Higher-Order Separation Logic in Isabelle/HOLCF

Carsten Varming[1,2] Lars Birkedal[3,4]

Carnegie Mellon University *IT Univsersity of Copenhagen*

Abstract

We formalize higher-order separation logic for a first-order imperative language with procedures and local variables in Isabelle/HOLCF. The assertion language is modeled in such a way that one may use any theory defined in Isabelle/HOLCF to construct assertions, e.g., primitive recursion, least or greatest fixed points etc. The higher-order logic ensures that we can show non-trivial algorithms correct without having to extend the semantics of the language as was done previously in verifications based on first-order separation logic [2,20]. We provide non-trivial examples to support this claim and to show how the higher-order logic enables natural assertions in specifications. To support abstract reasoning we have implemented rules for representation hiding and data abstraction as seen in [1].
The logic is represented as lemmas for reasoning about the denotational semantics of the programming language. This follows the definitional approach common in HOL theorem provers, i.e., the soundness of our model only relies on the soundness of Isabelle/HOL [6].
We use our formalization to give a formally verified proof of Cheney's copying garbage collector [4] using a tagged representation of objects. The proof generalizes the results in [2]. The proof uses an encoding of the separation logic formula $\mathsf{this}(h)$ to capture the heap from before the garbage collection and thus shows another novel use of higher-order separation logic.

1 Introduction

Separation logic [13,18] is a program logic for reasoning modularly about pointer-manipulating programs. Modular reasoning is achieved via the now well-known *frame rule*, expressed using the *separating conjunction* logical connective. Originally separation logic was devised for C-like languages and since then it has been extended to concurrent, higher-order, and object-oriented programming languages [12,3,11,14]. In [1] *higher-order* separation logic was proposed. "Higher-order" here refers to the possibility of quantifying over assertion-level predicates, both in the assertion logic and in the specification logic. In *loc. cit.* it was proposed that higher-order separation logic would be useful for two main purposes: (1) to model and reason

[1] Email: varming@cmu.edu

[2] Research supported in part by National Science Foundation Grants CCF-0541021, CCF-0429505 and by the FIRST Ph.D. school at the IT University of Copenhagen.

[3] Email: birkedal@itu.dk

[4] Research supported in part by the Danish Research Council (FNU Grant No. 272–07–9305)

about data abstraction via existential quantification over hidden resource invariants, and (2) to ease formalizations of separation logic by having one general expressive logic in which it is possible to *define* predicates, etc., needed for applications. The first purpose (data abstraction) was sketched in *loc. cit.* and has proved useful in further investigations [11,7,15]. In this paper we investigate the second purpose (ease of formalization) by developing a formalization of higher-order separation logic in Isabelle/HOL and by applying the formalization to give a formal separation logic proof of Cheney's copying garbage collector [4]. The formalization uses Isabelle/HOLCF [10] which is defined in Isabelle/HOL [16]. The formalized separation logic is for a programming language with simple procedures and address arithmetic. The resulting proof of Cheney's algorithm improves upon the proof in [2] qua the use of higher-order logic: In [2], following [20], the language of separation logic was retrofitted to express the correctness of Cheney's algorithm, but here we can define the needed assertions directly in the higher-order logic. Moreover, the formalization deals properly with procedures; we found that early versions of [1] had left out a side-condition regarding the free variables of the post condition of procedure calls, which is made explicit here.

Even though the ease of formalization purpose was suggested in [1], no suggestions of how to actually achieve this goal in practice were included in *loc. cit.* Here we achieve the goal by including the higher-order logic of Isabelle/HOL in the assertion language of higher-order separation logic. This lets the user use other packages developed in Isabelle/HOL to define assertions, including packages for least or greatest fixed points, total functions, etc. In particular, we show in this paper how the approach is used in the verification of Cheney's algorithm. The formalization is approximately 15000 lines of definitions and proofs. The proof of Cheney's algorithm takes up an additional 7500 lines.

Outline and overview of the technical development

We now give a brief overview of the technical development. In Section 2.1 we define the syntax of our programming language using a deep embedding of commands (by defining a new inductive type of commands) and a shallow embedding of expressions.

In Section 2.2 we present our Isabelle/HOL definition of the denotational semantics of the programming language. The definition makes use of the implementation of basic domain theory in Isabelle/HOL[10] (Isabelle/HOLCF). We also define the frame property expressing that commands behave locally [13], as needed for the verification of the frame rule, and prove that the meaning of any command satisfies the frame property.

In section 3 we present our shallow embedding og higher-order separation logic in Isabelle/HOL. We have chosen to use a shallow embedding since it makes it easy to use Isabelle/HOL to define new assertions, and as the logical implications in the rule of consequence can be defined directly as implications in Isabelle/HOL. Using the shallow embedding of the assertion logic of higher-order separation logic, we then define the validity of separation-logic triples. The proof rules of higher-order separation logic [1] are then formalized in Isabelle/HOL by means of a series of lemmas, stating essentially that if the triples in the antecedent of the rule are valid then also the triple in the conclusion is valid.

In Section 4 we present a couple of reasoning examples in the formalized higher-order separation logic. As already mentioned, our main example is a formal proof of Cheney's copying collector, but we also show an example of formal reasoning about a copying routine for trees and dags. We explain how the invariants of Cheney's algorithm can be defined in the implementation of higher-order separation logic and outline its correctness proof. The proof is a clear improvement over the one in [2] — all the necessary predicates can be defined *in* the logic, rather than extending the logic in an ad-hoc fashion as in [2]. Moreover, we avoid having to build up a mathematical representation of the heap in order to relate the current heap to the old heap as in [2], by making use of a (definable) predicate $\mathsf{this}(h)$ that holds only when the current heap can be described by h.

Finally, in Section 5 we discuss related work and conclude in Section 6.

2 Semantic Model of Programming Language

In this section we present our programming language model, as implemented in Isabelle/HOL. The programming language is a first-order imperative language with simple procedures, i.e., a language with a store (sometimes called a stack) and a heap. The language allows for full address arithmetic and has suitable commands for manipulating the heap, as is standard in separation logic [13,18]. Procedures are simple in that they can only refer to their formal parameters, not to global variables.

2.1 Syntax

Our model of the language uses a shallow embedding of expressions and a deep embedding of commands. This enables us to focus on the separation logic for commands. Of course, a real programming language will have a proper syntax for expression; the idea is that any such expression can easily be given meaning in our shallow embedding, and thus modeling its syntax only complicates the implementation and the uses thereof. In this presentation we use the syntax commonly seen in presentations of separation logic. (This makes the presentation easier to read, but as many of the symbols are already used in Isabelle/HOLCF and cannot be overloaded, the syntax of this presentation and the syntax used in our implementation are slightly different.)

2.1.1 Expressions

The set $\mathbb{S}$ of *stores* consists of the set of functions from $\mathbb{V}$, an infinite set of variables, to the set $\mathbb{Z}$ of integers. We define an equivalence relation on stores $- \simeq_{\_} -$ such that $s \simeq_V s'$ witnesses that s and s' agrees on $V \subseteq \mathbb{V}$.

An *expression* consists of a pair (f, V) where f is a function from stores $\mathbb{S}$ to values and V is a set, containing the free variables of the expression. An expression (f, V) is well-defined if and only if V is finite and $\forall s, s' \in \mathbb{S}.\ s \simeq_V s' \supset f\ s = f\ s'$. This complication enables us to reason about *non-aliasing* substitutions on commands. We say that an expression is an α-expression if and only if the expression evaluates to a value of type α. We model α-expressions in Isabelle/HOL by a new type α exp, defined as the set

$$\alpha\ \mathsf{exp} = \{(f, V) : (\mathbb{S} \to \alpha) \times (\mathbb{V}\ \mathsf{set}) \mid \forall s, s' \in \mathbb{S}.\ s \simeq_V s' \supset f\ s = f\ s'\}.$$

We define two operations on α-expressions: evaluating an expression in a store $\mathsf{ev}\ -\ - : \alpha\ \mathsf{exp} \to \mathbb{S} \to \alpha$, and to get the free variables of an expression $\mathsf{free}(-) : \alpha\ \mathsf{exp} \to \mathbb{V}\ \mathsf{set}$.

We have overloaded constants for numbers and arithmetic operations, extending them to appropriate expression types. We have also defined a constant $\mathsf{trivExp} : \mathbb{V} \to \mathbb{Z}\ \mathsf{exp}$ for encoding program variables (i.e., variables whose value is given by the store) as expressions, and some constants for comparisons between integer expressions. We have also defined a constant $\mathsf{funExp} : (\alpha \to \beta) \to \alpha\ \mathsf{exp} \to \beta\ \mathsf{exp}$ that lifts a function of type $\alpha \to \beta$ to a function defined on α expressions. This expression is quite useful for defining interesting assertions, but if used in the program text, care must be taken to avoid unimplementable expressions. In our program text for Cheney's collector this constant is used to form expressions out of the representation functions (see section 4.1).

2.1.2 Commands

The commands are given by this grammar:

$$\begin{aligned} c ::= {} & \mathsf{skip} \mid v{:=}e \mid v{:=}_l e \mid e_1{:=}_u e_2 \mid c_1;c_2 \mid \mathsf{if}\ b\ \mathsf{then}\ c_1\ \mathsf{else}\ c_2 \mid \mathsf{while}\ b\ \mathsf{do}\ c \\ & \mid v := \mathsf{cons}\ [e_1, \ldots, e_n] \mid \mathsf{dispose}\ e \mid \mathsf{newvar}\ v\ \mathsf{in}\ c \mid \mathsf{call}\ name([x_1, \ldots, x_n]; [y_1, \ldots, y_m]) \\ & \mid \mathsf{let} \begin{bmatrix} name_1 & = & \lambda[x_{1_1}, \ldots, x_{1_n}], [y_{1_1}, \ldots, y_{1_m}].\ c_1 \\ & \vdots & \\ name_k & = & \lambda[x_{k_1}, \ldots, x_{k_n}], [y_{k_1}, \ldots, y_{k_m}].\ c_k \end{bmatrix} \mathsf{in}\ c\ \mathsf{end} \end{aligned}$$

Most constructs are standard: $v{:=}e$ is assignment, $e_1{:=}_u e_2$ updates the heap location pointed to by e_1 with the value of e_2, $v{:=}_l e$ looks up the location pointed to by e in the heap and assigns the resulting value to v, $v := \mathsf{cons}\ [e_0, \ldots, e_n]$ stores the values of $[e_0, \ldots, e_n]$ at an address a and assigns a to v, $\mathsf{dispose}\ e$ disposes the location pointed to by e. Procedures can be defined using a let construct, and each procedure has a list of modifiable variables followed by a list of read-only variables. Having a list of modifiable variables enables a procedure to "return" multiple values from a procedure call.

We define two sets of free variables for commands: the set $\mathsf{mod}(c)$ of modifiable variables, and the set $\mathsf{read}(c)$ of read variables.

$$\begin{aligned} & \mathsf{mod}(\mathsf{call}\ name([x_1, \ldots, x_n]; [e_1, \ldots, e_m])) = \{x_1, \ldots, x_n\}, \\ & \mathsf{mod}(\mathsf{newvar}\ v\ \mathsf{in}\ c) = \mathsf{mod}(c) \backslash \{v\}, \\ & \mathsf{mod}(v{:=}e) = \{v\}, \\ & \mathsf{read}(\mathsf{call}\ name([x_1, \ldots, x_n]; [e_1, \ldots, e_m])) = \{x_1, \ldots, x_n\} \cup \left(\textstyle\bigcup_{i=1}^{m} \mathsf{free}(e_i)\right), \\ & \mathsf{read}(\mathsf{newvar}\ v\ \mathsf{in}\ c) = \mathsf{read}(c) \backslash \{v\}, \\ & \mathsf{read}(v{:=}e) = \mathsf{free}(e), \end{aligned}$$

and so forth. A procedure body may only mention the formal parameters of the procedure. Hence the set of modifiable variables is a superset of the variables that may be assigned to by a command, and the set of read variables is a superset of the variables a command may depend on. In the rest of this paper we will assume commands are well-formed, i.e., the formal parameters of a procedure are all distinct, a procedure assigns only to modifiable parameters, a procedure mentions only its formal parameters, in a call to a procedure the actual modifiable parameters are all distinct, and the two lists of actual parameters have the same lengths as the two lists of formal parameters. We define a constant $\mathsf{goodCmd} : (\mathsf{names} \to (\mathbb{N}_0 \times \mathbb{N}_0)\ \mathsf{option}) \times \mathsf{Cmd} \to \mathbb{B}$ such that if Γ encodes the length of the parameter lists for the free procedure names in C then $\mathsf{goodCmd}\ (\Gamma, C)$ asserts the well-formedness of C.

2.2 *Semantics*

Heaps (ranged over by h) are partial functions from the natural numbers to integers (represented in Isabelle/HOL as elements of type $\mathbb{N}_0 \to \mathbb{Z}\ \mathsf{option}$) and states are pairs of stores $\mathbb{S}$ and heaps $\mathbb{H}$.

2.2.1 *Expressions*

Expressions are modeled by their meanings; thus we can either evaluate an expression in a given store or get its free variables, as described in section 2.1.1.

2.2.2 *Commands*

The meaning of a command is defined relative to an environment providing meanings to the free procedure names in the command. An environment is a pair

$$(\Theta, \zeta) \in env = (\mathsf{names} \to_c \mathsf{state\ discr} \to_c \mathsf{state\ option\ set}) \times (\mathsf{names} \to (\mathbb{V}\ \mathsf{list} \times \mathbb{V}\ \mathsf{list}))$$

of functions. In the above display we use the Isabelle/HOL package for basic domain theory: for a type α, α discr is the discretely ordered cpo, and for cpos α and β, the type $\alpha \to_c \beta$ denotes the continuous functions from α to β. The notation α option is used for the Isabelle/HOL type of values of the form None or $\mathsf{Some}(a)$, for a in α, and α set denotes the powerset type of α, ordered by set inclusion. The option type is used to capture the possibility of abortion. The meaning of the procedure name n is $(\Theta\ n, \zeta\ n)$, i.e., a continuous state transformation function, a list of formal modifiable variables, and a list of formal read-only variables. (The environment consist of two functions only to make the encoding in Isabelle/HOLCF easier.) Given an environment (Θ, ζ) the meaning of a command is a triple:

$$((\mathsf{state\ discr} \to_c \mathsf{state\ option\ set}) \times (\mathbb{V}\ \mathsf{set} \times \mathbb{V}\ \mathsf{set}))$$

consisting of a continuous state transformation function, a set of modifiable variables, and a set of read variables. (Notice that in Isabelle/HOL a triple of type $\alpha \times \beta \times \gamma$ is represented by the type $\alpha \times (\beta \times \gamma)$.) The set of read variable may intersect with the set of modifiable variables, e.g., the meaning of $v{:=}2 * v$ both reads the value of v and modifies the store at v. We define two constants $\mathsf{fr} = \lambda t.\ \mathsf{snd}(\mathsf{snd}(t))$, $\mathsf{fm} = \lambda t.\ \mathsf{fst}(\mathsf{snd}(t))$, to get the free read variables and the free modifiable variables from the meaning of commands.

To ease the definition of the meaning of a command we have defined a few Isabelle/HOL constants: up that lifts functions in $\mathsf{state\ discr} \to_c \mathsf{state\ option\ set}$ to functions in $\mathsf{state\ option\ set} \to_c \mathsf{state\ option\ set}$, defined by

$$\mathsf{up}\ g\ X = \bigcup_{x \in X} \mathsf{case}\ x\ \mathsf{of\ None} \Rightarrow \{\mathsf{None}\} \mid \mathsf{Some}(x') \Rightarrow g\ x';$$

find that given a list of procedure declarations and a name will return the first procedure with that name or nothing if such a procedure isn't present in the list, and funZip that given a function f and two lists l_1, l_2 of equal length, returns the function whose graph is the graph of f restricted to dom $f \backslash (\mathsf{set}\ l_1)$ union the zip of l_1 and l_2. Besides these constants we also make use of constants defined in the Isabelle/HOL library: nat that maps non-negative integers to natural numbers and negative integers to 0, int that map natural numbers to integers, set that maps lists to sets, $\mathsf{override\_on}$ that given two functions $f, f' : A \to B$ and a set $C \subseteq A$ gives the function that is f outside C and f' on C, and Inv that given a set S and a function f such that $S \subseteq$ dom f and f is injective on S, gives a function that maps $f\ s$ to s for all $s \in S$.

A *semantic* substitution has four elements (f, M, g, R), a function $f : \mathbb{V} \to \mathbb{V}$ that maps modifiable variables, e.g. members of M, to modifiable variables, a function $g : \mathbb{V} \to \mathbb{Z}\ \mathsf{exp}$ that maps read variables, e.g. members of R, to expressions, and two sets M and R. A substitution (f, M, g, R) is well-defined if and only if

$$\forall v \in M.\ g\ v = \mathsf{trivExp}\ (f\ v) \qquad \forall v \in M \cup R.\ \mathsf{free}(g\ v) \cap \left(\bigcup_{x \in M \backslash \{v\}} \{f\ x\}\right) = \{\},$$

and can be applied to a command c if and only if M is a superset of $\mathsf{fm}(c)$ and R is a superset of $\mathsf{fr}(c)$.

We write $\mathsf{csubst}\ \sigma\ c$ for the result of applying a substitution (f, M, g, R) to a command c and define its meaning to be:

$$\begin{aligned}
&(\lambda(s,h).\ \mathsf{up}\\
&(\lambda(s',h').\ \{\mathsf{Some}(\lambda v.\ \mathsf{if}\ v \in \textstyle\bigcup_{v \in \mathsf{fm}(c)} \{f\ v\}\ \mathsf{then}\ s'\ (\mathsf{Inv}\ (\mathsf{fm}(c))\ f\ v)\ \mathsf{else}\ s\ v, h')\})\\
&\qquad\qquad\qquad (\mathsf{fst}(c)\ (\lambda v.\ \mathsf{ev}\ (g\ v)\ s, h)), \textstyle\bigcup_{v \in \mathsf{fm}(c)} \{f\ v\}, \bigcup_{v \in \mathsf{fr}(c)} \mathsf{free}(g\ v)).
\end{aligned}$$

Besides the constants defined above we also define a constant: $\mathsf{procsubst}$ that given a list of formal modifiable parameters, a list of actual modifiable parameters, a list of read-only parameters, and a set of variable names to avoid, will give a substitution σ and a list of variables rl such that σ substitutes the actual modifiable parameters for the formal modifiable parameters and rl for the formal read-only parameters. The elements of rl are, if possible, chosen outside the set of variables to be avoided, and ensures that the particular choice of names for the formal parameters doesn't limit the set of variables that can be used as actual modifiable parameters.

The meaning of commands is then given by Figure 1. Notice we have given the read only parameters of procedure calls a call-by-value semantics, and that we never allocate heap space at address 0. We also notice that the modifiable (resp. read) variables of the meaning of a command coincides with the modifiable (resp. read) variables derived from the syntax in sections 2.1.2.

The semantics satisfies two important properties. A well-behavedness property, expressing how the semantics of a command relates to its modifiable and readable variables, and the frame property.

Definition 2.1 A set of states S : $\mathsf{state\ option\ set}$ is less than another set of states S' on a set $V \subseteq \mathbb{V}$ (written $S \preceq_V S'$), if and only if

$$\forall \sigma \in S.\ \mathsf{case}\ \sigma\ \mathsf{of}\ \mathsf{None} \Rightarrow \mathsf{None} \in \sigma'\ |\ \mathsf{Some}(s,h) \Rightarrow \exists s', h'.\ \mathsf{Some}(s',h') \in S' \wedge s \simeq_V s' \wedge h = h'.$$

Definition 2.2 A set of states S : $\mathsf{state\ option\ set}$ is smaller on the V part of the store than another set of states S' (written $S \trianglelefteq_V S'$), if and only if

$$\forall \sigma \in S.\ \mathsf{case}\ \sigma\ \mathsf{of}\ \mathsf{None} \Rightarrow \mathsf{true}\ |\ \mathsf{Some}(s,h) \Rightarrow \exists s', h'.\ \mathsf{Some}(s',h') \in S' \wedge s \simeq_V s'.$$

$$
\begin{aligned}
[\![\mathsf{skip}]\!]^c(\Theta,\zeta) &= (\lambda(s,h).\ \{\mathsf{Some}(s,h)\},\emptyset,\emptyset)\\
[\![v{:=}e]\!]^c(\Theta,\zeta) &= (\lambda(s,h).\ \{\mathsf{Some}(s(v\mapsto(\mathsf{ev}\ e\ s)),h)\},\{v\},\mathsf{free}(e))\\
[\![c_1;c_2]\!]^c(\Theta,\zeta) &= (\lambda(s,h).\ \mathsf{up}\ (\mathsf{fst}([\![c_2]\!]^c(\Theta,\zeta)))\ (\mathsf{fst}([\![c_1]\!]^c(\Theta,\zeta))(s,h)),\\
&\qquad \mathsf{fm}([\![c_1]\!]^c(\Theta,\zeta))\cup\mathsf{fm}([\![c_2]\!]^c(\Theta,\zeta)),\\
&\qquad \mathsf{fr}([\![c_1]\!]^c(\Theta,\zeta))\cup\mathsf{fr}([\![c_2]\!]^c(\Theta,\zeta)))\\
[\![\mathsf{while}\ b\ \mathsf{do}\ c]\!]^c(\Theta,\zeta) &= (\mu F.\ \lambda(s,h).\ \mathsf{if}\ \mathsf{ev}\ b\ s\ \mathsf{then}\ \mathsf{up}\ F\ (\mathsf{fst}([\![c]\!]^c(\Theta,\zeta))(s,h))\ \mathsf{else}\ \{\mathsf{Some}(s,h)\},\\
&\qquad \mathsf{fm}([\![c]\!]^c(\Theta,\zeta)),\mathsf{fr}([\![c]\!]^c(\Theta,\zeta))\cup\mathsf{free}(b))\\
[\![\mathsf{if}\ b\ \mathsf{then}\ c_1\ \mathsf{else}\ c_2]\!]^c(\Theta,\zeta) &= (\lambda(s,h).\ \mathsf{if}\ \mathsf{ev}\ b\ s\ \mathsf{then}\ [\![c_1]\!]^c(\Theta,\zeta)(s,h)\ \mathsf{else}\ [\![c_2]\!]^c(\Theta,\zeta)(s,h),\\
&\qquad \mathsf{fm}([\![c_1]\!]^c(\Theta,\zeta))\cup\mathsf{fm}([\![c_2]\!]^c(\Theta,\zeta)),\\
&\qquad \mathsf{fr}([\![c_1]\!]^c(\Theta,\zeta))\cup\mathsf{fr}([\![c_2]\!]^c(\Theta,\zeta))\cup\mathsf{free}(b))\\
[\![\mathsf{dispose}\ e]\!]^c(\Theta,\zeta) &= (\lambda(s,h).\ \mathsf{case}\ h(\mathsf{nat}(\mathsf{ev}\ e\ s))\ \mathsf{of}\ \mathsf{None}\Rightarrow\{\mathsf{None}\}\\
&\qquad \mid \mathsf{Some}(n)\Rightarrow\{\mathsf{Some}(s,h(n\mapsto\mathsf{None}))\},\emptyset,\mathsf{free}(e))\\
[\![e_1{:=}_u e_2]\!]^c(\Theta,\zeta) &= (\lambda(s,h).\ \mathsf{case}\ h(\mathsf{nat}(\mathsf{ev}\ e_1\ s))\ \mathsf{of}\ \mathsf{None}\Rightarrow\{\mathsf{None}\}\\
&\qquad \mid \mathsf{Some}(n)\Rightarrow\{\mathsf{Some}(s,h(\mathsf{nat}(\mathsf{ev}\ e_1\ s)\mapsto\mathsf{ev}\ e_2\ s))\},\emptyset,\mathsf{free}(e_1)\cup\mathsf{free}(e_2))\\
[\![v{:=}_l e]\!]^c(\Theta,\zeta) &= (\lambda(s,h).\ \mathsf{case}\ h(\mathsf{nat}(\mathsf{ev}\ e\ s))\ \mathsf{of}\ \mathsf{None}\Rightarrow\{\mathsf{None}\}\\
&\qquad \mid \mathsf{Some}(n)\Rightarrow\{\mathsf{Some}(s(v\mapsto n),h)\},\{v\},\mathsf{free}(e))\\
[\![\mathsf{newvar}\ v\ \mathsf{in}\ c]\!]^c(\Theta,\zeta) &= (\lambda(s,h).\ \mathsf{up}\ (\lambda(s',h').\{\mathsf{Some}(s'(v\mapsto s(v)),h')\})\\
&\qquad (\mathsf{up}\ (\mathsf{fst}([\![c]\!]^c(\Theta,\zeta)))\ (\textstyle\bigcup_n\{\mathsf{Some}(s(v\mapsto n),h)\})),\\
&\qquad \mathsf{fm}([\![c]\!]^c(\Theta,\zeta))\backslash\{v\},\mathsf{fr}([\![c]\!]^c(\Theta,\zeta))\backslash\{v\})\\
[\![v:=\mathsf{cons}\ [e_1,\ldots,e_n]]\!]^c(\Theta,\zeta) &= (\lambda(s,h).\ \textstyle\bigcup_m \mathsf{if}\ \forall i.\ m<i\le m+n\supset h(i)=\mathsf{None}\\
&\qquad \mathsf{then}\ \{\mathsf{Some}(s(v\mapsto\mathsf{int}(m+1)),\\
&\qquad\quad h(m+1\mapsto\mathsf{ev}\ e_1\ s,\ldots,m+n\mapsto\mathsf{ev}\ e_n\ s))\}\\
&\qquad \mathsf{else}\ \emptyset,\{v\},\textstyle\bigcup_{0<i\le n}\mathsf{free}(e_i))\\
[\![\mathsf{call}\ p(\boldsymbol{x};\boldsymbol{e})]\!]^c(\Theta,\zeta) &= \ \mathsf{let}\ (\sigma,rl') \ = \mathsf{procsubst}\ (\mathsf{fst}(\zeta\ p))\ \boldsymbol{x}\ (\mathsf{snd}(\zeta\ p))\\
&\qquad\qquad (\mathsf{set}(\mathsf{fst}(\zeta\ p))\cup\mathsf{set}(\mathsf{snd}(\zeta\ p)))\\
&\quad \mathsf{in}\ (\lambda(s,h).\ \mathsf{up}\ (\lambda(s',h').\ \{\mathsf{Some}(\mathsf{override_on}\ s\ s'\ (\mathsf{set}\ \boldsymbol{x}),h')\})\\
&\qquad (\mathsf{fst}(\mathsf{csubst}\ \sigma\ (\Theta\ p,\mathsf{set}\ (\mathsf{fst}(\zeta\ p)),\mathsf{set}\ (\mathsf{fst}(\zeta\ p))\cup\mathsf{set}\ (\mathsf{snd}(\zeta\ p))))\\
&\qquad (\mathsf{funZip}\ s\ rl'\ (\mathsf{map}\ (\lambda e.\ \mathsf{ev}\ e\ s)\ \boldsymbol{e}),h)),\mathsf{set}\ \boldsymbol{x},\mathsf{set}\ \boldsymbol{x}\cup\bigcup_{e\in\boldsymbol{e}}\mathsf{free}(e))\\
[\![\mathsf{let}\ l\ \mathsf{in}\ c\ \mathsf{end}]\!]^c(\Theta,\zeta) &= \mathsf{let}\ vars\ = \lambda p.\ \mathsf{case}\ \mathsf{find}\ l\ p\ \mathsf{of}\ \mathsf{None}\Rightarrow\zeta\ p\mid\mathsf{Some}(c',ml,rl)\Rightarrow(ml,rl)\\
&\quad \mathsf{in}\ [\![c]\!]^c(\mu\Theta'.\ \lambda p.\ \mathsf{case}\ \mathsf{find}\ l\ p\ \mathsf{of}\ \mathsf{None}\Rightarrow\Theta\ p\\
&\qquad\qquad \mid\mathsf{Some}(c',ml,rl)\Rightarrow\mathsf{fst}([\![c']\!]^c(\Theta',vars)),vars)
\end{aligned}
$$

Figure 1. Language Semantics

We define the well-behavedness property $wb : ((\mathsf{state\ discr} \to_c \mathsf{state\ option\ set}) \times \mathbb{V}\ \mathsf{set} \times \mathbb{V}\ \mathsf{set}) \to \mathbb{B}$ by letting $wb\ c$ mean

$$\forall s,s',h.\ s\simeq_{\mathsf{fm}(c)\cup\mathsf{fr}(c)} s' \supset \mathsf{fst}(c)\ (s,h) \preceq_{\mathsf{fm}(c)} \mathsf{fst}(c)\ (s',h) \wedge \mathsf{fst}(c)\ (s,h) \trianglelefteq_{\mathbb{V}\backslash\mathsf{fm}(c)} \{\mathsf{Some}(s,h)\}.$$

Lemma 2.3 *For any command C and any environment (Θ,ζ) satisfying that, for every free procedure name n in C, we have $wb(\Theta\ n, \mathsf{set}\ (\mathsf{fst}(\zeta\ n)), \mathsf{set}\ (\mathsf{snd}(\zeta\ n)))$, it holds that $wb\ [\![C]\!]^c(\Theta,\zeta)$.*

Lemma 2.3 essentially shows that a command C may only change a store on the modifiable variables of C. And if C is executed from (s,h) and s and s' agrees on the free varaibles of C, then there exists an execution from (s',h) such that the two resulting stores match on the modifiable variables and the two resulting heaps are equal.

Definition 2.4 We say two heaps h,h' are disjoint (written $h\perp h'$) if and only if $\forall n,j.\ (h\ n = \mathsf{Some}(j) \supset h'\ n = \mathsf{None}) \wedge (h'\ n = \mathsf{Some}(j) \supset h\ n = \mathsf{None})$. Given two heaps h,h', let $h \uplus h'$ be the heap

$$h\uplus h' = \lambda n.\ \mathsf{if}\ (\exists n'.\ h\ n = \mathsf{Some}(n'))\ \mathsf{then}\ h\ n\ \mathsf{else}\ h'\ n.$$

$$
\begin{array}{rlrl}
\top &= \{h \mid \mathit{true}\} & \mathsf{emp} &= \{\lambda x.\ \mathsf{None}\} \\
\mathsf{pure}\ b &= \{h \mid b\} & \forall\ P &= \{h \mid \forall x:\ \alpha.\ h \in P\ x\} \\
P \mathrel{-\!\!*} Q &= \{h \mid \forall h_1.\ h \perp h_1 \wedge h_1 \in P \supset h \in Q\} & P \supset Q &= \{h \mid h \in P \supset h \in Q\}
\end{array}
$$

$$i \mapsto i' = \{h \mid h\ (\mathsf{nat}(i)) = \mathsf{Some}\ i' \wedge i > 0 \wedge \forall j.\ j \neq i \supset h\ j = \mathsf{None}\}$$

$$P * Q = \{h \mid \exists h_0, h_1.\ h = h_0 \uplus h_1 \wedge h_0 \perp h_1 \wedge h_0 \in P \wedge h_1 \in Q\}$$

Figure 2. Separation Logic Constants

We define the frame property as follows. Let $fp : (\mathsf{state\ discr} \to_c \mathsf{state\ option\ set}) \to \mathbb{B}$ be the function such that $fp\ f$ means

$$
\begin{aligned}
\forall s, s', h_1, h_2.\ h_1 \perp h_2 \supset (\mathsf{None} \in f\ (s, h_1 \uplus h_2) \supset \mathsf{None} \in f\ (s, h_1)) \wedge \\
(\mathsf{Some}(s', h') \in f\ (s, h_1 \uplus h_2) \wedge \mathsf{None} \notin f\ (s, h_1) \supset \\
\exists h_3.\ \mathsf{Some}(s', h_3) \in f\ (s, h_1) \wedge h' = h_3 \uplus h_2 \wedge h_3 \perp h_2).
\end{aligned}
$$

Lemma 2.5 (Frame Property) *For any command C and any environment (Θ, ζ) such that for every free procedure name n in C, the frame property $fp\ (\Theta\ n)$ holds, we have that the frame property $fp\ (\mathsf{fst}([\![C]\!]^c(\Theta, \zeta)))$ holds.*

3 Higher-Order Separation Logic

We define a shallow embedding of higher-order separation logic in Isabelle/HOL. Recall that predicates in higher-order separation logic are subsets of heaps. A shallow embedding enables us to use the meta-logic to define predicates. Thus to model separation logic we give suitable inhabitants of the type $\mathbb{H}$ set and define constants with these meanings.

Definition 3.1 Let the following constants be defined with meanings given in Figure 2.

$$
\begin{array}{rcr}
- \mapsto - : \mathbb{Z} \to \mathbb{Z} \to \mathbb{H}\ \mathsf{set} & \mathsf{pure}\ - : \mathbb{B} \to \mathbb{H}\ \mathsf{set} & - \mathrel{-\!\!*} - : \mathbb{H}\ \mathsf{set} \to \mathbb{H}\ \mathsf{set} \to \mathbb{H}\ \mathsf{set} \\
- * - : \mathbb{H}\ \mathsf{set} \to \mathbb{H}\ \mathsf{set} \to \mathbb{H}\ \mathsf{set} & \top : \mathbb{H}\ \mathsf{set} & - \supset - : \mathbb{H}\ \mathsf{set} \to \mathbb{H}\ \mathsf{set} \to \mathbb{H}\ \mathsf{set} \\
\forall\ - : (\alpha \to \mathbb{H}\ \mathsf{set}) \to \mathbb{H}\ \mathsf{set} & \mathsf{emp} : \mathbb{H}\ \mathsf{set} &
\end{array}
$$

As standard in higher-order logics, the existential quantifier is definable by:

$$\exists\ - : (\alpha \to \mathbb{H}\ \mathsf{set}) \to \mathbb{H}\ \mathsf{set} \quad \exists P = \forall \lambda Q :\ \mathbb{H}\ \mathsf{set}.\ ((\forall \lambda x :\ \alpha.\ (P\ x \supset Q)) \supset Q).$$

The above definitions of quantifiers satisfy the following expected properties:

$$\forall P = \bigcap_x P\ x \qquad \exists P = \bigcup_x P\ x$$

To define the rule for allocation it is convenient to have an array predicate. Thus we let $- \mapsto_l - : \mathbb{Z} \to \mathbb{Z}\ \mathsf{list} \to \mathbb{H}\ \mathsf{set}$ be defined by primitive recursion on its second argument: $i \mapsto_l [] = \mathsf{emp}$ and $i \mapsto_l (i' \# ir) = (i \mapsto i') * (i + 1 \mapsto_l ir)$. $(-\#-$ is Isabelle/HOL syntax for the cons operation on lists).

In Isabelle/HOL we can introduce special syntax for binding operators and thus given a lambda abstraction $\lambda x :\ \alpha.\ P$ we get syntactic translations: $\forall\ (\lambda x :\ \alpha.\ P) = \forall x :\ \alpha.\ P$ and $\exists\ (\lambda x :\ \alpha.\ P) = \exists x :\ \alpha.\ P$.

3.1 Assertions

Assertions are functions of type $\mathbb{A} = \mathbb{S} \to \mathbb{H}$ set. To make it easy to keep track of the free program variables of an assertion, we have defined suitable constants, lifting the constants from Definition 3.1 to assertions. This makes it easier to keep track of which program variables an assertion depends upon and is useful because many separation logic rules have side-conditions of the kind: "variable x must not occur free in assertion F." We encode such side-conditions using a constant $\mathsf{fv} : \mathbb{A} \to \mathbb{V}\ \mathsf{set} \to \mathbb{B}$ such that $\mathsf{fv}\ F\ V$ if and only if $\forall s, s'.\ s \simeq_V s' \supset F\ s = F\ s'$, i.e., the function F only depends on the values of the variables V in the store.

3.2 Validity

We encode the following partial correctness property as the meaning of the constant: $- \vdash \{-\}\ -\ \{-\} : \mathsf{env} \to (\mathbb{S} \to \mathbb{H}\ \mathsf{set}) \to \mathsf{Cmd} \to (\mathbb{S} \to \mathbb{H}\ \mathsf{set}) \to \mathbb{B}$.

Definition 3.2 The partial correctness formula $(\Theta, \zeta) \vdash \{P\}\ C\ \{Q\}$ holds iff, for all free procedure names n in C, $(\Theta\ n, \zeta\ n)$ is well-behaved in the sense of Lemma 2.3 and satisfies the frame property, C is well-formed as described in Section 2.1.2, and for all stores $s \in \mathbb{S}$ and heaps $h \in P\ s$, $\mathsf{fst}([\![C]\!]^c(\Theta,\zeta))(s,h)$ doesn't abort, i.e., $\mathsf{None} \notin \mathsf{fst}([\![C]\!]^c(\Theta,\zeta))(s,h)$, and for all stores s' and heaps h', if C executed in the state (s,h) terminates with (s',h') then $h' \in Q\ s'$, i.e., $\mathsf{Some}(s',h') \in \mathsf{fst}([\![C]\!]^c(\Theta,\zeta))(s,h) \supset h' \in Q\ s'$.

We prove a number of rules sound with respect to this definition. The rules fall in two categories: primitive rules and structural rules. The primitive rules for heap manipulating commands are tight, i.e., the pre- and post-condition specifies only the footprint of the command. The structural rules can be applied anywhere in a proof and bridge the gap between the tight primitive rules and complicated specifications. The frame rule and the rule of consequence are the most profound structural rules, but more rules comes in handy, especially a rule for introducing existential quantifiers.

By combining primitive and structural rules we can get a set of very useful rules. As an example of this technique we can prove the following specification for looking up values in the heap: Given arbitrary functions $f, q, r : \mathbb{Z}\ \mathsf{exp} \to \mathbb{A}$ such that v is not free in e, and for all x, v is not free in $f\ x$, and for all x, $q\ x$ and $r\ x$ do not depend on the value of v:

$$\begin{aligned}(\Theta,\zeta) \vdash \{\exists e'.\ (e \mapsto f\ e' * r\ e') \wedge q\ e'\}\ v{:=}_l e \\ \{\exists e'.\ \mathsf{pure}\ (\mathsf{trivExp}\ v = f\ e') \wedge e \mapsto \mathsf{trivExp}\ v * r\ e' \wedge q\ e'\}.\end{aligned}$$

The proof uses a tight rule for lookup, the rule of consequence, the frame rule, and the fact that we can factor out strictly exact parts of a formula, as expressed by the following lemma.

Lemma 3.3 *Assuming R is strictly exact (i.e., for all heaps h_0, h_1 and stores s, if $h_0 \in R\ s$ and $h_1 \in R\ s$ then $h_0 = h_1$) then $(P \wedge (Q * R)) \supset (R * (R \mathrel{-\!\!*} P \wedge (Q * R)))$.*

3.3 Procedures

The environment in the definition of validity is meant to capture the meaning of procedures. This could potentially complicate reasoning using our rules, but it doesn't have to. If a program doesn't introduce or call procedures the proof will make no assumptions about the environment, and thus we can universally quantify over it. If the program does use procedures then we want to make the interaction between environments as simple as possible. We do this by introducing a constant

$$\mathsf{va} : \mathsf{env} \to (\mathsf{name} \to (\mathbb{V}\ \mathsf{list} \times \mathbb{V}\ \mathsf{list} \times (\mathbb{A} \times \mathbb{A})\ \mathsf{set})) \to \mathbb{B}$$

that captures the pre- and post conditions we will assume of the procedures in scope. This constant relates a given environment to a set of pre- and post conditions a procedure satisfies, and to the formal parameters of the procedure. To make it clear why the environment doesn't complicate proofs we give a stylized and simplified version of the rule for let declarations:

$$\frac{\begin{array}{c}\mathsf{va}\ \Gamma\ Ctx \Longrightarrow \big(\Gamma', \mathsf{va}\ \Gamma'\ [Ctx|n \mapsto (x, y, \{(P,Q)\})] \Longrightarrow \Gamma' \vdash \{P_c\}\ c\ \{Q_c\}\big) \\ \mathsf{va}\ \Gamma\ Ctx \Longrightarrow \big(\Gamma', \mathsf{va}\ \Gamma'\ [Ctx|n \mapsto (x, y, \{(P,Q)\})] \Longrightarrow \Gamma' \vdash \{P\}\ c_n\ \{Q\}\big)\end{array}}{\mathsf{va}\ \Gamma\ Ctx \Longrightarrow \Gamma \vdash \{P_c\}\ \mathsf{let}\ n(x;y) = c_n\ \mathsf{in}\ c\ \mathsf{end}\ \{Q_c\}}$$

where $\Longrightarrow$ is ISABELLE/HOLs entailment on the meta-level. The rule assumes that for all fresh environments Γ' you can produce a proof of the client and a proof of the procedure body and the proofs may use the current assumptions about pre- and post conditions of procedures. As Γ' is a parameter it only takes up one symbol in a proof, and thus the environments doesn't complicate the use of the logic. The introduction of the constant va in the ISABELLE/HOL context enables us to forget about it until either we have a let declaration or a procedure call.

The meaning of $\mathsf{va}\ (\Theta, \zeta)\ Ctx$ is: for every procedure name n, assuming $Ctx\ n = (ml, rl, pqs)$ then $\zeta\ n = (ml, rl)$, the variables in $ml\ @\ rl$ are all distinct, and $(\Theta\ n, \zeta\ n)$ is well-behaved in the sense of Lemma 2.3 and satisfies the frame property. Moreover, for every pair of assertions $(P, Q) \in pqs$ we must have that $(P, \Theta\ n, Q)$ satisfies the partial correctness properties set out in the definition of validity, i.e., if P holds of a given heap and store then the procedure, executed in that state, does not abort, and if the procedure terminates then Q holds of the resulting heap and store.

Now that we have made assumptions about procedures concrete via va, we get a method for changing the assumptions by ensuring that the new assumptions hold of the same environment:

Lemma 3.4 *Assuming* $\mathsf{va}\ \Gamma\ Ctx$ *and* $Ctx\ n = (ml, rl, pqs)$ *and for every* $(P, Q) \in pqs'$, $\Gamma \vdash \{P\}\ \mathsf{call}\ n(ml; rl)\ \{Q\}$, *then* $\mathsf{va}\ \Gamma\ Ctx'$ *where* Ctx' *is* Ctx *updated with* $(n, (ml, rl, psq'))$.

Notice how Lemma 3.4 enables us to use any structural rule to change the assumptions about procedures.

The rule for procedure calls is then:

$$\frac{\begin{array}{c}\mathsf{va}\ \Gamma\ Ctx \quad \mathsf{distinct}\ x \quad (ml, rl, pqs) \in Ctx\ n \quad (P, Q) \in pqs \quad |ml| = |x| \quad |rl| = |y| \quad \mathsf{fv}\ Q\ FQ \\ \forall e \in \mathsf{set}\ y.\ \mathsf{free}(e) \cap \mathsf{set}\ x = \{\} \quad FQ \cap (\mathsf{set}\ x \backslash \mathsf{set}\ (ml@rl)) = \{\}\end{array}}{\Gamma \vdash \{\mathsf{lsubst}\ P\ (ml@rl)\ ((\mathsf{map\ trivExp}\ x)@y)\}\ \mathsf{call}\ n(x;y)\ \{\mathsf{lsubst}\ Q\ (ml@rl)\ ((\mathsf{map\ trivExp}\ x)@y)\}},$$

where $\mathsf{lsubst} : \mathbb{A} \to \mathbb{V}\ \mathsf{list} \to \mathbb{Z}\ \mathsf{exp\ list} \to \mathbb{A}$, and $\mathsf{lsubst}\ P\ vl\ es$ performs the denotational substitution specified by the zip of vl and es on P.

To support modularity and data abstraction (the first purpose of higher-order separation logic mentioned in the Introduction), we have a derived rule for let declarations that lets the assertions for the procedures be functions f of type $\alpha \to \mathbb{A}$. The procedure bodies are then verified using $f\ P$ for some $P : \alpha$, and the client is verified using $f\ x$ for a fresh parameter $x : \alpha$. Hence if α is the type $\mathbb{A}$ or $\mathbb{H}\ \mathsf{set}$ a group of procedures may share information about a piece of the heap but the information is kept secret from the client. This enables us to build modules as collections of procedures and abstract away their internal representations of data structures, as suggested in [1].

4 Examples

4.1 Cheney's Garbage Collector

We now give an overview of our formalization of Cheney's garbage collection algorithm [4]. As mentioned in the Introduction, the formalization and the proof improves upon the one in [2], which used a retrofitted version of first-order separation logic with additional basic predicates, etc., tailored to this particular algorithm. To ease the proof in [2], the authors simplified the algorithm to a representation that only allows pointers to pairs, and their extensions to the logic relied heavily on this representation, thus simplifying both the program and the proof. Our proof fits entirely in higher-order separation logic and makes use of Isabelle/HOL's library for defining functions over lists, finite sets, etc. We also discharge the actual layout of records in memory by assuming that we are given two representation functions that given a tag will tell us how big an record is and which fields contain pointers. It proved valuable to identify a fraction of the program as a copying routine and give a proof of its specification separately from the main proof. The main proof then uses the frame rule to utilize the copying routine.

4.1.1 Representation

We assume that we are given two functions $len : \mathbb{Z} \to \mathbb{N}_0\ \mathsf{option}$ and $isptr : \mathbb{Z} \to \mathbb{N}_0 \to \mathbb{B}$ that gives the representation of records. Each record is a block of continuous memory with a tag at its smallest address and size given by the len function, i.e., $len\ t$ is $\mathsf{Some}(l)$ if and only if t is a proper tag and the record it represents has length l. Given a proper record, i.e., an record with a tag t such that $len(t) = \mathsf{Some}(l)$ for some l, for all i such that $0 < i < l$, $isptr\ t\ i$ if and only if the i'th field in the record is a pointer. We require $\neg isptr\ t\ 0$, for all t. We reserve the tag -1 for the garbage collector and we don't allow skewed sharing between records. We disallow dangling pointers, and pointers not pointing to the least address of an record. That the records doesn't contain dangling pointers can be formalized as a closure property of a set of pointers:

Definition 4.1 A set P of pointers is *closed* if and only if for any $p \in P$, any pointer in the record pointed to by p is in P.

Notice that if P is closed and is a subset of the domain of the heap then every pointer in any record must be a member of P, and thus every pointer points to an record.

We say that a set P of pointers determines a heap if every pointer $p \in P$ points to an record in the heap and P is closed.

The main complications dealing with closed sets of pointers stems from the fact that if you take away a single pointer from the set, it may not be closed afterwards, i.e., our proofs by induction on finite closed sets of pointers have required some generalizations to heaps with dangling pointers.

In the following sections we need a few Isabelle/HOL constants: the of type α option $\to \alpha$ that is undefined on None and given $\mathsf{Some}(e)$ will return e, $-!-$ of type α list $\to \mathbb{N}_0 \to \alpha$ such that if n is less than the length of l then $l!n$ will return the n'th element in l and be undefined otherwise, $- \prec -$ such that given two functions $f, f' : \alpha \to \beta$ option, $f \prec f'$ states that if $f\ a = \mathsf{Some}(b)$ then $f'\ a = \mathsf{Some}(b)$ for all a, b.

4.1.2 Heap invariant

Given the representation assumptions above we need an appropriate heap invariant. Disallowing skewed sharing enables us to express every record by a formula and then use the separating conjunction to glue it all together. As the separating conjunction is associative and commutative this approach will give a unique set of heaps.

The heap invariant hinv *isptr len root* can be expressed in Isabelle/HOL by:

$$\mathsf{hinv}\ \mathit{isptr}\ \mathit{len}\ \mathit{root} = \exists P.\ \mathsf{pure}\ (\mathsf{funExp}(\lambda r.\ \mathsf{nat}(r) \in P \wedge \mathsf{finite}\ P)\ \mathit{root}) \wedge \\ (\lambda s.\ \mathsf{Finite\_Set.fold}\ (\mathsf{op}\ *)\ f\ \{\mathsf{empty}\}\ P)$$

where f is the function

$$f\ p = \exists t, il.\ \mathsf{int}(p) \mapsto_l (t\#il) \wedge \\ \mathsf{pure}\ (\mathit{len}\ t = \mathsf{Some}(\mathsf{length}\ (t\#il)) \ \wedge\ \mathsf{length}\ il > 0 \ \wedge\ t \neq -1 \wedge \\ \forall i.\ i < \mathsf{length}\ il \ \wedge\ \mathit{isptr}\ t\ (i+1) \supset il!i > 0 \wedge il!i \in P)$$

The meaning of Finite_Set.fold *op* $f\ u\ S$ is to apply f to every element of S and then fold *op* over the result starting with u. This is part of the Isabelle/HOL library. Intuitively, hinv *isptr len root* is saying that there exists a set of pointers P such that P is finite, the value of the expression *root* is in P, every pointer p in P points to a well-defined record, i.e., it has a tag t different from -1, the heap at $p, \ldots, p + \mathit{len}\ t - 1$ is defined and every pointer in the record is a member of P, i.e., P is closed and determines a heap.

4.1.3 Correctness

The main correctness criteria of a garbage collector is that it preserves the structure of the heap, i.e., the old heap and the new heap should be isomorphic, and it should preserve the heap invariant. To make this precise we define what it means to have a heap isomorphism.

Definition 4.2 [Heap Isomorphism] A function ϕ is a heap isomorphism between two heaps *oldh* and h, on a set of pointers *ALIVE* if and only if

$$\forall p \in \mathit{ALIVE}.\ \exists t.\ \mathit{oldh}\ p = \mathsf{Some}(t) \wedge t \in \mathrm{dom}\ \mathit{len} \wedge p \in \mathrm{dom}\ \phi\ \wedge \\ \forall i.\ i < \mathsf{the}(\mathit{len}\ t) \supset p + i \in \mathrm{dom}\ \mathit{oldh}\ \wedge (\neg \mathit{isptr}\ t\ i \supset h(\mathsf{the}\ (\phi\ p) + i) = \mathit{oldh}\ (p+i)) \\ \wedge (\mathit{isptr}\ t\ i \supset h(\mathsf{the}\ (\phi\ p) + i) = \mathsf{option\_map}\ \mathsf{int}\ \phi(\mathsf{nat}(\mathsf{the}\ (\mathit{oldh}\ (p+i))))).$$

We say two heaps h, h' are *heap isomorphic* if and only if there exists a heap isomorphism ϕ between h and h'.

4.1.4 Reachability

Cheney's collector only touches the part of the heap that is reachable from the root pointer. Separation logic prides itself as a logic of local reasoning and as such the proof should only deal with the reachable part of the heap. Hence we define a predicate $\mathsf{reachable} : (\mathbb{Z} \to \mathbb{N}_0 \to \mathbb{B}) \to (\mathbb{Z} \to \mathbb{N}_0\ \mathsf{option}) \to \mathbb{Z} \to \mathbb{N}_0\ \mathsf{list} \to \mathbb{H} \to \mathbb{Z}\ \mathsf{option}$ such that $\mathsf{reachable}\ \mathit{isptr}\ \mathit{len}\ r\ \mathit{nl}\ h = \mathsf{Some}(p)$ if and only if p is reachable from r in h by following pointers determined by nl (in the i'th step follow the pointer in the $\mathit{nl}!i$'th field). We also define a constant $\mathsf{reach} : (\mathbb{Z} \to \mathbb{N}_0 \to \mathbb{B}) \to (\mathbb{Z} \to \mathbb{N}_0\ \mathsf{option}) \to \mathbb{Z} \to \mathbb{H} \to \mathbb{N}_0\ \mathsf{set}$ with the meaning

$$\mathsf{reach}\ \mathit{isptr}\ \mathit{len}\ r\ h = \{p \mid \exists \mathit{nl}.\ \mathsf{reachable}\ \mathit{isptr}\ \mathit{len}\ r\ \mathit{nl}\ h = \mathsf{Some}(p) \wedge p > 0\}.$$

Lemma 4.3 *If the heap invariant holds for the set of pointers P and $r \in P$ then P can be divided into two disjoint sets P_1, P_2 such that P_1 is closed and contains r, and every pointer in P_1 is reachable from r. Hence the set of pointers P can be split into a set of pointers determining the part of the heap reachable from r and a set of pointers pointing to garbage.*

The division of pointers into two disjoint sets expressed in the above lemma lets us frame in the correctness proof of the collector next to the garbage to get a complete scenario.

4.1.5 Isomorphisms

Cheney's collector builds an isomorphism between heaps which we will express as a bijection between sets of pointers.

Definition 4.4 We define a constant $\mathsf{Iso} : \alpha\ \mathsf{set} \to \beta\ \mathsf{set} \to (\alpha \to \beta\ \mathsf{option}) \to \mathbb{B}$ with the meaning:

$$\mathsf{Iso}\ A\ B\ \phi = \mathrm{dom}\ \phi = A \wedge \forall a \in A.\ (\exists b \in B.\ \phi\ a = \mathsf{Some}(b))\ \wedge \forall b \in B.\ \exists! a \in A.\ \phi\ a = \mathsf{Some}(b)$$

If $\mathsf{Iso}\ A\ B\ \phi$ we say ϕ is an isomorphism from A to B (written $\phi : A \to B$).

Notice that from $\mathsf{Iso}\ A\ B\ \phi$ and $\phi\ a = \mathsf{Some}(b)$ we can prove $a \in A$ and $b \in B$. Given an isomorphism $\phi : A \to B$ and a $b \in B$ there is a unique a such that $\phi\ a = \mathsf{Some}(b)$. We define a constant $\mathsf{InvIso} : \alpha\ \mathsf{set} \to (\alpha \to \beta\ \mathsf{option}) \to \beta \to \alpha\ \mathsf{option}$ such that if $b \in B$ then $\mathsf{InvIso}\ A\ \phi\ b$ is $\mathsf{Some}(a)$ and if $b \notin B$ then $\mathsf{InvIso}\ A\ \phi\ b = \mathsf{None}$. Notice that if $b \in B$ then a must be a member of A. It turns out we can define this constant using the $\mathsf{Inv} : \alpha\ \mathsf{set} \to (\alpha \to \beta) \to \beta \to \alpha$ constant from the Isabelle/HOL library.

$$\mathsf{InvIso}\ A\ \phi\ b = \mathsf{if}\ \exists x.\ \phi\ x = \mathsf{Some}(b)\ \mathsf{then}\ \mathsf{Some}((\mathsf{Inv}\ A\ (\lambda x.\ \mathsf{the}\ (\phi\ x)))\ b)\ \mathsf{else}\ \mathsf{None}$$

4.1.6 Invariants of the collector

The loop-invariants of the collector is a conjunction of a pure part and a precise part. We start with the precise part. The invariant should relate the current heap with the heap as it was before the garbage collection. In our formalization of higher-order separation logic we encode a formula "$\mathsf{this}(h)$" that holds only for the heap

h. We use that to get a hold on the heap from before the execution of the collector and the following formulas to relate the current heap to the old heap.

Definition 4.5 Let $\mathsf{gh} : (\mathbb{N}_0 \to \mathbb{B}) \to (\mathbb{Z} \to \mathbb{Z}\ \mathsf{option}) \to \mathbb{N}_0 \to \mathbb{N}_0 \to \mathbb{N}_0 \to \mathbb{H} \to (\mathbb{Z}\ \mathsf{list})\ \mathsf{option}$ be a constant such that $\mathsf{gh}\ \mathit{ptest}\ f\ p\ l\ 0\ h$ returns the record pointed to by p of length l from the heap h, with f applied to every pointer in the record. gh is formally defined by primitive recursion on the length l.

Definition 4.6 Let $\mathsf{hs} : (\mathbb{Z} \to \mathbb{N}_0 \to \mathbb{B}) \to (\mathbb{Z} \to \mathbb{N}_0\ \mathsf{option}) \to \mathbb{N}_0\ \mathsf{set} \to \mathbb{N}_0\ \mathsf{set} \to (\mathbb{N}_0 \to \mathbb{N}_0\ \mathsf{option}) \to (\mathbb{N}_0 \to \mathbb{N}_0\ \mathsf{option}) \to \mathbb{H} \to \mathbb{H}\ \mathsf{set}$ be an ISABELLE/HOL constant with the meaning:

$$\mathsf{hs}\ \mathit{isptr}\ \mathit{len}\ P\ P'\ g\ f\ h = \mathsf{Finite\_Set.fold}\ (\mathsf{op}\ *)\ \Theta\ \{\mathsf{empty}\}\ P$$

where Θ is the function

$$\begin{aligned}\lambda p.\ \exists il, t.\ \mathsf{int}(p) \mapsto_l (t\#il) \wedge \mathsf{pure}\ \exists l, on.\ \mathit{len}\ t = \mathsf{Some}(l) \wedge f\ n = \mathsf{Some}(on) \wedge \\ \mathsf{gh}\ (\mathit{isptr}\ t)\ (\lambda i.\ \mathsf{option\_map}\ \mathsf{int}\ (g\ (\mathsf{nat}(i))))\ \mathit{on}\ l\ 0\ h) = \mathsf{Some}(t\#il) \wedge \\ l > 1 \wedge \forall i.\ i < l-1 \wedge \mathit{isptr}\ (i+1) \supset il!i > 0 \wedge \mathsf{nat}(il!i) \in P'\end{aligned}$$

Hence $\mathsf{hs}\ \mathit{isptr}\ \mathit{len}\ P\ P'\ g\ f\ h$ states that every pointer p in P points to an record ob such that every pointer in the record is in P'. And the heap h has an equivalent record at $f\ p$ such that if we map the pointers of that record with g we get ob.

Notice that if f and g are the everywhere defined identity functions ($\lambda n.\ \mathsf{Some}(n)$) then $\mathsf{hs}\ \mathit{isptr}\ \mathit{len}\ P\ P'\ g\ f\ h$ states that the current heap agrees with h on the records determined by P.

Definition 4.7 Let $\mathsf{fw} : (\mathbb{Z} \to \mathbb{N}_0\ \mathsf{option}) \to (\mathbb{N}_0 \to \mathbb{N}_0\ \mathsf{option}) \to \mathbb{N}_0\ \mathsf{set} \to \mathbb{H} \to \mathbb{H}\ \mathsf{set}$ be an ISABELLE/HOL constant with the meaning:

$$\mathsf{fw}\ \mathit{len}\ f\ P\ h = \mathsf{Finite\_Set.fold}\ (\mathsf{op}\ *)\ \Theta\ \{\mathsf{empty}\}\ P$$

where Θ is the function

$$\begin{aligned}\lambda p.\ \exists p', t, \mathit{rest}.\ \mathsf{int}(p') \mapsto_l (-1\#p'\#\mathit{rest}) \wedge \mathsf{pure}\ (f\ p = \mathsf{Some}(p') \wedge p' > 0 \wedge \\ h\ p = \mathsf{Some}(t) \wedge t \in \mathrm{dom}\ \mathit{len}\ \wedge \mathsf{Some}(\mathsf{length}\ \mathit{rest} + 2) = \mathit{len}\ t)\end{aligned}$$

Hence $\mathsf{fw}\ \mathit{len}\ f\ P\ h$ states that for every pointer p in P, p points to an record with tag -1 and a pointer $f\ p$. Moreover, the size of the record is given by the tag found at p in the heap h.

Now that we have the proper heap predicates we can state Lemma 4.3 precisely: Assuming $h \in \mathsf{hinv}\ \mathit{isptr}\ \mathit{len}\ \mathit{root}\ s$ and $\forall t.\ \neg \mathit{isptr}\ t\ 0$ then

$$\begin{aligned}h \in \exists P, P', h, h'.\ \mathsf{pure}\ (\mathsf{funExp}(\lambda r.\ \lambda r.\ \mathsf{nat}(r) \in P \wedge \mathsf{finite}\ P\ \wedge \mathsf{finite}\ P' \wedge \\ P \cap P' = \{\} \wedge P = \mathsf{reach}\ \mathit{isptr}\ \mathit{len}\ r\ h\ \wedge r > 0)\ \mathit{root}) \wedge \\ ((\lambda s.\ \{h\} \wedge \mathsf{hs}\ \mathit{isptr}\ \mathit{len}\ P\ P\ (\lambda n.\ \mathsf{Some}(n))\ (\lambda n.\ \mathsf{Some}(n))\ h) * \\ (\lambda s.\ \{h'\} \wedge \mathsf{hs}\ \mathit{isptr}\ \mathit{len}\ P'\ (P \cup P')\ (\lambda n.\ \mathsf{Some}(n))\ (\lambda n.\ \mathsf{Some}(n))\ h'))\end{aligned}$$

Notice the encoding of $\mathsf{this}(h)$ as $\{h\}$.

Assuming we have sets of pointers $QUEUE, FW, UNFORW, DONE, ALIVE$, an isomorphism ϕ, an old heap $oldh$, and an integer list d and that $ALIVE = FW \cup UNFORW$ then the impure part of the outer loop invariant is, given a store s:

$$\begin{aligned}&\mathsf{hs}\ isptr\ len\ UNFORW\ ALIVE\ (\lambda x.\,\mathsf{Some}(x))\ (\lambda x.\,\mathsf{Some}(x))\ oldh\ *\\&\mathsf{hs}\ isptr\ len\ DONE\ (DONE \cup QUEUE)\ \phi\ (\mathsf{InvIso}\ FW\ \phi)\ oldh\ *\\&\mathsf{hs}\ isptr\ len\ QUEUE\ ALIVE\ (\lambda x.\,\mathsf{Some}(x))\ (\mathsf{InvIso}\ FW\ \phi)\ oldh\ *\\&\mathsf{fw}\ len\ \phi\ FW\ oldh * s\ freep \mapsto_l d,\end{aligned}$$

which informally means that the heap can be split into: The part of the heap determined by $UNFORW$ which is as in $oldh$, the part determined by $DONE$ which has been updated according to ϕ, the part determined by $QUEUE$ which has yet to be updated, and we may find the records in $oldh$ by following the inverse of the isomorphism ϕ, the part of the heap pointed to by pointers in FW where each record represents a part of ϕ, and some free space.

To keep track of the space needed to complete the garbage collection we define a constant heap_size.

Definition 4.8 Let heap_size : $(\mathbb{Z} \to \mathbb{Z}\ \mathsf{option}) \to \mathbb{N}_0\ \mathsf{set} \to \mathbb{H} \to \mathbb{N}_0$ be the Isabelle/HOL constant with the meaning:

$$\mathsf{heap\_size}\ len\ P\ h = \textstyle\sum_{p \in P} \mathsf{case}\ h\ p\ \mathsf{of}\ \mathsf{None} \Rightarrow 0 \mid \mathsf{Some}(t) \Rightarrow \mathsf{case}\ len\ t\ \mathsf{of}\ \mathsf{None} \Rightarrow 0 \mid \mathsf{Some}(l) \Rightarrow l$$

The pure part of the heap invariant is derived from the assumptions of the following lemmas.

The following lemma shows that any record, pointed to by a pointer p reachable from r in the old heap $oldh$, has been been copied to the to-space, and thus no record alive is left behind.

Lemma 4.9 *Assume*

$$\begin{aligned}&\forall t.\ \neg isptr\ t\ 0 \wedge \mathsf{finite}\ DONE \wedge p \in \mathsf{reach}\ isptr\ len\ (\mathsf{int}(r))\ oldh \wedge r \in FW\ \wedge\\&\quad h \in \mathsf{hs}\ isptr\ len\ DONE\ DONE\ \phi\ (\mathsf{InvIso}\ FW \phi)\ oldh \wedge \mathsf{Iso}\ FW\ DONE\ \phi\end{aligned}$$

Then $p \in FW$.

The lemma below expresses that if $QUEUE$ is nonempty and the invariant ensures that the records determined by $QUEUE$ are densely packed in the interval $[n, \ldots, f[$, then we may conclude that the least pointer in $QUEUE$ is n.

Lemma 4.10 *Assume*

$$\begin{aligned}&\forall t.\ \neg isptr\ t\ 0 \wedge \mathsf{finite}\ DONE \wedge \mathsf{Iso}\ FW\ (DONE \cup QUEUE)\ \phi\ \wedge\\&DONE \cap QUEUE = \{\} \wedge QUEUE \neq \{\} \wedge QUEUE \subseteq [n, \ldots, f[\ \wedge\\&h \in \mathsf{hs}\ isptr\ len\ QUEUE\ ALIVE\ (\lambda x.\,\mathsf{Some}(x))\ (\mathsf{InvIso}\ FW \phi)\ oldh\ \wedge\\&\forall P, \psi.\ \mathsf{Iso}\ P\ QUEUE\ \psi \wedge \psi \prec \phi \supset n + \mathsf{heap\_size}\ isptr\ len\ P\ oldh = f\\&\forall p \in QUEUE.\ p + \mathsf{the}\ (len(\mathsf{the}\ (oldh(\mathsf{the}\ (\mathsf{InvIso}\ FW\ \phi))))) \leq f\end{aligned}$$

Then $n \in QUEUE$.

Using the lemmas above we have given a proof of the following Hoare triple in Isabelle/HOL.

$$\begin{gathered}
\Gamma \vdash \{\mathsf{pure}\ (\mathsf{funExp}(\lambda r.\ \mathsf{nat}(r) \in \mathit{ALIVE} \wedge \mathsf{finite}\ \mathit{ALIVE} \wedge \mathit{ALIVE} = \mathsf{reach}\ \mathit{isptr}\ \mathit{len}\ r\ \mathit{oldh} \wedge r > 0) \\
(\mathsf{trivExp}\ \mathit{rootp})) \wedge (\lambda s.\ \ \{\mathit{oldh}\} \cap \mathsf{hs}\ \mathit{isptr}\ \mathit{len}\ \mathit{ALIVE}\ \mathit{ALIVE}\ (\lambda x.\ \mathsf{Some}(x))\ (\lambda x.\ \mathsf{Some}(x))\ \mathit{oldh}) * \\
(\exists d.\ \mathsf{trivExp}\ \mathit{freep} \mapsto_l \mathsf{map}\ \mathsf{intExp}\ d \wedge \mathsf{pure}\ (\mathsf{boolExp}\ (|d| = \mathsf{heap\_size}\ \mathit{len}\ \mathit{ALIVE}\ \mathit{oldh})))\} \\
\mathit{cheney} \\
\{\exists \mathit{oldroot}, \mathit{DONE}, \phi.\ \mathsf{pure}\ (\mathsf{funExp}(\lambda r, f.\ \mathsf{nat}(\mathit{oldroot}) \in \mathit{ALIVE} \wedge 0 < \mathit{oldroot} \wedge \mathsf{finite}\ \mathit{ALIVE}\ \wedge \\
\mathit{ALIVE} = \mathsf{reach}\ \mathit{isptr}\ \mathit{len}\ \mathit{oldroot}\ \mathit{oldh} \wedge 0 < r \wedge \mathsf{Iso}\ \mathit{ALIVE}\ \mathit{DONE}\ \phi \wedge \mathsf{nat}(r) \in \mathit{DONE}\ \wedge \\
\mathit{DONE} \subseteq [\mathsf{nat}(r), \ldots, \mathsf{nat}(f)[)) \wedge (\lambda s.\ \mathsf{hs}\ \mathit{isptr}\ \mathit{len}\ \mathit{DONE}\ \mathit{DONE}\ \phi\ (\mathsf{InvIso}\ \mathit{ALIVE}\ \phi)\ \mathit{oldh}\ * \\
\hfill \mathsf{fw}\ \mathit{len}\ \phi\ \mathit{ALIVE}\ \mathit{oldh})\}
\end{gathered}$$

where *cheney* is Cheney's garbage collector.

It is easy to show that the post condition of Cheney's collector implies that the isomorphism $\phi : \mathit{ALIVE} \to \mathit{DONE}$ is a heap isomorphism in the sense of definition 4.2.

4.2 Copying a dag to a tree

In [18] John Reynolds gives a procedure $\mathsf{copytree}([j];[i])$ that copies a tree pointed to by i to a newly allocated tree pointed to by j. The procedure uses recursion, first on the left subtree and then on the right subtree. As sketched in [18], if you give the procedure a pointer to a dag then the result will be a tree where any shared structure in the dag is copied multiple times to eliminate the sharing. We did the exercise of formalizing this proof in our Isabelle/HOL implementation. The exercise illustrates the use of higher-order separation logic to quantify over heap sets across a Hoare triple, it exercises our proof rules for procedures, and Lemma 3.4 is needed to simplify the assumptions about the procedure to what is originally sought. The proof also relies crucially on the rule for lookup in Section 3.2. The main observation about this proof is that it fits entirely within higher-order separation logic in the sense that all the needed predicates are definable in our implementation.

We assume we are given a standard definition of trees as an inductive datatype tree in Isabelle/HOL. Then it is easy to define a predicate $\mathsf{htree} : \mathsf{tree} \to \mathbb{Z}\ \mathsf{exp} \to \mathbb{A}$ that given a tree and an expression specifies that the expression points to a representation of the tree in the heap. Following Reynolds we can also give a predicate $\mathsf{hdag} : \mathsf{tree} \to \mathbb{Z}\ \mathsf{exp} \to \mathbb{A}$ that given a tree and an expression asserts that the expression points to a dag:

Definition 4.11 Let $\mathsf{hdag} :\ \mathsf{tree} \to \mathbb{Z}\ \mathsf{exp} \to \mathbb{A}$ be defined by primitive recursion on tree as follows:

$$\begin{aligned}
\mathsf{hdag}\ \mathsf{Atom}\ \ \ e &= \mathsf{pure}\ (e = 0) \\
\mathsf{hdag}\ \mathsf{Node}(t_1, t_2)\ e &= \exists i_1, i_2 : \mathbb{Z}.\ \big(e \mapsto [\mathsf{int}(i_1), \mathsf{int}(i_2)] * \\
&\qquad (\mathsf{hdag}\ t_1\ \mathsf{int}(i_1) \wedge \mathsf{hdag}\ t_2\ \mathsf{int}(t_2))\big)
\end{aligned}$$

We would like to show

$$\forall t :\ \mathsf{tree}.\ \Gamma \vdash \{\mathsf{hdag}\ t\ e\}\ \mathsf{copytree}([j];[e])\ \{\mathsf{hdag}\ t\ e * \mathsf{htree}\ t\ j\} \qquad (1)$$

As Reynolds points out, following the proof for copying trees doesn't work, as we will be unable to give a proof of the necessary specification for the first recursive call. Instead we prove the stronger specification

$$\forall p : \mathsf{heap\ set}.\ \forall t : \mathsf{tree}.\ \Gamma \vdash \{(\lambda s.\ p) \wedge \mathsf{hdag}\ t\ i\}\ \mathsf{copytree}([j];[i])\ \{(\lambda s.\ p) * \mathsf{htree}\ t\ j\},$$

which uses quantification over arbitrary heap sets, a key property of higher-order separation logic.

The proof that the procedure body satisfies the above specification has two important parts. The first is that in order to look up the pointers to the sub-nodes of the dag we are faced with a precondition of the sort $(\lambda s.\ p) \wedge (v_0 \mapsto i_0 * v_0 + 1 \mapsto i_1)$ for some arbitrary p. We need the derived lookup rule from Section 3.2 to avoid the complication of Lemma 3.3 to get the standard lookup to work. The second important part is that in the first recursive call we can instantiate the first quantifier in the assumed specification with $p \wedge \mathsf{hdag}\ t_2\ i_2$, and then the proof works out.

Finally, we remark that the original specification in (1) of course follows from the stronger specification which we showed by means of Lemma 3.4 and the rule of consequence.

5 Related work

We have mentioned related work along the way. In this section we discuss other related work on formalizing separation logic.

In [19] Weber describes a formalization of first-order separation logic for a simple while-language without procedures in Isabelle/HOL and prove the soundness of the frame rule. He also includes a simple example, the verification of in-place list reversal.

Preoteasa [17] devised a formalization of first-order separation logic in PVS for a language with recursive procedures based on a predicate-transformer semantics. Again, the main result was the formal soundness proof the frame rule.

Here we extend these previous works with a formalization of higher-order separation logic for a language with procedures and substantial case studies, in particular the verification of Cheney's algorithm.

Our formalization of Cheney's garbage collector enables separate verification of mutators and the garbage collector, as is also the case in the recent work by McCreight, Shao, Lin and Li [9] on a Coq-implementation of Hoare logic for verification of garbage collectors written in assembly code.

6 Conclusion and Future Work

We have presented a formalization in Isabelle/HOL of higher-order separation logic for a language with simple procedures and address arithmetic. We have explored the applicability of the formalization by verifying non-trivial algorithms in it, and have found that the use of higher-order separation logic does indeed simplify formalized proofs as conjectured in [1]. In particular, our new formalized higher-order separation logic proof of correctness of Cheney's copying collector is at the same time both more general and simpler than the one in [2].

Future work includes further case studies of formal verification of programs, extending the formalization to languages with more features, e.g., concurrency, first class functions, and/or heaps with procedures. Future work also includes more automated reasoning in higher-order separation logic with, for instance, tactics for entailment of higher-order separation logic formulas.

7 Acknowledgements

We thank John Reynolds for interesting discussions regarding our system, and the anonymous referees for their helpful comments.

References

[1] Biering, B., L. Birkedal and N. Torp-Smith, *Bi-hyperdoctrines, higher-order separation logic, and abstraction*, ACM Trans. Program. Lang. Syst. **29** (2007).

[2] Birkedal, L., N. T. Smith and J. C. Reynolds, *Local reasoning about a copying garbage collector*, in: *Proceedings of the 31st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (2004), pp. 220–231.

[3] Brookes, S., *A semantics for concurrent separation logic*, Theoretical Computer Science **375** (2007), pp. 227–270.

[4] Cheney, C. J., *A nonrecursive list compacting algorithm*, Commun. ACM **13** (1970), pp. 677–678.

[5] Church, A., *A formulation of the simple theory of types*, The Journal of Symbolic Logic **5** (1940), pp. 56–68.

[6] Gordon, M., *Introduction to the hol system*, in: *HOL Theorem Proving System and Its Applications, 1991., International Workshop on the*, 1991, pp. 2–3.

[7] Krishnaswami, N., J. Aldrich and L. Birkedal, *Modular verification of the subject-observer pattern via higher-order separation logic*, in: *9th Workshop on Formal Techniques for Java-like Programs (FTfJP 2007)*, 2007.

[8] Lin, C., A. Mccreight, Z. Shao, Y. Chen and Y. Guo, *Foundational typed assembly language with certified garbage collection*, in: *TASE '07: Proceedings of the First Joint IEEE/IFIP Symposium on Theoretical Aspects of Software Engineering* (2007), pp. 326–338.

[9] McCreight, A., Z. Shao, C. Lin and L. Li, *A general framework for certifying garbage collectors and their mutators*, SIGPLAN Not. **42** (2007), pp. 468–479.

[10] Müller, O., T. Nipkow, D. Von Oheimb and O. Slotosch, *Holcf = hol + lcf*, J. Funct. Program. **9** (1999), pp. 191–223.

[11] Nanevski, A., A. Ahmed, G. Morrisett and L. Birkedal, *Abstract Predicates and Mutable ADTs in Hoare Type Theory*, in: *Proceedings of ESOP'07*, LNCS **4421**, 2007, pp. 189–204.

[12] O'Hearn, P., *Resources, concurrency, and local reasoning*, Theoretical Computer Science **375** (2007).

[13] O'Hearn, P. W., J. C. Reynolds and H. Yang, *Local reasoning about programs that alter data structures*, in: *CSL '01: Proceedings of the 15th International Workshop on Computer Science Logic* (2001), pp. 1–19.

[14] Parkinson, M., *When separation logic met java*, in: *FTfJP'06*, 2006.

[15] Parkinson, M. and G. Biermann, *Separation logic, abstraction and inheritance*, in: *Proc. 35th POPL*, 2008.

[16] Paulson, L. C., *Isabelle: The next seven hundred theorem provers*, in: *Proceedings of the 9th International Conference on Automated Deduction* (1988), pp. 772–773.

[17] Preoteasa, V., *Mechanical verification of recursive procedures manipulating pointers using separation logic*, in: *14th International Symposium on Formal Methods*, 2006, pp. 508–523.

[18] Reynolds, J. C., *Separation logic: A logic for shared mutable data structures* (2002).

[19] Weber, T., *Towards mechanized program verification with separation logic*, in: *Proceedings of CSL'04*, LNCS **3210** (2004), pp. 250–264.

[20] Yang, H., *An example of local reasoning in bi pointer logic: the schorr-waite graph marking algorithm* (2000).