Constraint-based Program Reasoning with Heaps and Separation

Gregory J. Duck, Joxan Jaffar, and Nicolas C. H. Koh

Department of Computer Science, National University of Singapore
{gregory, joxan}@comp.nus.edu.sg, kchuenho@dso.org.sg

Abstract. This paper introduces a constraint language $\mathcal{H}$ for *finite partial maps* (a.k.a. *heaps*) that incorporates the notion of *separation* from *Separation Logic*. We use $\mathcal{H}$ to build an extension of *Hoare Logic* for reasoning over *heap manipulating programs* using (constraint-based) *symbolic execution*. We present a sound and complete algorithm for solving quantifier-free (QF) $\mathcal{H}$-formulae based on *heap element propagation*. An implementation of the $\mathcal{H}$-solver has been integrated into a *Satisfiability Modulo Theories* (SMT) framework. We experimentally evaluate the implementation against *Verification Conditions* (VCs) generated from symbolic execution of large (heap manipulating) programs. In particular, we mitigate the *path explosion problem* using subsumption via interpolation – made possible by the constraint-based encoding.

Keywords: Heap Manipulating Programs, Symbolic Execution, Separation Logic, Satisfiability Modulo Theories, Constraint Handling Rules.

1 Introduction

An important part of reasoning over *heap manipulating programs* is the ability to specify properties local to *separate* (i.e. non-overlapping) regions of memory. Most modern formalisms, such as Separation Logic [20], Region Logic [2], and (Implicit) Dynamic Frames [16][22], incorporate some encoding of separation. Separation Logic [20] explicates separation between regions of memory through *separating conjunction* ($*$). For example, the *Separation Logic formula* $\mathbf{list}(l)*\mathbf{tree}(t)$ represents a program *heap* comprised of two separate sub-heaps: one containing a *linked-list* and the other a *tree* data-structure.

In this paper we explore a reformulation of Separation Logic in terms of a first-order *constraint language* $\mathcal{H}$ over *heaps* (i.e. finite partial maps between pointers and values). Under this approach, separating conjunction ($*$) is re-encoded as a *constraint* $H \doteq H_1 * H_2$ between heaps, indicating that: (1) heaps H_1 and H_2 are *separate* (i.e. disjoint domains) and (2) H is the *heap union* of H_1 and H_2. We can therefore re-encode the above Separation Logic formula as $\mathbf{list}(l, L) \wedge \mathbf{tree}(t, T) \wedge \bar{\mathcal{H}} \doteq L*T$ where **list** and **tree** are redefined to be predicates over heaps, and the special variable $\bar{\mathcal{H}}$ represents the *global heap* at the program point where it appears. We can also represent a *singleton heap* as a constraint $\bar{\mathcal{H}} \doteq (p \mapsto v)$.

The motivation behind $\mathcal{H}$ is to lift some of the benefits of Separation Logic to constraint-based reasoning techniques for heap manipulating programs, such as *constraint-based symbolic execution*. Our method is based on an extension of *Hoare Logic* [11] defined in terms of the constraint language $\mathcal{H}$. Whilst Separation Logic guarantees *total correctness* w.r.t. *memory safety* (e.g. no memory errors such as dereferencing dangling pointers, etc.), our reformulation allows for weaker axiomatizations, such as a version that drops the memory-safety requirement. This allows for a *Strongest Post Condition* (SPC) *predicate transformer semantics* [7] to be defined in terms of $\mathcal{H}$, which forms the basis of symbolic execution. The resulting *Verification Conditions* (VCs) can then be discharged using a suitable $\mathcal{H}$-*constraint solver/theorem prover*. This is illustrated with a simple example:

Example 1 (Heap Equivalence). Consider the following Hoare triple:

$$\{H = \bar{\mathcal{H}}\}\ x := \mathbf{alloc}(); \mathbf{free}(x)\ \{H = \bar{\mathcal{H}}\} \tag{1}$$

This triple states that the global heap *before* the code fragment is equal to the heap *after* the fragment, i.e. the global heap is *unchanged*. Here H is a *ghost variable* representing the initial state of the global heap $\bar{\mathcal{H}}$. Symbolic execution of the precondition $P \equiv (H = \bar{\mathcal{H}})$ yields the following $\mathcal{H}$-constraints:

$$Q \quad \equiv \quad \big(H = H_0 \wedge \underline{H_1 \doteq (x \mapsto \_) * H_0} \wedge \underline{H_1 \doteq (x \mapsto \_) * \bar{\mathcal{H}}}\big)$$

Here H_0 and H_1 represent the initial and intermediate values for $\bar{\mathcal{H}}$ respectively. The underlined $\mathcal{H}$-constraints encode the **alloc**() and **free**() respectively. Next we can employ an $\mathcal{H}$-constraint solver to prove that the postcondition is implied by Q, i.e. the *Verification Condition* (VC) $Q \rightarrow H = \bar{\mathcal{H}}$ holds, thereby proving the triple (1) valid. □

In order to discharge the VCs generated from symbolic execution we need a solver for the resulting $\mathcal{H}$-formulae. For this we present a simple decision procedure for *Quantifier Free* (QF) $\mathcal{H}$-formulae based on the idea of *heap membership propagation*. We show that the algorithm is both sound and complete, and is readily implementable using *Constraint Handling Rules* (CHR) [10]. We present an implementation of an $\mathcal{H}$-solver that has been integrated into a *Satisfiability Modulo Theories* (SMT) framework using SMCHR [8]. Our decision procedure is related to established algorithms for finite sets.

We use the $\mathcal{H}$-solver as the basis of a simple program verification tool using symbolic execution. In contrast to Separation Logic-based symbolic execution [4], which is based on a set of *rearrangement rules*, our version is based on constraint solving using the $\mathcal{H}$-solver as per Example 1 above. Our encoding allows for some optimization. Namely, we mitigate the *path explosion problem* of symbolic execution by employing *subsumption* via *interpolation* [14][17] techniques.

This paper is organized as follows: Section 2 introduces Hoare and Separation Logic, Section 3 formally introduces the $\mathcal{H}$-language, Section 4 introduces an extension of Hoare Logic based on the $\mathcal{H}$-language, Section 5 presents an $\mathcal{H}$-solver algorithm and implementation, and Section 6 experimentally evaluates the implementation. In summary, the contributions of this paper are the following:

– We define the $\mathcal{H}$-language that encodes separation as a constraint between heaps. We show that satisfiability of quantifier-free $\mathcal{H}$-formulae is decidable, and present a complete algorithm for solving $\mathcal{H}$-formulae.
– We present an extension of Hoare Logic based on the $\mathcal{H}$-language. Our extension is similar to Separation Logic, but allows for strongest post conditions, and is therefore suitable for program reasoning via constraint-based symbolic execution.
– We present an implementation of the $\mathcal{H}$-solver that has been integrated into an SMT framework. We experimentally evaluate the solver against VCs generated from symbolic execution of heap manipulating programs.

2 Preliminaries

This section presents a brief overview of Hoare and Separation Logic.

Hoare Logic [11] is a formal system for reasoning about program correctness. Hoare Logic is defined in terms of axioms over *triples* of the form $\{\phi\}\ C\ \{\varphi\}$, where ϕ is the *pre-condition*, φ is the *post-condition*, and C is some code fragment. Both ϕ and φ are formulae over the *program variables* in C. The meaning of the triple is as follows: for all program states σ_1, σ_2 such that $\sigma_1 \models \phi$ and executing σ_1 through C derives σ_2, then $\sigma_2 \models \varphi$. For example, the triple $\{x < y\}\ x := x + 1\ \{x \leq y\}$ is *valid*. Note that under this definition, a triple is automatically valid if C is non-terminating or otherwise has undefined behavior. This is known as *partial correctness*.

Separation Logic [20] is a popular extension of Hoare Logic for reasoning over *heap manipulating programs*. Separation Logic extends predicate calculus with new logical connectives (namely *empty heap* (**emp**), *singleton heap* ($p \mapsto v$), and *separating conjunction* ($H_1 * H_2$)) such that the structure of assertions reflects the structure of the underlying heap. For example, the pre-condition in the valid Separation Logic triple $\{x \mapsto \_ * y \mapsto 2\}\ [x] := [y]+1\ \{x \mapsto 3 * y \mapsto 2\}$ represents a heap comprised of two *disjoint singleton* heaps, indicating that both x and y are *allocated* and that location y points to the value 2. Here the notation $[p]$ represents pointer dereference. In the post-condition we have that x points to value 3, as expected. Separation Logic also allows *recursively-defined* heaps for reasoning over data-structures, such as $\mathbf{list}(l)$ and $\mathbf{tree}(t)$ from Section 1.

Separation Logic triples also have a slightly different meaning versus Hoare triples regarding *memory-safety*. A Separation Logic triple $\{\phi\}\ C\ \{\varphi\}$ additionally guarantees that any state satisfying ϕ will not cause a memory access violation in C. For example, the triple $\{\mathbf{emp}\}\ [x] := 1\ \{x \mapsto 1\}$ is invalid since x is a dangling pointer in any state satisfying the pre-condition.

3 Heaps with Separation

This section formally introduces the syntax and semantics of heaps with separation, which we denote by $\mathcal{H}$, that encodes some of the logical connectives

of Separation Logic. We assume as given a countably infinite set Values denoting *values*, e.g. $\mathsf{Values} = \mathbb{Z}$. A *heap* is a *finite partial map* between Values, i.e. $\mathsf{Heaps} = \mathsf{Values} \rightharpoonup_{\text{fin}} \mathsf{Values}$. This is the same definition as used by Separation Logic. Given a heap $h \in \mathsf{Heaps}$ with domain $D = dom(h)$, we sometimes treat h as the set of pairs $\{(p, v) \mid p \in D \wedge v = h(p)\}$.

The $\mathcal{H}$-language is the first-order language over heaps defined as follows:

Definition 1 (Heap Language). We define the $\mathcal{H}$-signature $\Sigma_{\mathcal{H}}$ as follows:
- *sorts*: Values, Heaps;
- *constants*: (empty heap) $\emptyset$ of sort Heaps;
- *functions*: (singleton heap) $(\_ \mapsto \_)$ of sort $\mathsf{Values} \times \mathsf{Values} \mapsto \mathsf{Heaps}$.
- *predicates*: (heap constraint) $(\_*\ldots*\_ \doteq \_*\ldots*\_)$ of sort $\mathsf{Heaps} \times \cdots \times \mathsf{Heaps} \mapsto \{\mathit{true}, \mathit{false}\}$.

The $\mathcal{H}$-language is the *first-order language* over $\Sigma_{\mathcal{H}}$. □

Example 1 used heap constraints of the form $H \doteq H_1 * H_2$, where H, H_1, and H_2 are variables. Throughout this paper we shall use upper-case letters H, I, J, etc., to denote *heap variables*, and lower-case letters p, v, etc., for *value variables*.

A *valuation* s (a.k.a. *variable assignment*) is a function mapping values to $\mathsf{Values} \cup \mathsf{Heaps}$. We define the semantics of the $\mathcal{H}$-language as follows:

Definition 2 (Heap Interpretation). Given a valuation s, the $\mathcal{H}$-interpretation $\mathcal{I}$ is a $\Sigma_{\mathcal{H}}$-interpretation such that:
- $\mathcal{I}(v, s) = s(v)$, where v is a variable;
- $\mathcal{I}(\emptyset, s) = \emptyset$ (as a Heap);
- $\mathcal{I}(p \mapsto v, s) = \{(q, w)\}$ where $q = \mathcal{I}(p, s)$ and $w = \mathcal{I}(v, s)$;
- $\mathcal{I}(H_1 * \ldots * H_i \doteq H_{i+1} * \ldots * H_n, s) = \mathit{true}$ iff for $h_i = \mathcal{I}(H_i, s)$ we have that:
  1. $dom(h_1) \cap \ldots \cap dom(h_i) = \emptyset$ and $dom(h_{i+1}) \cap \ldots \cap dom(h_n) = \emptyset$; and
  2. $h_1 \cup \ldots \cup h_i = h_{i+1} \cup \ldots \cup h_n$ □

Note that we treat each configuration of $(*)$ and $(\doteq)$ as a distinct predicate. Intuitively, a constraint like $H \doteq H_1 * H_2$ treats $(*)$ in essentially the same way as *separating conjunction* from Separation Logic, except that we give a name H to the conjoined heaps $H_1 * H_2$.

We define $\models_{\mathcal{H}} \ldots [s]$ as the *satisfaction relation* such that $\models_{\mathcal{H}} \phi\ [s]$ holds iff $\mathcal{I}(\phi, s) = \mathit{true}$ for all heap formulae ϕ. We also say that ϕ is *valid* if $\models_{\mathcal{H}} \phi\ [s]$ holds for all s, and *satisfiable* if $\models_{\mathcal{H}} \phi\ [s]$ holds for at least one s.

3.1 Normalization

In the absence of quantifiers, we can restrict consideration of $\mathcal{H}$-formula to a subset in *normal form* defined as follows:

Definition 3 (Normal Form). A quantifier-free (QF) $\mathcal{H}$-formula ϕ is in *normal form* if (1) all heap constraints are restricted to three basic forms:

$$H \doteq \emptyset \qquad H \doteq (p \mapsto v) \qquad H \doteq H_1 * H_2$$

where p, v, H, H_1, and H_2 are distinct variables, and (2) there are no negated heap constraints. □

$$
\begin{array}{rcll}
H \doteq E_1 * E_2 * S & \longrightarrow & H' \doteq E_1 * E_2 \wedge H \doteq H' * S & \\
H \doteq E_1 * E_2 & \longrightarrow & H' \doteq E_1 \wedge H \doteq H' * E_2 & (E_1 \text{ non-variable}) \\
H \doteq H_1 * E_2 & \longrightarrow & H' \doteq E_2 \wedge H \doteq H_1 * H' & (E_2 \text{ non-variable}) \\
H_1 \doteq H_2 & \longrightarrow & H' \doteq \emptyset \wedge H_1 \doteq H_2 * H' & \\
H \not\doteq E_1 * E_2 * S & \longrightarrow & \vee \begin{cases} E_1 \doteq (s \mapsto t) * H_1' \wedge E_2 \doteq (s \mapsto u) * H_2' \\ H' \doteq E_1 * E_2 \wedge H \not\doteq H' * S \end{cases} & \\
H \not\doteq E_1 * E_2 & \longrightarrow & H' \doteq E_1 \wedge H \not\doteq H' * E_2 & (E_1 \text{ non-variable}) \\
H \not\doteq H_1 * E_2 & \longrightarrow & H' \doteq E_2 \wedge H \not\doteq H_1 * H' & (E_2 \text{ non-variable}) \\
H \not\doteq \emptyset & \longrightarrow & H \doteq (s \mapsto t) * H' & \\
H \not\doteq (p \mapsto v) & \longrightarrow & \vee \begin{cases} H \doteq \emptyset \\ H \doteq (s \mapsto t) * H' \wedge (p \neq s \vee v \neq t) \end{cases} & \\
H \not\doteq H_1 * H_2 & \longrightarrow & \vee \begin{cases} H_1 \doteq (s \mapsto t) * H_1' \wedge H_2 \doteq (s \mapsto u) * H_2' \\ H' \doteq H_1 * H_2 \wedge H \not\doteq H' \end{cases} & \\
H_1 \not\doteq H_2 & \longrightarrow & \vee \begin{cases} H_1 \doteq (s \mapsto t) * H_1' \wedge H_1 \doteq (s \mapsto u) * H_2' \wedge t \neq u \\ H_1 \doteq I * H_1' \wedge H_2 \doteq I * H_2' \wedge H' \doteq H_1' * H_2' \wedge H' \not\doteq \emptyset \end{cases} &
\end{array}
$$

Fig. 1. $\mathcal{H}$-formulae normalization rewrite rules.

Any given QF $\mathcal{H}$-formula ϕ can be rewritten into *normal form* using the following steps: (1) push negation inwards using De Morgan's laws, and (2) transform the resulting formula using the rewrite rules from Figure 1. Here each rewrite rule is of the form ($\mathit{head} \longrightarrow \mathit{body}$), and E_i runs over *heap expressions* (H, $\emptyset$, $(p \mapsto v)$), S runs over $(*)$-sequences of heap expressions (E, $E{*}E$, etc.), and everything else runs over the variable symbols. A variable that appears in a rule body, but not the rule head, is taken to represent a fresh variable symbol that is introduced each time the rule is applied. For brevity we omit some rules, namely: normalizing the RHS of a ($\doteq$) to a heap variable (as this mirrors the LHS rules), and making variables unique. The main result for normalization is as follows:

Proposition 1 (Normal Form). *For all QF $\mathcal{H}$-formulae ϕ there exists a QF $\mathcal{H}$-formula φ such that (1) φ is in normal form and (2): for all valuations s there exists a valuation s' such that $\models_{\mathcal{H}} \phi\ [s]$ iff $\models_{\mathcal{H}} \varphi\ [s']$ and $s(v) = s'(v)$ for all $v \in \mathit{vars}(\phi)$.*

Proof. (Sketch) By the correctness of, and induction over, the normalization steps from Figure 1. □

Proposition 1 means that, at the expense of an increased formula size, we need only consider a limited subset of the $\mathcal{H}$-language that lacks negation.

3.2 Extensions

We may extend Definitions 1 and 2 to include other kinds of heap constraints, such as:

- *Heap union* $H \doteq H_1 \sqcup H_2$ holds iff there exists a $h \in \mathsf{Heaps}$ such that $h = s(H_1) \cup s(H_2)$ as sets and $s(H) = h$.
- *Heap intersection* $H \doteq H_1 \sqcap H_2$ holds iff $s(H) = s(H_1) \cap s(H_2)$ as sets.
- *Heap subset* $H_1 \sqsubseteq H_2$ holds iff $s(H_1) \subseteq s(H_2)$ as sets.

These constraints can similarly be reduced to the normal form from Definition 3.

For some applications we may extend $\mathcal{H}$ with ad hoc *user-defined* heap constraints. For this we can use *Constraint Logic Programming* (CLP) [13] over $\mathcal{H}$, i.e. CLP($\mathcal{H}$). For example, the following CLP($\mathcal{H}$) predicate $\mathbf{list}(l, L)$ specifies a skeleton *list constraint* under the standard *least model semantics* of CLP:

$$\begin{aligned} \mathbf{list}(0, L) &\ \texttt{:-}\ L \doteq \emptyset \\ \mathbf{list}(l, L) &\ \texttt{:-}\ l \neq 0 \wedge L \doteq (l \mapsto n) * L' \wedge \mathbf{list}(n, L') \end{aligned}$$

We can similarly define predicates for trees and arrays. The inclusion of CLP predicates requires stronger reasoning power in contrast to the base $\mathcal{H}$-language. For this we can employ standard (yet incomplete) methods such as [15].

4 Program Reasoning with $\mathcal{H}$

The core motivation of the $\mathcal{H}$-language is reasoning over heap manipulating programs. For this we consider the following extensions of *Hoare Logic* [11].

4.1 Direct Separation Logic Encoding

Separation Logic [20] is itself an extension of Hoare Logic. Given the similarity in the heap representations, we can re-encode the axioms of Separation Logic directly into Hoare axioms over $\mathcal{H}$-formulae, as shown in Figure 2(B). Each axiom is defined in terms of one of five auxiliary constraints: namely $\mathsf{alloced}$, access, assign, alloc, and free defined in Figure 2(A), which are themselves defined in terms of $\mathcal{H}$-formulae. The $\mathsf{alloced}(H, x)$ constraint represents that pointer x is *allocated* in heap H, i.e. $H \doteq (x \mapsto v) * H'$ for some v and H'. The remaining auxiliary constraints encode a *heap manipulation statement* as an $\mathcal{H}$-formula. The statements are:

- *heap access* $(x := [y])$ sets x to be the value pointed to by y;
- *heap assignment* $([x] := y)$ sets the value pointed to by x to be y;
- *heap allocation* $(x{:=}\mathbf{alloc}())$ sets x to point to a freshly allocated heap cell.[1]
- *heap free* $\mathbf{free}(x)$ deallocates the cell pointed to by x.

These axioms manipulate the *global heap* that is represented by a *distinguished heap variable* $\widetilde{\mathcal{H}}$. Under this treatment, $\widetilde{\mathcal{H}}$ is an *implicit program variable*[2] of type

[1] Here we assume the (de)allocation of single heap cells. This can be generalized.

[2] The variable is "implicit" in the sense that it is not explicitly represented in the syntax of the programming language.

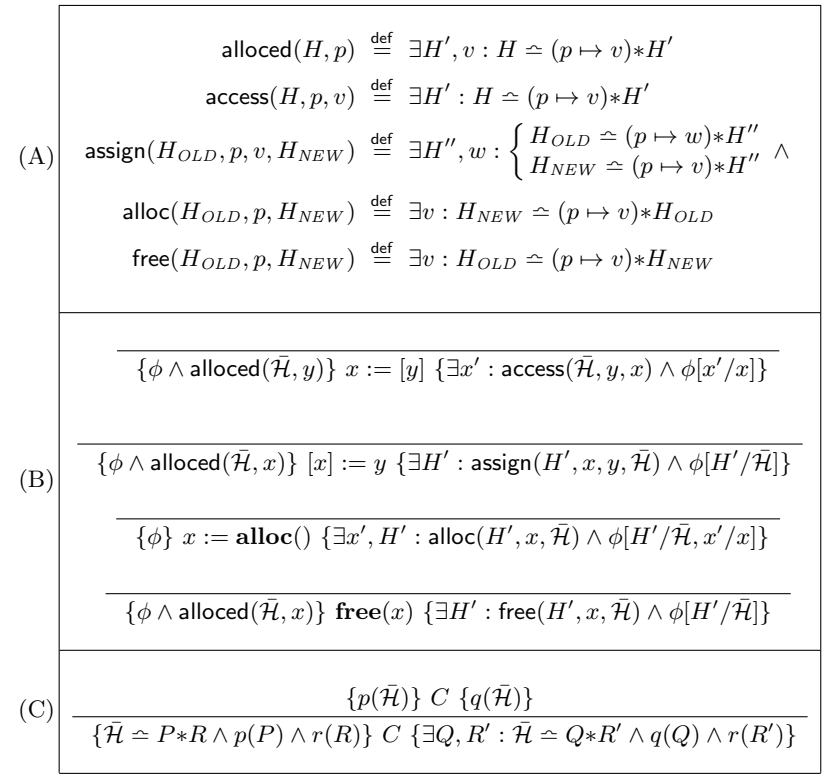
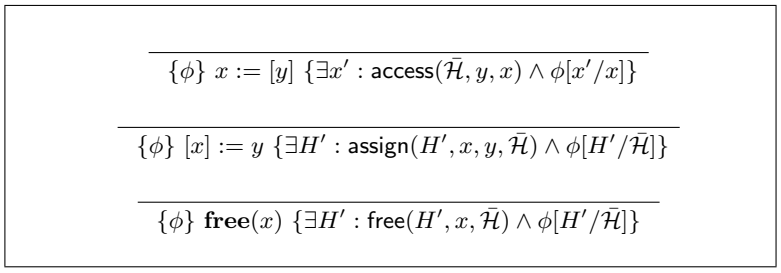


Fig. 2. (A) Auxiliary constraint definitions, (B) basic Hoare inference rules, and (C) the Frame Rule.

Heap that is assumed to be threaded throughout the program. Other axioms of Separation Logic, such as the *Frame Rule* [20], can similarly be re-encoded, as shown in Figure 2(C).

It is not surprising that Separation Logic can be re-formulated as Hoare axioms over the $\mathcal{H}$-language. However, there are some important differences to consider. Notably, the $\mathcal{H}$-encoding allows for *explicit heap variables* to express relationships *between* heaps across triples. In Example 1, we use the triple $\{H \doteq \bar{\mathcal{H}}\}\ C\ \{H \doteq \bar{\mathcal{H}}\}$ to express the property that the code fragment C does not change the global heap $\bar{\mathcal{H}}$ through an explicit variable H. Such a global property would require *second order* Separation Logic, e.g., $\forall h : \{h\}\ C\ \{h\}$. Furthermore, with explicit heap variables, we can strengthen the *Frame Rule* by R' for R in the post-condition of Figure 2(C).

The $\mathcal{H}$-based encoding tends to be more verbose compared to Separation Logic, which favors more concise formulae. Whilst not so important for *automated* systems, the $\mathcal{H}$-based encoding is likely less suitable for *manual* proofs of correctness.



$$\frac{}{\{\phi\}\ x := [y]\ \{\exists x' : \mathsf{access}(\bar{\mathcal{H}}, y, x) \wedge \phi[x'/x]\}}$$

$$\frac{}{\{\phi\}\ [x] := y\ \{\exists H' : \mathsf{assign}(H', x, y, \bar{\mathcal{H}}) \wedge \phi[H'/\bar{\mathcal{H}}]\}}$$

$$\frac{}{\{\phi\}\ \mathbf{free}(x)\ \{\exists H' : \mathsf{free}(H', x, \bar{\mathcal{H}}) \wedge \phi[H'/\bar{\mathcal{H}}]\}}$$

Fig. 3. Alternative Hoare inference rules.

4.2 Strongest Post-Condition Encoding

Separation Logic and the corresponding $\mathcal{H}$-encoding from Figure 2 (B) enforces *total correctness* w.r.t. *memory safety*. That is, a valid triple $\{\phi\}\ C\ \{\varphi\}$ additionally ensures that any state satisfying ϕ will not cause a *memory fault* (e.g. dereferencing a dangling pointer) when executed by C. This is enforced by the *access*, *assignment*, and *free* axioms of Figure 2 (B) by requiring that the pointer x be allocated in the global heap $\mathcal{H}$ in the pre-condition via the $\mathsf{alloced}(\bar{\mathcal{H}}, x)$ constraint.

Memory safety has implications for *forward reasoning* methods such as *symbolic execution*. For example, to symbolically execute a formula ϕ through an assignment $[x] := v$, we must first prove that $\phi \rightarrow \mathsf{alloced}(\bar{\mathcal{H}}, x)$. Such a proof can be arbitrarily difficult in general, e.g. for formulae with quantifiers or recursively-defined CLP($\mathcal{H}$) predicates. Furthermore, if memory safety is *not* a property of interest, this extra work is unnecessary.

By decoupling the heap representation ($\mathcal{H}$) from the logic, we can experiment with alternative axiomatizations. One such axiomatization that is *partially correct* modulo memory safety is shown in Figure 3.[3] This version drops the requirement that x be allocated in $\bar{\mathcal{H}}$ in the pre-condition, and therefore treats memory errors the same way as *undefined behavior* (or *non-termination*) in classic Hoare Logic.

There are several advantages to the weaker axiomatization of Figure 3. Firstly, the axioms of Figure 3 specify a *Strongest Post Condition* (SPC) *predicate transformer semantics* and is therefore immediately suitable for automated forward based reasoning techniques such as symbolic execution. This is in contrast to symbolic execution in Separation Logic [4] (or the corresponding axioms from Figure 2), where symbolic execution requires the $\mathsf{alloced}$ condition to be separately proven. The SPC axiomatization allows for weaker, more concise, specifications.

[3] The axiom for *heap allocation* is the same as Figure 2 (B).

Example 2 (Double List Reverse). For example, consider the following triples in the spirit of Example 1:

$$\{H \doteq \bar{\mathcal{H}}\}\ l := \mathbf{reverse}(\mathbf{reverse}(l))\ \{H \doteq \bar{\mathcal{H}}\} \quad (2)$$

$$\{\bar{\mathcal{H}} \doteq L{*}H' \wedge \mathbf{list}(L, l) \wedge H \doteq \bar{\mathcal{H}}\}\ l := \mathbf{reverse}(\mathbf{reverse}(l))\ \{H \doteq \bar{\mathcal{H}}\} \quad (3)$$

Both attempt to state the same property: that double *in-place list-reverse* leaves the global heap $\bar{\mathcal{H}}$ unchanged. Suppose that the only property of interest is the heap equivalence (i.e. not memory safety). Triple (2) is valid under the weaker Figure 3 axiomatization, but not the stronger Figure 2 (B) version which requires memory safety. The latter requires a more complex specification, such as Triple (3), where the recursively defined property $\mathbf{list}(L, l)$ ensures l points to a valid allocated list. □

There are also some disadvantages to consider. For obvious reasons, the SPC axiomatization is unsuitable if memory safety *is* a property of interest. Furthermore, the soundness of Separation Logic's *Frame Rule* (or Figure 2 (C)) depends on memory safety, and thus is not valid under the new interpretation. Therefore the SPC axiomatization is not suitable for Separation Logic-style *local reasoning* proofs. In essence, this is a trade-off between local reasoning vs. making symbolic execution "easier", highlighting the flexibility of our overall approach.

5 A Solver for $\mathcal{H}$-formulae

Automated symbolic execution depends on an $\mathcal{H}$-solver to discharge the generated *Verification Conditions* (VCs). In this section we present a simple, yet sound and complete, algorithm for solving the quantifier-free (QF) fragment of the $\mathcal{H}$-language.

Algorithm The $\mathcal{H}$-solver algorithm is based on the *propagation* of *heap membership* and *(dis)equality* constraints. Heap membership (a.k.a. heap element) is represented by an auxiliary $\mathsf{in}(H, p, v)$ constraint, which is defined as follows:

Definition 4 (Heap Membership). We extend Definitions 1 and 2 to include the *heap membership* constraint $\mathsf{in}(H, p, v)$ defined as follows:

$$\models_{\mathcal{H}} \mathsf{in}(H, p, v)\ [s] \quad \text{iff} \quad (s(p), s(v)) \in s(H)$$

where H, p, and v are variables. □

Heap element $\mathsf{in}(H, p, v)$ is analogous to set membership $x \in S$ from set theory. (Dis)equality is propagated via the usual $x = y$ and $x \neq y$ constraints.

The $\mathcal{H}$-solver operates over conjunctions of *normalized* $\mathcal{H}$-constraints as per Definition 3. Arbitrary QF $\mathcal{H}$-formula ϕ can be normalized to a φ using the rules from Figure 1, such that the solutions to ϕ and φ correspond as per Proposition 1. The arbitrary Boolean structure of φ can be handled using the *Davis-Putnam-Logemann-Loveland* (DPLL) algorithm [6] modulo the $\mathcal{H}$-solver.

$$\mathsf{in}(H,p,v) \wedge \mathsf{in}(H,p,w) \Longrightarrow v = w \quad (1)$$
$$H \doteq \emptyset \wedge \mathsf{in}(H,p,v) \Longrightarrow \mathit{false} \quad (2)$$
$$H \doteq (p \mapsto v) \Longrightarrow \mathsf{in}(H,p,v) \quad (3)$$
$$H \doteq (p \mapsto v) \wedge \mathsf{in}(H,q,w) \Longrightarrow p = q \wedge v = w \quad (4)$$
$$H \doteq H_1 * H_2 \wedge \mathsf{in}(H,p,v) \Longrightarrow \mathsf{in}(H_1,p,v) \vee \mathsf{in}(H_2,p,v) \quad (5)$$
$$H \doteq H_1 * H_2 \wedge \mathsf{in}(H_1,p,v) \Longrightarrow \mathsf{in}(H,p,v) \quad (6)$$
$$H \doteq H_1 * H_2 \wedge \mathsf{in}(H_2,p,v) \Longrightarrow \mathsf{in}(H,p,v) \quad (7)$$
$$H \doteq H_1 * H_2 \wedge \mathsf{in}(H_1,p,v) \wedge \mathsf{in}(H_2,q,w) \Longrightarrow p \neq q \quad (8)$$

Fig. 4. $\mathcal{H}$-solver CHR propagation rules.

We specify the $\mathcal{H}$-solver as a set of *Constraint Handling Rules* [10] with disjunction (CHR$^\vee$) [1] as shown in Figure 4. Here each rule (*Head* $\Longrightarrow$ *Body*) encodes *constraint propagation*, where the constraints *Body* are added to the store whenever a matching *Head* is found. Rule (1) encodes the functional dependency for finite partial maps; rules (2)–(4) encode propagation for *heap empty* $H = \emptyset$ and *heap singleton* $H \doteq (p \mapsto v)$ constraints; and rules (5)–(8) encode heap membership propagation through *heap separation* $H \doteq H_1 * H_2$ constraints. Most of these rules are self-explanatory, e.g., rule (6) states that if $H \doteq H_1 * H_2$ and $\mathsf{in}(H_1,p,v)$, then it must be the case that $\mathsf{in}(H,p,v)$, since H_1 is a sub-heap of H. We assume a complete solver for the underlying equality theory ($x = y$, $x \neq y$).

The $\mathcal{H}$-solver employs the standard CHR$^\vee$ execution algorithm with the rules from Figure 4. We shall present a semi-formal summary below. The input is a *constraint store* S defined to be a set[4] of constraints (representing a conjunction). Let Rules be the rules from Figure 4, then the algorithm $\mathsf{hsolve}(S)$ is recursively defined as follows:

- (*Propagation Step*) If there exists $R \in \mathsf{Rules}$ of the form ($h_1 \wedge \ldots \wedge h_n \Longrightarrow$ *Body*), a subset $\{c_1,\ldots,c_n\} \subseteq S$ of constraints, a subset $E \subseteq S$ of *equality* constraints, and a *matching substitution* θ such that: $E \rightarrow (\theta.h_i = c_i)$ for $i \in 1..n$ then rule R is *applicable* to the store S. We *apply* rule R as follows:
  - If *Body* = *false* then return *false*;
  - If *Body* $= d_1 \wedge \ldots \wedge d_m$ then return $\mathsf{hsolve}(S \cup \theta.\{d_1,\ldots,d_m\})$; else
  - If *Body* $= d_1 \vee \ldots \vee d_m$ then let $S_i := \mathsf{hsolve}(S \cup \theta.\{d_i\})$ for $i \in 1..m$. If there exists an $S_i \neq \mathit{false}$ then return S_i, else return *false*.
- Else if no such R exists, return S.

Propagation proceeds until failure occurs or a fixed point is reached.

Example 3 ($\mathcal{H}$-Solving). Consider the following goal G:

$$H \doteq (p \mapsto v) \wedge H \doteq I * J \wedge J \doteq (p \mapsto w) \wedge v \neq w$$

[4] We assume a set-based CHR semantics.

$\{\underline{H \doteq (p \mapsto v)}, H \doteq I * J, J \doteq (p \mapsto w), v \neq w\}$ (3)
$\{\underline{H \doteq (p \mapsto v)}, H \doteq I * J, \underline{J \doteq (p \mapsto w)}, v \neq w, \mathsf{in}(H, p, v)\}$ (3)
$\{H \doteq (p \mapsto v), \underline{H \doteq I * J}, J \doteq (p \mapsto w), v \neq w, \mathsf{in}(H, p, v), \underline{\mathsf{in}(J, p, w)}\}$ (7)
$\{H \doteq (p \mapsto v), H \doteq I * J, J \doteq (p \mapsto w), v \neq w, \underline{\mathsf{in}(H, p, v)}, \underline{\mathsf{in}(J, p, w)}, \underline{\mathsf{in}(H, p, w)}\}$ (1)
$\{H \doteq (p \mapsto v), H \doteq I * J, J \doteq (p \mapsto w), \underline{v \neq w}, \mathsf{in}(H, p, v), \mathsf{in}(J, p, w), \underline{v = w}\}$ (E)
false

Fig. 5. $\mathcal{H}$-solving constraint propagation steps.

We wish to show that this goal is unsatisfiable using the $\mathcal{H}$-solver from Figure 4. Initially the constraint store contains the initial goal G. Constraint propagation proceeds as shown in Figure 5. Here we apply rules (3), (3), (7), (1), (E) to the underlined constraint(s) in order, where (E) represents an inference made by the underlying equality solver. Propagation leads to failure, and there are no branches – therefore goal G is unsatisfiable. □

Since all the rules from Figure 4 are *propagation rules*, the solving algorithm $\mathsf{hsolve}(G)$ will always terminate with some final store S. The $\mathcal{H}$-solver is both sound and complete w.r.t. (un)satisfiability.

Proposition 2 (Soundness). *For all G, S, if $\mathsf{hsolve}(G) = S$, then for all valuations s, $\models_{\mathcal{H}} G\ [s]$ iff $\models_{\mathcal{H}} S\ [s]$.*

Proof. (Sketch) By the correctness of the rules from Figure 4 w.r.t. Definitions 2 and 4. □

Proposition 3 (Completeness). *For all G, S such that $\mathsf{hsolve}(G) = S$, then $\not\models_{\mathcal{H}} G\ [s]$ for all valuations s (i.e. G is unsatisfiable) iff $S = \mathit{false}$.*

Proof. (Sketch) The "⇐" direction follows from Proposition 2. We consider the "⇒" direction. The rest is proof by contrapositive: assuming $S \neq \mathit{false}$ we show that there exists a valuation s such that $\models_{\mathcal{H}} G\ [s]$. Let s_E be a valuation for the underlying equality subset of S over integer variables, then let $s(v) = s_E(v)$ for integer variables, and

$$s(H) = \{(s_E(p), s_E(v)) \mid \mathsf{in}(H, p, v) \in S\} \tag{4}$$

for all heap variables H. Assume that $\not\models_{\mathcal{H}} S\ [s]$. By case analysis of Definition 2 we find that a rule must be applicable:

- Case $s(H) \notin \mathsf{Heaps}$: Rule (1);
- Case $H \doteq \emptyset$ and $s(H) \neq \emptyset$: Rule (2);
- Case $H \doteq (p \mapsto v)$ and $s(H) \neq \{(p, v)\}$: Rules (3) or (4);
- Case $H \doteq H_1 * H_2$ and $s(H) \neq s(H_1) \cup s(H_2)$: Rules (5), (6), or (7);
- Case $H \doteq H_1 * H_2$ and $\mathit{dom}(s(H_1)) \cap \mathit{dom}(s(H_2)) \neq \emptyset$: Rule (8)

This contradicts the assumption that S is a final store, therefore if $S \neq \mathit{false}$ then $\models_{\mathcal{H}} S\ [s]$, and therefore $\models_{\mathcal{H}} G\ [s]$ by Proposition 2 completes the proof. □

$$H \not\simeq \emptyset \implies \mathsf{in}(H, s, t) \tag{9}$$

$$H \not\simeq (p \mapsto v) \implies \vee \begin{cases} H \simeq \emptyset \\ \mathsf{in}(H, s, t) \wedge (s \neq p \vee t \neq v) \end{cases} \tag{10}$$

$$H \not\simeq H_1 * H_2 \implies \vee \begin{cases} \mathsf{in}(H, s, t) \wedge \neg\mathsf{in}(H_1, s, t) \wedge \neg\mathsf{in}(H_2, s, t) \\ \mathsf{in}(H_1, s, t) \wedge \neg\mathsf{in}(H, s, t) \\ \mathsf{in}(H_2, s, t) \wedge \neg\mathsf{in}(H, s, t) \\ \mathsf{in}(H_1, s, t) \wedge \mathsf{in}(H_2, s, u) \end{cases} \tag{11}$$

$$\mathsf{access}(H, p, v) \iff \mathsf{in}(H, p, v) \tag{12}$$

$$\mathsf{assign}(H_0, p, v, H_1) \implies \mathsf{in}(H_0, p, w) \wedge \mathsf{in}(H_1, p, v) \tag{13}$$

$$\mathsf{assign}(H_0, p, v, H_1) \wedge \mathsf{in}(H_0, q, w) \implies p = q \vee \mathsf{in}(H_1, q, w) \tag{14}$$

$$\mathsf{assign}(H_0, p, v, H_1) \wedge \mathsf{in}(H_1, q, w) \implies p = q \vee \mathsf{in}(H_0, q, w) \tag{15}$$

$$\mathsf{alloc}(H_0, p, H_1) \implies \mathsf{in}(H_1, p, v) \tag{16}$$

$$\mathsf{alloc}(H_0, p, H_1) \wedge \mathsf{in}(H_0, q, w) \implies p \neq q \wedge \mathsf{in}(H_1, q, w) \tag{17}$$

$$\mathsf{alloc}(H_0, p, H_1) \wedge \mathsf{in}(H_1, q, w) \implies p = q \vee \mathsf{in}(H_0, q, w) \tag{18}$$

$$\mathsf{free}(H_0, p, H_1) \iff \mathsf{alloc}(H_1, p, H_0) \tag{19}$$

Fig. 6. Extended $\mathcal{H}$-solver propagation rules.

The proof for Proposition 3 is constructive; namely, (4) can be used to construct a *solution* for a satisfiable goal G. Furthermore, we can combine the normalization of Proposition 1 and DPLL(hsolve) to derive a sound and complete algorithm for solving arbitrary QF $\mathcal{H}$-formulae ϕ.

5.1 Extensions

The propagation rules from Figure 4 define a solver for the base $\mathcal{H}$-language. We can use heap membership propagation to define rules for other kinds of $\mathcal{H}$-constraints, as shown in Figure 6.

Rules (9)–(11) handle the negations of the base $\mathcal{H}$-constraints from Definition 3. These rules are an alternative to the decomposition from Figure 1. We can also define rules for directly handling the auxiliary constraints from Figure 2 (A) for program reasoning. For example, rules (13)–(15) handle the $\mathsf{assign}(H_0, p, v, H_1)$ constraint. We similarly provide rules for the other auxiliary constraints. Here, variables appearing in a rule body but not in the rule head are interpreted the same way as with Figure 1.

6 Experiments

In this section we test an implementation of the $\mathcal{H}$-solver against *verification conditions* (VCs) derived from symbolic execution. We compare against Verifast [12]

(version 12.12), a program verification system based on Separation Logic. Our motivation for the comparison is: (1) Verifast is based on forward symbolic execution, and (2) Verifast incorporates the notion of separation (via Separation Logic). That said, the Verifast execution algorithm [12] is very different from the $\mathcal{H}$-solver.

We have implemented a version of the $\mathcal{H}$-solver as part of the *Satisfiability Modulo Constraint Handling Rules* (SMCHR) [8] system.[5] SMCHR is a *Satisfiability Modulo Theories* (SMT) framework that supports theory (T) solvers implemented in CHR. The SMCHR system also supports several "built-in" theories, such as a linear arithmetic solver based on [9], that can be combined with the $\mathcal{H}$-solver to handle the underlying (dis)equality constraints. The SMCHR system has also been extended to support *disjunctive propagators* [19] for rules with disjunctive bodies, such as Rule (5).

For these benchmarks we either restrict ourselves to the fragment of Verifast that is fully automatable, or we provide the minimal annotations where appropriate. For the $\mathcal{H}$-solver, we have implemented a prototype symbolic execution tool as a GCC plug-in. Our tool symbolically executes GCC's internal GIMPLE representation to generate *path constraints*. Given a *safety condition* φ, we generate the corresponding *verification condition* $(\exists \bar{x} : \phi) \models \varphi$, which is *valid* iff $\phi \wedge \neg\varphi$ is *unsatisfiable*. Here $\bar{x}$ represents existential variables introduced during symbolic execution. Unsatisfiability is tested for using the $\mathcal{H}$-solver.

A well-known problem with forward symbolic execution is the so-called *path explosion problem*. The number of paths through a (loop-free) program fragment can easily be exponential. We can mitigate this problem using *subsumption via interpolation* [14][17]. The basic idea is as follows: given a VC $\phi_1 \models \varphi$ that holds for path ϕ_1, we generate an *interpolant* ψ_1 for ϕ_1, that, by definition, satisfies $\phi_1 \models \psi_1 \models \varphi$. As symbolic execution continues, we can prune (subsume) all other paths with constraints ϕ_2 such that $\phi_2 \models \psi_1$. The key is that this pruning can occur *early*, as we construct the constraint for each path.

Our interpolation algorithm is based on an improved version of the *constraint deletion* idea from [14]. Given a path constraint $\phi = c_1 \wedge \ldots \wedge c_n$ we find a subset $I \subseteq \{c_1, \ldots, c_n\}$ such that $I \wedge \neg\varphi$ remains *unsatisfiable*. For this we simply re-use the SAT solver's *Unique Implication Point* UIP algorithm over the implication graph formed by the $\mathcal{H}$-solver propagation steps.

We test several programs that exhibit the path explosion problem. These include: subsets_N - sum-of-subsets size N; expr_N - simple virtual machine executing N instructions; stack_N - for all $M \leq N$, do N-pushes, then N-pops; filter_N - filter for TCP/IP packets; sort_N - bubble-sort of length N; search234_N - 234-tree search; insert234_N - 234-tree insert. Most of our examples are derived from unrolling loops of smaller programs.

The results are shown in Table 7. Here Safety indicates the safety condition (defined below), LOC indicates the number of *lines-of-code*, type indicates heap operations used (with r = read, w = write, and a = allocation/deallocation), #bt is the number of backtracks for our prototype tool, and #forks is the number of

[5] SMCHR is available from http://www.comp.nus.edu.sg/~gregory/smchr/

				Heaps		Verifast	
Bench.	Safety	LOC	type	time(s)	#bt	time(s)	#forks
subsets_16	F	50	rw-	0.00	17	10.69	65546
expr_2	F	69	rw-	0.05	124	18.38	136216
stack_80	F	976	rwa	8.66	320	68.20	9963
filter_1	F	192	r--	0.03	80	0.75	8134
filter_2	F	321	r--	0.11	307	–	–
sort_6	F	178	rw-	0.03	54	2.66	35909
search234_3	F	251	r--	0.02	46	0.67	1459
search234_5	F	399	r--	0.05	76	90.65	118099
insert234_5	F	839	rwa	1.19	120	52.87	36885
expr_2	$\sqsubseteq$	69	rw-	0.20	1329	n.a.	n.a.
stack_80	$\sqsubseteq$	976	rwa	8.07	322	n.a.	n.a.
filter_2	OP	321	r--	0.00	2	n.a.	n.a.
stack_80	A	976	rwa	8.90	320	65.68	9801
insert234_5	A	839	rwa	1.50	60	40.64	55423
subsets_16	$\emptyset$	50	rw-	0.00	33	n.a.	n.a.

Fig. 7. Theorem proving and symbolic execution benchmarks.

symbolic execution forks for Verifast, and corresponds to the number of paths through the code. All experiments were run on GNU/Linux x86_64 with a Intel® Core™ i5-2500K CPU clocked at 4GHz. A timeout of 10 minutes is indicated by a dash (–). The safety conditions correspond to (some variant of) the following triples:

- *Framing* (F) with $\{\bar{\mathcal{H}} \doteq (p \mapsto v) * F\} C \{\exists F' : \bar{\mathcal{H}} \doteq (p \mapsto v) * F'\}$ where p is outside the footprint of the code C;
- *Operations* (OP) where $OP \in \{\sqsubseteq, \sqsupseteq, \doteq\}$ with $\{H \doteq \bar{\mathcal{H}}\} C \{H \ OP \ \bar{\mathcal{H}}\}$;
- *Allocation* (A) with $\{\ldots\} C \{\exists F', v : \bar{\mathcal{H}} \doteq (p \mapsto v) * F'\}$ for p allocated by C;
- *Empty* ($\emptyset$) with $\{\bar{\mathcal{H}} \doteq \emptyset\}\ C\ \{\mathit{false}\}$, i.e. C will always fault on memory.

Some safety conditions, namely (OP) and ($\emptyset$), cannot be encoded directly in Separation Logic or Verifast, and are marked by "n.a.".

Overall our tests exhibit significant search-space pruning thanks to interpolation. In contrast Verifast explores the entire search-space, and thus has exponential runtime behavior. However the *time-per-path* ratio favors Verifast, suggesting that Verifast would perform better on examples that do not have a large search-space, or when interpolation fails to subsume a significant number of branches. Our tool and SMT solver implementation are preliminary and can likely be further optimized.

7 Related Work

Several systems [3][5][12] implement Separation Logic-based symbolic execution, as described in [4]. However, due to the memory-safety requirements of Separation Logic, symbolic execution is limited to formulae over the footprint of the code. Our symbolic execution is based on the SPC Hoare Logic extension and

therefore works for arbitrary formulae. This is convenient when memory-safety is not a property of interest, such as Example 2.

Several automatic theorem provers for Separation Logic triples/formulae have been developed, including [4][5][18]. These systems generally rely on a set of *rearrangement rules*, and are usually limited to a subset of all formulae, e.g. those with no non-separating conjunction, etc. In contrast our $\mathcal{H}$-solver uses a different algorithm based on heap-membership propagation, and handles any arbitrary QF $\mathcal{H}$-formulae.

Other formalisms, such as (Implicit) Dynamic Frames [16][22] and Region Logic [2], also encode separation. The underlying approach is to represent the heap H as a (possibly implicit) *total map* over all possible addresses, and to represent access or modification rights as sets of addresses F. Separation is represented as set disjoint-ness, i.e. $F_1 \cap F_2 = \emptyset$. One difficulty is that we must relate H with F, which can make reasoning comparatively more difficult. For example, consider the following VCs:

$$p \notin F \wedge \mathbf{list}(H, F, l) \wedge \mathsf{assign}'(H, p, v, H') \models \mathbf{list}(H', F, l) \quad (5)$$

$$H \doteq L{*}R \wedge R \doteq (p \mapsto w){*}R' \wedge \mathbf{list}(L, l) \wedge \mathsf{assign}(H, p, v, H') \models L \sqsubseteq H' \quad (6)$$

where assign' is a suitable re-encoding of assign for total heaps. Both VCs are natural encodings of the same problem: we wish to prove that l is still a list after writing to a (separate) pointer p. VC (6) holds independently of the recursively defined **list** relation, and can be trivially disposed of using our $\mathcal{H}$-solver. In contrast, VC (5) depends on the recursively-defined **list** predicate as it relates H with F, and is therefore more difficult to prove.

Our $\mathcal{H}$-solving algorithm is related to analogous algorithms for finite sets, such as [23]. Although formalized differently, the basic idea is similar, i.e. based on the propagation of *set membership* $x \in S$ constraints. In [21] this idea was adapted into a decision procedure for Region Logic. Our approach works directly with heaps rather than indirectly via sets.

8 Future Work and Conclusions

In this paper we presented a reformulation of the key ideas behind Separation Logic as a first-order constraint-language $\mathcal{H}$ over heaps. Here we express separation as a constraint between heaps. We present an SPC extension of Hoare Logic based on encoding of heap-manipulating statements in terms of $\mathcal{H}$-formulae. Our extension is suitable for forward reasoning via constraint-based symbolic execution. We present a sound and complete solver for QF $\mathcal{H}$-formulae and have implemented a version as part of an SMT framework. Experimental evaluation yields promising results.

There is significant scope for future work, such as: building theorem provers for recursively-defined properties based on the $\mathcal{H}$-solver, or further developing program verification tools using $\mathcal{H}$-language-based symbolic execution.

References

1. S. Abdennadher and H. Schütz. CHR$^{\vee}$: A flexible query language. In *Proceedings of the 3rd International Conference on Flexible Query Answering Systems*, volume 1495, pages 1–14. Springer, 1998.
2. A. Banerjee, D. Naumann, and S. Rosenberg. Regional logic for local reasoning about global invariants. In *Proceedings of the 22nd European conference on Object-Oriented Programming*, pages 387–411. Springer-Verlag, 2008.
3. J. Berdine, C. Calcagno, and P. O'Hearn. Smallfoot: Modular automatic assertion checking with separation logic. In *In International Symposium on Formal Methods for Components and Objects*, pages 115–137. Springer, 2005.
4. J. Berdine, C. Calcagno, and P. O'Hearn. Symbolic execution with separation logic. In *Proceedings of the Third Asian conference on Programming Languages and Systems*, pages 52–68. Springer-Verlag, 2005.
5. M. Botinčan, M. Parkinson, and W. Schulte. Separation logic verification of c programs with an SMT solver. *Electronic Notes in Theoretical Computer Science*, 254:5–23, October 2009.
6. M. Davis, G. Logemann, and D. Loveland. A machine program for theorem-proving. *Communications of the ACM*, 5(7):394–397, July 1962.
7. E. Dijkstra. Guarded commands, nondeterminacy and formal derivation of programs. *Communcations of the ACM*, 18(8):453–457, August 1975.
8. G. Duck. SMCHR: Satisfiability modulo constraint handling rules. *Theory and Practice of Logic Programming*, 12(4-5):601–618, 2012. Proceedings of the 28th international conference on Logic Programming.
9. B. Dutertre and L. De Moura. A fast linear-arithmetic solver for DPLL(T). In *Proceedings of the 18th international conference on Computer Aided Verification*, pages 81–94. Springer, 2006.
10. T. Frühwirth. Theory and practice of constraint handling rules. *Special Issue on Constraint Logic Programming, Journal of Logic Programming*, 37, October 1998.
11. C. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–580, October 1969.
12. B. Jacobs, J. Smans, P. Philippaerts, F. Vogels, W. Penninckx, and F. Piessens. VeriFast: a powerful, sound, predictable, fast verifier for C and Java. In *Proceedings of the 3rd international conference on NASA Formal methods*, pages 41–55. Springer-Verlag, 2011.
13. J. Jaffar and M. J. Maher. Constraint logic programming: A survey. *J. LP*, 19/20:503–581, May/July 1994.
14. J. Jaffar, A. Santosa, and R. Voicu. An interpolation method for CLP traversal. In *Proceedings of the 15th international conference on Principles and practice of constraint programming*, pages 454–469. Springer-Verlag, 2009.
15. J. Jaffar, A. E. Santosa, and R. Voicu. A coinduction rule for entailment of recursively defined properties. In *Proceedings of the 14th International Conference on Principles and Practice of Constraint Programming*, volume 5202 of *LNCS*, pages 493–508. Springer, 2008.
16. I. Kassios. Dynamic frames: Support for framing, dependencies and sharing without restrictions. In *International Symposium on Formal Methods*, pages 268–283. Springer, 2006.
17. K. McMillan. Lazy annotation for program testing and verification. In *Proceedings of the 22nd International Conference on Computer Aided Verification*, volume 6174, pages 104–118. Springer Berlin / Heidelberg, 2010.

18. H. Nguyen, C. David, S. Qin, and W. Chin. Automated verification of shape and size properties via separation logic. In *Proceedings of the 8^{th} international conference on Verification, model checking, and abstract interpretation*, pages 251–266. Springer-Verlag, 2007.
19. O. Ohrimenko, P. Stuckey, and M. Codish. Propagation via lazy clause generation. *Constraints*, 14:357–391, 2009.
20. J. C. Reynolds. Separation logic: A logic for shared mutable data objects. In *17^{th} IEEE Symposium on Logic in Computer Science*, pages 55–74. IEEE Computer Society Press, 2002.
21. S. Rosenberg, A. Banerjee, and D. Naumann. Decision procedures for region logic. In *Proceedings of the 13^{th} international conference on Verification, Model Checking, and Abstract Interpretation*, pages 379–395. Springer-Verlag, 2012.
22. J. Smans, B. Jacobs, and F. Piessens. Implicit dynamic frames: Combining dynamic frames and separation logic. In *Proceedings of the 23^{rd} European conference on Object-Oriented Programming*, pages 148–172. Springer Berlin Heidelberg, 2009.
23. C. Zarba. Combining sets with elements. In *Verification: Theory and Practice*, pages 762–782. Springer, 2004.