coreStar: The Core of jStar

Matko Botinčan[1], Dino Distefano[2], Mike Dodds[1],
Radu Grigore[2], Daiva Naudžiūnienė[1], and Matthew J. Parkinson[3]

[1] University of Cambridge,
{matko.botincan,mike.dodds,daiva.naudziuniene}@cl.cam.ac.uk
[2] Queen Mary, University of London,
{dino.distefano,radu.grigore}@eecs.qmul.ac.uk
[3] Microsoft Research Cambridge, mattpark@microsoft.com

Abstract. Separation logic is a promising approach to program verification. However, currently there is no shared infrastructure for building verification tools. This increases the time to build and experiment with new ideas. In this paper, we outline coreStar, the verification framework underlying jStar. Our aim is to provide basic support for developing separation logic tools. This paper shows how a language can be encoded into coreStar, and gives details of how coreStar works to enable extensions.

1 Introduction

Separation logic [23] based approach to program verification has gained a lot of attention recently. It enables sound and precise reasoning about complex heap (and, generally, resource) properties which pose a major challenge to other approaches. For making separation logic an interesting hammer for program verification the crucial thing was development of tools that: (1) have automated the reasoning with separation logic [4,14,17,18,10] and (2) have scaled it to programs of substantial size [25,9,6]. These efforts were focused on automated proving of shape properties for low-level C programs, however, separation logic based verification of programs written in higher-level languages has also been approached. For instance, Distefano and Parkinson have developed jStar, a tool for verifying Java programs [15]. What is common about all these tools is that each one of them had to incorporate in one way or another a number of core techniques such as symbolic execution with separation logic [5] or loop invariants inference (e.g. like in [14]) for ensuring termination of the symbolic execution.

In general, building a program verification tool is a daunting task and requires knowledge of many domains (such as decision procedures, theorem proving, formal semantics, verification condition generation, abstract interpretation, compilation, etc.) and a complex software engineering. The line of work on the Boogie program verifier [1,20] has identified this problem and has offered a solution in terms of an intermediate verification language and a program verifier for this language. The end result has shown a lot of success and Boogie has been used as a verification backend for a number of tools including Spec# [3], VCC [11] and Dafny [19].

Motivated by the Boogie approach to program verification, this paper proposes a similar agenda for separation logic based program verification. We present coreStarIL — an intermediate language for program verification with separation logic, and coreStar — a tool enabling automated verification of coreStarIL programs. Like with Boogie, coreStarIL programs can be written manually, however, the goal is that they are automatically generated by program verifiers that take programs written in higher-level languages and encode the semantics of the input program by translating it to coreStarIL.

coreStar is the result of efforts to make the core of jStar generic and reusable. It has been successfully applied as a backend to a couple of other separation logic based program verifiers: MultiStar [24] for reasoning about multiple related abstractions with its frontend for Eiffel and VMC [8,7] for verifying multicore C programs with asynchronous memory operations. Nevertheless, many of its design choices are still open, and design decisions already been made have to be justified further. The purpose of this paper is to describe the current state of coreStar and encourage its critique and suggestions from the verification community for the further development.

2 A crash course on separation logic

Separation logic [23] helps in achieving *local reasoning* when dealing with heap allocated data structures. The idea is that verification should focus on what changes, not what stays the same. To achieve this, separation logic takes a different starting perspective than Hoare logic: instead of pre- and postconditions describing the global state, it just describes a part of the state. A precondition must describe resources that a command accesses, and everything not mentioned in the precondition is implicitly left unchanged. If the command terminates, then the resulting partial state satisfies the postcondition.

To deal with this property formally, separation logic introduces a new logical connective $*$ that expresses disjointness of partial states. The conjunction $P_1 * P_2$ says that the state can be split into two disjoint parts, one satisfying P_1 and the other P_2. This connective enables the so-called *frame rule* which allows us to extend any Hoare triple $\{P\}\, C\, \{Q\}$ by an arbitrary frame F that is unchanged by the command C, and in this way enable the local reasoning.

Any application of separation logic to automated program analysis and verification requires support of a separation logic theorem prover. The purpose of the prover is to answer queries involving separation logic formulae that occur during symbolic execution. Existing techniques for automated theorem proving in separation logic deal only with a limited fragment of separation logic (often referred to as "symbolic heaps"). In this fragment there is no negation and formulae are required to be of the form $\Delta = \Pi \wedge \Sigma$ where Π is a $\wedge$-separated sequence of pure (first-order) formulae, and Σ is a $*$-separated sequence of spatial formulae. This fragment has been proven in practice to be quite effective in terms of expressiveness as well as computational tractability.

coreStar is intended to be generic. Hence, the underlying symbolic heap representation in the tool does not hard-code any particular pure or spatial predicates (even the very basic points-to predicate $\mapsto$ is not included). A user defines his own version of separation logic to be used by providing logic rules that define how the predicates are manipulated. We provide a more detailed description about logic rules and how coreStar's separation logic theorem prover works in Sec. 5.

3 coreStar's input language coreStarIL

coreStarIL is a simple untyped imperative language that can be seen as a variant of Dijkstra's guarded commands [12]. Variants of Dijkstra's guarded commands have also been used as a basis of intermediate representation in ESC/Java [16] and Boogie [2]. The coreStarIL syntax looks as follows:

$$
\begin{aligned}
\mathit{Program} &::= \{\Delta_P\}\ (\mathit{CoreStmt};)^+\ \{\Delta_Q\} \\
\mathit{CoreStmt} &::= \bar{x} := \{\mathsf{Pre}\}\{\mathsf{Post}\} \\
&\quad \mid\ \mathsf{goto}\ l_1, \ldots, l_n \\
&\quad \mid\ \mathsf{label}\ l \\
&\quad \mid\ \mathsf{abs}
\end{aligned}
$$

Here Δ_P, Δ_Q (the pre- and postcondition of the program), Pre and Post (the pre- and postcondition of commands) are symbolic heaps, $\bar{x}$ is a list of variables (we use () to denote the empty list), and $l, l_1, \ldots, l_n$ are labels. A symbolic heap is a formula of the form $\Pi \wedge \Sigma$ where Π and Σ are pure and spatial assertions, respectively, defined by:

$$
\begin{aligned}
\Pi &::= \mathsf{true} \mid E = E \mid E \neq E \mid p(\overline{E}) \mid \Pi \wedge \Pi \\
\Sigma &::= \mathsf{emp} \mid s(\overline{E}) \mid \Sigma * \Sigma
\end{aligned}
$$

(E ranges over expressions, $p(\overline{E})$ is a family of pure predicates and $s(\overline{E})$ a family of spatial predicates).

The key difference compared to the other representations [12,16,2] is use of so called *specification assignment* $\bar{x} := \{\mathsf{Pre}\}\{\mathsf{Post}\}$. We explain the semantics of the specification assignment in more details in Sec. 4, but, intuitively, its purpose is to support state-modifying commands in a way that is "natural" for separation logic: the part of the state that is not touched by the command is framed away and the specification of the command is used to replace the local pre-state with the local post-state. In specification assignments we allow special variables _v that are implicitly existentially quantified (over the whole statement). The postcondition of the specification assignment can contain variables $\mathsf{ret}_1, \ldots, \mathsf{ret}_n$ that get bound to $x_1, \ldots, x_n$.

The rest of the statements are fairly standard: goto performs demonic non-deterministic jump to one of the labels denoted by label statement and abs triggers the abstraction[4] (the meaning of which is explained in Sec. 6).

3.1 coreStarIL encoding of a simple heap-manipulating language

To illustrate how to translate from a higher level language to coreStarIL we show a translation of a Smallfoot [4]-like heap-manipulating language. We consider a slightly simplified sequential version of Smallfoot with the commands of the following syntax:

$$
\begin{array}{lll}
A ::= \mathsf{empty} & & \text{empty command} \\
\quad | \quad x := E & & \text{variable assignment} \\
\quad | \quad x := [E] & & \text{heap lookup} \\
\quad | \quad [E] := F & & \text{heap mutation} \\
\quad | \quad \mathsf{new}(x) & & \text{allocation} \\
\quad | \quad \mathsf{dispose}(x) & & \text{deallocation} \\
C ::= A \mid C; C \mid x := f(E) & & \\
\quad | \quad \mathsf{if}(E)\,\{C\}\,\mathsf{else}\,\{C\} \mid \mathsf{while}(E)\,\{C\} & &
\end{array}
$$

Atomic commands. Since atomic commands A allow reading and mutation of heap we have to somehow represent the heap contents. The standard way in separation logic is to use the spatial predicate $\mapsto$ such that $x \mapsto v$ iff the local heap contains a single cell x that points to the value v. Following the semantics of the atomic commands [5] we translate them to coreStarIL statements as follows:

$$
\begin{array}{rcl}
tr(\mathsf{empty}) & = & () := \{\}\{\}; \\
tr(x := E) & = & x := \{\}\{\mathsf{ret} = E\}; \\
tr(x := [E]) & = & x := \{E \mapsto \_\mathsf{v}\}\{E \mapsto \_\mathsf{v} \wedge \mathsf{ret} = \_\mathsf{v}\}; \\
tr([E] := F) & = & () := \{E \mapsto \_\mathsf{v}\}\{E \mapsto F\}; \\
tr(\mathsf{new}(x)) & = & () := \{\}\{x \mapsto \_\mathsf{v}\}; \\
tr(\mathsf{dispose}(x)) & = & () := \{x \mapsto \_\mathsf{v}\}\{\};
\end{array}
$$

In the translated statements _v is a fresh unifiable variable. For readability we have omitted explicitly denoting the translation of expressions.

If-else and while. coreStarIL does not contain assert and assume statements like languages in [12,16,2] as these statements can in fact be encoded using specification assignment:

$$
\begin{array}{rcl}
\mathsf{assert}(E) & \triangleq & () := \{E\}\{\} \\
\mathsf{assume}(E) & \triangleq & () := \{\}\{E\}
\end{array}
$$

[4] Although in principle we can perform abstraction automatically at dominators of basic blocks, we want to allow frontend to decide when to perform the abstraction.

Using assume statements the if-else and while commands can be translated in a standard way:

$$\begin{array}{rcl} tr(\mathsf{if}(E)\,\{C_1\}\,\mathsf{else}\,\{C_2\}) & = & \mathsf{goto}\ l_1, l_2; \\ & & \mathsf{label}\ l_1;\ \mathsf{assume}(E);\ tr(C_1)\ \mathsf{goto}\ l_3; \\ & & \mathsf{label}\ l_2;\ \mathsf{assume}(\neg E);\ tr(C_2)\ \mathsf{goto}\ l_3; \\ & & \mathsf{label}\ l_3; \\ tr(\mathsf{while}(E)\,\{C\}) & = & \mathsf{label}\ l_1;\ \mathsf{abs};\ \mathsf{goto}\ l_2, l_3; \\ & & \mathsf{label}\ l_2;\ \mathsf{assume}(E);\ tr(C)\ \mathsf{goto}\ l_1; \\ & & \mathsf{label}\ l_3;\ \mathsf{assume}(\neg E); \end{array}$$

where l_1, l_2 and l_3 are fresh labels. Note that we are putting the abs statement at the head of the loop to allow abstraction (though a translation equivalent to what the Smallfoot tool does would not contain it).

Procedure calls. A procedure call is translated by using the callee's specification and encoding it into the specification assignment. If $\lambda y\,.\,\{P\}f(y)\{Q\}$ is the specification of a function f then

$$tr(x := f(E)) \quad = \quad x := \{P[E/y]\}\{Q[E/y]\}$$

In order to avoid capture, the variables in the specification of f need to be freshened before the substitution takes place.

4 Symbolic execution in coreStar

In coreStar, the symbolic execution is performed on the control-flow graph of the input program. The symbolic states are defined as pairs of a CFG node and a symbolic heap. Statements are executed symbolically over the symbolic states, i.e., at each step the current symbolic state is updated to reflect the abstract effect of the statement. The process executes until it reaches a fix-point where no new symbolic states can be reached.

The input to the symbolic execution is a coreStarIL program of the form $\{\Delta_P\}s_1; \ldots s_n; \{\Delta_Q\}$. coreStar provides two kinds of symbolic execution:

- symbolic execution with frame inference;
- bi-abductive symbolic execution.

In the classical symbolic execution with frame inference, assuming Δ_P statements are symbolically executed aiming to reach at the end a symbolic state implying Δ_Q. However, this approach assumes that we have given a sufficient precondition for $s_1; \ldots s_n;$. Sometimes we want to allow incomplete preconditions (e.g., because we want to relieve the user of the burden to write down the specification or because we may just not know the complete specification). coreStar can try to find out the missing part of the precondition Δ_M such that $\{\Delta_P * \Delta_M\}s_1; \ldots; s_n\{\Delta_Q\}$ (even if Δ_P is empty). This approach requires using a technique called bi-abduction [9] so we are referring to it as bi-abductive symbolic execution.

4.1 Symbolic execution with frame inference

Classical symbolic execution for separation logic requires frame inference, that is: given Δ_1 and Δ_2 find Δ_F such that $\Delta_1 \vdash \Delta_2 * \Delta_F$ holds. The frame inference allows us to execute a command C with a precondition P and a postcondition Q from a current state Δ_H in the following way:

$$\frac{\{P\}\, C\, \{Q\} \qquad \Delta_H \vdash P * \Delta_F}{\{\Delta_H\}\, C\, \{Q * \Delta_F\}}$$

If there exists Δ_F such that $\Delta_H \vdash P * \Delta_F$ then this rule says that from a pre-state Δ_H we can propagate $Q * \Delta_F$ to the post-state.

4.2 Bi-abduction

Bi-abduction [9] can be seen as a generalisation of the frame inference: given Δ_1 and Δ_2 the goal is to find the "missing" assumption (anti-frame) Δ_A and the frame Δ_F such that $\Delta_1 * \Delta_A \vdash \Delta_2 * \Delta_F$ holds. The authors of [9] have considered solving the bi-abduction query by utilizing separate proof systems for frame inference and abduction. In coreStar we solve the bi-abduction query using a single proof system as it will be explained in Sec. 5.

Using the Hoare's rule of consequence, the procedure for bi-abduction gives rise to a bi-abductive version of the frame rule:

$$\frac{\{P\}\, C\, \{Q\} \qquad \Delta_H * \Delta_A \vdash P * \Delta_F}{\{\Delta_H * \Delta_A\}\, C\, \{Q * \Delta_F\}}$$

The rule tells us that given a pre-state Δ_H and a specification of a command $\{P\}\, C\, \{Q\}$ we can compute Δ_F and Δ_A such that $\Delta_H * \Delta_A$ is sufficient to execute C in order to obtain the post-state $Q * \Delta_F$.

4.3 Bi-abductive symbolic execution

Now we describe how the symbolic execution with bi-abduction operates in coreStar. To simplify the presentation we represent the input program by a set of nodes N, and the functions $succ : N \to N$ returning the successor node which represents the statement that follows in the input program and $stm : N \to \mathcal{S}$ returning the statement associated with the node ($\mathcal{S}$ denoting the set of coreStarIL statements). Symbolic execution operates by traversing pairs of the form $(n, (\Delta_H, \Delta_M))$ where $n \in N$ is a node, Δ_H is the symbolic heap representing the current state and Δ_M is the missing part of the heap discovered so far which needs to be added to the precondition.

Fig. 1 shows the symbolic execution rules that define the effect of each coreStarIL statement on a symbolic state. If $(n, (\Delta_H, \Delta_M)) \overset{s}{\rightsquigarrow} (n', (\Delta'_H, \Delta'_M))$ then executing s in the state $(n, (\Delta_H, \Delta_M))$ leads to the new state $(n', (\Delta'_H, \Delta'_M))$. All rules apart from the rule for specification assignment are straightforward. To execute $\bar{x} := \{\mathsf{Pre}\}\{\mathsf{Post}\}$ in a pre-state (Δ_H, Δ_A) we proceed as prescribed

$$
\begin{array}{lll}
(n,(\Delta_H,\Delta_M)) & \overset{\bar{x}:=\{\mathsf{Pre}\}\{\mathsf{Post}\}}{\rightsquigarrow} & (succ(n),(\Delta_F * \mathsf{Post}[\bar{x}/\bar{\mathsf{ret}}],\Delta_A * \Delta_M)) \text{ where} \\
 & & \Delta_H * \Delta_A \vdash \mathsf{Pre} * \Delta_F \\
(n,(\Delta_H,\Delta_M)) & \overset{\mathsf{goto}\ l_1,\ldots,l_n}{\rightsquigarrow} & (n_{l_i},(\Delta_H,\Delta_M)) \text{ where } stm(n_{l_i}) = \mathsf{label}\ l_i,\ i = 1..n \\
(n,(\Delta_H,\Delta_M)) & \overset{\mathsf{label}\ l}{\rightsquigarrow} & (succ(n),(\Delta_H,\Delta_M)) \\
(n,(\Delta_H,\Delta_M)) & \overset{\mathsf{abs}}{\rightsquigarrow} & (succ(n),abs(\Delta_H,\Delta_M))
\end{array}
$$

Fig. 1: Symbolic execution rules for coreStarIL statements.

by the bi-abductive version of the frame rule. We first invoke the prover to provide us with the pair of symbolic heaps (Δ_F, Δ_A) solving the associated bi-abduction query, and then conjoin (Δ_F, Δ_A) with $(\mathsf{Post}[\bar{x}/\bar{\mathsf{ret}}], \Delta_M)$ to obtain $(\Delta_F * \mathsf{Post}[\bar{x}/\bar{\mathsf{ret}}], \Delta_A * \Delta_M)$ as a post-state.

The first and the second rule in Fig. 1 are nondeterministic (the nondeterminism in the first rule may arise due to multiple answers to the frame inference question, and in the second due to multiple labels) thus the transition to a new symbolic state may introduce more than just a single symbolic heap pair associated with the destination node. Thus in coreStar the nodes are associated with sets of symbolic heap pairs and are traversed by employing a worklist algorithm. Each such set of symbolic heaps pairs associated to a node represents a disjunction over its elements.

After the symbolic execution completes we are left with a missing part in the anti-frame that together with the starting Δ_P forms a new candidate precondition $\Delta_{P'}$. To check that the $\Delta_{P'}$ is sufficient to execute the program we re-run the symbolic execution starting with precondition $\Delta_{P'}$, this time with the frame inference only.

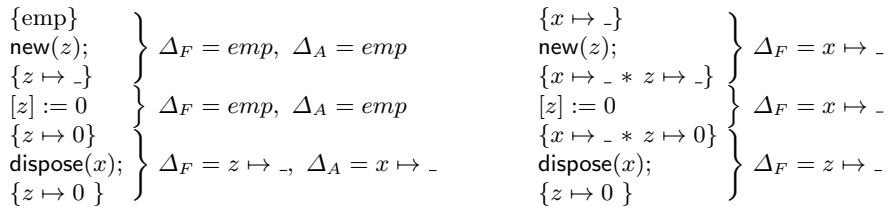
Example. To illustrate how the bi-abductive symbolic execution works let us consider the example in Fig. 2 (a). Symbolic execution with frame inference only would fail at the third statement program since we do not have a heap cell for x. Bi-abductive symbolic execution at this step finds the missing part $\Delta_A = x \mapsto \_$. After forming a new candidate precondition we run the symbolic execution with frame inference as shown in Fig. 2 (b).

5 The coreStar's separation logic theorem prover

As we have seen in Sec. 4, the purpose of the separation logic theorem prover in coreStar is to provide an answer to three types of queries:

- the frame inference;
- the bi-abductive frame inference; and
- deciding implications between symbolic heaps.

The last type of query occurs when checking whether the final disjunctive state in the symbolic execution satisfies the desired postcondition. Answering all three types of queries reduces to a particular proof search task.



(a) Bi-abductive frame inference (b) Frame inference

Fig. 2: Symbolic execution example

5.1 Prover's internals

The design of the coreStar's separation logic prover builds on the design of the entailment prover in Smallfoot [5] and descendent tools. In contrast to these prior tools, the coreStar's prover works with an internal representation that allows performing the bi-abductive frame inference directly. We first explain how the prover performs the frame inference, and then show what is different for the other two types of queries.

Frame inference. To perform frame inference coreStar's prover works with sequents $\Sigma \mid \Delta_A \vdash \Delta_G$, where Δ_A is the assumed formula, Δ_G is the goal formula and Σ is the subtracted (spatial) formula. The semantics of this judgment is $\Sigma * \Delta_A \;\Rightarrow\; \Sigma * \Delta_G$.

As in Smallfoot, in order to answer the frame inference question, starting with a sequent $\mathsf{emp} \mid \Delta_1 \vdash \Delta_2$ the prover searches for sequents of the form $\Sigma \mid \Delta_L \vdash \mathsf{true} \wedge \mathsf{emp}$. All such leftover formulae Δ_L are collected from the leaves of the proof search tree and their disjunction forms the frame that was searched for. Namely, an incomplete proof

$$\begin{array}{c} \Sigma \mid \Delta_L \vdash \mathsf{true} \wedge \mathsf{emp} \\ \vdots \\ \mathsf{emp} \mid \Delta_1 \vdash \Delta_2 \end{array}$$

can be transformed into the desired proof of $\Delta_1 \vdash \Delta_2 * \Delta_L$ by spatially adding Δ_L to the right hand side at every proof step, and in addition, allowing $*$-introduction of Δ_L at the top of the proof.

Bi-abductive frame inference. The proof search in bi-abductive frame inference is performed using sequents $\Sigma \mid \Delta_A \vdash \Delta_G \dashv \Delta_M$, where Δ_A is the assumed formula, Δ_G is the goal formula, Σ is the subtracted (spatial) formula and Δ_M represents the anti-frame part of the formula. The sequents that the prover searches for in this case are of the form $\Sigma \mid \Delta_L \vdash \mathsf{true} \wedge \mathsf{emp} \dashv \Delta_M$. The frame and the anti-frame are formed from a disjunction of all leftover pairs of formulae Δ_L and Δ_M.

Semantics of our bi-abductive proof sequents can be described in the following way. Assume that the sequent $\Sigma \mid \Delta_A \vdash \Delta_G \dashv \Delta_M$ appears in the proof search that has started from a symbolic execution pre-state (Δ_H, Δ_P). Then the semantics of the sequent is given by

$$\Sigma * \Delta_A * \Delta_M * \Delta_P \Rightarrow \Sigma * \Delta_G * \Delta_H.$$

In other words, one can see the symbolic execution with bi-abductive frame inference as performing a proof search for the given sequence of statements using the sequents of the form

$$\Sigma \mid \Delta_A \vdash \Delta_G \dashv \Delta_M \dashv\vdash \Delta_H, \Delta_P$$

with the above semantics.

Deciding implication. To decide the implication the prover works like in the frame inference case but searches for sequents of the form $\Sigma \mid \mathsf{emp} \vdash \mathsf{true} \wedge \mathsf{emp}$.

5.2 Logic rules

The prover engine of coreStar is designed to be agnostic to specific details of the underlying logical theory. Except for basic constants true and emp, equalities, disequalities and the multiplicative connective, no other pure or spatial predicates are predefined in coreStar. Support for a particular version of separation logic is provided by specifying rewrite and proof rules in input files. This allows coreStar to deal not just with heap-specific reasoning (like most tools for shape analysis with separation logic do) but to reason in general about any objects that can be represented by means of abstract predicates [21] like e.g., threads, locks, abstract datatypes, etc.

Proof rules. Some general structural rules that need to be used with any kind of separation logic proof system are hard-coded into the prover. However, the rules that define how the reasoning within a particular logic theory is performed are specified externally and have to be provided by the user.[5] coreStar loads the proof rules into the separation logic prover and during the proof search applies the rules in order as they are specified in the input file. coreStar proof rules are specified using the syntactic construct `rule` [13]:

```
rule X: | Concl-L |- Concl-R -| Concl-A
if SubtFPre | Prem-L |- Prem-R -| Prem-A
```

The keyword `if` separates the conclusion from the premise. Mathematically this syntax corresponds to

$$\frac{\Sigma * \texttt{SubtFPre} \mid \Delta_A * \texttt{Prem-L} \vdash \Delta_G * \texttt{Prem-R} \dashv \Delta_M * \texttt{Prem-A}}{\Sigma \mid \Delta_A * \texttt{Concl-L} \vdash \Delta_G * \texttt{Concl-R} \dashv \Delta_M * \texttt{Concl-A}}\ \text{X},$$

for some Σ, Δ_A, Δ_G and Δ_M. Notice that this means that proof rules are implicitly framed – everything that is not mentioned in the definition of the proof rule is left unchanged.

[5] At the moment of writing this paper coreStar does not check that user-defined rules are consistent. Therefore, the user should ensure the soundness of rules.

```
rule pto_match:  | x ↦ y  |-  x ↦ t
without  y ≠ t
if  x ↦ y  |  |-  y = t
or  x ↦ y  |  y = t  |-  -|  y = t
```

```
rule pto_missing:  |  |-  x ↦ y
if  |  |-  -|  x ↦ y
```

Fig. 3: Proof rules for the points-to predicate.

Example. The coreStar proof rules for bi-abductive reasoning with standard separation logic points-to predicate are shown in Fig.3. The empty parts of the sequents stand for empty heap emp (and therefore are simply preserved), and the question mark is used to denote variables that can be unified with any expression.

The rule `pto_match` says if we have a heap cell with the same address in both the assumed and the goal formula of an entailment, then move the cell to the matched (subtracted) formula, and either add a proof obligation that the corresponding values are the same to the goal formula, or abduct this equality and add it to the assumption and the anti-frame. The `without` clause prevents firing of the rule if we already know the cells have different values (this way we can avoid applying the rule infinitely). The rule `pto_missing` performs abduction of a heap cell and adds it to the anti-frame in case the rule `pto_match` did not fire and we still have the heap cell in the obligation.

Rewrite rules. The prover also can be extended with rewrite rules which are used to simplify terms. For example, the following rewrite rules would be used to simplify list terms:

`rewrite cons_hd:` $hd(cons(x, y)) = x$
`rewrite cons_tl:` $tl(cons(x, y)) = y$

6 Abstraction

Symbolic execution alone does not converge in many cases. To ensure termination, symbolic states need to be abstracted. In coreStar, abstraction is invoked on abs statements. Abstraction is critical for the success of the symbolic execution. If abstraction forgets too much information then next steps of symbolic execution may fail due to too weak symbolic state. If abstraction keeps too much information then the computation may never converge.

6.1 Abstraction rules

Abstraction rules help the convergence of coreStar's computation by syntactic rewriting of predicates. The idea is that an abstraction rule simplifies a formula so that it remains within a restricted class of heaps for which simplification is known to converge. Abstraction rules are of the form:

$$\frac{\text{condition}}{\Delta_H * \Delta'_H \rightsquigarrow \Delta_H * \Delta''_H} \text{ (Abs Rule)}$$

Heap Δ'_H gets replaced by Δ''_H if the condition holds. Δ''_H should be more abstract (simpler) than Δ'_H since some unnecessary information is removed (abstracted away). Heap Δ_H is an arbitrary context preserved by the abstraction rule.

Given a set of abstraction rules, coreStar tries to use any rules that can be applied to a heap. When no rules are applicable the resulting heap is maximally abstracted. Note that to ensure termination of this strategy, abstraction rules should be chosen so that each application strictly simplifies the heap. Moreover, for soundness, the abstraction rules *must* be true implications in separation logic. When designing abstraction rules checking this implication gives an easy sanity condition for the soundness of the resulting fixed-point computation.

Example. The abstraction rules used for linked lists are based on how programmers typically deal with linked list programs. For example, suppose we had a predicate $\mathsf{node}(x, y)$ representing a node at address x with the next pointer y, and a predicate $\mathsf{lseg}(x, y)$ representing a linked list starting at y and ending with a pointer to y. We might want to have the following rules for abstraction [14]:

$$\exists x' \,.\, \mathsf{node}(x, x') \ast \mathsf{node}(x', \mathsf{nil}) \rightsquigarrow \mathsf{lseg}(x, \mathsf{nil})$$
$$\exists x' \,.\, \mathsf{lseg}(x, x') \ast \mathsf{node}(x', \mathsf{nil}) \rightsquigarrow \mathsf{lseg}(x, \mathsf{nil})$$
$$\exists x' \,.\, \mathsf{lseg}(x, x') \ast \mathsf{lseg}(x', \mathsf{nil}) \rightsquigarrow \mathsf{lseg}(x, \mathsf{nil})$$

In coreStar, the last rule, for instance, would be specified as

$$\frac{\_x \notin Context \cup \{x\}}{\mathsf{lseg}(x, \_x) \ast \mathsf{lseg}(\_x, \mathsf{nil}) \rightsquigarrow \mathsf{lseg}(x, \mathsf{nil})}$$

The heap Δ_H is implicitly added to both sides of $\rightsquigarrow$. The condition in the rule says that $\_x$ does not occur syntactically in the rest of the heap (i.e., $\_x \notin Var(\Delta_H)$) and that x cannot be instantiated to $\_x$.

7 Conclusions and future work

The tool coreStar retains the Java-agnostic parts of the program verifier jStar — a prover and a symbolic interpreter. The prover answers three types of queries: Given Δ_1 and Δ_2, (1) find Δ_A and Δ_F such that $\Delta_1 \ast \Delta_A \vdash \Delta_2 \ast \Delta_F$ holds, (2) find Δ_F such that $\Delta_1 \vdash \Delta_2 \ast \Delta_F$ holds, and (3) decide $\Delta_1 \vdash \Delta_2$. Here, Δs are separation logic formulas. A few symbols, such as equality and emp, are interpreted by the prover directly; a few other symbols, such as addition, are interpreted by an SMT solver, which is asked to reason about the pure parts of the formulas; all other symbols, including predicates like points-to, are interpreted according to a set of logical rules provided by the user.

The symbolic interpreter works on programs written in coreStarIL, which is a very simple language. Its control flow is unstructured and the only interesting statement is the specification assignment. The interpreter (tries to) answer two types of questions: Given a program P with precondition Δ_1 and postcondition Δ_2, (4) find Δ_A such that $\{\Delta_1 \ast \Delta_A\}\, P\, \{\Delta_2\}$ is a valid Hoare triple

and (5) decide $\{\Delta_1\}\, P\, \{\Delta_2\}$. In both cases, it over-approximates the possible executions according to a set of user-provided abstraction rules.

The interface to the prover (tasks 1–3 above) is a fairly stable OCaml module signature; the interface to the symbolic interpreter (tasks 4 and 5 above) is coreStarIL. We believe that many tools based on separation logic contain implementations of special cases of tasks 1–5. For example, Sec. 3.1 essentially shows how the Smallfoot tool [4] could be reimplemented as a small frontend for coreStar.

Much of the burden of ensuring soundness, however, remains with the frontend, because coreStar believes, without checking, that all the (logical and abstraction) rules it is given are sound. We are exploring several approaches to make coreStar less gullible. One option is to formalise coreStar's proof system in a higher-order prover and accept only rules that come with a soundness proof. Another option is to design a higher level language for rules that curtails expressivity such that the soundness of logical rules is decidable. We could also restrict abstraction rules so that their confluence and termination is decidable.

On the other hand, there are situations where we feel that the current abstraction rules are not flexible enough, which is why coreStar has a prototype mechanism to allow plugging in arbitrary abstract domains. When such plugins are in use, how much coreStar trusts its user is not the only issue — now it becomes important how much the user should trust coreStar. To tackle this problem, we are considering generating proofs (and requiring plugins to provide proofs) that can be checked easily.

Currently, coreStar provides no support for verifying large programs. For example, in jStar it is the frontend's responsibility to encode method calls as coreStarIL specification assignments. Once coreStarIL has procedures and procedure calls, coreStar should implement several global analyses, such as specification inference for mutually recursive procedures via bi-abduction [9] or RHS [22].

References

1. Michael Barnett, Bor-Yuh Evan Chang, Robert DeLine, Bart Jacobs, and K. Rustan M. Leino. Boogie: A modular reusable verifier for object-oriented programs. In *FMCO*, pages 364–387, 2005.
2. Michael Barnett and K. Rustan M. Leino. Weakest-precondition of unstructured programs. In *PASTE*, pages 82–87, 2005.
3. Mike Barnett, K. Rustan M. Leino, and Wolfram Schulte. The spec# programming system: An overview. In *CASSIS*, 2004.
4. Josh Berdine, Cristiano Calcagno, and Peter W. O'Hearn. Smallfoot: Modular automatic assertion checking with separation logic. In *FMCO*, pages 115–137, 2005.
5. Josh Berdine, Cristiano Calcagno, and Peter W. O'Hearn. Symbolic execution with separation logic. In *APLAS*, pages 52–68, 2005.
6. Josh Berdine, Byron Cook, and Samin Ishtiaq. Slayer: Memory safety for systems-level code. In *CAV*, 2011.
7. Matko Botinčan, Mike Dodds, Alastair F. Donaldson, and Matthew J. Parkinson. Proving the safety of asynchronous memory operations in multicore programs. *Submitted*.

8. Matko Botinčan, Mike Dodds, Alastair F. Donaldson, and Matthew J. Parkinson. Automatic safety proofs for asynchronous memory operations. In *PPOPP*, pages 313–314, 2011.
9. Cristiano Calcagno, Dino Distefano, Peter W. O'Hearn, and Hongseok Yang. Compositional shape analysis by means of bi-abduction. In *POPL*, pages 289–300, 2009.
10. Wei-Ngan Chin, Cristina David, Huu Hai Nguyen, and Shengchao Qin. Automated verification of shape, size and bag properties. In *ICECCS*, pages 307–320, 2007.
11. Ernie Cohen, Markus Dahlweid, Mark A. Hillebrand, Dirk Leinenbach, Michal Moskal, Thomas Santen, Wolfram Schulte, and Stephan Tobies. Vcc: A practical system for verifying concurrent c. In *TPHOLs*, pages 23–42, 2009.
12. Edsger W. Dijkstra. Guarded commands, nondeterminacy and formal derivation of programs. *Commun. ACM*, 18(8):453–457, 1975.
13. Dino Distefano, Mike Dodds, and Matthew J. Parkinson. How to verify java program with jStar: a tutorial. `http://www.jstarverifier.org/jstar.tutorial.pdf`.
14. Dino Distefano, Peter W. O'Hearn, and Hongseok Yang. A local shape analysis based on separation logic. In *TACAS*, pages 287–302, 2006.
15. Dino Distefano and Matthew J. Parkinson. jStar: towards practical verification for java. In *OOPSLA*, pages 213–226, 2008.
16. Cormac Flanagan and James B. Saxe. Avoiding exponential explosion: generating compact verification conditions. In *POPL*, pages 193–205, 2001.
17. Alexey Gotsman, Josh Berdine, and Byron Cook. Interprocedural shape analysis with separated heap abstractions. In *SAS*, pages 240–260, 2006.
18. Bart Jacobs, Jan Smans, and Frank Piessens. A quick tour of the verifast program verifier. In *APLAS*, pages 304–311, 2010.
19. K. Rustan M. Leino. Dafny: An automatic program verifier for functional correctness. In *LPAR*, pages 348–370, 2010.
20. K. Rustan M. Leino and Philipp Rümmer. A polymorphic intermediate verification language: Design and logical encoding. In *TACAS*, pages 312–327, 2010.
21. Matthew J. Parkinson and Gavin M. Bierman. Separation logic and abstraction. In *POPL*, pages 247–258, 2005.
22. Thomas W. Reps, Susan Horwitz, and Shmuel Sagiv. Precise interprocedural dataflow analysis via graph reachability. In *POPL*, pages 49–61, 1995.
23. John C. Reynolds. Separation logic: A logic for shared mutable data structures. In *LICS*, 2002.
24. Stephan van Staden and Cristiano Calcagno. Reasoning about multiple related abstractions with multistar. In *OOPSLA*, pages 504–519, 2010.
25. Hongseok Yang, Oukseh Lee, Josh Berdine, Cristiano Calcagno, Byron Cook, Dino Distefano, and Peter W. O'Hearn. Scalable shape analysis for systems code. In *CAV*, pages 385–398, 2008.