Shape Analysis Using Separation Logic

Tomáš Brablec

Abstract

This work describes the implementation of a static analysis of C programs, that tries to find bugs related to dynamically allocated memory. The analysis is implemented using the Frama-C framework, and it is done by dataflow analysis using a solver for separation logic. This work also explores possible improvements to this prototype.

1 Introduction

The way dynamic memory allocation is handled is one of the key aspects that determines the runtime performance of a language. The usual choice for high-level languages is garbage collection, which tracks allocations at runtime, and frees memory when an algorithm determines that it is no longer reachable. This has a performance impact that can be unacceptable in certain applications, such as embedded systems, operating system kernels, or real-time applications. Other languages, notably C and C++, let the user's code manage memory manually. While being the most performant option, this allows for a range of errors resulting from incorrect handling of allocated memory. For example, null pointer dereferences are a common mistake, where a fallible function indicates an error by returning a `NULL` pointer, and the programmer forgets to check for it. Use-after-free is a situation where a pointer to a freed allocation is used to read or write data to memory, resulting in undefined behavior. The opposite situation, where the user does not free memory when it is no longer needed, is called a memory leak. Another common error is double free, where a pointer to already freed allocation is passed to the `free` function again, resulting again in undefined behavior.

To mitigate these problems, a number of methods have been developed. One option is the instrumentation of program binaries with extra code that is able to detect invalid memory accesses at runtime, and safely stop the program. An example of this is Valgrind [1], or LLVM's AddressSanitizer [2]. Even though these methods are efficient at mitigating the security risks associated with memory errors, these tools come with a performance penalty, and obviously cannot be used to check programs ahead of time. Some languages are built to prove a program's memory safety statically during compilation; this is achieved by placing additional restrictions on the code, as, for example, Rust does with its ownership and borrowing rules. While being a reasonably safe option, this makes implementing some programs difficult, or requires writing unsafe code. Static analysis tools exist for other languages, including C, which will be the topic of this work. For example, the static analyzer Infer can perform analyses of memory allocations.

2 Frama-C framework

Frama-C [3] is a framework for building analysis tools of C99 source code. Frama-C itself is written mainly in the OCaml programming language. Unlike other tools, which focus on finding common bugs using heuristics, Frama-C specializes in verification tools, which guarantee that after successful completion, the program is correct. The framework itself is composed of a kernel, multiple plugins, and a GUI to present the results of analyses. The kernel provides common functionality for multiple plugins. The main component is an adapted form of the *C Intermediate Language* (CIL) [4], constructed by the Frama-C kernel for use within plugins, as well as an API for its manipulation. CIL has the form of an abstract syntax tree of the input source

code, with extra semantic information added to it. This includes types of variables, whether a variable is initialized at a certain node, and other information. Frama-C also transforms the input code, making operations like type casts explicit, and otherwise making the code more suitable for static analysis. For example, all `return` statements in a function are replaced with `goto` statements leading to a single `return` at the end of the function. A complete description of CIL can be found in the module `Frama_c_kernel.Cil_types` of the Frama-C API documentation [5]. The modules `Cil`, `Cil_datatype`, and `Ast_info` inside `Frama_c_kernel` contain other useful functions for adding content to the AST.

One of the functions of Frama-C is to help with the development of custom analyses. This is done using its plugin system, where a separate plugin binary is built and linked dynamically to the Frama-C runtime. Frama-C then handles command-line argument parsing, reporting results, storing intermediate states of analyses, and exposing APIs to the plugin for access to the input's CIL.

2.1 Dataflow analysis

Besides a general framework for crafting analyses, Frama-C provides generic implementations of common algorithms used for static analysis. One of these is dataflow analysis, implemented in module `Frama_c_kernel.Dataflow2`. Dataflow analysis works by assigning an initial value to each node of a Control Flow Graph (CFG), and then systematically updating the values of nodes using a transfer function, following the edges between them. When a node is reached for the second time, the data for this node is computed again based on the data from the previous node, and then joined with the previous data stored for the node. Typically, a CFG is a structure where a node represents a basic block – a sequence of instructions that will be executed in order, with no loops or conditionals inside. Edges then represent the control flow of a program – branches and loops.

However, Frama-C implements dataflow analysis in a slightly different way. In dataflow analysis as it is implemented in Frama-C, instructions (`Cil_types.instr`) have a separate transfer function from statements (`Cil_types.stmt`). Instruction is a kind of statement that does not change the control flow of the current function, such as variable definition, assignment, or function call. To implement a dataflow analysis, the user needs to provide the type of data that will be stored and updated for each node of the CFG, and the implementation of the following functions. Before running the analysis, the data for the first statement must be set manually.

- `computeFirstPredecessor` – this function is called when the analysis reaches a statement that has no data associated with it. The input for this instruction is the result of the previous statement's transfer function, and the statement itself. The result of this function is the initial state for the given statement, which will be stored. Note that because you set the initial state for the first statement manually, this function is not called for the first statement.
- `doStmt` – this function is used by the transfer function for statements – it gets the statement itself, and data provided by the previous statement's transfer function, and it decides what to do. One option is not to continue the analysis of this statement, another option (default) is to continue the analysis of the inside of this statement, and the third is to also continue, but with modified data. Note that the full transfer function for compound statements (if, loop, etc.) is implemented by the dataflow module itself, the user is not supposed to call `doInstr` inside `doStmt`. Also note that the result of the full transfer function – the new computed data for each statement, is not stored. It is only sent to the next statement's transfer function.
- `doInstr` – this is the transfer function for instructions, it receives data from the englobing statement, and generates new data that will be used by the transfer function of the englob-

ing statement. Note that the result of `doInstr` is also not stored, only sent to the transfer function that called `doInstr` internally.

- `combinePredecessors` – this is the join operation. It is called when the analysis computes a new state for a node that already has some data associated with it. This includes situations, where the analysis goes through two branches of an `if` statement, and then joins data from both branches on the statement immediately following the conditional block. Input for this function is the statement itself, an old state – the data currently stored for the statement, and a new state – the data that was just computed by the transfer function. The returned value is then stored as the data for this statement.
- `doGuard` – this function is called when an `if` statement is reached. It receives the result of the transfer function for the `if` statement, and the condition expression, and generates two states, each to be used in one of the branches.
- `doEdge` – called between analyzing two statements. The function receives both statements, and the result of the first statement's transfer function. The function can modify this data, and the modified version will be passed into the second statement's transfer function.

Note that this is not the full API required for implementing dataflow analysis in the `Dataflow2` module, but the other functions, such as pretty-printers for stored data, are not important to the actual analysis, and are therefore omitted.

2.2 Visitor mechanism

Frama-C provides a convenient way to modify the AST of the analyzed program using a user-provided visitor object, as described in [6]. The plugin constructs an object inheriting from a class inside Frama-C, and overrides some of the methods corresponding to the AST node that it visits. For example, when the plugin overrides the `vstmt` method, the method will be called at each statement of the AST, and the statement will be passed into the method as input. The method returns a value of type `Cil_types.visitAction`, which allows the visitor to either leave the node as is, change it, or continue with the visits of its children.

There are two kinds of visitors, `Cil.CilVisitor` and visitors derived from `Visitor.frama_c_visitor`. The former does not update the internal state of Frama-C when the AST is modified, and it is therefore unsuitable for making larger changes, such as adding or removing statements. The latter visitor is able to update the internal state of Frama-C after a pass over the AST, but this update must be done explicitly by the plugin. The way of doing this update was not obvious from the available documentation, so I mention it here explicitly:

```
Ast.mark_as_changed (); (* tell Frama-C that the AST was changed *)
Cfg.clearFileCFG file;
Cfg.computeFileCFG file; (* recompute the CFG, which is used for dataflow analysis *)
```

These lines must run after adding or removing a statement (`Cil_types.stmt`), or after changing the type of a statement, for example after changing an instruction to a block.

Frama-C also provides a different kind of visitor called a *copy* visitor, as opposed to the *inplace* visitor described above. The copy visitor does not have the same state synchronization problem as the inplace visitor, but it creates a new *project* inside Frama-C, instead of modifying the AST of the default, existing one. A project in Frama-C is a collection of all configuration and input data provided by the user. This includes the values of command-line arguments, the AST of the input program, the CFG that constructed internally by Frama-C, and other data. Switching into a new project resets all these values, making it impossible to access the command-line plugin configuration.

2.3 Ivette

Frama-C currently supports two GUI frameworks. There is a legacy GTK application called `frama-c-gui`, and a new Electron-based application called Ivette [7]. Both applications can be extended to show custom data, but in the case of `frama-c-gui`, this requires manually implementing new GTK widgets, which is inconvenient. Also, the codebase is considered unmaintainable by the authors and is intended to be replaced by Ivette.

Ivette is a desktop application written in TypeScript using a client-server architecture. Ivette, the client, asynchronously polls the server (Frama-C plugin) for data and displays it. The server has to first register, what data has to be shown. The protocol is composed of data in the JSON format, with a predefined structure. The server can notify the client about changes using signals, which also have to be registered ahead of time. The API for registering and using Ivette can be found in module `Server` inside the library `frama-c-server.core`.

Ivette is currently in development, and the API is subject to change in future versions of Frama-C. There is an undocumented requirement for plugins to be used with Ivette: the plugin must have a command-line argument, which enables the analysis. For example, the Eva plugin has the `-eva` command line argument, without which the plugin does not do anything. This requirement stems from the way Ivette processes command-line arguments and executes `frama-c` multiple times, first without user-provided arguments. In this first pass, the analysis must not run.

3 Separation logic

A common technique in program analysis is to generate logical formulae describing the possible states of the program at each point, and then use a solver to prove the correctness of the program. However, earlier analyses of programs with dynamic memory have faced a problem with globality – an abstract state of the program would contain the description of the whole heap in such a way, that an update of a single value would require updating the whole program state. Separation logic (SL) aims to solve this inefficiency by describing the heap in a way that allows for local updates of the heap. In this work, I will describe the syntax and semantics of SL [8], for which Tomáš Dacík implemented a solver called Astral [9]. Testing the solver's performance on formulae generated during the analysis is a secondary goal of this work.

3.1 Syntax

Let $x, y \in \mathbf{Var}$, where $\mathbf{Var}$ is an infinite set of variables, with a special variable $\mathrm{nil} \in \mathbf{Var}$. The syntax of SL is defined by the following grammar.

$$
\begin{aligned}
\varphi &::= x = y \mid x \neq y && \text{(pure atoms)} \\
&\quad \mid x \mapsto y \mid \mathrm{ls}(x, y) && \text{(spatial atoms)} \\
\varphi &::= \varphi \wedge \varphi \mid \varphi \vee \varphi \mid \varphi \wedge_{\neg} \varphi \mid \neg\varphi && \text{(boolean connectives)} \\
\varphi &::= \varphi * \varphi \mid \varphi \mathrel{-\!\!\circledast} \varphi && \text{(spatial connectives)}
\end{aligned}
$$

The atomic formulae $x = y$ and $x \neq y$ simply describe equality and inequality of two variables, $x \mapsto y$ corresponds to a pointer from x to y. The atom $\mathrm{ls}(x, y)$ describes a sequence of pointers from x to y, or in other words, an acyclic linked list. The boolean connectives have their usual meaning, the spatial connective $\varphi * \varphi$, called separating conjunction, says basically that the heap can be split into two disjoint parts, each satisfying one of the formulae. The septraction operator $\mathrel{-\!\!\circledast}$ has roughly the following meaning: formula $\varphi_1 \mathrel{-\!\!\circledast} \varphi_2$ is satisfied for all heaps, for which exists another heap satisfying φ_1, which can be merged with our heap, together satisfying φ_2.

3.2 Semantics

The models of SL formulae are so-called stack-heap models. Let **Loc** be an ordered set of memory locations, then stack s is a partial function from **Var** to **Loc**, and heap h is a partial function from **Loc** to **Loc**, where $h(s(\text{nil})) = \perp$. The symbol $\perp$ says that the function h is not defined for this input, which corresponds to an invalid memory access in the analyzed program. Stack-heap model is then simply a pair (s, h). The semantics of equality and inequality atoms are satisfied on all models, where $s(x) = s(y)$, or $s(s) \neq s(y)$ respectively, and the heap is empty. Points-to atom $x \mapsto y$ is satisfied on a stack-heap model, where the heap contains only a single allocated location, $h = \{s(x) \mapsto s(y)\}$. The list segment predicate $\text{ls}(x, y)$ is satisfied on every model where either $s(x) = s(y)$ and the heap is empty, or where the heap is a series of distinct allocated locations $h = \{l_0 \mapsto l_1, ..., l_{n-1} \mapsto l_n\}$ of length at least $n = 1$, where $s(x) = l_0$ and $s(y) = l_n$.

Semantics of boolean connectives are defined as usual. Separating conjunction $\varphi_1 * \varphi_2$ is satisfied on models, for which exist two heaps h_1, h_2, such that

$$(s, h_1) \vDash \varphi_1 \wedge (s, h_2) \vDash \varphi_2 \wedge h_1 \uplus h_2 \neq \perp \wedge h = h_1 \uplus h_2.$$

Septraction is defined similarly,

$$(s, h) \vDash \varphi_1 \mathbin{-\!\circledast} \varphi_2 \Leftrightarrow \exists h_1 : (s, h_1) \vDash \varphi_1 \wedge h_1 \uplus h \neq \perp \wedge (s, h_1 \uplus h) \vDash \varphi_2.$$

The operation $\uplus$ is defined as a union of two heaps, but it is defined only for heaps, whose domains are disjoint, and also share only named locations, i.e. locations, which are in the image of s.

3.3 Fragment used for analysis

Although any formula generated by the mentioned grammar is a well formed SL formula, I will use just a subset of all these formulae. This fragment has the form $\varphi = \varphi_1 * \varphi_2 * ... * \varphi_n$, where $\varphi_1, ..., \varphi_n$ are atomic formulae for equality, inequality, points-to relation, and list segments. Note that pure atoms are satisfied only on the empty heap, which is why separating conjunction is used for both spatial and pure atoms.

4 Shape analysis

Shape analysis is a kind of static analysis that aims to detect the shapes of data structures in the heap, such as linked lists, trees, and variations of those, and to use this knowledge to describe the state of a program's memory in more detail than would be otherwise possible. In [10], the authors propose a method to analyze programs with singly linked lists using separation logic for better scalability. However, they work with SL of slightly different semantics (pure atoms are satisfied on any heap), and they also implement shape analysis of a minimal language. They also do not use a dedicated solver for separation logic. My objective is to adapt their method to support at least a part of the C programming language, and to work with SL as implemented by Astral.

4.1 Implementation

In its current state, the plugin can analyze programs without function calls except allocation functions, multiple dereferences on any side of assignments, and global variables. Most of these missing aspects can be implemented without any new code in the analysis itself, merely by preprocessing the CIL AST and adding some heuristics. A more interesting extension might be

the to cover doubly linked lists and nested lists, since predicates for these structures are also available in recent versions of Astral. More on this in Section 5.

Let us start with the type of data that will be stored for each CFG node during the dataflow analysis. Currently, this is a list of SL formulae (`SSL.t list`). Each formula represents a possible state the program could be in. For example, after a statement `x = malloc(sizeof(void *));`, the program's memory could be described by any of the two formulae $\varphi_1 = (x \mapsto y')$, or $\varphi_2 = (x = \text{nil})$. This expresses the two possible outcomes of calling `malloc` – either the allocation succeeded, and x is now an allocated memory location, or it failed, and `x` is equal to `NULL`. The y' in the formula simply represents an arbitrary memory location that x is now pointing to, and it is a *fresh* variable.

Variables in the formulae are of two kinds, named and fresh variables. Named variables correspond to variables in the analyzed code, simply by having an identical name. Fresh variables, in this text marked with a tick x', are variables corresponding to memory locations unnamed in the analyzed code, for example the inner nodes inside a linked list, or the target of a dangling pointer. Note that Astral does not differentiate between named and fresh variables in any way, the only distinction is how this analysis treats each kind. Internally, fresh variables are differentiated by their names containing an exclamation mark.

The implementation of the analysis itself is done through the `Dataflow2` module API.

- `computeFirstPredecessor` – This function is implemented as identity.
- `doInstr` – As mentioned, this is the transfer function for instructions, and therefore much of the logic of the analysis is implemented here. Note that an instruction in the CIL naming convention is any C statement that doesn't affect the control flow of the current function, which means that a function call is also an instruction. As input, the function gets the instruction itself, and the state from the previously analyzed instruction, and it must return a new state for this instruction based on the inputs. For this analysis, I am interested in the following instructions: `LocalInit`, `Set`, and `Call`.
  - `LocalInit` is the initialization of a local variable. Initializations of non-pointer variables are currently ignored, and the previous state is returned. For pointer variables, the action depends on the initializing value. If the variable is initialized with a call to an allocation function (currently, this is detected simply by its name `malloc`), for each previous state, two new states are generated. One of them represents successful allocation, and therefore (<name> $\mapsto$ <fresh>) is appended to it using separating conjunction. `name` is simply the name of the newly initialized variable, and `fresh` is a globally unique (fresh) variable name generated by Astral. Any other initialization is handled with the same logic as an assignment.
  - `Set` is an assignment instruction. Again, this analysis is only interested in assignments to pointer variables, or assignments where the right-hand side contains a dereference. For other assignments, the previous state is returned.

    Simple assignment `a = b;`, where the type of the variables is a pointer type, results in the following change to the previous state. First, all occurrences of the variable `a` in the formula are substituted with a new, fresh variable, and then the atom $(a = b)$ is added to the formula.

    The assignment of a dereferenced variable, `a = *b;`, is more complicated. We must first find the variable, to which `b` is pointing, and then do essentially the same as with simple

assignment. To find the variable, to which `b` is pointing, the formula is first transformed into a shape, in which `b` or its alias is not a part of a list segment predicate, but a simple points-to predicate. Alias here means another variable equal to `b`. Then, the transitive closure of equality $C(b)$ is found for `b`, and then the set of all variables that are pointed to by this closure is found simply by iterating through all the atoms of the formula: $T = \{t; \exists c \in C(b) : c \mapsto t\}$. This set T ("target") can be empty – this corresponds to the possibility of dereferencing an invalid pointer in the program. In this case, the analysis is stopped. T can contain exactly one element $T = \{t\}$, then `a` is substituted with a new variable name as in the first case, and atom $a \mapsto t$ is appended to the substituted formula. This is the computed state. T cannot contain more than a single element, because such a formula would be unsatisfiable. Unsatisfiable formulae are filtered out in `doEdge`.

For write to a dereferenced variable `*a = b;`, the single-element set $T = \{t\}$ with the target of `a` is again found, and then the spatial atom $s \mapsto t$ is changed to $b \mapsto t$. `s` is a member of the equivalence class $C(a)$. Notice that in this case, no substitutions are made, because no additional equality is being added.

- `Call` is a function call instruction. Currently, only allocation functions are supported. The implementation is the same as in allocation during initialization, with the extra step that before creating two states for two outcomes of the allocation, the name of the variable `x` in `x = malloc(sizeof(void *));` would be first substituted in the whole formula with a fresh variable.

  The function `free` is also handled here. As in the previous cases, the set of aliases of the freed variable is again found, and the points-to atom leading from one of these variables is removed.

- `combinePredecessors` – This function is called when the analysis reaches a CFG node for the second time, and computes new data for this node. `combinePredecessors` gets the old state and the new state as input parameters, and returns these states joined together, as well as information on whether the old state was updated in any way. This is then used by the dataflow analysis to decide whether to continue updating nodes following this one. Note that the state of a CFG node is internally a list of formulae, but the meaning is a logical disjunction of these (the models of a state for a single CFG node are the union of models of all the formulae in the list).

  In theory, we could simply take the old state φ_{old} and new state φ_{new}, and check the entailment $\varphi_{\text{old}} \vDash \varphi_{\text{new}}$. If this were true, all models of the new state would have been contained in the old state, and we could have returned that the state had not changed. Otherwise, we would have returned the disjunction of new and old states (internally, a list concatenation). However, this would be imprecise, a single formula in φ_{new}, of which a single model would not be a model of φ_{old} would cause adding the entirety of φ_{new}.

  Instead, the formulae of both old and new states are all placed in a single list φ_{all}, and a new list φ_{unique} is constructed by the following algorithm. In each iteration, a single formula φ_{next} is removed from φ_{all}, and the following entailment is computed:

$$\varphi_{\text{next}} \vDash \bigvee_{\varphi \in \varphi_{\text{all}} \cup \varphi_{\text{unique}}} \varphi$$

  If the entailment is true, φ_{next} is simply thrown out, since its models are fully contained in either φ_{all} or φ_{unique}. If the entailment is false, φ_{next} is added to φ_{unique}. This is done until φ_{all}

does not contain any formulae. The joined state for this CFG node is then the list φ_{unique}. This method proved to be efficient in eliminating duplicate formulae, but it is also probably the most expensive part of the analysis, because when computing the entailment, all fresh variables in the formulae are existentially quantified.

- `doStmt` – This function decides, whether to continue with analysis upon reaching a statement. This is independent of the actual algorithm for dataflow analysis, which decides when the analysis is complete – that is, when the state of all statements cannot be updated. Currently, this is set to always continue analysis.
- `doGuard` – called when the analysis reaches an `if` statement. The function gets the state from the previous node and the condition expression, and returns the states to use in each of the branches. Currently, all conditions that are not in the form `a == b` or `a != b` are considered nondeterministic. When reaching a nondeterministic condition, the analysis simply uses the input state as the state for both branches, because it gained no additional information about what is true in the branches. If the condition is `a == b`, then all formulae of the input state φ_{in_i} are tested separately using the following method. For the "then" branch, each formula is appended with $(a = b)$ using separating conjunction, and checked for satisfiability. Unsatisfiable formulae are filtered out, the rest is used in the "then" branch. For the "else" branch, the same method is used, only the formula is appended with $(a \neq b)$. This way, only satisfiable formulae are passed into their respective branches, simplifying further analysis. For the case of condition `a != b`, the algorithm is the same, only the states for the two branches are swapped. If there is no formula left for a certain branch after filtering out unsatisfiable formulae, the branch is marked as unreachable. Note that this method still allows for the analysis of conditions such as `(list != NULL)`, because the `NULL` constant will be swapped for the special `__nil` variable, see Section 4.3 for more details.
- `doEdge` – called between the updates of nodes, it can modify the state that is sent from the previous node to the next one. In this function, the satisfiability of all formulae in the state is checked, and unsatisfiable formulae are removed from the state. All formulae are also simplified using Astral's `Simplifier.simplify` function, which performs many simple syntactic optimizations of the formula without changing its semantics, such as flattening of nested operations. Most importantly, this is where the abstraction into list segments is performed, and where the formulae are further simplified, this time taking advantage of the different semantics of named and fresh variables. More on this below.

4.2 Abstraction and simplifications

The following simplifications are done on each formula separately in this order. The main purpose of most of these is to reduce the number of fresh variables, since these must lie under a quantifier when checking an entailment. The other purpose is to reduce the size of the formula itself, mainly by eliminating useless information, such as irrelevant equalities and inequalities.

- The formula is syntactically simplified using Astral's `Simplifier`.
- The formula is checked for satisfiability, and if unsatisfiable, it is removed.
- Variables going out of scope on this transition between statements are substituted with fresh variables. A list of all variables in scope at each statement is computed before starting the analysis using a preprocessor pass, more in Section 4.3. Variables going out of scope are the set difference between the sets of variables in scope at the previous and next statement.

- *Junk* atoms are removed from the formula. Junk atoms are atoms $x' \mapsto y$, $\mathrm{ls}(x', y)$, or $x' = y$, in which x' is fresh (as indicated by the tick), and it appears only once in the formula. Removing the atom $x' = y$ does not have any effect on the analysis, since the variable x' is mentioned only in this equality. A points-to or list segment atom starting at x' corresponds to a memory leak, since x' is an unnamed variable, and it is therefore no longer reachable from the analyzed code. Removing this information will not affect the result of the analysis, but it will simplify the formula.
- Fresh variables equal to nil are substituted with nil. This is done by finding the equivalence class for the special variable nil, and substituting all fresh variables in this class with nil. This is done to reduce the number of fresh variables in the formula.
- Equivalence classes of fresh variables are reduced to a minimal size. The equivalence class $E_{x'}$ for a fresh variable x' is computed, and if it contains a named variable y, all fresh variables in $E_{x'}$ are substituted with y. If not, all fresh variables in $E_{x'}$ are substituted with x'. After this, all equalities with the same variable on both sides are removed. This simplification, like the previous, is done to reduce the number of variables in the formula.
- Consecutive points-to atoms are abstracted into a list segment atom. This is done by joining two spatial atoms (points-to atom, or a list segment) into a single list segment, while removing the fresh variable in their middle:

  $$P(x, y') * P(y', z) \rightsquigarrow \mathrm{ls}(x, z) * x \neq z$$

  where

  $$P(x, y) \coloneqq x \mapsto y \mid \mathrm{ls}(x, y)$$

  Since Astral allows list segments to have the length zero, the inequality $x \neq z$ is added. This is done for all created list segments, and all other steps in this analysis assume that this inequality is present for any list segment.

  However, several conditions need to be met for this abstraction to be valid. Firstly, x and z must be distinct before the transformation. This is checked by checking the satisfiability of the original formula with the $x = z$ atom added. If the resulting formula is satisfiable, the abstraction cannot be done.

  Secondly, y' cannot be a part of any other equality or pointer predicate anywhere else in the formula because this would not be compatible with the semantics of the list segment predicate. This property is checked syntactically in the formula itself.
- All fresh variables occurring solely in inequalities are removed along with the inequality atoms themselves. These inequalities are left in the formula after joining existing list segments and can be removed, since they do not contain any meaningful information.
- Formulae are checked for redundancy using the same algorithm that is used when computing a join of two states in `combinePredecessors`. The set of initial formulae φ_{all} at the start of the algorithm is simply the whole set of input formulae.

4.3 Preprocessing

Preprocessing is done on the AST of the analyzed program using visitors (described in Section 2.2). Writing the analysis rules themselves to support all syntactical constructs in the C language would be tedious, a much better option is to support basic constructs in the analysis

itself, and to transform any program into this basic form before the analysis. Currently, the following transformations are done in this order:

- All variable names in the analyzed program are made globally unique. This is done as a precaution for the future, when the analysis will work with local as well as global variables, and to make the output of the analysis easier to read and interpret.
- All pointer or integer constants in the file are replaced with a special variable `_nil`, which is set to to `NULL` at the start of the program. Thanks to this, the analysis can omit the support for constants everywhere besides initialization, notably in the evaluation of conditions, e.g. `x == NULL`.
- Field accesses on structs are either converted to dereferences, or removed from the AST altogether. Consider the following structure:

  ```
  struct List {
      List *next;
      uint32_t data;
  }
  ```

  When visiting a statement, in which a field of this structure is accessed, the structure field is analyzed. If the structure itself has more than two self-referential pointers (`List *`), the statement is discarded (removed from the AST), and a warning about a skipped structure is printed. This structure likely represents an ADT which is not yet supported, such as a doubly linked list, a binary tree, a graph, etc. If the structure has a single self-referential pointer, and it is the currently accessed field, the field access is converted to a simple dereference. For example, `list->next` would be converted into `*list`.

  If the accessed field is not self-referential, no matter whether the structure itself contains such a field or not, the whole statement (e.g. `list->data = 42;`) containing the field is removed.
- For each statement, its local variables are stored in an associative array for use in the simplification Section 4.2. This is done simply by storing the local variables of each block scope when traversing the AST.
- Type casts are removed from the AST. The analysis does not currently take types of variables into account, apart from a broad distinction between pointer and non-pointer types, and it is therefore simpler to remove all type casts from the AST, eliminating the need to create rules for these in the analysis. This is the only visitor using the low-level `Cil.CilVisitor` instead of the `Visitor` module itself, because the high-level visitor introduces pointer casts (e.g. `void *` into `List *`) back into the AST. This is also why this step is done as the last one.

4.4 GUI

As mentioned in Section 2.3, the API for interfacing with Ivette is not yet stable, or documented, so it would not make sense to create complex visualizations of the analysis with this version of Frama-C. However, mostly for debugging purposes, I added a simple text field for each statement, which shows the final state (a set of formulae) reached in each statement. The plan is to expand this GUI to show the whole progress of the analysis in real time, leveraging the asynchronous nature of Ivette, and to provide a user-friendly result summary including errors and warnings in the relevant parts of the source code.

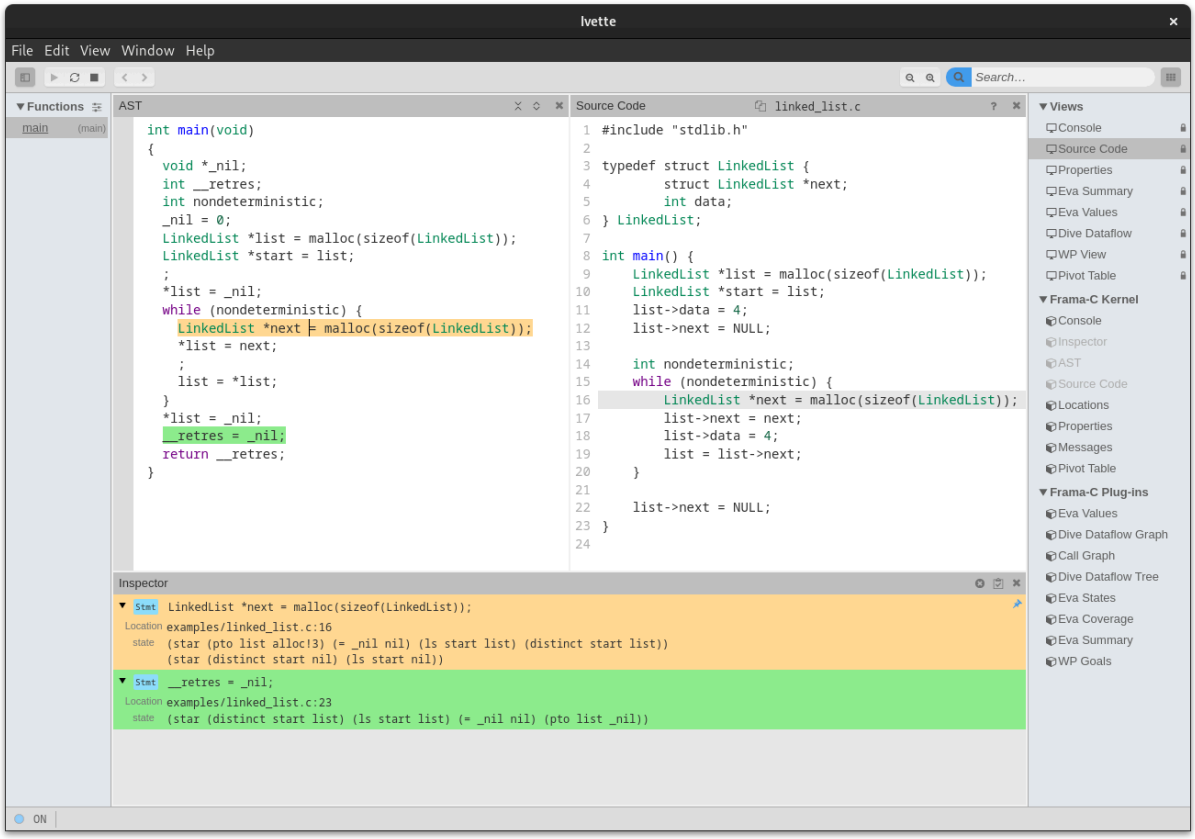

Figure 1: Ivette showing formulae computed for two statements. The right panel shows the original code, the left panel shows the code after preprocessing.

4.5 Example

The following example shows the analysis of a simple program, which constructs a singly linked list of a nondeterministic, unbounded length. The program then iterates through the list, modifying the data stored in each node. Finally, the list is deallocated. The full source code is available in Appendix A.

After the allocation of the list, the state is the following formula:

$$\_\text{nil} = \text{nil} * \text{ls}(\text{start}, \_\text{nil}) * \text{start} \neq \_\text{nil}$$

This demonstrates many things, firstly that the abstraction to a list segment was successful, otherwise the analysis would not have terminated at all, because the state formulae would grow in length with each iteration of the loop, indefinitely. Secondly, the small size of the formula shows that the elimination of variables going out of scope works, otherwise we would see other variables from within the block inside the formula. Lastly, the fact that only one formula remained after analyzing the block shows that the simplifications done in `doEdge` are effective in simplifying and deduplication of the formulae.

After the second block, which traverses the list, the state is not changed. This should be expected, as the code does not modify the structure of the list itself, only the data contained within it.

After the third block, which frees the list, we get the following state:

$$\_\text{nil} = \text{nil} * \text{start} \neq \_\text{nil}$$

This formula describes the empty heap, indicating that the analysis correctly followed the actions of the code. The atom (start ≠ _nil) contains the information that `start` is now a dangling pointer, since it is not allocated, but it is not a null pointer. If we add a statement to the end of the program dereferencing the pointer, for example `start->next = NULL;`, the analysis will report an error:

```
[SLplugin] examples/linked_list.c:54: Failure:
    detected a dereference of an unallocated variable
```

5 Future work

A small shortcoming of the current implementation is that statements with multiple composite l-values are not supported by the analysis. However, these composite statements can be broken down into a sequence of the following four basic statements:

```
a = b;
a = *b;
*a = b;
a = malloc(...);
```

For example, the statement `*a = **b;` can be broken down into this sequence, using temporary variables:

```
tmp_1 = *b;
tmp_2 = *tmp_1;
*a = tmp_2;
```

This will be simple to implement using a new AST visitor. The main reason why this has not been implemented already is that I encountered some difficulties with adding statements to the AST, which were resolved only recently. Moreover, this missing feature was not essential when testing the analysis, so the priority for implementing this preprocessing pass was low.

Another missing feature is the analysis of function calls other than allocation functions. This can be done in multiple ways, for example programs without recursion can be inlined into a single function. This is not a big limitation, since recursion is generally avoided in low-level code. Another option is to create function summaries – pairs of input formulae and changes done by the function, stored for each function. Ideally, only the relevant part of the formula should be passed into a function for analysis, which would reduce the time needed for analysis and help create more general summaries, allowing for the simple reuse of the computed result of a function call.

Global variables can be converted to local variables in the entrypoint function, and passed into every other function as extra input parameters. This can be further optimized by not passing variables irrelevant to the content of the called functions.

Most importantly, Astral now supports more predicates, which can be used for an abstraction of doubly linked lists, and nested lists. Therefore, the implementation of more abstractions is possible. This would mean going beyond the abstraction rules laid out in [10], but it would allow for the analysis of a much wider range of programs, as doubly linked lists are used more commonly than singly linked lists.

6 Conclusion

In the current state, the analysis is able to process simple programs operating on singly linked lists. The main limitations include the lack of support for analysis across function boundaries,

and the inability to analyze more complex data structures. However, Astral itself supports other, currently unused abstractions, making it possible to implement the analysis of more complex structures in the future.

Bibliography

[1] N. Nethercote and J. Seward, "Valgrind: a framework for heavyweight dynamic binary instrumentation," *SIGPLAN Not.*, vol. 42, no. 6, pp. 89–100, Jun. 2007, doi: 10.1145/1273442.1250746.

[2] K. Serebryany, D. Bruening, A. Potapenko, and D. Vyukov, "AddressSanitizer: a fast address sanity checker," in *Proceedings of the 2012 USENIX Conference on Annual Technical Conference*, in USENIX ATC'12. Boston, MA: USENIX Association, 2012, p. 28–29.

[3] J. Signoles, P. Cuoq, F. Kirchner, N. Kosmatov, V. Prevosto, and B. Yakobowski, "Frama-c: a Software Analysis Perspective," 2012, p. . doi: 10.1007/s00165-014-0326-7.

[4] G. C. Necula, S. McPeak, S. P. Rahul, and W. Weimer, "CIL: Intermediate Language and Tools for Analysis and Transformation of C Programs," in *Compiler Construction*, R. N. Horspool, Ed., Berlin, Heidelberg: Springer Berlin Heidelberg, 2002, pp. 213–228.

[5] "Frama-C API documentation." [Online]. Available: https://frama-c.com/download/frama-c-27.1-Cobalt-api.tar.gz

[6] J. Signoles, T. Antignac, L. Correnson, M. Lemerre, and V. Prevosto, "Frama-C Plug-in Development Guide." [Online]. Available: http://frama-c.com/download/frama-c-plugin-development-guide.pdf

[7] L. Correnson, "Ivette: A Modern GUI for Frama-C," 2023, pp. 116–131. doi: 10.1007/978-3-031-26236-4_10.

[8] T. Dacík, "A Decision Procedure for Strong-Separation Logic," 2022. [Online]. Available: https://www.fit.vut.cz/study/thesis/25151/

[9] T. Dacík, "Astral solver." [Online]. Available: https://github.com/TDacik/Astral

[10] D. Distefano, P. W. O'Hearn, and H. Yang, "A Local Shape Analysis Based on Separation Logic," in *Tools and Algorithms for the Construction and Analysis of Systems*, H. Hermanns and J. Palsberg, Eds., Berlin, Heidelberg: Springer Berlin Heidelberg, 2006, pp. 287–302.

Appendix A

This is the full source code for the example in Section 4.5.

```
#include "stdlib.h"

typedef struct List {
        struct List *next;
        int data;
} List;

int main() {
    List *start = malloc(sizeof(List));
    if (start == NULL)
        return 1;

    { // construct a linked list of unknown size
        List *list = start;
        list->next = NULL;
        int nondeterministic;
        while (nondeterministic) {
            List *next = malloc(sizeof(List));
            if (next == NULL)
                return 1;
            list->next = next;
            list = list->next;
        }
        list->next = NULL;
    }

    { // walk to the end of the list
        List *list = start;
        while (list != NULL) {
            List *next = NULL;
            next = list->next;
            list->data = 42;
            list = next;
        }
    }

    { // free the list
        List *list = start;
        while (list != NULL) {
            List *next = NULL;
            next = list->next;
            free(list);
            list = next;
        }
    }

    // this would be detected as an error
    // start->next = NULL;
}
```