Consider the following C program:

```
void foo(int a)
{
  int x;
```

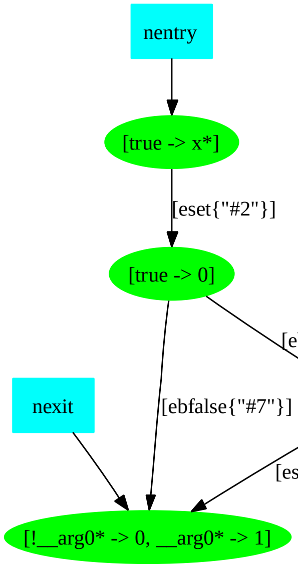

Figure 5.7: DOT graph for Example 5.6 showing values of variable `x`

```
  int y[100];
  if (a) {
    x = 0;
    y[0] = 0;
  }
}
```

Execute the 'memory/valdot.clp' program. The resulting graphs, in 'foo_x_val.dot' and 'foo_y_val.dot', are shown in Figures 5.8 and 5.9, respectively. Note that for local variable `x`, the guards at exit from the function are disjoint, while for local array variable `y`, the guards overlap - the initial (uninitialized) value of `y` is preserved, indicating that even when the branch is taken not all elements of `y` were initialized.

5.3.6 Cross-procedure trace propagation

Almost any interprocedural analysis will need to propagate information about memory locations from a `sum` to its callers (or vice versa). Since the locations represented by any given trace can differ between each `sum`, we get the notion of a trace *scope*. A trace in the scope of a particular `sum` always represents the same set of locations. Moreover, when propagating from one `sum` to another, traces in the scope of the original `sum` need to be converted to traces in the scope of the new `sum`.

- `trace_visible(in SUM:sum,in T:t_trace)`: Indicates whether `T` is a memory location that is either heap-allocated, statically-allocated, or stack-allocated

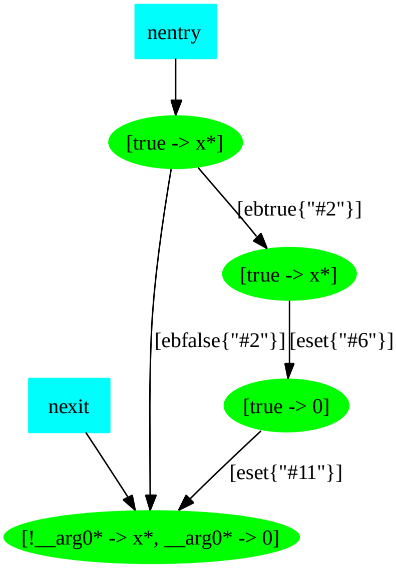

Figure 5.8: DOT graph for Example 5.7 showing values of variable x

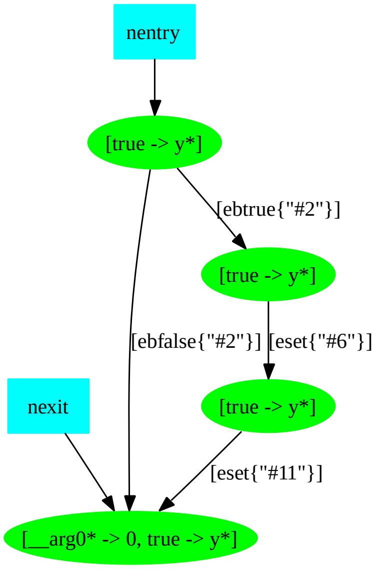

Figure 5.9: DOT graph for Example 5.7 showing values of array y

within the frame for a caller of SUM (i.e. outer loop/function or call). Traces which are **not** visible are the stack locations of temporaries for inner loops of a function, and temporaries, locals and arguments for the function itself.

- inst_trace(in I:sum,in P:pp,in CT:t_trace,T:t_trace_val,G:g_guard): For an isum edge isum(P,_,I), trace CT in the scope of I converts to trace T in the scope of the sum containing point P, when G holds. T may be an intermediate integer which was passed to the callee I, for CT = drf{PCT} where PCT is an integer-typed trace in I. inst_trace is only defined if trace_visible(I,CT).
- td_inst_trace(in I:sum,in P:pp,in T:t_trace,CT:t_trace,G:g_guard): Reverse of inst_trace: for an isum edge isum(P,_,I), trace T in the scope of the sum containing point P converts to trace CT in the scope of I, when G holds.
- inst_may_access(in I:sum,in CT:t_trace,AT:access_type): For an isum edge isum(_,_,I), trace CT in the scope of I may be accessed as read or write as indicated by AT. Useful for filtering the results of td_inst_trace, though td_inst_trace will still be generated for traces that are not accessed by the target. This information is computed by the alias analysis (Section 5.4.3) and this predicate will never hold if the alias analysis has not been run.

Predicates inst_trace and td_inst_trace bear a close relation with the meaning of traces themselves. Recall from Section 5.3.1 that the drf{T} trace refers to the initial target of T at entry to the sum that T is in the scope of. For an isum call or loop SUM then, we can convert a drf{CT} trace in the scope of SUM by converting CT to some trace T, then simply evaluating T *at the call site itself* using val.

Example 5.8 Consider the following C program:

```
void bar(int *x);
void foo(int c)
{
  int a, b;
  int *x;
  if (c) x = &a;
  else   x = &b;
  bar(x);
}
```

Execute the following CLP program:

```
import "memory/scalar_sat.clp".
analyze session_name("cil_body").

predicate call_target(T:t_trace,GSTR:string).
dircall(I,"bar"), icall(P,_,I),
    inst_trace(s_call{I},P,drf{root{arg{0}}},trace{T},G),
    guard_string(G,GSTR),
    +call_target(T,GSTR).

?- call_target(T,G).
```

The above program converts the target of the first parameter to the call to 'bar' to traces in the scope of 'foo' itself. The following output should be printed:

```
call_target(root{local{"a"}},"__arg0*").
call_target(root{local{"b"}},"!__arg0*").
```

The first parameter to 'bar' may be either of the local variables in 'foo', though under different conditions. `inst_trace` maintains the same properties of `val` in that different traces are converted to under disjoint conditions except when the traces are soft.

It may, however, be the case that the same trace is converted to by two different callee traces, in the presence of aliasing. `inst_trace` may overapproximate the guards in this case; see 5.4 for more information.

Other types of values which include traces have, in effect, the same scope as those traces and need to be converted when propagating from one `sum` to another.

- `inst_guard(in I:sum,in P:pp,in CG:g_guard,G:g_guard)`: Guard `CG` in the scope of callee `I` at `P` converts to `G` in the scope of the `sum` containing point `P`.
- `inst_scalar(in I:sum,in P:pp,in CS:scalar,S:scalar)`: Scalar `CS` in the scope of callee `I` at `P` converts to `S` in the scope of the `sum` containing point `P`.

5.3.7 Freshly allocated data

Data that was allocated via `malloc` or similar function calls (and their wrappers) are represented as `drf{root{temp{T,W}}}` where `T` is the temporary storing the return value of the function call. While within the procedure these are treated like any other trace, propagating information about them to callers in cases where they escape (returned or assigned through side effects to caller pointers) is trickier as the caller has no trace to refer to them. In cases where the freshly allocated value escapes, the alias analysis identifies the trace through which it escapes and supplies this to the memory analysis:

- `exit_trace_rename(in SUM:sum,in T:t_trace,NT:t_trace)`: If `T` was freshly allocated within the current `SUM` (i.e. the trace root is `temp{_,_}`), then `NT` is the trace through which `T` escapes. If `T` escapes through multiple different pointers, the alias analysis will choose a single one and reflect the aliasing introduced through other edges (Section 5.4). If `T` is not freshly allocated, then `NT` is equal to `T`.

Example 5.9 Consider the following C program:

```
void* malloc(int len);
void foo(int **fill)
{
  int *v = malloc(sizeof(int));
  *fill = v;
}
```

Run the alias analysis 'aliasing/aliasing.clp' and then execute the following CLP program:

```
import "memory/scalar_sat.clp".
analyze session_name("cil_body").

predicate escape_target(T:t_trace,NT:t_trace).
dircall(I,"malloc"), icall(P,_,I),
    inst_trace(s_call{I},P,drf{root{return}},trace{T},_),
    exit_trace_rename(s_func,T,ESCAPE),
    +escape_target(T,ESCAPE).

?- escape_target(T,NT).
```

The above program identifies the trace through which the return value of 'malloc' escapes within 'foo'. The following output should be printed:

```
escape_target(drf{root{temp{"ciltmp","malloc"}}},drf{drf{root{arg{0}}}}).
```

Note that within the caller, the alias analysis (and as a result, the memory analysis) will not introduce a fresh location for the target `fill` after the call, and effectively merges the original target of `fill` with the location allocated by 'foo'. There is currently no clean way to differentiate the two values (typically the old one is freed, or both values are part of an unbounded soft structure), and for some analyses this can degrade precision. This will be addressed in a future release.

5.4 Alias Analysis

The Saturn alias analysis give a conservative and fairly precise approximation of all the inter-procedural aliasing information for the functions and summaries in a program. The alias analysis and memory model (Section 5.3) work in tandem; the alias analysis generates only inter-procedural information, i.e. that applying to summaries, types, or globals as a whole, and the memory model refines this to generate points-to graphs and related for each point in each function. The memory model (or rather, analyses built on top of the memory model) may be run without aliasing information, though the results will not be conservative.

The alias analysis itself is built on top of the memory model. Because of this interdependency between the two, the alias analysis is responsible not just for computing inter-procedural aliasing information (Section 5.4.1), but also all other inter-procedural information the memory model depends on. This includes identifying soft traces (Section 5.4.2), use/mod information for summaries (Section 5.4.3), and indirect call targets (Section 5.4.4).

The alias analysis includes several CLP files:

- 'aliasing/aliasing.clp': Main analysis that runs over each function and computes aliasing, soft trace, and use/mod information.
- 'aliasing/aliasinginit.clp': Analysis that runs over each global variable and computes information derived from static initializers.
- 'funptr/funptr.clp': Analysis that identifies indirect call targets and function pointers passed into each function.

The above files should be run as a cofixpoint. If one or more of the files is omitted then the analysis results will be incomplete with respect to the information computed by those files. Also note that if the analyzed C program itself is open, i.e. missing callers or callees of certain functions, then the analysis results will be incomplete with respect to the missing code and will **not** be conservative.

```
clpa --timeout 100 aliasing/aliasing.clp aliasing/aliasinginit.clp funptr/funptr.clp
```

The alias analysis CLP files above produce no text output. Alternative versions 'aliasing/aliasingprint.clp', 'aliasing/aliasinginitprint.clp' and 'funptr/funptrprint.clp' behave identically except that they print out the main facts generated. This is helpful for quickly seeing what the analysis does, and these versions of the files are used in the examples below.

Regardless of which way it is run, the alias analysis populates the following summary databases. To do a fresh restart of the alias analysis, just delete these databases first:

- `sum_entry.db`, `sum_exit.db`, `sum_comp.db`, `sum_glob.db`: Inter-procedural aliasing information.
- `sum_usemod.db`, `sum_usemod_comp.db`, `sum_usemod_glob.db`: Soft trace and usemod information.
- `sum_funptr.db`, `sum_funptr_entry.db`: Indirect call targets and associated information.

The alias analysis also generates UI databases 'display.db' and 'search.db' which can be consumed by the Saturn UI (7.5) to show not only which aliasing relationships were added by the analysis, but also why they were.

5.4.1 Interprocedural aliasing

Several different kinds of aliasing information are produced by the alias analysis. These include side effects on summaries, entry aliasing specific to summaries, global invariants and type invariants. These all indicate behaviors other than the default used by the memory model, that each trace `S` always points to `drf{S}`; all interprocedural information generated indicates may-aliasing, that two pointers may be aliased rather than must be aliased. When generating the entry points-to graph for a summary, or applying side effects on a trace at each `isum` CFG edge, the memory model takes all the following kinds of aliasing into account to different kinds of aliasing into account, introducing new unconstrained bits into the `val` relation for each possible alias introduced.

Summary side effects

Side effects on loop and functions are assignments of pointers which might affect the behavior of any summary which calls that loop or function.

Example 5.10 Consider the following C program:

```
void foo(int **px, int *y)
{
    *px = y;
}
int* bar(int *z)
{
    int *a;
    foo(&a,z);
    return a;
}
```

Run the alias analysis, including ‘aliasing/aliasingprint.clp’. The following output should be printed:

```
str_spoints("foo",s_func,fn_exit,"__arg0*","__arg1*").
str_spoints("bar",s_func,fn_exit,"return","__arg0*").
```

The first line indicates that at exit from ‘foo’ `*px` may have been updated to point to the target of `y`. This was then propagated up to ‘bar’, where the second line indicates that at exit from ‘bar’ its return value may point to the target of its first argument `z`.

Now execute the ‘memory/valdot.clp’ program, and view the ‘bar_a_val.dot’ DOT graph produced (See Figure 5.10). Variable `a` in ‘bar’ may point to the initial target of `z` after the call to ‘foo’, depending on whether the unconstrained bit `exit_bit(s_call{"#2"},__arg0*,__arg1*)` holds.

Summary entry aliasing

Entry aliasing on a loop or function indicate pointers passed in by any caller may be aliased. Entry aliasing is only generated when the aliasing may affect the behavior of the loop or function. The use/mod information is examined, and if one or both of the pointers are not directly accessed (say, when a global variable is passed into

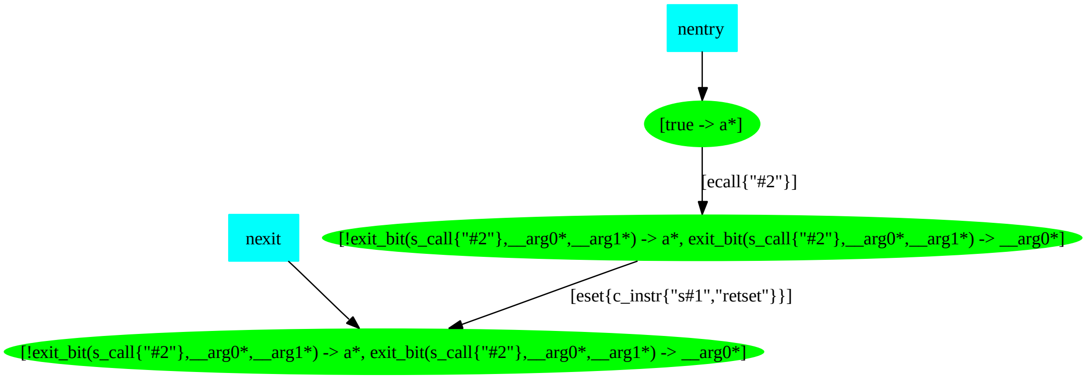
nentry
[true -> a*]
[ecall{"#2"}]
nexit
[!exit_bit(s_call{"#2"},__arg0*,__arg1*) -> a*, exit_bit(s_call{"#2"},__arg0*,__arg1*) -> __arg0*]
[eset{c_instr{"s#1","retset"}}]
[!exit_bit(s_call{"#2"},__arg0*,__arg1*) -> a*, exit_bit(s_call{"#2"},__arg0*,__arg1*) -> __arg0*]

Figure 5.10: DOT graph for Example 5.10

a function), or if neither of the pointers is written through, then the entry aliasing is not generated.

Example 5.11 Consider the following C program:

```
void foo(int *x, int *y)
{
    *x = *y;
}
void bar(int **px, int *y)
{
    foo(*px, y);
}
void baz()
{
    int a;
    int *pa = &a;
    bar(&pa, &a);
}
```

Run the alias analysis, including ‘aliasing/aliasingprint.clp’. The following output should be printed:

```
str_spoints("bar",s_func,fn_entry,"__arg0*","__arg1*").
str_spoints("foo",s_func,fn_entry,"__arg0","__arg1*").
```

The first line is generated during the analysis of ‘baz’ and indicates that at entry to ‘bar’ that `*px` and `y` may be aliased and point to the same location. This was then propagated down to the call to ‘foo’, where the second line indicates that at entry to ‘foo’ its two parameters may be aliased.

Now execute the ‘memory/valdot.clp’ program, and view the ‘foo__arg0_val.dot’ DOT graph produced (See Figure 5.11). The first argument to ‘foo’ may be

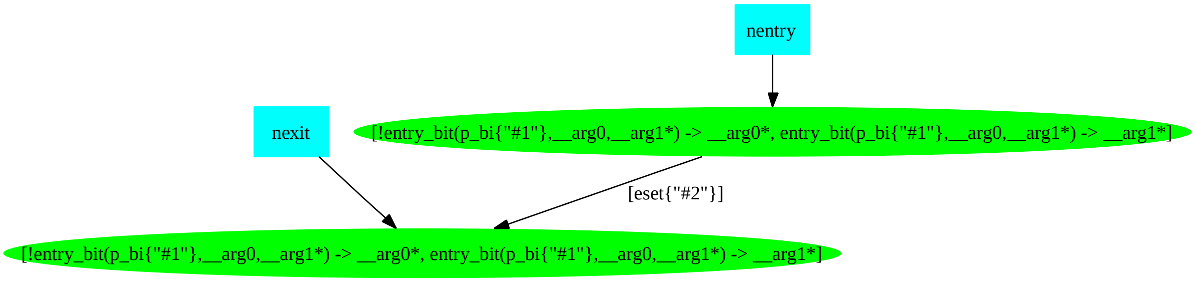
nentry
nexit
[!entry_bit(p_bi{"#1"},__arg0,__arg1*) -> __arg0*, entry_bit(p_bi{"#1"},__arg0,__arg1*) -> __arg1*]
[eset{"#2"}]
[!entry_bit(p_bi{"#1"},__arg0,__arg1*) -> __arg0*, entry_bit(p_bi{"#1"},__arg0,__arg1*) -> __arg1*]

Figure 5.11: DOT graph for Example 5.11

aliased with its second argument, depending on whether the unconstrained bit `entry_bit(p_bi{"#1"},_arg0,_arg1*)` holds.

Note that the entry aliasing in this example would not be generated if ‘foo’ did not write to `x`, or if ‘foo’ were a function prototype with no implementation (and hence no use/mod information).

Global invariant aliasing

Global invariants capture aliasing that is generally expected to hold of particular global variables. These may hold at entry/exit from any summary, and thus don’t need to be explicitly accounted for as side effects or entry aliasing on particular summaries.

Global invariants are generated whenever any two global variables are found to share aliased data. This behavior can be tuned to specific code bases by creating a new file that imports ‘aliasing/aliasing.clp’, and adding new instances of the `omit_glob_spoints` predicate described therein.

Example 5.12 Consider the following C program:

```
int *f, *g;
void foo(int *x, int *y)
{
    *x = *y;
}
void bar()
{
    foo(f, g);
}
void baz()
{
    f = malloc(4);
    g = f;
}
```

Run the alias analysis, including 'aliasing/aliasingprint.clp' and 'aliasing/aliasinginitprint.clp'. The following output should be printed:

```
str_glob_spoints("f","f","g*").
str_spoints("foo",s_func,fn_entry,"__arg0","__arg1*").
```

The first line is generated during the analysis of 'baz' and indicates that global pointers f and g may generally alias one another at entry/exit to any summary in the program. This was then propagated to the analysis of 'bar' where the second line indicates that at entry to 'foo' its two parameters may be aliased.

A subtlety with global pointers is that if a pointer references heap-allocated data, any aliasing it is involved in will be added only if the initialization of that pointer occurs within the analyzed program. If the `f = malloc(4);` line were omitted from the above program, the aliasing would not be added unless another function initialized f with malloc'ed data. This is only an issue when analyzing open programs without full initialization code or a 'main' function.

Type invariant aliasing

Type invariants capture aliasing that is generally expected to hold of any value of a particular type. As with global invariants, these may hold at entry/exit to any summary and aren't explicitly summarized except on the type itself. The type of a value is determined by the memory model by looking both at the given C type in the program as well as any casts performed. This assumes that individual memory locations in the program are never cast between incompatible types, a memory safety property that will be checked by a forthcoming analysis.

Type invariants are generated whenever two fields (or chains of field/derefs) from a structure may alias within the same structure, as well as when a field of a structure is updated directly by a function to point to some other global variable. This behavior can be tuned to specific code bases by creating a new file that imports 'aliasing/aliasing.clp', and adding new instances of the `omit_glob_spoints` predicate described therein.

Example 5.13 Consider the following C program:

```
struct str {
  int *f, *g;
};
void foo(int *x, int *y)
{
    *x = *y;
}
void bar(void *v)
{
    struct str *s = v;
    foo(s->f, s->g);
}
```

```
void baz(struct str *s)
{
    s->f = s->g;
}
```

Run the alias analysis, including ‘aliasing/aliasingprint.clp’. The following output should be printed:

```
str_comp_spoints("str",".f",".g*").
str_spoints("foo",s_func,fn_entry,"__arg0","__arg1*").
```

The first line is generated during the analysis of ‘baz’ and indicates that fields f and g of values of type str may generally alias one another at entry/exit to any summary in the program. This was then propagated through the cast in ‘bar’ where the second line indicates that at entry to ‘foo’ its two parameters may be aliased.

5.4.2 Soft trace identification

As described in Section 5.3.2, the memory model and analyses depending on it must identify which traces are soft and can represent more than one actual memory location. Flat stack or global arrays of known length are handled directly by the memory model, but trickier constructs require softness information computed by the alias analysis.

It destroys precision to assume every pointer passed into a function is an array, or that every location that contains recursive links is actually traversed. The alias analysis identifies as soft only those pointers used in arithmetic, and only those locations whose links are traversed. This must be done transitively through summary calls as well, pushing information bottom up as well as through globals and types to identify all soft traces.

Example 5.14 Consider the following C program:

```
void foo(int **buf)
{
    buf[1] = buf[2] = 0;
}
void bar(int **x)
{
    *x = 0;
    foo(x);
}
```

Run the alias analysis and then execute the following program:

```
import "memory/scalar_sat.clp".
analyze session_name("cil_body").

predicate inter_soft(FN:string,T:t_trace,ST:soft_type).
```

```
cil_curfn(FN), access(_,T,_), trace_soft(s_func,T,ST),
    +inter_soft(FN,T,ST).

?- inter_soft(FN,T,ST).
```

The following output should be printed:

```
inter_soft("foo",drf{root{arg{0}}},ptarray).
inter_soft("bar",drf{root{arg{0}}},ptarray).
```

The buffer target is marked as soft in both 'foo' and 'bar', even though it is only accessed as an array within 'foo'. Were it not marked as soft within 'bar', then at the call to 'foo' it would appear from the memory model as though every cell in the buffer was 0, when that is not the case.

5.4.3 Use/mod information

Use/mod information is very useful to most analyses. Through the `inst_may_access` predicate described in Section 5.3.6, the memory model uses it to identify which integers should be clobbered after each `isum`, the alias analysis uses it to trim down the relevant entry aliases, and client analyses can use it to restrict top-down propagation.

Use/mod information also typically dominates the remaining information computed by the alias analysis, and can be a significant obstacle to scalability. To alleviate this, the alias analysis aims to compact the use/mod information as much as possible while still retaining precision. This results in several ways to summarize use/mod information, which are folded together through the `inst_may_access` predicate:

- Per-summary use/mod information: it is prohibitively expensive to record every single trace that may be accessed by every single summary. To restrict the amount of information computed per-summary then, Precise per-trace information is only kept down to a certain number of dereferences (zero for globals, one for function arguments and loop local variables). Beyond this, blanket `deepread` and `deepwrite` access information is recorded on traces which covers all data transitively reachable from those pointers.

- Per-type use/mod information: the above per-summary strategy is too imprecise by itself; many structures have read-only fields, and if a `deepwrite` is applied to the base of the structure due to some other field being written, then the targets of the read-only fields will be marked as written as well. The set of structure fields through which writes are ever performed within the program is recorded by the alias analysis and used within `inst_may_access` to refine the set of writes, avoiding the above situation.

- Per-global use/mod information: in large systems many global variables are immutable beyond whichever value they were assigned by their static initializer. These include fixed integer values and large global arrays of data.

Use/mod information for these globals is not meaningful, and so the alias analysis computes the set of globals which could ever be written by the program and records use/mod only for those globals.

5.4.4 Indirect call targets

Indirect call targets are computed by the stand-alone analysis 'funptr/funptr.clp'. For conservative results this file must be run as a cofixpoint with the other alias analysis files, but for flexibility (and reasonably close to conservative results) the other two alias analysis files may be run first followed by the function pointer analysis. This is a top-down analysis (vs. the bottom-up 'aliasing/aliasing.clp') that augments the memory model with entry information about function pointer targets passed into the function. This is then used to compute and store the complete set of which functions each indirect call could target, and which may be consumed by other analyses via the `anycall` predicate.

Example 5.15 Consider the following C program:

```
void fn1(int);
void fn2(int);
void (*pfn)(int) = fn1;
void foo()
{
    pfn = fn2;
}
void bar(int x)
{
    (*pfn)(x);
}
```

Run the alias analysis, making sure to include all three files. The following output should be printed:

```
str_indirect("bar","*pfn","fn1").
str_indirect("bar","*pfn","fn2").
```

The indirect call within 'bar' may target either 'fn1' or 'fn2'. There is currently no additional information supplied indicating the conditions under which indirect calls may target their various possible callees.

Note that the points-to edges for global variable `pfn` used to identify these targets are not explicitly printed out by 'aliasing/aliasingprint.clp' nor 'aliasing/aliasinginitprint.clp'. These aliasing relationships for immutable global data such as functions are recorded differently by the alias analysis and not directly incorporated by the memory model (aliasing over immutable data does not change the possible effects of the assignments considered by the memory model).

5.5 The NULL Dereference Analysis

This section describes how to run the provided null dereference bug finder.

5.5.1 Running NULL

The null analysis is run by typing:

```
clpa --no-fixpoint --timeout 100 CLPA_DIR/analysis/null/null.clp
```

Since the null analysis is a bug finder, it does not support fixpointing and will enter an infinite loop if fixpointing is enabled. Also, it is highly recommended to set a resonable time out limit to avoid thrashing on very large functions.

Important Facts and Advice:

- The null analysis performs its own side effect analysis independent from the provided alias analysis. It is **not** recommended to run the null analysis in the presence of aliasing summaries; if these summaries are present then some warnings may be missing from the results.

- The null analysis reports significantly more false positives in the presence of arrays and pointer arithmetic. The reason for this is that since no strong updates are performed on these constructs, any `NULL` assignment contaminates other elements of the same data structure. Thus, be aware that you may observe strange error messages in the presence of such data structures.

- Sources of unsoundness: It is important to note that this analysis is unsound; that is, it may not find all the null bugs that a program may have. Some of the souces of unsoundness that we are aware of are the following: First, the null analysis uses an unsound side-effect analysis. Therefore, it does not necessarily account for all function(and loop) side effects that cause a pointer to be assigned to `NULL`. Another obvious source of unsoundness is that the null analysis does not do a fixpoint computation. Function summaries are computed exactly once, and not until the summaries stabilize.

5.5.2 Errors detected

In this section, we discuss the four different classes of errors that the null dereference analysis tracks and present examples illustrating each of these error classes.

NULL flow errors

This part of the null analysis detects errors where a variable is assigned to `NULL` and there exists a feasible execution path such that this `NULL` value can flow to a site of dereference.
Example 1:

```
void test(int* p, int* q, bool flag)
{
  int a;
  if(flag) p=NULL; (*)
  q = p;
  a = *q;  (**)
}
```

In this example, `p` is set to `NULL`, aliased by `q` and eventually dereferenced. Since there is an execution path where the `NULL` value assigned to p at (*) can flow to the dereference at (**), the analysis reports an error.

Null flow errors can also be inter-procedural:

Example 2:

```
void bar(int *p, bool flag)
{
  int a;
  if(flag) a = *p;
  else a = -1;
}

void foo(bool flag)
{
  int *p;
  if(flag)  p=NULL;
  else p=malloc(sizeof(int));
  bar(p, flag);
}
```

The analysis reports an error for the call to `bar` in `foo` because `p` is set to `NULL` if `flag` is true, and `bar` dereferences `p` if flag is true, resulting in an error. However, changing `foo` to:

```
void foo(bool flag)
{
  int *p;
  if(!flag)  p=NULL;
  else p=malloc(sizeof(int));
  bar(p, flag);
}
```

will not cause the analysis to report an error since there is indeed no error in the call to `bar`.

The analysis is fully path-sensitive within one function and uses *the correlation analysis* to achieve selective path-sensitivity accross function boundaries. In cases where using the correlation analysis is too imprecise and results in many false positives, the analysis can be made to report less false positives by enabling predicate `certain_deref_only` in the run script.

Conditional misuse errors

This part of the analysis detects errors that arise from conditional misuse, i.e. a conditional ensures that a variable is `NULL` before dereferencing it.

Example: Error within one function

```
void test(int* p, int flag){
  if(!p || flag)
    *p = 8;

}
```

In this example, `p` is dereferenced whenever `p` is `NULL`, resulting in a run-time crash. Thus, the analysis reports an error.

Example: Interprocedural error

```
void bar(int *p){
 *p = 7;
}

void foo(int* p, int flag) {
  if(!p || flag) bar(p);
}
```

This example is similar to the one above except that it involves a function call. `foo` calls `bar` whenever `p` is `NULL`, but `bar` in turn dereferences `p`. Thus, the analysis reports an error for the call to `bar`.

Inconsistency errors

In addition to the errors presented above, the null analysis also detects *inconsistency errors.* Informally, an inconsistency arises if a variable is checkd for `NULL` before dereferencing it at one location, but not checked for `NULL` at another location which dereferences it. The detection of inconsistencies is not syntactic, but rather semantic: The analysis does not pattern match on conditionals that compare a variable with a given name against `NULL`. The analysis reports an inconsistency error if the following condition holds: For two pointer expressions that have the same guarded points-to set, one pointer is dereferenced without ensuring it is non-`NULL`, but the other one is dereferenced after checking it is `NULL`. This technique of detecting inconsistencies allows us to naturally capture any possible aliasing relationships between pointers. Furthermore, we determine if a location is checked for `NULL` by querying the statement guard. This allows the analysis to detect even unconventional forms of `NULL` checks.

Example 1: Simple Inconsistency

```
void foo(int* p){
 if(p) *p = 2;
 *p = 3;
}
```

The analysis reports an inconsistency error in `foo` because the first line of the function assumes `p` could be `NULL`, while the second line ignores this possibility. Therefore, either the check on the first line is defensive or `p` is actually allowed to be `NULL` and leads to a crash on the second line.

Example 2: A more complicated inconsistency error

```
void foo(int* p){
 int * q = p;
 bool flag = (p==0);
 if(!flag) *p = 2;
 *q = 3;
}
```

The analysis again reports an inconsistency error in `foo`. The reason for this is as follows. First, note that `p` and `q` are aliases of one another, so the assumptions made about `p` should be the same as those made about `q`. Next, the check `!flag` ensures that `p` is non-`NULL` at line 3, but `q` is dereferenced without making sure that it is non-`NULL`. Thus, the analysis reports an inconsistency error.

The null analysis also tracks inconsistencies across function boundaries. If a function assumes that a pointer may be `NULL` by testing it for `NULL` and then passes the same pointer (or its alias) to another function which dereferences it without testing it against `NULL`, the analysis reports an inconsistency error.

Interprocedural Inconsistency Example:

```
void bar(int* q){
  *q = 8;
}
void foo(int* p)
{
  int* q =p;
  if(p) *p = 8;
  bar(q);
}
```

In function `foo`, `p` is tested for being `NULL`, indicating the possibility that `foo` may have been called with a `NULL` parameter. However, an alias of `p` is unconditionally dereferenced inside a callee of `foo`, causing the analysis to report an inconsistency error.

NULL return errors

Another source of null dereference errors is the dereference of return values of functions which may return `NULL`. Our analysis reports a return-`NULL` error if a function

dereferences a possibly-NULL return value and the analysis cannot prove that the conditions at the call site imply that a non-NULL value will be returned by the called function.
Example:

```
int** bar(int size){
  int** p = malloc(sizeof(int*));
  if(!p) return NULL;
  for(int i=0; i<size; i++){
    p[i] = malloc(sizeof(int));
    if(!p[i]) return NULL;
  }
  return p;
}
void foo(int size)
{
    int** arr = bar(size);
    arr[0][0] = -1; (*)
}
```

The analysis reports a return-NULL error for the line marked with (*) because even though bar can return NULL, the return value of bar is dereferenced without testing it is non-NULL so that a NULL dereference is possible.

Chapter 6

Abstract Semantics

This chapter gives an abstract description of the memory analysis. Section 6.1 gives the preliminaries. Section 6.2 describes how expressions, lvalues, and conditions are evaluated. Section 6.3 describes how locations are instantiated at call sites. Section 6.4 describes what interprocedural aliasing information the memory analysis consumes. Section 6.5 describes how the memory analysis computes sound, flow-sensitive, path-sensitive points-to information of a procedure.

6.1 Preliminaries

This chapter uses the following functions to distinguish among different kinds of program variables:

- Global is the set of global variables.
- $\mathit{Var}(P)$ is the set of all formal parameter and local variables of procedure P.
- $\mathit{FormalParam}(P, i)$ is the ith formal parameter variable of procedure P.
- $\mathit{FormalReturn}(P)$ is the formal return variable of procedure P.
- $\mathit{ActualParam}(P, k, i)$ is ith actual parameter variable passed to call site k in procedure P.
- $\mathit{ActualReturn}(P, k)$ is a actual return variable of a call site k in procedure P.

Let $\mathit{Location}^P$ be the set of *abstract locations* of procedure P. The superscript is omitted when the procedure is clear from context. The abstract locations of a procedure are disjoint from the abstract locations of any other procedure, so $\mathit{Location}^P \cap \mathit{Location}^Q = \emptyset$ for any $P \neq Q$. Henceforth, *abstract locations* are called simply *locations*. Let $\mathit{Env}^P \in (\mathit{Global} \cup \mathit{Var}(P)) \rightarrow \mathit{Location}^P$ be a total, injective function that assigns locations to the global, formal, and local variables of P. The range of Env^P is the set of *root* locations, Roots^P.

Let β^P be a set of boolean variables. Let Guard^P be the set of propositional formulas over β^P and atomic predicates on locations in $\mathit{Location}^P$. Let Guard^P_β be the propositional formulas only over the boolean variables in β^P; locations do not appear these formulas.

A *guarded points-to graph* $\gamma^P \in \mathcal{G} = (\mathit{Location}^P \times \mathit{Location}^P) \rightarrow \mathit{Guard}^P$ gives the condition under which one location points to another location. If $\gamma^P(l_1, l_2) = \phi$, then l_1 may point to l_2 only when formula ϕ is true. The *default points-to graph*, $\gamma_{def} \in (\mathit{Location}^P \times \mathit{Location}^P) \rightarrow \{\mathit{true}, \mathit{false}\}$, satisfies the following conditions:

- Every location that is not a root is pointed to by at least one location.

$$(\forall l_2 \in \mathit{Location} - \mathit{Roots})\ (\exists l_1 \in \mathit{Location})\ \gamma_{def}(l_1, l_2) \tag{6.1}$$

- Every root is not pointed to by any location.

$$(\forall l_1 \in \mathit{Location})\ (\forall l_2 \in \mathit{Roots})\ \neg\gamma_{def}(l_1, l_2) \tag{6.2}$$

- Every location points to at least one location.

$$(\forall l_1 \in \mathit{Location})\ (\exists l_2 \in \mathit{Location})\ \gamma_{def}(l_1, l_2) \tag{6.3}$$

- Every location points to at most one location.

$$(\forall l_1, l_2, l_3 \in \mathit{Location})\ ((\gamma_{def}(l_1, l_2) \wedge \gamma_{def}(l_1, l_3)) \Rightarrow l_2 = l_3) \tag{6.4}$$

- Every location is pointed to by at most one location.

$$(\forall l_1, l_2, l_3 \in \mathit{Location})\ ((\gamma_{def}(l_2, l_1) \wedge \gamma_{def}(l_3, l_1)) \Rightarrow l_2 = l_3) \tag{6.5}$$

We use the following operations on points-to graphs. The least upper bound of points-to graphs is

$$(\gamma_1 \sqcup \gamma_2)(l_i, l_j) = \gamma_1(l_i, l_j) \vee \gamma_2(l_i, l_j)$$

The *Refine* $\in (\mathit{Guard} \times \mathcal{G}) \rightarrow \mathcal{G}$ function *refines* a points-to graph by restricting all points-to edges by a guard:

$$\mathit{Refine}(\phi, \gamma)(l_i, l_j) = \gamma(l_i, l_j) \wedge \phi$$

Let an *unguarded* points-to graph be a points-to graph where all points-to edges are unconditionally *true* or *false*. We define an operation σ that replaces the guards in an unconditional points-to graph with formulas enforcing that a location points to at most one location at a time. Let $\sigma \in \mathcal{G} \rightarrow (\mathit{Location} \times \mathit{Location}) \rightarrow \mathit{Guard}_\beta$ be a function that assigns formulas over boolean variables to points-to edges in a given points-to graph and satisfies the following conditions:

- Every location points to at most one location at a time.

$$\forall \gamma \in \mathcal{G}\ \forall l_1, l_2, l_3 \in \mathit{Location}.\ l_2 \neq l_3 \Rightarrow \neg(\sigma(\gamma)(l_1, l_2) \wedge \sigma(\gamma)(l_1, l_3)) \tag{6.6}$$

- Every location points to at least one location.

$$\forall \gamma \in \mathcal{G}\ \forall l_1 \in \mathit{Location}.\ \left[\bigvee_{l_2 \in \mathit{Location}} \sigma(\gamma)(l_1, l_2) \right] \tag{6.7}$$

- If a points-to relationship does not exist in the unguarded points-to graph, it does not exist in the guarded points-to graph.

$$\forall \gamma \in \mathcal{G}\ \forall l_1, l_2 \in \mathit{Location}\ \neg\gamma(l_1, l_2) \Rightarrow \neg\sigma(\gamma)(l_1, l_2) \tag{6.8}$$

6.2 Expression and Lvalue Evaluation

Expression evaluation is defined by inference rules with judgements of the form

$$\Gamma \vdash_{exp} e : \mathcal{E}$$

where Γ is a points-to graph in which the expression e is evaluated and $\mathcal{E} \in \mathit{Location} \rightarrow \mathit{Guard}$ is a function mapping each location l to the condition under which the value of e points to l. Note that expressions are side-effect free; the points-to graph is not affected by expression evaluation.

Lvalue evaluation is defined by inference rules with judgements of the form

$$\Gamma \vdash_{lval} e : \mathcal{E}$$

where again Γ is a points-to graph, e is an lvalue, and $\mathcal{E} \in \mathit{Location} \rightarrow \mathit{Guard}$ is a function mapping each location l to the condition under which l is the lvalue e.

Boolean condition evaluation is defined by inference rules with judgements of the form

$$\Gamma \vdash_{cond} e : \mathcal{C}$$

where Γ is a points-to graph, e a boolean expression, and $\mathcal{C} \in \mathit{Guard} \times \mathit{Guard}$ is a set of pairs of formulas where $(\phi, \psi) \in \mathcal{C}$ means e represents the condition ϕ under guard ψ.

6.3 Instantiation

A *location instantiation* $\mathcal{I}_k \in (\mathit{Location}^Q \times \mathit{Location}^P) \rightarrow \mathit{Guard}$ maps a location l^Q of a callee Q and a location l^P of a caller P at call site k in P to the condition under which l^Q and l^P represent the same location. The location instantiation is defined by the inference rules below, which use judgements of the form

$$\Gamma^P \vdash_{inst_k} l^Q : l^P, \phi^P$$

where l^Q is a location of procedure Q, l^P is a location of procedure P, and ϕ^P is the condition under which l^Q and l^P represent the same set of concrete locations.

- Consider a global variable g. A location l^Q in callee Q instantiates to location l^P in caller P if l^Q is the location assigned to global g in procedure Q, and l^P is the location assigned to global g in procedure P.

$$\frac{g \in \mathit{Global} \quad l^Q = \mathit{Env}^Q(g) \quad l^P = \mathit{Env}^P(g)}{\Gamma^P \vdash_{inst_k} l^Q : l^P, \mathit{true}}$$

- Consider the ith actual parameter variable a_i at a call site k in a caller P and the ith formal paramater variable f_i in a callee Q. Then, the location l^Q instantiates to a location l^P under the condition where the location of f_i points to l^Q in γ_{def} and the location of a_i points-to l^P in Γ^P.

$$\frac{a_i = \mathit{ActualParam}(P, k, i) \quad f_i = \mathit{FormalParam}(Q, i)}{\Gamma^P \vdash_{inst_k} l^Q : l^P, \Gamma^P(\mathit{Env}^P(a_i), l^P) \wedge \gamma^Q_{def}(\mathit{Env}^Q(f_i), l^Q)}$$

- Consider the actual return variable a at a call site k in a caller P and the formal return variable f in a callee Q. Then, the location l^Q instantiates to a location l^P under the condition where the location of f points to l^Q in γ_{def} and the location of a points-to l^P in Γ^P.

$$\frac{a = \mathit{ActualReturn}(P, k) \quad f = \mathit{FormalReturn}(Q)}{\Gamma^P \vdash_{inst_k} l^Q : l^P, \gamma^P_{def}(\mathit{Env}^P(a), l^P) \wedge \gamma^Q_{def}(\mathit{Env}^Q(f), l^Q)}$$

- Consider a location l'^Q in callee Q that instantiates to location l'^P in caller P under condition ϕ. Then, a location l^Q in Q instantiates to location l^P in P under the conjunction of ϕ, the condition under which l'^Q points to l^Q in Q's default points-to graph, and the condition under which l'^P points to l^P in P's points-to graph at call site k.

$$\frac{\Gamma^P \vdash_{inst_k} l'^Q : l'^P, \phi}{\Gamma^P \vdash_{inst_k} l^Q : l^P, \phi \wedge \gamma^Q_{def}(l'^Q, l^Q) \wedge \Gamma^P(l'^P, l^P)}$$

6.4 Alias Analysis

The alias analysis computes an aliasing summary for each procedure consisting of an entry points-to graph and an exit points-to graph. The alias analysis also computes a points-to graph for each global variable and a points-to graph for each *type*. A type points-to graph of a structure type expresses which fields of the structure type may point to the other fields of the same structure type over all instances of the structure type in the program. Define

$$\begin{array}{rcll} \mathit{ProcSummary}(P) & = & (\gamma^P_{entry}, \gamma^P_{exit}) & \text{the summary of procedure } P \\ \mathit{GlobalSummary}(G) & = & \gamma^G_{global} & \text{the summary of global } G \\ \mathit{TypeSummary}(T) & = & \gamma^T_{type} & \text{the summary of type } T \end{array}$$

6.5 Summary Generation

The memory analysis consumes the summaries generated by the alias analysis to compute sound, flow-sensitive, path-sensitive points-to information of a procedure. Define $\gamma^P_{entry}, \gamma^P_{exit}, \gamma^P_{global}, \gamma^P_{type}$ as

$$\begin{array}{rcl} (\gamma^P_{entry}, \gamma^P_{exit}) & = & \mathit{ProcSummary}(P) \\ \gamma^P_{global} & = & \bigsqcup_{G \in Global} \mathit{InstGlobal}^P(\mathit{GlobalSummary}(G)) \\ \gamma^P_{type} & = & \bigsqcup_{T \in Type} \mathit{InstType}^P(\mathit{TypeSummary}(T)) \end{array}$$

The functions $\mathit{InstGlobal}^P$ and $\mathit{InstType}^P$ are location instantiation functions that instantiate the locations used in the global and type summaries to locations of P.

The following inference rule states a condition under which the summary for a procedure is sound. The rule uses judgements of the form $\Gamma \vdash s; \Gamma'$ where Γ is the points-to graph in which statement s is executed, and Γ' is the points-to graph after which statement s is executed.

$$\frac{\begin{array}{c}\Gamma_{init} = \mathit{Initialize}(\gamma_{def}, \gamma_{entry}, \gamma_{global}, \gamma_{type}) \\ \Gamma_{init} \vdash s; \Gamma_{final}\end{array}}{\Gamma_{init} \vdash \texttt{proc P s}, \Gamma_{final}}$$

Section 6.5.1 explains the *Initialize* function. Section 6.5.2 gives inference rules for statements.

6.5.1 Initial Points-to Graph

The memory analysis first computes a sound initial points-to graph, $\Gamma_{init} = \mathit{Initialize}(\gamma^P_{def}, \gamma^P_{entry}, \gamma^P_{global}, \gamma^P_{type})$. The function $\mathit{Initialize} \in \mathcal{G}^4 \to \mathcal{G}$ computes the initial points-to graph by merging the default, entry, global, and type points-to graphs and associating a guard with each points-to edge which encodes disjointness among the targets of a particular location. Let γ_{merge} be the points-to graph that merges the default, entry, global and type points-to graphs defined by

$$\gamma_{merge} = \bigsqcup\{\gamma^P_{def}, \gamma^P_{entry}, \gamma^P_{global}, \gamma^P_{type}\}$$

Then,

$$\mathit{Initialize}(\gamma_{def}, \gamma_{entry}, \gamma_{global}, \gamma_{type})(l_i, l_j) = \sigma(\gamma_{merge})(l_i, l_j)$$

6.5.2 Statements

The statements that may appear in a procedure are assignments, branches, and calls. This section describes the inference rule for each kind of statement.

Assignments

The inference rule for assignment statements is

$$\frac{\Gamma \vdash_{lval} e_1 : \mathcal{E}_1 \quad \Gamma \vdash_{exp} e_2 : \mathcal{E}_2}{\Gamma \vdash \mathrm{e}_1 := \mathrm{e}_2; \mathit{Assign}(\Gamma, \mathcal{E}_1, \mathcal{E}_2)}$$

The function $\mathit{Assign} \in (\mathcal{G} \times (\mathit{Location} \to \mathit{Guard}) \times (\mathit{Location} \to \mathit{Guard})) \to \mathcal{G}$ is the points-to graph transfer function for an assignment statement. Define

$$\mathit{Assign}(\Gamma, \mathcal{E}_1, \mathcal{E}_2)(l_i, l_j) = [\mathcal{E}_1(l_i) \wedge \mathcal{E}_2(l_j)] \vee \left[\bigvee_{l \neq l_i} \mathcal{E}_1(l) \wedge \Gamma(l_i, l_j) \right] \vee [\mathit{Soft}(l_i) \wedge \Gamma(l_i, l_j)]$$

where $\mathit{Soft}(l)$ is the condition under which the location l represents more than one concrete memory location. For example, an array may be modeled by a single soft location.

Branches

The inference rule for branch statements is

$$\frac{\begin{array}{c}\Gamma \vdash s_1; \Gamma_1 \\ \Gamma \vdash s_2; \Gamma_2 \\ b \text{ is a fresh boolean variable}\end{array}}{\Gamma \vdash \text{if e } s_1 s_2; \mathit{Refine}(b, \Gamma_1) \sqcup \mathit{Refine}(\neg b, \Gamma_2)}$$

The rule joins the points-to graphs of each branch while preserving disjointness of points-to relationships by refining the points-to graph of one branch with a fresh boolean variable b and refining the points-to graph of the other branch with its negation $\neg b$.

Calls

The inference rule for call statements is

$$\frac{\mathit{ProcSummary}(Q) = (\gamma^Q_{entry}, \gamma^Q_{exit})}{\Gamma^P \vdash \texttt{call}_k \ \texttt{Q}; \mathit{Apply}(\Gamma^P, \gamma^Q_{exit})}$$

The function $\mathit{Apply} \in \mathcal{G}^2 \to \mathcal{G}$ computes the resulting points-to graph after applying the exit summary the callee at a call site k. Define a new domain called $\mathit{Location}_\perp$ that adds a special location called $\perp$ to $\mathit{Location}$:

$$\mathit{Location}_\perp = \mathit{Location} \cup \{\perp\}$$

Define a new domain $\mathcal{G}_\perp$ that adds the special location called $\perp$ to points-to graphs:

$$\mathcal{G}_\perp = (\mathit{Location} \times \mathit{Location}_\perp) \to \mathit{Guard}$$

Define a function $\mathit{Extend}_\perp \in \mathcal{G} \rightarrow \mathcal{G}_\perp$ that extends a points-to graph with an edge between every location and $\perp$ with a trivial guard:

$$\mathit{Extend}_\perp(\gamma)(l_1, l_2) = \begin{cases} \mathit{true} & l_2 = \perp \\ \gamma(l_1, l_2) & l_2 \neq \perp \end{cases}$$

Define a function $\mathit{Effect} \in \mathcal{G} \rightarrow \mathcal{G}_\perp$ that identifies the condition of points-to edges in a points-to graph extended with edges to $\perp$ and refined by σ:

$$\mathit{Effect}(\gamma)(l_1, l_2) = \sigma(\mathit{Extend}_\perp(\gamma))(l_1, l_2)$$

Define a function $\mathit{Pure} \in \mathcal{G} \rightarrow \mathit{Location} \rightarrow \mathit{Guard}$ that identifies the condition of a points-to edge pointing to $\perp$ in a points-to graph extended with edges to $\perp$ and refined by σ:

$$\mathit{Pure}(\gamma)(l) = \mathit{Effect}(\gamma)(l, \perp)$$

Finally,

$$\mathit{Apply}(\Gamma^P, \gamma^Q_{exit})(l^P_i, l^P_j) = \left[\Gamma^P(l^P_i, l^P_j) \wedge \psi_1\right] \vee \psi_2 \tag{6.9}$$

where

$$\psi_1 = \bigwedge_{l^Q_m} \left[\neg \mathcal{I}_k(l^Q_m, l^P_i) \vee \mathit{Pure}(\gamma^Q_{exit})(l^Q_m)\right]$$

$$\psi_2 = \bigvee_{l^Q_m, l^Q_n} \left[\mathcal{I}_k(l^Q_m, l^P_i) \wedge \mathcal{I}_k(l^Q_n, l^P_j) \wedge \mathit{Effect}(\gamma^Q_{exit})(l^Q_m, l^Q_n)\right]$$

The formula ψ_1 which appears in the first disjunct in equation 6.9 is the condition under which the points-to edge (l^P_i, l^P_j) is preserved across the call site. It is conjoined with $\Gamma^P(l^P_i, l^P_j)$, the guard of the points-to edge (l^P_i, l^P_j) of the points-to graph in which the call statement is executed. The formula ψ_1 is a conjunction of formulas where each conjunct is the condition under which a particular callee label l^Q_m does not instantiate to l^P_i, or l^Q_m instantiates to l^P_i but no side effect to l^Q_m appears in the exit summary of the callee.

The formula ψ_2 which appears as the second disjunct in equation 6.9 is the condition under which the points-to edge (l^P_i, l^P_j) is introduced into caller P as a side effect by callee Q at call site k. The formula ψ_2 is a disjunction of formulas where each disjunct is a conjunction of the condition that l^Q_m instantiates to l^P_i and the condition that l^Q_n instantiates to l^P_j and the condition of the callee side-effect $(l^Q_m$,$l^Q_n)$.

Chapter 7

Saturn Tools Reference

7.1 Overview

This chapter describes the various tools included with Saturn, how they interact and how they can be used to run analyses and view results. The sections are organized as follows:

- Section 7.2 describes how all syntax trees, summaries, and other intermediate data and results are stored on disk.

- Section 7.3 describes the various methods for generating syntax trees encoding the source program to be analyzed.

- Section 7.4 describes how to run the Saturn interpreter over the generated syntax trees and produce results.

- Section 7.5 describes how to set up and use the Saturn UI for viewing error reports.

7.2 Data Storage and Management

Almost all on-disk data used by Saturn is stored using BDB databases, which are simple key-value maps. These have the extension '.db' and can be easily copied around, deleted, reverted, and so forth. One only has to worry about a handful of files rather than enormous directories full of data. A few utility apps can be used to query these databases directly:

- `dbkeys file.db` : Prints out all keys in the named database.

- `dbfind file.db key` : Prints out the value for the given key. Most useful for databases with plaintext values.

- `dbfindc file.db key` : Prints out the plaintext contents of the session named 'key', provided that file.db is used to store session contents. Databases storing session contents are named according to the corresponding session, e.g. the contents of all `cil_body` sessions are stored in `cil_body.db`.

Classes of databases generated and used by Saturn are as follows:

7.2.1 Syntax Tree Databases

Databases are used to store the abstract syntax trees generated for the program to be analyzed (See Section 7.3). For each session name used to store syntax information, a single database will be generated. With CIL, these have the form:

```
cil_body.db
cil_comp.db
cil_enum.db
cil_glob.db
cil_init.db
```

`cil_body.db` stores all function bodies in the program, `cil_comp.db` stores all information on composite types, etc. The `dbkeys` and `dbfindc` programs can be used to query these databases.

7.2.2 Process Order Database

The file `process.db` is generated during syntax tree parsing and stores all process order information generated for the set of syntax trees. For function bodies, this file indicates the bottom up ordering of functions over the direct call graph. It may also be updated by running other analyses, e.g. generating the `cil_sum_body` sessions will add new process order edges.

7.2.3 Preprocessed Files Database

The file `ppfile.db` is generated during parsing and stores the contents of each source file after preprocessing. This is used by the UI for toggling between the original and preprocessed source code. Each value is stored as plaintext, and the database may be queried with `dbkeys` and `dbfind`.

7.2.4 Summary Databases

Running an inter-procedural analysis using the Saturn interpreter will populate a number of databases with summary information. These databases correspond one-to-one with the summary session names used by the analysis, e.g. if the analysis stores information in a summary session `sum_func`, then `sum_func.db` will contain all such summaries. These databases have the exact same form as syntax tree session databases (the interpreter makes no distinction between the two) and can be queried using `dbkeys` and `dbfindc`.

Note that to do a clean restart of a run without regenerating syntax trees, just delete the summary session databases.

7.2.5 Plaintext Output Databases

Some packages also store output in plaintext output databases separate from summary sessions. In particular, the UI display package stores display information in `display.db`, and search information in `search.db`. These are consumed directly by the UI, but can also be queried with `dbkeys` and `dbfind`.

7.3 Abstract Syntax Trees

Before using the interpreter to run a program analysis over a program, the source first needs to be compiled into an Abstract Syntax Tree (AST) representation stored in a session database. Since the C frontend that we use is the C Intermediate Language (CIL), we frequently refer to these ASTs as CIL trees.

7.3.1 Compiling Individual Source Files

To compile a single source file, use the `cilcc` command. `cilcc` is a program that takes preprocessed source files and compiles them into CIL trees. Since `cilcc` expects preprocessed source files as input, you will usually have to run your source file through the C preprocessor before passing it to `cilcc`. For example, consider the traditional “Hello world” program `hello.c`:

```
#include <stdio.h>

int main(void) {
    puts("Hello world");
    return 0;
}
```

To build `hello.c` using `cilcc`, you need to execute the following commands:

```
$ gcc -E hello.c -o hello.i
$ cilcc hello.i
```

The first command executes the GCC preprocessor to expand out the `#include` preprocessor directive, producing preprocessed source `hello.i`, and the second command invokes cilcc on the preprocessed source to produce the following CLPA session database files:

```
cil_body.db
cil_comp.db
cil_enum.db
cil_glob.db
cil_init.db
process.db
```

Some files may be missing; if the original source file has no static initializers, `cil_init.db` will be missing, and if the original source has no composite type or enum definitions, `cil_comp.db` or `cil_enum.db` will be missing, and so forth.

7.3.2 Compiling Large Systems

Most software systems consist of many source files, and use some sort of automated build software like `make`. CLPA provides two methods for obtaining CIL trees automatically from the build process of a software package.

The first of these is `clpamake.pl`, which is a script that scrapes through the output generated by the make process looking for calls to GCC. For each such call, it executes the corresponding command to build CIL trees from the given source files. In general this approach is unreliable since most codebases will produce more than one executable, and the `clpamake.pl` script will simply collapse all the source files from all of the binaries together. However it does have the advantage of being reasonably simple, portable and transparent.

The second is an adaption of the Berkeley build interceptor, which instruments the build process of an application, intercepting calls to the C compiler and linker and embedding information in the object files in order so it can reliably reconstruct the preprocessed source files that were compiled to produce a given object file. The main disadvantage is it is less portable since it relys on dynamic linker tricks and monkeying with the internals of GCC. It is only tested on Linux.

7.3.3 Using the Build Interceptor

Before using the build interceptor, it needs to be customized for the layout of the compiler binaries on your system. The configuration file is `build-intercept/interceptor.config` in the CLPA source tree.

Firstly, the `paths` section needs to be filled in with three values, each of which must be an absolute path:

- `intercept_scripts` should be the build-intercept/ subdirectory of your CLPA source tree.
- `intercept_library` should be the location of `libintercept.so` in the same directory.
- `intercept_home` can be anywhere where the interceptor can store temporary files from the `clpa-intercept` script for use by the `clpa-make` script. Since a lot of I/O is performed in this directory it should be on a local disk with a lot of space available.

In the `[Redirections]` section, you need to provide the location of the compiler binaries on your system. Examples are given for the compiler in Fedora Core 4, but you will probably need to add additional entries. In particular the location of `cc1` C compiler backend differs greatly between compiler versions and linux distributions.

The system provides two commands clpa-intercept, which intercepts a build process to add source file information to the object files produced by the build process, and clpa-extract, which takes one or more object files produced by a build process instrumented with clpa-intercept, extracts exactly those source files which were used in producing a given object file, and builds CIL trees from those source files.

To build openssh, for example:

```
$ tar zxvf openssh-*.tar.gz
$ cd openssh
$ ./configure
$ clpa-intercept make
$ mkdir ../ssh-trees
$ clpa-extract ../ssh-trees sshd
```

and you'll have some shiny new clpa trees corresponding to the ssh daemon (sshd) in ../ssh-trees.

Note that you have to specify the compiled binaries that you want to analyze. If there are multiple programs produced by your source package just pick the main one. Only the source files that went into producing the specified binaries will be included in the CIL trees.

NB. The C frontend does not handle symbol disambiguation in a particularly sophisticated way, so it is strongly recommended for correctness that you do not attempt to combine multiple binaries into the same set of CIL trees.

Building Linux with the Build Interceptor

To build CIL trees for Linux 2.6 together with all drivers (large):

```
$ make allyesconfig
$ make menuconfig
```

Go to the ”Loadable module support” item, and turn off ”Enable loadable module support”. This ensures that all of the drivers get linked together statically rather than built as separate modules. Quit menuconfig, saving the configuration.

```
$ clpa-intercept make vmlinux
```

Wait a couple of hours, and eventually a binary called vmlinux will be produced in the top level directory.

```
$ mkdir ../my-trees
$ clpa-extract ../my-trees vmlinux >\& extract.log
```

After about 12 hours some shiny new trees will be in ../my-trees. You can follow the progress of the build using ”tail -f extract.log”, including any errors that turn up. The files are built in alphabetical order so you should have a rough idea of the progress of the build. Unfortunately the cilcc compiler is quite slow.

7.4 CLP Interpreter

Once abstract syntax tree databases have been generated for the program of interest, the CLP interpreter `clpa` can be used to run a CLP analysis over those databases. The simplest way to use `clpa` is to `cd` to the directory containing the syntax tree databases, and enter:

```
clpa /path/to/analysis.clp
```

The following sections describe the main aspects of the interpreter and how they can be tuned via command line arguments.

7.4.1 Intra-procedural Analysis

Within each function or other session that is being analyzed, `clpa` simply continually executes rules from the analysis until there are no more rules to execute. The analysis rules use as input the syntax predicates in the currently analyzed function, as well as any other syntax sessions or summary sessions of other functions, and produce as output new predicates to add to summary sessions (as well as plaintext output to the screen and package output to other databases on disk).

The main way to control the interpreter's behavior is to set a timeout. By using the command line argument `--timeout N`, after `N` seconds of running the interpreter on any given function it will stop, print to the screen any output generated so far (as well as an error message), discard any modifications to sessions performed so far, and move onto the next function.

7.4.2 Inter-procedural Analysis

Across the entire source program being analyzed by `clpa` will analyze each function in turn, going in bottom-up order by default or top-down if specified within the analysis. It may also have to fixpoint over functions, reanalyzing the same function potentially many times, as summary sessions change. For example, if "foo" makes a direct call to "bar" in the source program, then the analysis of `cil_body("foo")` may depend on the summary for "bar", say `sum_func("bar")`. If "foo" is analyzed and `sum_func("bar")` later changes, then the behavior of the analysis in "foo" may be affected and "foo" must be reanalyzed.

These dependencies between analysis sessions and summary sessions are discovered on the fly as `clpa` runs, and all necessary reanalysis and fixpointing will be performed by default. This behavior can, however, be turned off via the `--no-fixpoint` command line argument to `clpa`, which restricts each function such that it is only analyzed once, in a single pass through the code.

7.4.3 Cluster-based Distributed Runs

For analyzing extremely large code bases, or even for much faster analysis of merely large ones, `clpa` can be run in a distributed mode on a cluster of computers. The main `clpa` process acts as a server, managing the function analysis worklist, reads and writes to databases, and printing any per-function output to the screen. Multiple workers are then spawned, each of which opens a connection with the server, receive and process functions to be analyzed, then send any output back to the server. Workers are stateless and do not write anything to their local disk.

Performance speedup can vary greatly depending on the worker CPU/RAM and network speed/latency. Our own runs on a 50 core cluster with a switched Gigabit network typically yield a 40-45x speedup over a single core, though ad-hoc clusters of computers with low-latency connections between them should also perform well.

The simplest way to start a distributed run is to run `clpa` with all the arguments that would normally be used in the single threaded mode, but add the `--use-workers` command line argument as well:

```
clpa --use-workers other-clpa-args
```

Rather than running the analysis itself, this tells `clpa` to act as a server, and it will print the `address:port` it is listening on for workers to connect. A worker process can be spawned as `clpa-worker address port`, which will immediately connect to the given server address and start analyzing functions. The `clpa-worker` processes can be started up manually or via a script on different machines, or can be spawned via cluster submission utilities.

Starting up workers this way on a cluster can be problematic if the cluster is busy and other users need time; the `clpa-worker` process will by default run until the entire fixpointing analysis has finished, and will not timeout or share the resources in any way. This can be fixed by adding the `--spawn-workers` and `--spawn-cmd` command line arguments to `clpa`:

```
clpa --use-workers --spawn-workers NUM --spawn-cmd CMD other-clpa-args
```

When these arguments are set, the server itself is responsible for spawning workers via the `CMD` command. This command should take two arguments for the address and port, and spawn a `clpa-worker` with those arguments (for example, for the SUN gridengine the `CMD` might be `"qsub -b y -N analysis-name /path/to/clpa-worker"`). In addition, each worker will exit after 10-20 minutes regardless of whether the main interprocedural analysis has finished yet; the server will continually respawn workers to make sure there are always `NUM` workers either actively analyzing functions, or waiting in the cluster queue system for a node to run on.

7.4.4 Checkpointing

For very long single threaded or distributed runs, being able to checkpoint the `clpa`'s progress can be useful. If `clpa` is killed using the SIGINT signal then it will clean up its state before exiting, but if it is killed any other way then the databases will probably be left in a corrupt state.

Adding the `clpa` comand line argument `--checkpoint N` causes a checkpoint to be created after analyzing every `N` functions. Only the latest checkpoint is stored, and it is placed by default in the `checkpt` directory (this can be changed with the `--checkpoint-dir DIR` argument). This directory will contain copies of all the summary databases at the checkpoint, as well as a dump of the current worklist state in file `work.chk`. If the plug gets pulled or another unfortunate event kills `clpa` later, delete all the (probably corrupt) summary databases and rerun `clpa` with the `--checkpoint-resume` argument, and it will copy in the checkpointed summaries and worklist state and resume where it left off.

7.4.5 Controlling Output

There are various options for controlling the level of output produced by `clpa`.

Quiet

Adding the `--quiet` argument to `clpa` suppresses printing of functions analyzed, timing and allocation info, and everything else except the actual analysis output. This is most useful when writing regression scripts, as a plain sorted diff of the output against expected will suffice if `--quiet` is on.

Print level

The `--print-level` *n* option allows you to specify a limit on the depth of subterms that should be printed when converting a term to a string. Terms below the specified depth will be printed as "...". The default depth limit is 15 terms. To disable depth limiting, pass a depth limit parameter of -1.

Statistics

A variety of statistics flags can be turned on via the `--stats flag` argument to `clpa`. These cover both single function information (SAT queries performed, etc.) as well as whole run information (functions analyzed, memory usage, etc.). The full list of possible stats flags with descriptions is given by the `--list-stats` argument to `clpa`.

Debugging

A variety of debugging flags can also be turned on via the `--debug flag` argument to `clpa`. The easiest way to debug analyses is via inserting print() adds in the relevant places, but some of the debug flags can also be useful. The full list of possible debug flags with descriptions is given by the `--list-debug` argument to `clpa`.

7.5 User Interface

The Saturn user interface is a web-based viewer for the error reports and other summary information generated by Saturn analyses. The UI structure is such that each analysis may generate a set of *displays*, which specify a section of the source C program to display and directions for highlighting particular lines and adding additional text. Each display describes a single property the analysis has discovered for a single function. Displays have unique names, so that interprocedural analyses can construct links between their various displays which can be browsed by the user either as either statically or dynamically generated HTML pages.

Individual Saturn analyses can generate output either as UI displays via the `display` package (Section 8.13), plaintext (queries, etc.), or (usually) both. The `display` package generates a display database 'display.db' and search database 'search.db'.

Section 7.5.2 describes how to use the UI to convert the above databases into a set of static HTML pages. This does not require a web server, but for large sets of displays (in the 10,000s or 100,000s) the amount of time taken to generate the HTML becomes very long and the size of the resulting HTML becomes very large in comparison to the original display database.

Section 7.5.3 describes how to set up the UI on a web server to generate HTML dynamically. This requires little disk space per run besides the display database itself, and also allows the user to search the displays generated by multiple analyses for particular functions.

7.5.1 Configuration Files

The static and dynamic UI both use configuration files to specify where the relevant files and directories they require for each display database are. These files have the extension '.conf', and consist of several lines of key-value pairs with whitespace between the key and value. The possible keys and meaning of the associated values are as follows:

- `display`: Specifies the display database 'display.db' containing all displays which were produced by the Saturn analysis.

- `srcpath`: Specifies the root directory of the C source tree the CIL databases were generated from for the Saturn analysis. The individual displays contain file offsets into this directory which the UI will use when generating the HTML.

- `ppfile`: Optional location of the 'ppfile.db' generated by the frontend. This file contains preprocessed versions of the various source files under `srcpath`, and if specified in the configuration then in the resulting displays the user will be able to toggle the various source code lines between the original and preprocessed versions.

- `style`: Specifies the file or URL location of the CSS style sheet to use with the HTML. Most analyses have a style sheet associated with them that is an extension of 'ui/www/styles/default.css'.

- `binpath`: Specifies the directory containing the `dbkeys` and `dbfind` to use, generally the 'bin' output directory of the Saturn make process.

- `search`: Used only by the dynamic UI, specifies the search database 'search.db' containing display search terms.
- `name`: Used only by the dynamic UI, specifies a name to print for this analysis when the user is doing a search.

7.5.2 Static UI

The UI script 'ui/scripts/static.pl' generates a complete, inter-linked directory of static HTML files and associated table of contents for a single display database. 'static.pl' is invoked as follows:

```
static.pl file.conf [category]
```

The configuration file is required and specifies the location of the configuration file to use for finding and rendering the displays. The category is optional and if specified restricts the generated table of contents to only those displays with that category (See Section 8.13).

After running, a new subdirectory will be created in the current directory with the same name as the display database. This directory contains file 'index.html' with a table of contents for all the displays, and separate HTML files for each display. Open the 'index.html' file in the output directory to view and browse the displays.

7.5.3 Dynamic UI

To set up the UI to generate HTML dynamically, perform the following steps:

1. Place the 'search.pl' and 'report.pl' files and all Perl modules from the 'ui/scripts' directory into a directory in which CGI script execution is enabled. For information about setting up mod_perl, see `http://perl.apache.org`.
2. Create a 'configs' subdirectory in the directory from step 1.
3. Within the 'configs' subdirectory, create one or more configuration files describing the different display databases you want to be accessible from the web interface.

The dynamic UI can then be used in two ways. First, the various display databases can be searched by opening the URL of the 'search.pl' script via a web browser. This will provide a drop-down list of the various databases and a search box. The only terms that searching will work for are those that were added to the search database when the analysis ran (See Section 8.13).

Second, new tables of contents can be generated statically which link to various displays within the dynamic UI. Invoke the 'ui/scripts/dyntoc.pl' as follows:

```
dyntoc.pl file.conf reportURL [category]
```

The configuration file is required and must be within the ‘configs’ subdirectory setup earlier. The reportURL is also required and must point to the ‘report.pl’ file on the server; this will be used for display links in the table of contents. The category is optional and if specified restricts the generated table of contents to only those displays with that category.

After running a single file ‘index.html’ will be produced in the current directory. This can be moved or renamed as desired, and then opened up in a web browser to view and browse the displays.

Chapter 8

Saturn CLP Packages Reference

8.1 Overview

Packages provide interfaces for CLP programs to interact with external routines written in other languages such as OCaml and C. These routines may, for example, provide parsing information for the source code being analyzed, or perform constraint based analyses.

Each package defines a set of symbols for inclusion in logic programs. Programs may access the contents of a package via the 'using' directive. The following types of symbols may be included in a package:

- Type definitions. These are regular type definitions which can be used in package as well as user-defined predicates. Package types are either composite sumtypes or are left abstract.
- Add predicates. These are predicate definitions that can only be used in add operations. Whenever they are added they commit information to the package via an internal handler.
- Find predicates. These are predicate definitions that can only be used in find operations. Whenever a query is made for them information is extracted from the package via an internal handler. Find predicates can have more than one mode.
- Collection predicates. These are predicate definitions that can only be used in the right side of collection operations (any non-add predicate may be used in the left side). An internal handler in the package specifies how to combine the set of facts being quantified over.
- Predicate definitions. These are regular predicates which can be used in any operation, and have no special handlers in the package.

- Session definitions. These are regular session functions which can be used as with user-defined session functions.

The remainder of this section lists and describes all symbols included in the available packages.

8.2 Builtin

Package builtin is used by default. The builtin predicates define generally useful routines for writing logic programs.

- `notequal(in V0:T,in V1:T)` **Find**

`notequal` holds if `V0` and `V1` (which must be the same type) are different ground values.

- `lessthan(in V0:T,in V1:T)` **Find**

`lessthan` holds if `V0` is less than `V1` (which must be the same type), under a total ordering dependent on the types of `V0` and `V1`.

- `tostring(in V:T,out S:string)` **Find**

Binds `S` to the string representation of `V`.

- `toint(in V:T,out N:int)` **Find**

If `V` has an integer representation, binds `N` to that representation. Will succeed if `V` is an integer, float, or integer string. If `V` is a floating point value, `N` will be truncated.

- `tofloat(in V:T,out N:float)` **Find**

If `V` has a floating point representation, binds `N` to that representation. Will succeed if `V` is an integer, float, or floating point string.

- `print(...)` **Add**

Adding `print` prints the string representation of its arguments (of which there may be any number) to the screen.

- `warning(STR:string,...)` **Add**

Adding `warning(STR,...)` prints a warning `STR` along with any extra arguments (of which there may be any number).

- `profile(...)` **Add**

The number of times `profile` is added with a particular set of arguments is counted up by the system. After the current session finishes executing (including normal termination as well as premature timeouts), this total will be printed for every distinct set of arguments.

- `list_mem(in L:list[T],out V:T)` **Find**

Binds `V` to any member of the list `L`.

- `list_length(in L:list[T],out N:int)` **Find**

Binds `N` to the length of list `L`.

- `list_sort(in L:list[T],out V:list[T])` **Find**

Binds `V` to the sorted list (according to the ordering given by `lessthan()`) containing all elements of `L`.

- `list_append(in L0:list[T],in L1:list[T],out V:list[T])` **Find**

Binds `V` to the result of appending list `L1` to the end of `L0`.

- `list_reverse(in L:list[T],out V:list[T])` **Find**

Binds `V` to the reverse of list `L`.

- `list_all(in V:T,out L:list[T])` **Collection**

Used in collection operations to extract the `V` value of all facts matching a query into a list `L`. For example, the collection operation \/`pair(0,V):list_all(V,L)` binds `L` to a list containing each value `V` that the constant 0 is paired with.

The lists bound by `list_all` will be sorted, but may contain duplicate entries if multiple facts unify with `V` to the same value. Note that if additional facts are later added that also match the query, an error message will be generated.

- `t_pair[T0,T1] ::= pair{V0:T0,V1:T1}` **Type**

A pair of two values of any type.

- `bool ::= true | false` **Type**

A simple two-valued type for boolean constants.

- `maybe[T] ::= yes{T} | no` **Type**

A polymorphic type representing an optional value. Similar to the option type in the ML family of languages.

8.3 Integer Operations

Package intops provides numerous operations on integers. These operations do not introduce constraints, and as such all arguments must be bound to constant integers, excepting the result `NR` if present.

- `int_neg(in N:int,out NR:int)` **Find**

Binds `NR` to `-N`.

- `int_add(in N0:int,in N1:int,out NR:int)` **Find**

Binds `NR` to `N0 + N1`.

- `int_sub(in N0:int,in N1:int,out NR:int)` **Find**

Binds `NR` to `N0 - N1`.

- `int_mul(in N0:int,in N1:int,out NR:int)` **Find**

Binds `NR` to `N0 * N1`.

- `int_div(in N0:int,in N1:int,out NR:int)` **Find**

Binds `NR` to `N0 / N1`.

- `int_mod(in N0:int,in N1:int,out NR:int)` **Find**

Binds `NR` to `N0 % N1`.

- `int_max(in N0:int,in N1:int,out NR:int)` **Find**

Binds `NR` to the larger of `N0` and `N1`.

- `int_min(in N0:int,in N1:int,out NR:int)` **Find**

Binds `NR` to the smaller of `N0` and `N1`.

- `int_band(in N0:int,in N1:int,out NR:int)` **Find**

Binds `NR` to the bitwise-and of `N0` and `N1`.

- `int_bor(in N0:int,in N1:int,out NR:int)` **Find**

Binds `NR` to the bitwise-or of `N0` and `N1`.

- `int_bxor(in N0:int,in N1:int,out NR:int)` **Find**

Binds `NR` to the bitwise-xor of `N0` and `N1`.

- `int_eq(in N0:int,in N1:int)` **Find**

Holds when `N0 == N1`.

- `int_ne(in N0:int,in N1:int)` **Find**

Holds when `N0 != N1`.

- `int_lt(in N0:int,in N1:int)` **Find**

Holds when `N0 < N1`.

- `int_gt(in N0:int,in N1:int)` **Find**

Holds when `N0 > N1`.

- `int_le(in N0:int,in N1:int)` **Find**

Holds when `N0 <= N1`.

- `int_ge(in N0:int,in N1:int)` **Find**

Holds when `N0 >= N1`.

- `int_min_all(in N:int,out Min:int)` **Collection**

Compute the minimum of a set of integers given by collection over `N`. Fails if the collection is empty.

- `int_max_all(in N:int,out Max:int)` **Collection**

Compute the maximum of a set of integers given by collection over `N`. Fails if the collection is empty.

8.4 String Operations

Package strops provides numerous operations on strings. These operations do not introduce constraints, and as such all arguments where noted must be bound to constant values.

- `str_len(in S:string,out LENR:int)` **Find**

Binds `LENR` to the length of constant string `S`.

- `str_cat(in S0:string,in S1:string,out SR:string)`
  `str_cat(in S0:string,out S1:string,in SR:string)`
  `str_cat(out S0:string,in S1:string,in SR:string)` **Find**

Binds `SR` to the concatenation of constant strings `S0` and `S1`. Alternatively, if `SR` and at least one of `S0` and `S1` is constant, binds the remaining values as appropriate.

- `str_sub(in S:string,in POS:int,in LEN:int,out SR:string)`
  `str_sub(in S:string,out POS:int,out LEN:int,in SR:string)` **Find**

Binds `SR` to the substring of constant string `S` denoted by constant integers `POS` and `LEN`. Alternatively, if `SR` is a constant string, binds `POS` and `LEN` according to any/all occurrences of `SR` within `S`.

8.5 Map ADT

Package `map` provides an implementation of the map abstract data type.

- `map[Key,Value]` **Type**

Maps are an abstract type with key and value type parameters.

- `map_empty(out M:map[K,V])` **Find**

Binds `M` to an empty map.

- `map_insert(in Key:K, in Value:V, in M0:map[K,V],`
  `           out M1:map[K,V])` **Find**

Insert a mapping from `Key` to `Value` into map `M0` to produce a new map `M1`. Any existing mapping for `Key` is overwritten.

- `map_search(in M:map[K,V], out Key:K, out Value:V)` **Find**

Search for `Key` in map `M`, returning the corresponding value `Value` on success. If `Key` is unbound, then all key-value mappings in the map will be returned.

- `map_remove(in Key:K, in M0:map[K,V], out M1:map[K,V])` **Find**

Remove any mapping for `Key` from map `M0` to produce a new map `M1`. If `Key` is not present in `M0` then the map is unchanged.

- `map_to_sorted_list(in M:map[K,V], out L:list[t_pair[K,V]])` **Find**

Convert a map `M` to a list of key-value pairs `L`, sorted by key.

- `map_of_list(in L:list[t_pair[K,V]], out M:map[K,V])` **Find**

Convert an unordered list of key-value pairs `L` to a map. If duplicate keys are present in the list, the last key-value mapping will be used.

- `map_all(in Key:K, in Value:V, out M:map[K,V])` **Collection**

Collect all key-value pairs matching a query into a map.

8.6 Set ADT

Package `set` provides an implementation of the set abstract data type.

- `set[Value]` **Type**

Saps are an abstract type with key and value type parameters.

- `set_empty(out S:set[V])` **Find**

Binds `S` to an empty set.

- `set_singleton(in Value:V, out S:set[V])` **Find**

Create a singleton set `S` containing `Value`.

- `set_insert(in Value:V, in S0:set[V], out S1:set[V])` **Find**

Insert an element `Value` into set `S0` to produce a new set `S1`.

- `set_member(in S:set[V], out Value:V)` **Find**

Test whether element `V` is present in set `S`. If `V` is unbound, then the predicate will succeed once with each element of the set bound to `V`.

- `set_remove(in Value:V, in S0:set[V], out S1:set[V])` **Find**

Remove element `V` from set `S0` to produce a new set `S1`. If `V` is not present in `S0` then the set is unchanged.

- `set_union(in S0:set[V], in S1:set[V], out S2:set[V])` **Find**

Set `S2` to the union of `S0` and `S1`.

- `set_intersect(in S0:set[V], in S1:set[V], out S2:set[V])` **Find**

Set `S2` to the intersection of `S0` and `S1`.

- `set_difference(in S0:set[V], in S1:set[V], out S2:set[V])` **Find**

Set `S2` to the set difference of `S0` and `S1`.

- `set_to_sorted_list(in S:set[V], out L:list[V])` **Find**

Convert a set `S` to a list of values `L`, sorted by key.

- `set_of_list(in L:list[V], out S:set[V])` **Find**

Convert an unordered list of values `L` to a set `S`. Duplicate elements are ignored.

- `set_all(in V:V, out S:set[V])` **Collection**

Collect all values matching a query together into a set. Equivalent to calling `list_all` followed by `set_of_list`.

8.7 Boolean Formula Construction

Package biteval is used by default. Biteval provides support for constructing and combining boolean formulas, principally through `#id_g`, `#and`, `#or`, and `#not`. Boolean formulas are an abstract type polymorphic in the values of the leaf variables. Package solve_sat may be used to solve boolean formulas where the leaves are left as uninterpreted boolean variables, while package solve_mip may be used to construct and solve formulas where the leaves are linear formulas over integer and floating point variables.

- `bval[T]` **Type**

The abstract type of boolean formulas. Each leaf `V` in the formula representing a boolean variable has type `T`.

- `#bool_g(in B:bool,out GR:bval[T])` **Find**

Binds `GR` to the constant formula (of any leaf type) equal to the boolean `B`.

- `#id_g(in V:T,out GR:bval[T])` **Find**

Binds `GR` to the formula for the unconstrained boolean variable represented by `V`. If the same `V` is used multiple times, the same formula will result.

- `#not(in G:bval[T],out GR:bval[T])` **Find**

Binds `GR` to the negation of `G`.

- `#and(in G0:bval[T],in G1:bval[T],out GR:bval[T])` **Find**

Binds `GR` to the conjunction of `G0` and `G1`.

- `#or(in G0:bval[T],in G1:bval[T],out GR:bval[T])` **Find**

Binds `GR` to the disjunction of `G0` and `G1`.

- `#and_all(in G:bval[T],out GR:bval[T])` **Collection**

Used in collection operations to compute the conjunction `GR` over the set of formulas `G` as indicated by the results of the query used for the collection. The principal difference between using `#and_all` for collection rather than `list_all` is that for `list_all` the query is performed immediately to yield the resulting list, whereas for `#and_all` the query is delayed until the exact value of the formula is needed for a SAT query. As long as SAT queries are delayed until after all formulas have been computed, none will have their results invalidated (with resulting error messages) by newly introduced formulas.

- `#or_all(in G:bval[T],out GR:bval[T])` **Collection**

Used in collection operations to compute the disjunction `GR` over the set of formulas `G` as indicated by the results of the query used for the collection. The same considerations as with `#and_all` apply for `#or_all`.

- `#g_id(in G:bval[T],out VR:T)` **Find**

Binds `VR` to a leaf in the formula `G`. Separate predicates will be instantiated for each leaf in `G`.

- `#g_size(in G:bval[T],out N:int)` **Find**

Binds `N` to the number of distinct non-constant sub-formulas of `G`.

- `#simplify(in G:bval[T],out GR:bval[T])` **Find**

Simplifies `G`, resolving all collection operations and applying as much constant folding and other simplification as possible according to a simplification level `L` on each sub-formula of `G`, binding `GR` to that simplified representation. Simplification levels are provided for each new formula's construction; `#not`, `#and`, and `#or` each use level 100, the maximum. Other packages that construct formulas may use smaller levels that perform fewer simplifications (but lead to an overall faster running time). Note that simplification is performed by all SAT queries, as well as other operations such as adding to a summary. This predicate is only necessary to get a (fairly) canonical representation of a formula to allow cheap and accurate formula comparison and indexing.

- #not_lv(in G:bval[T],in L:int,out GR:bval[T])
  #not_lv(out G:bval[T],out L:int,in GR:bval[T]) **Find**

Variant on #not that makes the simplification level L used explicit. May be used with two modes, either to construct a negation formula with the desired simplification level, or to break GR down into G and L, provided that GR is itself a negation formula.

- #and_lv(in G0:bval[T],in G1:bval[T],in L:int,out GR:bval[T])
  #and_lv(out G0:bval[T],out G1:bval[T],out L:int,
  in GR:bval[T]) **Find**

Variant on #and that makes the simplification level L used explicit. May be used with two modes, either to construct a conjunction formula with the desired simplification level, or to break GR down into G0, G1 and L, provided that GR is itself a conjunction.

- #or_lv(in G0:bval[T],in G1:bval[T],in L:int,out GR:bval[T])
  #or_lv(out G0:bval[T],out G1:bval[T],out L:int,
  in GR:bval[T]) **Find**

Variant on #or that makes the simplification level L used explicit. May be used with two modes, either to construct a disjunction formula with the desired simplification level, or to break GR down into G0, G1 and L, provided that GR is itself a disjunction.

8.8 Boolean Constraint Solving

Package solve_sat provides an interface with the SAT solvers MiniSAT and zChaff for solving boolean formulas over unconstrained variables. These formulas are created using the predicates from package biteval, which is used by default.

- #sat(in G:bval[T]) **Find**

Holds if G is satisfiable.

- bval_asn[T] **Type**

The type of a satisfying assignment for a boolean formula over unconstrained variables of type T.

- `#satasn(in G:bval[T],out ASN:bval_asn[T])` **Find**

Holds if `G` is satisfiable, and binds `ASN` to a satisfying assignment of booleans to each boolean variable in `G`.

- `#asng(in ASN:bval_asn[T],in G:bval[T])` **Find**

Holds if `G` is true under boolean assignment `ASN`. Variables in `G` not used in the formula tested to construct `ASN` are treated as false.

8.9 Bitvector Operations

Package `solve_bitvector` provides numerous predicates for constructing and manipulating vectors of boolean formulas, which may then be solved using the predicates in package `solve_sat`. Bit vectors are an abstract type over the type of values used to construct unconstrained vectors, as well as an additional type which may be used to insert individual unconstrained variables into formulas.

- `bvec[T,U]` **Type**

The type of a bitvector. Each bitvector is a signed or unsigned vector of boolean formulas, each of which has leaf formulas constructed either from unconstrained vector IDs of type `T`, or from individual unconstrained bits of type `U`. The format of these leaves is given by `bvecbit[T,U]` below.

- `bvecbit[T,U] ::= b_vbit{V:T,N:int} | b_abit{V:U}` **Type**

The type of a leaf boolean variable in a formula used in bitvectors. Leaf variables `b_vbit{V,N}` indicate bit position `N` of the ID `V`, while variables `b_abit{V}` indicate individual unconstrained variables unique to `V`.

- `#id_bv(in V:T,in S:bool,in O:bool,in LEN:int,`
  `out BVR:bvec[T,U])` **Find**

Binds `BVR` to a vector of unconstrained boolean variables unique to `V`. The sign and length of the vector are indicated by `S` and `LEN` respectively. If `O` is true, then the overflow bit of the vector is an unconstrained boolean variable, whereas if `O` is false then the overflow bit is a constant 0 bit. If the same `V` is used multiple times, the same bit vector will be bound.

- `#int_bv(in N:int,in S:bool,in LEN:len,`
  `out BVR:bvec[T,U])` **Find**

Binds `BVR` to a bit vector of sign/length `S/LEN` representing the constant `N`.

- `#g_bv(in G:bval[bvecbit[T,U]],in S:bool,in LEN:int,`
  `out BVR:bvec[T,U])` **Find**

Binds `BVR` to a bit vector of sign/length `S`/`LEN` whose first bit is `G`.

- `#bv_cast(in BV:bvec[T,U],in S:bool,in LEN:int,`
  `out BVR:bvec[T,U])` **Find**

Binds `BVR` to a bit vector cast from `BV`, changing the signed/unsigned flag `S` and vector length `LEN`.

- `#bv_int(in BV:bvec[T,U],out NR:int)` **Find**

Binds `NR` to an integer representation of `BV`, provided that `BV` is a constant vector.

- `#bv_neg(in BV:bvec[T,U],out BVR:bvec[T,U])` **Find**

Binds `BVR` to `-BV`.

- `#bv_bnot(in BV:bvec[T,U],out BVR:bvec[T,U])` **Find**

Binds `BVR` to `~BV`.

- `#bv_add(in BV0:bvec[T,U],in BV1:bvec[T,U],`
  `out BVR:bvec[T,U])` **Find**

Binds `BVR` to `BV0 + BV1`.

- `#bv_sub(in BV0:bvec[T,U],in BV1:bvec[T,U],`
  `out BVR:bvec[T,U])` **Find**

Binds `BVR` to `BV0 - BV1`.

- `#bv_band(in BV0:bvec[T,U],in BV1:bvec[T,U],`
  `out BVR:bvec[T,U])` **Find**

Binds `BVR` to `BV0 & BV1`.

- `#bv_bor(in BV0:bvec[T,U],in BV1:bvec[T,U],`
  `out BVR:bvec[T,U])` **Find**

Binds `BVR` to `BV0 | BV1`.

- `#bv_bxor(in BV0:bvec[T,U],in BV1:bvec[T,U],`
  `out BVR:bvec[T,U])` **Find**

Binds `BVR` to `BV0 ^ BV1`.

- `#bv_mul(in BV:bvec[T,U],in N:int,out BVR:bvec[T,U])` **Find**

Binds `BVR` to `BV * N`, where `N` is a constant integer.

- `#bv_shl(in BV:bvec[T,U],in N:int,out BVR:bvec[T,U])` **Find**

Binds `BVR` to `BV << N`, where `N` is a constant integer.

- `#bv_shr(in BV:bvec[T,U],in N:int,out BVR:bvec[T,U])` **Find**

Binds `BVR` to `BV >> N`, where `N` is a constant integer.

- `#bv_eqz(in BV:bvec[T,U],out GR:bval[bvecbit[T,U]])` **Find**

Binds `GR` to a formula holding if `BV` is zero.

- `#bv_nez(in BV:bvec[T,U],out GR:bval[bvecbit[T,U]])` **Find**

Binds `GR` to a formula holding if `BV` is non-zero.

- `#bv_eq(in BV0:bvec[T,U],in BV1:bvec[T,U],`
  `out GR:bval[bvecbit[T,U]])` **Find**

Binds `GR` to a formula holding if `BV0 == BV1`.

- `#bv_ne(in BV0:bvec[T,U],in BV1:bvec[T,U],`
  `out GR:bval[bvecbit[T,U]])` **Find**

Binds `GR` to a formula holding if `BV0 != BV1`.

- `#bv_lt(in BV0:bvec[T,U],in BV1:bvec[T,U],`
  `out GR:bval[bvecbit[T,U]])` **Find**

Binds `GR` to a formula holding if `BV0 < BV1`.

- `#bv_gt(in BV0:bvec[T,U],in BV1:bvec[T,U],`
  `out GR:bval[bvecbit[T,U]])` **Find**

Binds `GR` to a formula holding if `BV0 > BV1`.

- #bv_le(in BV0:bvec[T,U],in BV1:bvec[T,U],
  out GR:bval[bvecbit[T,U]]) **Find**

Binds GR to a formula holding if BV0 <= BV1.

- #bv_ge(in BV0:bvec[T,U],in BV1:bvec[T,U],
  out GR:bval[bvecbit[T,U]]) **Find**

Binds GR to a formula holding if BV0 >= BV1.

- #asnbv(in ASN:bval_asn[bvecbit[T,U]],
  in BV:bvec[T,U],out NR:int) **Find**

Binds NR to the value of BV under boolean assignment ASN.

- #bv_bit(in BV:bvec[T,U],in N:int,
  out GR:bval[bvecbit[T,U]]) **Find**

Binds GR to the formula in bit position N of BV. Bit positions are ordered from least to most significant.

- #bv_oflow(in BV:bvec[T,U],out GR:bval[bvecbit[T,U]]) **Find**

Binds GR to the overflow bit for BV, which indicates whether BV represents the result of an overflowing computation.

- #bv_split(in BV:bvec[T,U],
  out SR:bool,out OGR:bval[bvecbit[T,U]],
  out BITS:list[bval[bvecbit[T,U]]]) **Find**

Splits BV into its component formulas. SR indicates whether BV is signed, OGR indicates whether BV is the result of an overflowing computation, and BITS are all the individual bits in BV, from least to most significant order. If BV is variable, binds BV to the bitvector specified by the remaining arguments (which must all be constant).

- #bv_guard(in BV:bvec[T,U],in G:bval[bvecbit[T,U]],
  out BVR:bvec[T,U]) **Find**

Binds GR to the guarded bitvector formed by conjoining each bit of BV with G.

- `#bv_all(in BV:bvec[T,U],in G:bval[bvecbit[T,U]], out BVR:bvec[T,U])` **Collection**

Used in collection operations to compute the disjunction `BVR` over a series of conjoined bitvector/guard pairs `BV/G`. If `BV0/G0`...`BVM/GM` are the bitvectors and guards used in performing the collection, then bit `N` of `BVR` is equal to (`G0`∧`BV0[N]`)∨...∨(`GM`∧`BVM[N]`).

8.10 Linear and Integer Constraint Solving

Package `solve_mip` provides an interface with the mixed integer linear program solver lp_solve for solving boolean formulas over linear and integer constraints.

Constraints compare linear formulas over floating point or integer variables with constant floating point values. Formulas are constructed with `#id_ipval` and `#flt_ipval`, combined with `#ip_add`, `#ip_sub`, `#ip_mul`, and `#ip_div`, and formed into constraints with `#ip_cst`. These constraints can be combined using the predicates in biteval, and optimized or tested for satisfiability with `#ip_sat`, `#ip_minimize`, and `#ip_maximize`.

As with unconstrained boolean variables and bit vectors, linear formulas and linear constraints are polymorphic in the type of value used to construct unconstrained linear variables.

- `ipval[T]` **Type**

The type of a linear formula. Each formula is a linear combination of floating point or integer variables, each of which is constructed from an ID value `V` of type `T`.

- `ipcst[T,U]` **Type**

The type of a leaf constraint in a boolean formula over linear constraints. Each leaf is either a linear constraint comparing a formula over values of type `T` with a constant value, or is an opaque boolean variable of type `U`. A boolean formula over linear constraints has type `bval[ipcst[T,U]]`.

- `ipkind ::= ip_float | ip_int | ip_binary` **Type**

The possible kinds of variables used in linear constraints, distinguished by the possible values they can take.

- `ip_float`: Any floating point value.
- `ip_int`: Any integer.
- `ip_binary`: Either 0 or 1.

- `#id_ipval(in V:T,in KIND:ip_kind,out IVR:ipval[T])` **Find**

Binds `IVR` to a formula containing a single variable with kind `KIND`. If the same `ID` is used multiple times, the same formula will be bound.

- `#flt_ipval(in N:float,out IVR:ipval[T])` **Find**

Binds `IVR` to a constant formula for the specified floating point constant `N`.

-
  ```
  #ip_add(in IV0:ipval[T],in IV1:ipval[T],
          out IVR:ipval[T])
  ```
  Find

Binds `IVR` to the sum of formulas `IV0` and `IV1`.

-
  ```
  #ip_sub(in IV0:ipval[T],in IV1:ipval[T],
          out IVR:ipval[T])
  ```
  Find

Binds `IVR` to the difference of formulas `IV0` and `IV1`.

- `#ip_mul(in IV:ipval[T],in N:float,out IVR:ipval[T])` **Find**

Binds `IVR` to the result of multiplying `IV` by the floating point constant `N`.

- `#ip_div(in IV:ipval[T],in N:float,out IVR:ipval[T])` **Find**

Binds `IVR` to the result of dividing `IV` by the floating point constant `N`.

-
  ```
  ipval_terms[T] ::= term_ax{A:float,V:T,TAIL:ipval_terms[T]}
                   | term_z
  ```
  Type

The type of a decomposition of the terms in a linear formula. `term_ax{A,V,TAIL}` indicates the sum of `A*V` and the value represented by `TAIL`. `term_z` indicates an empty set of terms, zero.

-
  ```
  #ip_split(in IV:ipval[T],
            out TERMS:ipval_terms[T],out B:float)
  ```
  Find

Splits `IV` into its component terms and constants. `TERMS` is bound to a decomposition of all the terms in the formula. `B` is bound to the fixed constant offset of the formula.

-
  ```
  ipcstop ::= ip_lt | ip_gt | ip_le | ip_ge
            | ip_eq | ip_ne
  ```
  Type

The possible ways to compare a linear formula with a constant value.

-
  ```
  #ip_cst(in IV:ipval[T],in OP:ipcstop,in N:float,
          out GR:bval[ipcst[T,U]])
  #ip_cst(out IV:ipval[T],out OP:ipcstop,out N:float,
          in GR:bval[ipcst[T,U]])
  ```
  Find

Binds GR to a boolean formula which holds when the formula IV bears the relation given by OP with the constant N. Note that for ip_ne, ip_lt, and ip_gt, the resulting constraint will treat IV as if it has an integer value. Specifically, IV != N is converted to IV <= N-1 | IV >= N+1, IV < N is converted to IV <= N-1, and IV > N is converted to IV >= N+1.

-
  ```
  #ip_bit(in X:U,out GR:bval[ipcst[T,U]])
  #ip_bit(out X:U,in GR:bval[ipcst[T,U]])
  ```
  Find

Binds GR to the formula for the opaque boolean variable X.

- `#ip_sat(in G:bval[ipcst[T,U]]` **Find**

Holds if G is satisfiable.

-
  ```
  #ip_minimize(in G:bval[ipcst[T,U]],in IV:ipval[T],
               out NR:float)
  ```
  Find

Holds if G is satisfiable, binding NR to the minimum value of IV when G is satisfiable.

-
  ```
  #ip_maximize(in G:bval[ipcst[T,U]],in IV:ipval[T],
               out NR:float)
  ```
  Find

Holds if G is satisfiable, binding NR to the maximum value of IV when G is satisfiable.

- `ipval_asn[T,U]` **Type**

The type of a satisfying assignment for a boolean formula over linear constraints over unconstrained floating point or integer variables of type T and opaque boolean variables of type U.

- `#ip_sat_asn(in G:bval[ipcst[T,U]],out IASN:ipval_asn[T,U])` **Find**

Holds if G is satisfiable, and binds IASN to a satisfying assignment of values to each linear variable in G.

- `#ip_minimize_asn(in G:bval[ipcst[T,U]],in IV:ipval[T],`
  `out NR:float,`
  `out IASN:ipval_asn[T,U])` **Find**

Holds if `G` is satisfiable, binding `NR` to the minimum value of `IV` when `G` is satisfiable, and `IASN` to a satisfying linear variable assignment minimizing `IV`.

- `#ip_maximize_asn(in G:bval[ipcst[T,U]],in IV:ipval[T],`
  `out NR:float,`
  `out IASN:ipval_asn[T,U])` **Find**

Holds if `G` is satisfiable, binding `NR` to the maximum value of `IV` when `G` is satisfiable, and `IASN` to a satisfying linear variable assignment maximizing `IV`.

- `#ip_asn(in IASN:ipval_asn[T,U],in IV:ipval[T],`
  `out NR:float)` **Find**

Binds `NR` to the value of formula `IV` under linear variable assignment `IASN`. Variables in `IV` not mentioned in the formula used to create `IASN` are treated as zero.

- `#ip_asn_g(in IASN:ipval_asn[T],in G:bval[ipcst[T]])` **Find**

Holds if `G` is true under linear variable assignment `IASN`. Variables in `IV` not mentioned in the formula used to create `IASN` are treated as zero.

- `#ip_asn_var(in IASN:ipval_asn[T,U],out XR:T,out NR:float)` **Find**

Binds `XR` and `NR` to any variable-value linear variable pair in assignment `IASN`. Will only be generated for variables actually relevant to the test used to generate `IASN`.

- `#ip_asn_bit(in IASN:ipval_asn[T,U],out XR:U,out BR:bool)` **Find**

Binds `XR` and `NR` to any variable-value opaque boolean variable pair in assignment `IASN`. Will only be generated for boolean variables actually relevant to the test used to generate `IASN`.

- `#ip_asn_print(in IASN:ipval_asn[T,U],...)` **Add**

Adding will print all relevant variable-value pairs in the specified assignment, prefixed with any extra arguments (of which there may be any number).

- `#ip_pretty_ipval(in IV:ipval[string],`
  `out SR:string)` **Find**

Converts a linear formula over strings `IV` to a string `SR`, cleaning up the result as much as possible.

- `#ip_pretty_bval(in G:bval[ipcst[string,string]],`
  `out SR:string)` **Find**

Converts a boolean formula over linear constraints over strings `G` to a string `SR`, cleaning up the result as much as possible.

8.11 CIL Translation

The package translatecil encodes the CIL representation of C language constructs in a format suitable for use in logic programs. This consists of several components:

- Abstract types representing instances of CIL constructs (types, expressions, etc.). For example, `c_exp` is the type of CIL expressions.
- Factory predicates for constructing new instances of the abstract types. For example, `make_cil_exp(X,Y,E)` gets a `c_exp` `E` unique to `X` and `Y`.
- Enumerated sum-types for certain CIL types that can take on only a finite set of values. For example, `binop` is the type of a binary operation.
- Additional syntax predicates for encoding the CIL syntax tree itself. These relate instances of the abstract types with each other and with constant strings, integers, etc. For example, `cil_exp_cast(E:c_exp,CE:c_exp,T:c_type)` indicates that `E` is the result of casting expression `CE` to type `T`.
- Session functions for storing the syntax predicates generated during parsing. A typical C program is far too large to store all syntax in a single session. Each instance of a session function stores all syntax related to a particular C symbol. For example, `cil_func('foo')` stores the syntax for function foo's body, while `cil_comp('str')` stores the syntax for all fields in composite type (i.e. struct or union) str.

We go over each of these components in turn and give a brief description of all defined symbols. For a more comprehensive overview of CIL syntax and what these values mean, please consult the CIL documentation.

8.11.1 Abstract Types

All the CIL constructs represented by abstract types are given by the following table. The 'CIL Type' column gives the name of the corresponding construct as defined in `cil/src/cil.ml`

Type	CIL Type	Description
`c_type`	typ	C language type
`c_comp`	compinfo	C language composite struct/union
`c_field`	fieldinfo	Field of a struct/union
`c_enum`	enuminfo	C language enum
`c_var`	varinfo	Local or global variable
`c_init`	init	Static initializer for a global variable
`c_exp`	exp	C language expression
`c_const`	constant	C language constant value (5,'foo',etc.)
`c_lval`	lval	C language l-value
`c_offset`	offset	Series of field/array accesses in an l-value
`c_fundec`	fundec	Function body definition
`c_block`	block	Series of C statements
`c_stmt`	stmtkind	C language statement (if,while,return,etc.)
`c_instr`	instr	C language instruction (assignment,call)
`c_attr`	attribute	A C or GCC-extension attribute
`c_attr_arg`	attrparam	An attribute argument
`c_macro`	macro	A C macro expansion

8.11.2 Factory Predicates

All factory predicates are have the form `cil_make_*(X,Y,V)` where `X` and `Y` are values unique to the value `V` bound by the predicate. If `X` and `Y` are the same, then the same `V` will always result. All factory predicates are 'Find' predicates with the first two arguments as `in` and the last as `out`.

Predicate	Description
`make_cil_type(X:XT,Y:YT,V:c_type)`	Make a `c_type` `V` from `X` and `Y`
`make_cil_comp(X:XT,Y:YT,V:c_comp)`	Make a `c_comp` `V` from `X` and `Y`
`make_cil_field(X:XT,Y:YT,V:c_field)`	Make a `c_field` `V` from `X` and `Y`
`make_cil_enum(X:XT,Y:YT,V:c_enum)`	Make a `c_enum` `V` from `X` and `Y`
`make_cil_var(X:XT,Y:YT,V:c_var)`	Make a `c_var` `V` from `X` and `Y`
`make_cil_init(X:XT,Y:YT,V:c_init)`	Make a `c_init` `V` from `X` and `Y`
`make_cil_exp(X:XT,Y:YT,V:c_exp)`	Make a `c_exp` `V` from `X` and `Y`
`make_cil_const(X:XT,Y:YT,V:c_const)`	Make a `c_const` `V` from `X` and `Y`
`make_cil_lval(X:XT,Y:YT,V:c_lval)`	Make a `c_lval` `V` from `X` and `Y`
`make_cil_offset(X:XT,Y:YT,V:c_offset)`	Make a `c_offset` `V` from `X` and `Y`
`make_cil_fundec(X:XT,Y:YT,V:c_fundec)`	Make a `c_fundec` `V` from `X` and `Y`
`make_cil_block(X:XT,Y:YT,V:c_block)`	Make a `c_block` `V` from `X` and `Y`
`make_cil_stmt(X:XT,Y:YT,V:c_stmt)`	Make a `c_stmt` `V` from `X` and `Y`
`make_cil_instr(X:XT,Y:YT,V:c_instr)`	Make a `c_instr` `V` from `X` and `Y`
`make_cil_attr(X:XT,Y:YT,V:c_attr)`	Make a `c_attr` `V` from `X` and `Y`
`make_cil_attr_arg(X:XT,Y:YT,V:c_attr_arg)`	Make a `c_attr_arg` `V` from `X` and `Y`
`make_cil_macro(X:XT,Y:YT,V:c_macro)`	Make a `c_macro` `V` from `X` and `Y`

8.11.3 Enumerated Types

The following tables give the values and meaning of each enumerated type.

Type `ikind` corresponds to the CIL `ikind` representing different types of integers, and takes on the following values:

Value	CIL Value	C type
`ichar`	IChar	char
`ischar`	ISChar	signed char
`iuchar`	IUChar	unsigned char
`iint`	IInt	int
`iuint`	IUInt	unsigned int
`ishort`	IShort	short
`iushort`	IUShort	unsigned short
`ilong`	ILong	long
`iulong`	IULong	unsigned long
`ilonglong`	ILongLong	long long
`iulonglong`	IULongLong	unsigned long long

Type `fkind` corresponds to the CIL `fkind` representing different types of floating point values, and takes on the following values:

Value	CIL Value	C type
`ffloat`	FFloat	float
`fdouble`	FDouble	double
`flongdouble`	FLongDouble	long double

Type `unop` corresponds to the CIL `unop` representing different unary operations, and takes on the following values:

Value	CIL Value	Operation
`u_neg`	Neg	unary minus
`u_bnot`	BNot	bitwise not
`u_lnot`	LNot	logical not

Type `binop` corresponds to the CIL `binop` representing different binary operations, and takes on the following values:

Value	CIL Value	Operation
`b_plusa`	PlusA	integer + integer
`b_pluspi`	PlusPI	pointer + integer
`b_indexpi`	IndexPI	pointer + integer
`b_minusa`	MinusA	integer - integer
`b_minuspi`	MinusPI	pointer - integer
`b_minuspp`	MinusPP	pointer - pointer
`b_mult`	Mult	multiplication
`b_div`	Div	division
`b_mod`	Mod	modulus
`b_shiftlt`	Shiftlt	left shift
`b_shiftrt`	Shiftrt	right shift
`b_lt`	Lt	less than compare
`b_gt`	Gt	greater than compare
`b_le`	Le	less than or equal compare
`b_ge`	Ge	greater than or equal compare
`b_eq`	Eq	equal compare
`b_ne`	Ne	not equal compare
`b_band`	BAnd	bitwise and
`b_bxor`	BXor	bitwise exclusive-or
`b_bor`	BOr	bitwise or
`b_land`	LAnd	logical and
`b_lor`	LOr	logical or

8.11.4 Syntax Predicates Overview

Since the syntax predicates mimic the original CIL datatype structure, they principally encode parent-child relations We then divide syntax predicates according to the abstract type of the 'parent' that they describe. Note however that these predicates can be used in any fashion within logic programs, for either adding or finding, and in the later case with any or all arguments variable (allowing for simple searching or traversing of the syntax tree).

8.11.5 `c_type` Syntax Predicates

- `cil_type_void(T:c_type)` : Void type.
- `cil_type_int(T:c_type,K:ikind)` : Int type with kind `K`.
- `cil_type_float(T:c_type,K:fkind)` : Float type with kind `K`.
- `cil_type_ptr(T:c_type,DT:c_type)` : Pointer type to type `DT`.
- `cil_type_array(T:c_type,ET:c_type,E:c_exp)` : Array type with element type `ET` and length `E`.
- `cil_type_func(T:c_type,RT:c_type,VARGS:bool)` : Function type with return type `RT` and `VARGS` indicating whether the function is varargs.

- `cil_type_func_arg(T:c_type,A:int,AS:string,AT:c_type)` : Single argument `A` of a function type, with name `AS` and type `AT`.

- `cil_type_func_arg_attr(T:c_type,A:int,ATTR:c_attr)` : Single function type argument attribute.

- `cil_type_named(T:c_type,NAME:string,NT:c_type)` : Named alias type `NAME` for type `NT`.

- `cil_type_comp(T:c_type,COMP:string)` : Type for composite struct/union named `COMP`.

- `cil_type_enum(T:c_type,ENUM;string)` : Type for enum named `ENUM`.

- `cil_type_valist(T:c_type)` : Special valist type.

- `cil_type_x_attr(T:c_type,ATTR:c_attr)` : Single type attribute.

- `cil_type_x_bytes(T:c_type,N:int)` : Type width is `N` bytes. Not present for types that don't have widths (such as function types).

8.11.6 `c_comp` Syntax Predicates

- `cil_comp_name(C:c_comp,NAME:string,STRUCT:bool)` : Composite with name `NAME`. if `STRUCT` is `true`, the type is a struct, while if `STRUCT` is false, the type is a union.

- `cil_comp_field(C:c_comp,POS:int,F:c_field)` : Single field `F` of a composite, with index into the composite `POS`.

- `cil_comp_attr(C:c_comp,ATTR:c_attr)` : Single composite attribute.

- `cil_comp_bytes(C:c_comp,N:int)` : Width of this composite is `N` bytes.

- `cil_comp_location(C:c_comp,FILE:string,LINE:int)` : Source file location of the beginning (opening 'struct') of a composite's definition.

- `cil_comp_end_location(C:c_comp,FILE:string,LINE:int)` : Source file location of the ending (closing '}') of a composite's definition.

8.11.7 `c_field` Syntax Predicates

- `cil_field_name(F:c_field,NAME:string,T:c_type,BITS:int,BITW:int)` : Field with name `NAME`, type `T`, which starts at bit position `BITS` into its parent structure, and has bit width `BITW`.

- `cil_field_attr(F:c_field,ATTR:c_attr)` : Single field attribute.

8.11.8 `c_enum` Syntax Predicates

- `cil_enum_name(E:c_enum,NAME:string)` : Enum with name `NAME`.
- `cil_enum_item(E:c_enum,POS:int,S:string,E:c_exp)` : Single item `S` of an enum, with value given by `E`.
- `cil_enum_attr(E:c_enum,ATTR:c_attr)` : Single enum attribute.
- `cil_enum_location(C:c_enum,FILE:string,LINE:int)` : Source file location of the beginning (opening 'enum') of an enum's definition.
- `cil_enum_end_location(C:c_enum,FILE:string,LINE:int)` : Source file location of the ending (closing '}') of an enum's definition.

8.11.9 `c_var` Syntax Predicates

- `cil_var_name(X:c_var,NAME:string,T:c_type)` : Variable with name `NAME` and type `T`.
- `cil_var_global(X:c_var)` : Flag for global variables.
- `cil_var_inline(X:c_var)` : Flag for inline function variables.
- `cil_var_static(X:c_var)` : Flag for static variables.
- `cil_var_register(X:c_var)` : Flag for register variables.
- `cil_var_init(X:c_var,I:c_init)` : Static initializer for global variables.
- `cil_var_attr(X:c_var,ATTR:c_attr)` : Single variable attribute.
- `cil_var_location(X:c_var,FILE:string,LINE:int)` : Source file location of a variable definition. Only present for global variables.

8.11.10 `c_init` Syntax Predicates

- `cil_init_single(I:c_init,E:c_exp)` : Single initializer with value `E`.
- `cil_init_cmpnd_type(I:c_init,T:c_type)` : Compound initializer, initializing a composite or array type `T`.
- `cil_init_cmpnd_field(I:c_init,F:c_field,FI:c_init)` : Part of a compound initializer, initializing field `F` with `FI`.
- `cil_init_cmpnd_index(I:c_init,E:c_exp,EI:c_init)` : Part of a compound initializer, initializing index `E` of an array with `EI`.

8.11.11 c_exp Syntax Predicates

- cil_exp_const(E:c_exp,C:c_const) : Constant expression.
- cil_exp_lval(E:c_exp,LV:c_lval) : L-value expression, value is that held by LV at the point the expression is evaluated.
- cil_exp_sizeof(E:c_exp,ST:c_type) : Sizeof a type.
- cil_exp_sizeofe(E:c_exp,SE:c_exp) : Sizeof an expression type.
- cil_exp_sizeofstr(E:c_exp,SS:string) : Sizeof a string.
- cil_exp_alignof(E:c_exp,AT:c_type) : Alignof a type.
- cil_exp_alignofe(E:c_exp,AE:c_exp) : Alignof an expression type.
- cil_exp_unop(E:c_exp,OP:unop,RE:c_exp,T:c_type) : Unary operation OP on RE, with result type T.
- cil_exp_binop(E:c_exp,OP:binop,LE:c_exp,RE:c_exp,T:c_type) : Binary operation OP on LE and RE, with result type T.
- cil_exp_cast(E:c_exp,CE:c_exp,T:c_type) : Cast expression of CE into type T.
- cil_exp_addr(E:c_exp,LV:c_lval) : Address-of expression, get the address of LV.
- cil_exp_start(E:c_exp,LV:c_lval) : Implicit address-of expression used for arrays, get the address of LV[0].
- cil_exp_x_intval(E:c_exp,N:int) : Expression constant-folds to integer value N. Not present for expressions that don't fold to a constant value.
- cil_exp_x_location(E:c_exp,FILE:string,BLINE:int,ELINE:int,BBYTE:int,EBYTE:int) : Source file location of expression E, between lines BLINE ELINE, and file byte ranges BBYTE and EBYTE. Note that these ranges are for the preprocessed code.
- cil_exp_x_macro(E:c_exp,M:c_macro) : Expression is contained within macro expansion M. Only available if the modified macro-generating GCC was used for compilation.

8.11.12 c_const Syntax Predicates

- cil_const_int(C:c_const,K:ikind,N:int) : Integer N with kind K.
- cil_const_str(C:c_const,S:string) : ANSI string S.
- cil_const_wstr(C:c_const,LEN:int) : Wide string with LEN characters.
- cil_const_char(C:c_const,N:int) : Character with value N.
- cil_const_real(C:c_const,K:fkind,N:float) : Float N with kind K.

8.11.13 `c_lval` Syntax Predicates

- `cil_lval_var(LV:c_lval,X:c_var,OFF:c_offset)` : Offset `OFF` into a local or global variable `X`.
- `cil_lval_mem(LV:c_lval,ME:c_exp,OFF:c_offset)` : Offset `OFF` into the target of a pointer expression `ME`.

8.11.14 `c_offset` Syntax Predicates

- `cil_off_none(OFF:c_offset)` : Empty offset.
- `cil_off_field(OFF:c_offset,F:c_field,NOFF:c_offset)` : Offset given by taking offset `NOFF` of field `F` of this composite.
- `cil_off_index(OFF:c_offset,E:c_exp,NOFF:c_offset)` : Offset given by taking offset `NOFF` of index `E` of this array.

8.11.15 `c_fundec` Syntax Predicates

As every function body is declared in a separate session, there will be at most one function definition. This allows the special predicate `cil_curfn` to identify this function's name.

- `cil_curfn(FNAME:string)` : `FNAME` is the name of the currently analyzed function, provided that the current session is indeed a function body.
- `cil_fundec_name(FN:c_fundec,NAMEX:c_var,BODY:c_block)` : Function definition with function variable `NAMEX` and body `BODY`.
- `cil_fundec_formal(FN:c_fundec,A:int,AX:c_var)` : Single formal parameter `A` of a function definition, accessed via variable `AX`.
- `cil_fundec_local(FN:c_fundec,X:c_var)` : Single local variable `X` of a function definition.
- `cil_fundec_location(FN:c_fundec,FILE:string,LINE:int)` : Source file location of the beginning (function symbol) of a function's definition.
- `cil_fundec_end_location(FN:c_fundec,FILE:string,LINE:int)` : Source file location of the ending (closing '}') of a function's definition.

8.11.16 `c_block` Syntax Predicates

- `cil_block_stmts(B:c_block,STMTS:list[c_stmt])` : Block for a sequence of statements `STMTS`.
- `cil_block_attr(B:c_block,ATTR:c_attr)` : Single block attribute.

8.11.17 `c_stmt` Syntax Predicates

An extra sum-type `cases` is used by switch statements to encode a series of case and default labels.

```
cases ::= case{E:c_exp,S:c_stmt}
        | default{S:c_stmt}
```

Each `cases` is a single case in a switch statement, either a test/target pair or just the target for a default label.

The statement predicates are as follows:

- `cil_stmt_instrs(S:c_stmt,INSTS:list[c_instr])` : Sequence of individual instructions.
- `cil_stmt_return(S:c_stmt)` : Any return statement.
- `cil_stmt_return_exp(S:c_stmt,E:c_exp)` : Expression `E` is returned by the statement.
- `cil_stmt_goto(S:c_stmt,TGT:c_stmt)` : Goto statement with target `TGT`.
- `cil_stmt_break(S:c_stmt)` : Break statement.
- `cil_stmt_continue(S:c_stmt)` : Continue statement.
- `cil_stmt_if(S:c_stmt,E:c_exp,TB:c_block,FB:c_block)` : If statement testing `E` and branching to `TB` if the result is non-zero, or to `FB` if the result is zero.
- `cil_stmt_switch(S:c_stmt,E:c_exp,B:c_block,CASES:list[c_cases])` : Switch statement with body `B` and individual cases `CASES`.
- `cil_stmt_loop(S:c_stmt,B:c_block)` : Loop statement executing block `B` repeatedly. Any control flow exiting the loop will be given by interior break, continue, or goto statements.
- `cil_stmt_block(S:c_stmt,B:c_block)` : Nested block of statements.
- `cil_stmt_tryfinally(S:c_stmt)` : Try-finally statement, MSVC-only and contents are not currently available.
- `cil_stmt_tryexcept(S:c_stmt)` : Try-except statement, MSVC-only and contents are not currently available.
- `cil_stmt_x_location(S:c_stmt,FILE:string,LINE:int)` : Source file location of any statement, always present but sometimes with an unknown location.

8.11.18 c_instr Syntax Predicates

- cil_instr_set(I:c_instr,LV:c_lval,E:c_exp) : Assignment instruction, storing the value of expression E in the location given by LV.

- cil_instr_call(I:c_instr,FNE:c_exp) : Call instruction to function expression FNE.

- cil_instr_call_ret(I:c_instr,RLV:c_lval) : Return l-value for a call instruction which assigns its return value somewhere.

- cil_instr_call_arg(I:c_instr,A:int,AE:c_exp) : Single argument number A of a call, with value AE. Arguments are numbered starting from 0.

- cil_instr_asm(I:c_instr,TEMPLATES:list[string]) : Assembly instruction executing a sequence of individual assembly templates TEMPLATES.

- cil_instr_asm_out(I:c_instr,N:int,L:maybe[string],X:string,LV:c_lval) : Output l-value for an assembly instruction. The Nth argument to the assembly instruction is an output which assigns the value of location template X to LV. Within the assembler code this location may be referred to using the optional symbolic name L.

- cil_instr_asm_in(I:c_instr,N:int,L:maybe[string],X:string,E:c_exp) : Input expression for an assembly instruction. The Nth argument to the assembly instruction is an input which takes the value of E in location template X. Within the assembler code this location may be referred to using the optional symbolic name L.

- cil_instr_asm_clobber(I:c_instr,X:string) : Single register value X clobbered by an assembly instruction.

- cil_instr_asm_attr(I:c_instr,ATTR:c_attr) : Single assembly instruction attribute (const or volatile).

- cil_instr_x_location(I:c_instr,FILE:string,LINE:int) : Source file location of any instruction, always present but sometimes with an unknown location.

8.11.19 c_attr Syntax Predicates

- cil_attr_name(A:c_attr,NAME:string) : Attribute with name NAME.

- cil_attr_arg(A:c_attr,P:int,PARG:c_attr_arg) : Single argument A of an attribute, with value PARG.

8.11.20 `c_attr_arg` Syntax Predicates

- `cil_attr_arg_int(A:c_attr_arg,N:int)` : Constant integer argument.
- `cil_attr_arg_str(A:c_attr_arg,S:string)` : Constant string argument.
- `cil_attr_arg_cons(A:c_attr_arg,CONS:c_attr)` : Constructed argument from another attribute.
- `cil_attr_arg_sizeof(A:c_attr_arg,ST:c_type)` : Sizeof type argument.
- `cil_attr_arg_sizeofe(A:c_attr_arg,SA:c_attr_arg)` : Sizeof argument.
- `cil_attr_arg_alignof(A:c_attr_arg,AT:c_type)` : Alignof type argument.
- `cil_attr_arg_alignofe(A:c_attr_arg,AA:c_attr_arg)` : Alignof argument.
- `cil_attr_arg_unop(A:c_attr_arg,OP:unop,RA:c_attr_arg)` : Unary operation argument.
- `cil_attr_arg_binop(A:c_attr_arg,OP:binop,LA:c_attr_arg,RA:c_attr_arg)` : Binary operation argument.
- `cil_attr_arg_dot(A:c_attr_arg,FA:c_attr_arg,F:string)` : 'Field' access argument.

8.11.21 `c_macro` Syntax Predicates

- `cil_macro_builtin(M:c_macro)` : `M` is a built-in macro, such as `sizeof`.
- `cil_macro_ident(M:c_macro,TEXT:string)` : `M` is an identifier macro, such as `#define SIZE 3`. `TEXT` is the string version of the macro.
- `cil_macro_func(M:c_macro,TEXT:string)` : `M` is a function macro, such as `#define MAX(a,b) a<b?b:c`. `TEXT` is the string version of the macro.
- `cil_macro_func_arg(M:c_macro,A:int,FORMAL:string,ACTUAL:string)`: Argument `A` to function macro `M` has formal string `FORMAL` and actual argument string `ACTUAL`. Arguments are numbered starting from zero.
- `cil_macro_x_name(M:c_macro,NAME:string)` : The (human-readable) name of a macro.
- `cil_macro_x_location(M:c_macro,FILE:string,LINE:int,BBYTE:int,EBYTE:int)` : Source file location of a macro expansion, with file start and end bytes.
- `cil_macro_x_parent(M:c_macro,PM:c_macro)` : The expansion of `M` is contained within the expansion of `PM`.

8.11.22 Translation Sessions

Each session stores all syntax predicates associated with a particular global symbol. Additional syntax for another symbol not included in the current session (for example, a global variable accessed in a callee of the currently analyzed function) can be accessed by either explicitly using the session name, or by using `merge_preds` with that session and then accessing the syntax as normal.

- `cil_body(FUNC:string)` : The type and definition of the function `FUNC`. Includes all statements/expressions/etc. in the function, as well as type information for all variables and fields explicitly accessed. Does **NOT** include information for globals accessed only by callees, or composite type and field information beyond that explicitly accessed.
- `cil_glob(GLOB:string)` : The type of global variable `GLOB`.
- `cil_init(GLOB:string)` : Any static initializer for global variable `GLOB`. This is kept separately from `cil_glob` because typically only the global's type is needed, and the initialization information can be quite huge (outgrowing even the total size of the function definitions in a program).
- `cil_comp(COMP:string)` : The type and field information for the composite struct/union `COMP`.
- `cil_enum(ENUM:string)` : The item information for the enum `ENUM`.

8.12 Graphviz Visualization

Package dotty provides predicates for generating graphviz description files for use by such programs as `dot` and `dotty`. More information on graphviz can be found at http://www.graphviz.org.

- `dotgraph` **Type**

The type of a dotty graph.

- `dotty_graph(NAME:string,DIRECTED:bool,GID:dotgraph)` **Find**

Gets the graph `GID` associated with `NAME`. If no graph has been associated with `NAME`, creates a new graph which is either directed or undirected according to the flag `DIRECTED`. This graph and all nodes and edges added to it will be written out to the file '`NAME.dot`'.

- `dotty_node(GID:dotgraph,VNAME:_)` **Add**

Adds to graph `GID` a node with name `VNAME`.

- `dotty_edge(GID:dotgraph,ENAME:_,SNAME:_,TNAME:_)` **Add**

Adds to graph `GID` an edge with name `ENAME` going from `SNAME` to `TNAME`. Adds nodes `SNAME` and `TNAME` if they have not been yet been added.

- `dotty_attr(GID:string,NAME:_,ATTR:string,VS:string)` **Add**

Adds to graph `GID` an attribute `ATTR` with string value `VS` on the node or edge with name `NAME`. A few special names may be used to specify attributes of other items:

- graph : Specifies an attribute of the graph as a whole.
- node : Specifies a default attribute for every node.
- edge : Specifies a default attribute for every edge.

A full list of attribute names and possible values is available at http://www.graphviz.org. Some of the more useful ones are below:

- color : The color of an edge or node.
- label : The label attached to an edge, or the name of a node.
- shape : The shape of a node. Examples values include 'box', 'ellipse', 'circle', 'point', and 'plaintext'.
- style : The style of an edge or node. Examples values include 'dashed', 'dotted', 'solid', 'bold', and 'filled'.

8.13 User Interface Generation

Package display provides predicates for generating displays for use by the user interface (Section 7.5). All displays will be stored (as compressed XML) in file 'display.db', and all generated search terms will be stored in file 'search.db'. These databases will typically be consumed by the UI itself, but the `dbkeys` and `dbfind` utility apps may also be used to form queries. The XML schema used for displays is given in Appendix B.

- `display_add(DISPLAY:string,REPLACE:bool,CATEGORY:string,TITLE:string,`
  `FUNCTION:string,FILE:string,MINLINE:int,MAXLINE:int,`
  `FOCUSLINE:int,DITEMS:list[displayitem])` **Add**

Generates a display with the globally unique identifier `DISPLAY`, with `REPLACE` indicating whether this display should replace any existing display with that name, or instead be dropped. `CATEGORY` describes the general class of displays this should be bucketed with, while `TITLE` gives the user-readable title of the display. `FUNCTION` gives the name of the source language function the display concerns, with `FILE`, `MINLINE`, and `MAXLINE` giving the piece of the source code to render. The initial focus in the rendered page will be on line FOCUSLINE, with DITEMS specifying the actual highlighting and text to render alongside the code.

- `displayitem ::= d_line_style{LINE:int,CSSCLASS:string}`
  `| d_line_text{LINE:int,CSSCLASS:string,`
  `POSITION:displayposition,TEXT:string,`
  `LINKS:list[displaylink]}` **Type**

Each `displayitem` modifies the way the code is rendered by the UI.

A `d_line_style` specifies a new `CSSCLASS` to use in rendering the specified line. `CSSCLASS` must be defined in whatever 'file.css' is used for rendering the displays, and will typically contain foreground or background coloring information.

A `d_line_text` specifies `TEXT` and `LINKS` to other displays to be added when rendering the specified line. `CSSCLASS` is the file.css class used when rendering the text, and `POSITION` specifies where in relation to the line of code to render the text.

- `displayposition ::= d_left | d_right | d_bottom | d_top` **Type**

A position relative to a line of code to render a `d_line_text` item.

- `displaylink ::= d_link{DISPLAY:string,TEXT:string}` **Type**

A clickable link to a display. The display `DISPLAY` should have been created separately (possibly during analysis of another function) using the `display_add` predicate. The UI will render a link with text `TEXT` which, when clicked by the user, will direct them to the rendered `DISPLAY`.

- `search_add(TERM:string,DISPLAY:string)` **Add**

Generates a search term `TERM` mapped to display `DISPLAY`. When doing a search within the UI, search term `TERM` will generate a list including `DISPLAY` as well as any other displays associated with `TERM`.

Appendix A

Tutorial Locking Analysis

This appendix includes the full locking analysis developed during the CLP tutorial (Section 3).

```
% interprocedural locking analysis.

import "../memory/scalar_sat.clp".

analyze session_name("cil_body").

% PREDICATES

% the possible spinlock states
type lockstate ::= locked | unlocked | error.

% at program point P, trace T is in state S if G holds
predicate state(P:pp,T:t_trace,S:lockstate,G:g_guard).

% Summary edge on a call: if callee trace CT is in state SIN, it transitions
% to state SOUT.
predicate cedge(I:c_instr,T:t_trace,SIN:lockstate,SOUT:lockstate).

% extra unconstrained bits, true if T is locked at entry
type g_xrep ::= t_locked{t_trace}.

% SUMMARIES

% session with the lockstate summary of FN
session sum_locking(FN:string) containing [sedge].

% summary edge: if trace T is in state SIN, it transitions to state SOUT
predicate sedge(T:t_trace,SIN:lockstate,SOUT:lockstate).
```

```
% RULES

% call transitions

% add transitions for direct calls to lock/unlock/trylock
dircall(I,"lock"),
    +cedge(I,drf{root{arg{0}}},locked,error),
    +cedge(I,drf{root{arg{0}}},unlocked,locked).
dircall(I,"unlock"),
    +cedge(I,drf{root{arg{0}}},locked,unlocked),
    +cedge(I,drf{root{arg{0}}},unlocked,error).

% add additional transitions according to summary information generated
% on the targets of direct calls
dircall(I,F), sum_locking(F)->sedge(T,SIN,SOUT), +cedge(I,T,SIN,SOUT).

% transfer functions

% adding will merge G into the condition under which T is in state S at P
predicate smerge(P:pp,T:t_trace,S:lockstate,G:g_guard).
smerge(P,T,S,_), pp_order(P,DEP,DIST), $order(DEP,DIST,0),
    \/smerge(P,T,S,G):#or_all(G,MG), +state(P,T,S,MG).

% LKG and UKG are conditions under which T is locked or unlocked at entry
predicate entry_locked(in T:t_trace,LKG:g_guard,UKG:g_guard).
?entry_locked(T,_,_), #id_g(br_abit{ar_extra{t_locked{T}}},LKG), #not(LKG,UKG),
    +entry_locked(T,LKG,UKG).

% compute the initial lock states for any trace T whose lock state might
% change during the function's execution (there is a call with a transition
% on T). generate a boolean variable indicating whether T is locked (LKG) or
% unlocked (UKG) at entry, and add the two states
entry(PIN), icall(P0,_,I), cedge(I,CT,_,_),
    inst_trace(s_call{I},P0,CT,trace{T},_), entry_locked(T,LKG,UKG),
    +state(PIN,T,locked,LKG), +state(PIN,T,unlocked,UKG).

% sets and branches don't affect lockstate
iset(P0,P1,_), state(P0,T,S,G), eguard(P0,P1,G,EG), +smerge(P1,T,S,EG).
branch(P,P0,_,_), state(P,T,S,G), eguard(P,P0,G,EG0), +smerge(P0,T,S,EG0).
branch(P,_,P1,_), state(P,T,S,G), eguard(P,P1,G,EG1), +smerge(P1,T,S,EG1).

% call transfer function for locks where a transition is known
icall(P0,P1,I), cedge(I,CT,SIN,SOUT),
    inst_trace(s_call{I},P0,CT,trace{T},BG),
    state(P0,T,SIN,SG), #and(BG,SG,G),
    eguard(P0,P1,G,EG), +smerge(P1,T,SOUT,EG).

% for locks which do not have known transitions on the callee, we treat the
% call as a nop. since there may be multiple aliases of the lock, we need to
% get the conjunction over the negations of all conditions under which some
% alias for the lock does NOT have a transition on the callee

% NG is a negated condition for when T has a transition on the call at P.
% we assume that if there are any transitions, there will be transitions for
% all combinations of SIN (locked/unlocked only)
predicate edge_negate(P:pp,T:t_trace,NG:g_guard).
icall(P,_,I), cedge(I,CT,_,_),
    inst_trace(s_call{I},P,CT,trace{T},G),
    #not(G,NG), +edge_negate(P,T,NG).

% call transfer function for locks which do NOT have callee transitions.
% get the conjunction on the negated conditions and propagate forward unchanged
icall(P0,P1,_), state(P0,T,S,SG),
    pp_order(P0,DEP,DIST), $order(DEP,DIST,1),
    \/edge_negate(P0,T,NG):#and_all(NG,MNG), #and(MNG,SG,G),
    eguard(P0,P1,G,EG), +smerge(P1,T,S,EG).

% call transfer function for locks in the error state
icall(P0,P1,_), state(P0,T,error,G),
    eguard(P0,P1,G,EG), +smerge(P1,T,error,EG).

% summary computation

% for trace T, the SIN -> SOUT transition occurs when G holds
predicate trace_trans(T:t_trace,G:g_guard,SIN:lockstate,SOUT:lockstate).

% compute the transition conditions for each trace T. get the lock state
% at exit and combine with the boolean variable indicating whether the lock
% was locked (LKG) or unlocked (UKG) at entry
exit(P), state(P,T,S,SG), entry_locked(T,LKG,UKG),
    #and(SG,LKG,LKGG), +trace_trans(T,LKGG,locked,S),
    #and(SG,UKG,UKGG), +trace_trans(T,UKGG,unlocked,S).

% function FN has summary edge A/SIN/SOUT
predicate fedge(FN:string,T:t_trace,SIN:lockstate,SOUT:lockstate).

% combine the transition conditions with the return-non-zero conditions
% to compute all the summary transitions on the current function
cil_curfn(F), trace_trans(T,SG,SIN,SOUT), guard_sat(SG),
    +fedge(F,T,SIN,SOUT), +sum_locking(F)->sedge(T,SIN,SOUT).

% print out the results
?- fedge(FN,A,SIN,SOUT).
```

Appendix B

UI Display Schema

This appendix includes the XML schema for the displays rendered by the UI (Section 7.5). Running dbfind on a 'display.db' file produced by an analysis produces an XML snippet following this schema (e.g. run the aliasing or null analysis for examples). While Saturn analyses do not have to worry about the form of this schema as it is generated transparently by the display package (Section 8.13), other tools wishing to use the UI simply need to generate XML of this form accessible via a command or script with the same interface as dbfind.

```
<!-- each "display" is a single view into the code base,
     and can have any number of highlighted lines or additional text -->
<element name="display" minOccurs="1" maxOccurs="1">
  <complexType>
    <sequence>

      <!-- unique name for this display, not shown on screen -->
      <element name="name" type="string"/>

      <!-- category and printed title for this display -->
      <element name="category" type="string"/>
      <element name="title" type="title"/>

      <!-- source code info, with min and max line #s to fetch -->
      <element name="function" type="string"/>
      <element name="file" type="string"/>
      <element name="minline" type="integer"/>
      <element name="maxline" type="integer"/>

      <!-- line to put initial viewer focus on -->
      <element name="focusline" type="integer"/>

      <!-- draw a particular line with a particular style -->
      <element name="linestyle" minOccurs="0" maxOccurs="unbounded">
        <complexType>
          <sequence>
            <element name="line" type="integer"/>
            <element name="cssclass" type="string"/>
          </sequence>
        </complexType>
      </element>

      <!-- draw some text around a particular line -->
      <element name="linetext" minOccurs="0" maxOccurs="unbounded">
        <complexType>
          <sequence>
            <element name="line" type="integer"/>
            <element name="cssclass" type="string"/>

            <!-- where to draw the text relative to the line itself -->
            <element name="position">
              <simpleType>
                <restriction base="string">
                  <enumeration value="left"/>
                  <enumeration value="right"/>
                  <enumeration value="top"/>
                  <enumeration value="bottom"/>
                </restriction>
              </simpleType>
            </element>

            <element name="text" type="string"/>

            <!-- clickable link to another named display -->
            <element name="link" minOccurs="0" maxOccurs="unbounded">
              <complexType>
                <sequence>
                  <element name="name" type="string"/>
                  <element name="text" type="string"/>
                </sequence>
              </complexType>
            </element>
          </sequence>
        </complexType>
      </element>
    </sequence>
  </complexType>
</element>
```