Theoretical Computer Science 410 (2009) 4216–4233

Contents lists available at ScienceDirect

Theoretical Computer Science

journal homepage: www.elsevier.com/locate/tcs

Frame rule for mutually recursive procedures manipulating pointers

Viorel Preoteasa *

Department of Information Technologies, Åbo Akademi University, Joukahaisenkatu 3-5 B, 20520 Turku, Finland

ARTICLE INFO

Article history:
Received 11 March 2008
Received in revised form 12 May 2009
Accepted 13 May 2009
Communicated by D. Sannella

Keywords:
Predicate transformer semantics
Mechanical verification of programs
Mutually recursive procedures
Pointers
Separation logic
Frame rule

ABSTRACT

Using a predicate transformer semantics of programs, we introduce statements for heap operations and separation logic operators for specifying programs that manipulate pointers. We prove a powerful Hoare total correctness rule for mutually recursive procedures manipulating pointers. The rule combines earlier proof rules for (mutually) recursive procedures with the frame rule for pointer programs. The theory, including the proofs, is implemented in the theorem prover PVS. In this implementation program variables and addresses can store values of almost any type of the theorem prover.

1. Introduction

Pointers are an important programming concept and they provide an effective and efficient solution to many programming tasks. Moreover, object oriented languages rely explicitly (C++, Pascal), or implicitly (Java, C#, Python, Eiffel) on pointers. Burstall [5] has introduced a logic for reasoning about programs with pointers. Based on Burstall's ideas Reynolds [19] describes the separation logic, a more general and abstract logic for reasoning about correctness of pointer programs. This logic combines ideas from [12,23,9].

Pointer manipulating programs are difficult to construct and even more difficult to verify mainly due to aliasing. For example in C++ language we could have two pointers to integer numbers: int $*x$, $*y$. However, after the assignment $*x := 7$ we cannot say anything about the value of $*y$. If addresses x and y are different, then $*y$ is unchanged, otherwise $*y$ is 7. We cannot say just by looking at the program $*x := 7$ what will be the effect on $*y$. If we know that $x \neq y \wedge *x = 1 \wedge *y = 1$, then after executing the assignment given above we know that $x \neq y \wedge *x = 7 \wedge *y = 1$ will be true. In other words: we have to know more about x and y in order to know that $*y$ is not modified.

Separation logic was introduced for specifying in a more abstract manner non-aliasing properties about pointers. Within separation logic two new predicate operators were introduced: *singleton heap* ($\mapsto$) and *separation conjuction* ($*$). The predicate $x \mapsto a$ is true in those computation states where the heap contains only one address x, and the value stored at address x is a. The predicate $p * q$ is true in a computation state s if we can split the heap of s such that p is true for one component of the heap and q is true for the second component. For example the predicate $(x \mapsto 1) * (y \mapsto 1)$ is true in a state where the heap contains two distinct addresses (x, y), and 1 is stored at both addresses x, and y. Using separation logic, if we know that $(x \mapsto 1) * (y \mapsto 1)$ is true before executing the program $*x := 7$, then after the execution is true $(x \mapsto 7) * (y \mapsto 1)$.

Most approaches of reasoning about pointer programs treat the heap globally, even if programs modify only a small and well-defined part of it. Separation logic has introduced a *frame rule* [23] which enables *local reasoning* about pointer programs. The original *Hoare frame rule* states that if the *Hoare triple* $p \{\!| S |\!\} q$ is true and r is a predicate which does not

* Tel.: +358 405609056.

0304-3975/$ – see front matter
doi:10.1016/j.tcs.2009.05.016

contain variables modified by S, then the triple $p \wedge r \ \{\!|\ S\ |\!\}\ q \wedge r$ is also true. However, this rule does not hold for pointer programs. Using this frame rule we would be able to prove $*x = 1 \wedge *y = 1 \ \{\!|\ *x := 7\ |\!\}\ *x = 7 \wedge *y = 1$ which is not true if addresses x and y are the same. If instead of conjuction in the Hoare frame rule we use separation conjuction, then the new rule holds also for pointer programs.

$$p \ \{\!|\ S\ |\!\}\ q \Rightarrow p * r \ \{\!|\ S\ |\!\}\ q * r$$

provided that r does not contain variables modified by S.

The frame rule becomes very important when reasoning about programs with procedures. Assume that the predicate $tree.x$ is true on those states where the heap contains only a tree with the root at address x. The specification of a procedure which disposes a tree from memory could be written:

$$tree.x \ \{\!|\ \mathsf{DisposeTree}(x)\ |\!\}\ \mathsf{emp} \tag{1}$$

where emp is a predicate which is true in those states where the heap is empty. When we prove the correctness of DisposeTree we would like to deal only with addresses changed as specified in (1). However, we would like to use DisposeTree in contexts where the heap also contains some other addresses:

$$p * tree.x \ \{\!|\ \mathsf{DisposeTree}(x)\ |\!\}\ p * \mathsf{emp} \tag{2}$$

The frame rule enables us to use (2), but prove only (1). Originally the frame rule was proved in [23] for a simple while language without procedures.

Separation logic [18,12,19,23] is a powerful tool for proving correctness of imperative programs that manipulate pointers. However, without theorem prover support, such tasks are unfeasible. By employing Isabelle/HOL [11] theorem prover and separation logic, Weber [22] implements relatively complete Hoare [8] logics for a simple while programming language extended with heap operations. Nevertheless, his implementation does not treat (recursive) procedures and local variables.

Mutually recursive procedures are also a very important programming concept which is used for example in programs written in an object oriented language. Nipkow [10] has introduced a complete Hoare logic for a language with parameterless mutually recursive procedures.

In this paper, we introduce a predicate transformer semantics for imperative programs with pointers and define separation logic constructs. We treat mutually recursive procedures with parameters and local variables. The contributions of this paper could be summarized as follows.

(1) We introduce an abstract Hoare total correctness rule for mutually recursive procedures. This rule is a generalization of rules from [10,2,16] and can be specialized in a rule combining the frame rule [9,23] and the rule for mutually recursive procedures [10], but allowing procedures with value and result parameters and local variables.
(2) We verify a complex example of a collection of mutually recursive procedures which parses arithmetical expressions.
(3) Our work is implemented in the theorem prover PVS [13] and it is based on a previous formalization [2] of Refinement Calculus [3] with recursive procedures.

Extending the frame rule to a language with mutually recursive procedures with parameters is both motivating and challenging. As shown in the example above, it is desirable to prove the correctness statement of a procedure independent of the contexts in which the procedure would operate. Later, if the procedure is used in a specific context, then the procedure correctness statement could be extended using the frame rule to be applicable to that context. The challenge comes from the fact that the frame rule is proved by structural induction on programs, therefore adding recursive procedure calls means that the frame rule must be proved for them too. One may argue that the semantics of recursive procedures is similar to the semantics of the while statement. This is partially true, but mutually recursive procedures are more general than a simple while statement and there is an additional complication to be considered when dealing with recursive procedures with result parameters. The formulas that can be added to a correctness statement $p \ \{\!|\ S\ |\!\}\ q$ in the frame rule are those which do not contain free variables modified by the program S. If we have for example a procedure $\mathsf{parse}(s, x)$ which recursively builds in the result parameter x a pointer representation of the parsing tree of the expression contained in the string s, then the correctness statement associated to the procedure parse is parameterized by the possible actual result parameters and so are the formulas to be added in the frame rule.

This paper is a completion of the paper [16]. In [16] we studied the separation logic and the frame rule for programs with recursive procedures, but we only stated the theorem about recursive procedures. Here we concentrate on the proof of the Hoare correctness rule for mutually recursive procedures, and we introduce some definitions about separation logic. The paper [16] treats exhaustively the separation logic concepts which are mentioned here.

2. Related work

Following [23], Weber [22] implements in theorem prover Isabelle/HOL relatively complete Hoare logics for total and partial correctness of a simple while language with pointers where variables and addresses store only natural numbers. Nevertheless, his implementation does not treat (recursive) procedures and local variables.

In [23,22] memory faults are modeled by transitions to a special state *fault*. When giving semantics of partial correctness Hoare triples, the programs are required to avoid the state *fault*. In our approach memory faults are modeled by non-termination and our semantics is equivalent to the total correctness semantics from [23,22].

Reynolds, Yang, O'Hearn and Weber [19,23,22] require an infinite supply of addresses and the assumption that only a finite number of them are allocated during the program execution. This assumption is needed for address allocation statement which should always succeed. We do not need these restrictions. By using the demonic update statement [3] to define address allocation we obtain a simpler semantics which yields the same proof rules as in [19,23,22]. In our approach, if new addresses are not available then the program terminates successfully. This treatment is equivalent to the one where we require that addresses are always available for allocation. Both of these treatments are unfeasible in practice, but most approaches to pointer semantics use one of them.

The proof of the frame rule in [23] is a consequence of the *frame* and *safety (termination) monotonicity* properties and these are proved by induction on the program structure. Although we could state the termination monotonicity property in our semantics, it does not seem obvious how to represent the frame property. Our proof of the frame rule is done directly by induction on programs.

Similarly to [23,22], non-deterministically choosing a new available address in the allocation statement is essential in proving the frame rule.

In [19,23,22] addresses are natural numbers and variables and addresses can store only natural numbers. The fields of pointer structures are recorded at successive addresses. Although it is possible to reason in this manner about high level programming languages, the semantics is at the level of an assembly language and we cannot take advantage of any type checking mechanism that would simplify the verification work. In our approach a given address can store only values of a specified type and this fact is ensured by the theorem prover. We can have record types and addresses can store these records. We could easily implement address arithmetic, but we would use it for representing dynamically allocated arrays of arbitrary types rather than record fields.

In [14], Parkinson introduces local reasoning and separation logic for a significant fragment of Java. He proves Hoare partial correctness rules for this language with respect to an operational semantics. Similar to our case, the separation logic is defined in a context where addresses could hold values of various types, and not only integers as in the traditional separation logic. The rule for procedures is introduced for static method calls, and then extended to dynamic dispatch based on a new formulation of behavioral subtyping called specification compatibility. In our framework the specification compatibility could be expressed as refinement between the base class method specification and the extension class method specification. The abstract predicates from [14] seem to correspond to the parametric predicates introduced in [1,2] which we use in the present paper.

Using separation logic, Varming and Birkedal [21] have implemented in the Isabelle theorem prover the semantics for a language with mutually recursive procedures manipulating pointers. As in our case, their specifications can contain higher-order assertions easing the reasoning about complex algorithms with specifications requiring higher-order constructs. Unlike us, they treat only partial correctness and their program variables range only over integers. The procedures in [21] can refer only local variables, but not global.

Birkedal and Yang [4] are building a logic for proving that a program using a module would perform independently of the actual implementation of the module. Their major concern is to solve the problem that in standard models of separation logic the identity of addresses can be observed in the model. This leads in some cases to the impossibility to prove that two similar programs compute the same result if final results of the two programs differ only on the addresses from the heap. A similar problem occurs also in our case. In our approach, and many works in separation logic, one would not be able to prove the frame rule for the allocation statement unless this statement is non-deterministic. Although Birkedal and Yang prove a very general frame rule for an imperative programming language with pointers, they have a simplifying assumption. The variables of their language are immutable, that is, they are local variables which are set when introduced, and they are not changed within their scope. This makes it possible to derive a very general frame rule ($\alpha \Rightarrow \alpha \otimes P$) which does not need any syntactic condition that P does not contain free variables modified by a program occurring in α. They have mutually recursive programs, but they do not have result parameters. In our work the key element is the interaction between the recursion and result parameters.

3. Preliminaries

In this section, we introduce a predicate transformer semantics of imperative programs. We do not consider programs as syntactic entities, but rather we work directly with their semantic interpretations (monotonic predicate transformers). We use higher-order logic [6] as the underlying logic. We recall some facts about complete lattices and fixpoints [7], and about the refinement calculus [3].

If $f : A \to B, g : B \to C$ are two functions, and $x \in A$, then we denote function application by $f.x$, forward function composition by $f ; g$ ($(f ; g).x = g.(f.x)$), and backward function composition by $g \circ f$ ($(g \circ f).x = g.(f.x)$). As an exception to the above notations, $S ; T$ for predicate transformer denotes also backward function composition.

3.1. Complete lattices & least fixpoints

A partially ordered set $\langle L, \leq \rangle$ is called a *complete lattice* if for every subset A of L the least upper bound of A, denoted $\bigvee A$, exists. If L is a complete lattice then every subset A of L has also greatest lower bound ($\bigwedge A$) and L has least and greatest

elements denoted by $\bot$ and $\top$ respectively. By Knaster–Tarski [20] Theorem we know that all monotonic functions $f : L \to L$ have the least fixpoint denoted by μf.

Lemma 1. *If $L' \subseteq L$ is a complete sublattice of L and if $f : L \to L$ is monotonic such that $f.L' \subseteq L'$ then $\mu_L f \in L'$, and $\mu_L f = \mu_{L'} f$.*

Proof. See Corollary 6, page 18 from [17]. □

If A_i is a family of non-empty sets indexed by $i \in I$ then we denote by $\prod_{i\in I} A_i$ or just $\prod_i A_i$ when I is fixed, the Cartesian product of A_i's. If $a \in \prod_i A_i$ then $a_i \in A_i$ denotes the ith component of a. Conversely, if for every $i \in I$, $b_i \in A_i$, then $(b_i)_{i\in I} \in \prod_i A_i$ denotes the tuple containing the elements b_i. If $f \in \prod(A_i \to B_i)$ and $x \in \prod A_i$, then we define $f.x \in \prod_i B_i$ by $(f.x)_i \mathrel{\hat{=}} f_i.x_i$.

If L is a complete lattice and A a non-empty set, then $A \to L$ together with the pointwise extensions of all operations on L to $A \to L$ is a complete lattice. Similarly, if for each $i \in I$, L_i is a complete lattice, then $\prod_i L_i$ together with the component-wise extensions of all operations from L_i to $\prod_i L_i$ is a complete lattice.

Theorem 2. *If $f : \prod_i L_i \to \prod_i L_i$ is monotonic and $\hat{f} : \prod_i(A_i \to L_i) \to \prod_i(A_i \to L_i)$ is given by $(\hat{f}.x)_i.a_i = (f.(\bigvee_{b\in A} x.b))_i$, then $\hat{f}$ is monotonic and $(\forall a \in A \bullet (\mu \hat{f}).a = \mu f)$, where $A = \prod_i A_i$.*

Proof. The fact that $\hat{f}$ is monotonic follows directly from the definition.

We show that $(\mu \hat{f}).a = \mu f$ by showing that $(\mu \hat{f}).a$ is a fixpoint of f and $(\lambda a_i \bullet \mu f_i)_{i\in I}$ is a fixpoint of $\hat{f}$. First we prove $(\forall a, c \in A \bullet (\mu \hat{f}).a = (\mu \hat{f}).c)$:

$$(\mu \hat{f}).a = \hat{f}.(\mu \hat{f}).a = f.\left(\bigvee_{b\in A} (\mu \hat{f}).b\right) = \hat{f}.(\mu \hat{f}).c = (\mu \hat{f}).c$$

We have

$$f.((\mu \hat{f}).a) = f.\left(\bigvee_{b\in A} (\mu \hat{f}).b\right) = \hat{f}.(\mu \hat{f}).a = (\mu \hat{f}).a$$

and

$$\hat{f}.((\lambda a_i \bullet \mu f_i)_{i\in I}).a = f.\left(\bigvee_{b\in A} (\lambda a_i \bullet \mu f_i)_{i\in I}.b\right) = f.\left(\bigvee_{b\in A} \mu f\right) = f.(\mu f) = \mu f$$

It follows that $(\mu \hat{f}).a = \mu f$. □

3.2. Predicates & predicate transformers

Let Σ be the state space. Predicates, denoted Pred, are the functions from $\Sigma \to$ Bool. We denote by $\subseteq$, $\cup$, and $\cap$ the predicate inclusion, union, and intersection respectively. The type Pred together with inclusion forms a complete boolean algebra.

Prog is the type of all monotonic functions from Pred to Pred. Programs are modeled as elements of Prog. If $S \in$ Prog and $p \in$ Pred, then $S.p \in$ Pred are all states from which the execution of S terminates in a state satisfying the postcondition p. The program *sequential composition* denoted $S\ ;\ T$ is modeled by the functional composition of monotonic predicate transformers, i.e. $(S\ ;\ T).p = S.(T.p)$. We denote by $\sqsubseteq$, $\sqcup$, and $\sqcap$ the pointwise extensions of $\subseteq$, $\cup$, and $\cap$, respectively. The type Prog, together with the pointwise extension of the operations on predicates, forms a complete lattice. The partial order $\sqsubseteq$ on Prog is the *refinement relation* [3]. The predicate transformer $S \sqcap T$ models *non-deterministic* choice — the choice between executing S or T is arbitrary.

We often work with predicate transformers based on functions or relations. A deterministic program can be modeled by a function $f : \Sigma \to \Sigma$ where the interpretation of $f.\sigma$ is the state computed by the program represented by f that starts from the initial state σ. We can model a non-deterministic program by a relation on Σ, i.e. a function $R : \Sigma \to \Sigma \to$ Bool. The state σ' belongs to $R.\sigma$ if there exists an execution of the program starting in σ and ending in σ'.

If $b, p, q \in$ Pred, $R : \Sigma \to \Sigma \to$ Bool, $f : \Sigma \to \Sigma$, then we define

- *Deterministic update* statement

$$[f] : \text{Prog} \mathrel{\hat{=}} (\lambda q \bullet \lambda \sigma \bullet q.(f.\sigma))$$

- *Demonic update* statement

$$[R] : \text{Prog} \mathrel{\hat{=}} (\lambda q \bullet \lambda \sigma \bullet \forall \sigma' \bullet R.\sigma.\sigma' \Rightarrow q.\sigma')$$

- *Assert* statement

$$\{p\} : \text{Prog} \mathrel{\hat{=}} (\lambda q \bullet p \cap q)$$

- *Conditional* statement

$$\text{if } b \text{ then } S \text{ else } T \text{ fi} : \text{Prog} \mathrel{\hat{=}} (\{b\} \ ; \ S) \sqcup (\{\neg b\} \ ; \ T)$$

- *Skip* statement

$$\text{skip} : \text{Prog} = (\lambda p \bullet p)$$

- *Postcondition* statement

$$[\![p]\!] : \text{Prog} \mathrel{\hat{=}} (\lambda q \bullet \lambda \sigma \bullet p \subseteq q)$$

Definition 3. If α and β are predicates and S is a program, then a *Hoare total correctness triple*, denoted $\alpha \ \{\!|\ S\ |\!\}\ \beta$ is true if and only if $\alpha \subseteq S.\beta$.

Lemma 4. *If p, q are predicates, R, Q are relations and S_i, T are predicate transformers then the following hold:*

(i) $\{p \wedge q\} = \{p\} \ ; \ \{q\}$
(ii) $[R \ ; \ Q] = [R] \ ; \ [Q]$
(iii) $(\bigsqcap_{i\in I} S_i) \ ; \ T = \bigsqcap_{i\in I}(S_i \ ; \ T)$
(iv) $(\bigsqcup_{i\in I} S_i) \ ; \ T = \bigsqcup_{i\in I}(S_i \ ; \ T)$
(v) $R \subseteq Q \Rightarrow [Q] \sqsubseteq [R]$
(vi) $\{S.p\} \ ; \ [\![p]\!] \sqsubseteq S$
(vii) $\text{skip} \sqsubseteq \{[\![p]\!].p\}$
(viii) $\{p\} \ ; \ [\![q]\!] \sqsubseteq S \Leftrightarrow p \ \{\!|\ S\ |\!\}\ q$

4. Program variables, addresses, constants & expressions

We assume that we have a type Value that contains all program variables, program addresses, and constants. The type Value is the global type of all values that could be assigned to program variables. We can have program variables of type address, or type integer, and, although not used here, we could have program variables that store other program variables (references). We assume that we have the disjoint subtypes Location and Constant of Value, and the element nil $\in$ Constant. Moreover, we assume that Variable, and Address are disjoint subtypes of Location. The elements of Variable, Address, and Constant represent the program variables, program addresses, and program constants, respectively. The element nil represents the null address. For example, the type of integer numbers, Int, is a subtype of Constant.

For all $x \in$ Location, we introduce the type of x, denoted T.x, as an arbitrary subtype of Value. T.x represents all values that can be assigned to x. For a type $X \subseteq$ Value we define the subtypes Vars.$X \subseteq$ Variable, Addrs.$X \subseteq$ Address, and AddrsNil.$X \subseteq$ Address $\cup$ {nil} by

$$\text{Vars}.X \mathrel{\hat{=}} \{x \in \text{Variable} \mid \text{T}.x = X\}$$
$$\text{Addrs}.X \mathrel{\hat{=}} \{x \in \text{Address} \mid \text{T}.x = X\}$$
$$\text{AddrsNil}.X \mathrel{\hat{=}} \text{Addrs}.X \cup \{\text{nil}\}$$

The type Vars.X represents the program variables of type X. The elements of Addrs.X are the addresses that can store elements of type X. An element of AddrsNil.X is either nil or is an address that can store an element of type X. For example, the program variables of type addresses to natural numbers are Vars.(AddrsNil.Nat). Based on these addresses we will define the heap and the heap operations in Section 5.

In the C++ programming language, and in most imperative programming languages, a binary tree structure will be defined by something like:

$$\text{struct btree\{ int label ; btree } *\text{left ; btree } *\text{right ; \}} \tag{3}$$

In our formalism, binary trees, labeled with elements from an arbitrary type A, are modeled by a type Ptree.A. Elements of Ptree.A are records with three components: $a \in A$, and $p, q \in$ AddrsNil.(Ptree.A). Formally the record structure on Ptree.A is given by a bijective function

$$\text{ptree} : A \times \text{AddrsNil}.(\text{Ptree}.A) \times \text{AddrsNil}.(\text{Ptree}.A) \to \text{Ptree}.A.$$

If $a \in A$, and $p, q \in$ AddrsNil.(Ptree.A), then ptree.(a, p, q) is the record containing the elements a, p, q. The inverse of ptree has three components (label, left, right) having the types:

$$\text{label} : \text{Ptree}.A \to A \text{ and left, right} : \text{Ptree}.A \to \text{AddrsNil}.(\text{Ptree}.A).$$

The type Ptree.Int corresponds to btree from definition (3), and the type AddrsNil.(Ptree.Int) corresponds to (btree $*$) from (3).

We access and update program locations using two functions.

$$\text{val}.x : \Sigma \to \text{T}.x \quad \text{and} \quad \text{set}.x : \text{T}.x \to \Sigma \to \Sigma$$

For $x \in \mathsf{Location}$, $\sigma \in \Sigma$, and $a \in \mathsf{T}.x$, $\mathsf{val}.x.\sigma$ is the value of x in state σ, and $\mathsf{set}.x.a.\sigma$ is the state obtained from σ by setting the value of location x to a.

Local variables are modeled using two statements (add and del), which intuitively correspond to stack operations — adding a location to the stack and deleting it from the stack. Of the two statements, only del is a primitive in our calculus, whereas add is defined as the relation inverse of del

$$\mathsf{del}.x : \Sigma \to \Sigma$$

The behavior of the primitives val, set and del is described using the following axioms.

(a) $\mathsf{val}.x.(\mathsf{set}.x.a.\sigma) = a$
(b) $x \neq y \Rightarrow \mathsf{val}.y.(\mathsf{set}.x.a.\sigma) = \mathsf{val}.y.\sigma$
(c) $\mathsf{set}.x.a \ ; \ \mathsf{set}.x.b = \mathsf{set}.x.b$
(d) $x \neq y \Rightarrow \mathsf{set}.x.a \ ; \ \mathsf{set}.y.b = \mathsf{set}.y.b \ ; \ \mathsf{set}.x.a$
(e) $\mathsf{set}.x.(\mathsf{val}.x.\sigma).\sigma = \sigma$
(f) $\mathsf{del}.x$ is surjective
(g) $x \neq y \Rightarrow \mathsf{del}.x \ ; \ \mathsf{val}.y = \mathsf{val}.y$
(h) $\mathsf{set}.x.a \ ; \ \mathsf{del}.x = \mathsf{del}.x$
(i) $x \neq y \Rightarrow \mathsf{set}.x.a \ ; \ \mathsf{del}.y = \mathsf{del}.y \ ; \ \mathsf{set}.x.a$

Note that although the intuition behind del is described in terms of a stack, the axioms do not make any assumptions about the structure of the program state. Axioms (a)–(e) are the same as the assumptions considered in [3]. The axioms (f)–(i) were introduced in [1,2] to model local variables and procedure parameters.

We extend the operations on program variables to lists of program variables. If $x = (x_1, \ldots, x_n)$ is a list of program variables, then

$$\begin{aligned}
\mathsf{T}.x &= \{(a_1, \ldots, a_n) \mid \forall i \bullet a_i \in \mathsf{T}.x_i\} \\
\mathsf{val}.x.\sigma &= (\mathsf{val}.x_1.\sigma, \ldots, \mathsf{val}.x_n.\sigma) \\
\mathsf{set}.x.a &= \mathsf{set}.x_1.a_1 \ ; \ \ldots \ ; \ \mathsf{set}.x_n.a_n \\
\mathsf{del}.x &= \mathsf{del}.x_1 \ ; \ \ldots \ ; \ \mathsf{del}.x_n
\end{aligned}$$

where $a = (a_1, \ldots, a_n) \in \mathsf{T}.x$. If variables are replaced by lists of variables and the condition $x \neq y$ is replaced by "x, y have no common variables", then the axioms (b)–(i) are true. Moreover, if x is a list with distinct variables, then the axiom (a) holds also for x. We denote the list of program variables by VarList.

Program expressions of type A are functions from Σ to A. We denote by $\mathsf{Expr}.A$ the type of all program expressions of type A. We lift all operations on basic types to operations on program expressions. For example, if $\oplus : A \times B \to C$ is an arbitrary binary operation, then $\oplus : \mathsf{Expr}.A \times \mathsf{Expr}.B \to \mathsf{Expr}.C$ is defined by $e \oplus e' \mathrel{\hat{=}} (\lambda\sigma \bullet e.\sigma \oplus e'.\sigma)$. To avoid confusion, we denote by $(e \doteq e')$ the expression $(\lambda\sigma \bullet e.\sigma = e'.\sigma)$.

For a *parametric boolean expression (predicate)* $\alpha : A \to \Sigma \to \mathsf{Bool}$, we define the boolean expressions

$$\exists .\alpha \mathrel{\hat{=}} \lambda\sigma \bullet \exists a : A \bullet \alpha.a.\sigma \quad \text{and} \quad \forall.\alpha \mathrel{\hat{=}} \lambda\sigma \bullet \forall a : A \bullet e.a.\sigma$$

and denote by $\exists\, a \bullet \alpha.a$ and $\forall a \bullet \alpha.a$ the expressions $\exists .\alpha$ and $\forall.\alpha$, respectively.

If $e \in \mathsf{Expr}.A$, $x \in \mathsf{VarList}$, and $e' \in \mathsf{Expr}.(\mathsf{T}.x)$, then we define $e[x := e']$, the substitution of e' for x in e by $e[x := e'].\sigma = e.(\mathsf{set}.x.(e'.\sigma).\sigma)$.

We also introduce the notion of x-independence for an expression $e \in \mathsf{Expr}.A$, as the semantic correspondent to the syntactic condition that x does not occur free in e. Given $f \in \Sigma \to \Sigma$ and $e \in \mathsf{Expr}.A$, then we say that e is f-independent if $f \ ; \ e = e$. We say that e is $\mathsf{set}.x$-independent if e is $\mathsf{set}.x.a$-independent for all $a \in \mathsf{T}.x$.

Let $x, y \in \mathsf{VarList}$ such that $\mathsf{T}.x = \mathsf{T}.y$ and $e \in \mathsf{Expr}.(\mathsf{T}.x)$. We recall the definition of the assignment statement from [3] and the definition of local variables manipulation statements from [2].

- *Assignment* statement

$$x := e \mathrel{\hat{=}} [\lambda\sigma \bullet \mathsf{set}.x.(e.\sigma).\sigma]$$

- *Add local variable* statement

$$\mathsf{Add}.x \mathrel{\hat{=}} [(\lambda\sigma, \sigma' \bullet \sigma = \mathsf{del}.x.\sigma')]$$

- *Add and initialize local variable* statement

$$\mathsf{Add}.x.e \mathrel{\hat{=}} [(\lambda\sigma, \sigma' \bullet \exists\sigma_0 \bullet \sigma = \mathsf{del}.x.\sigma_0 \wedge \mathsf{set}.x.(e.\sigma).\sigma_0 = \sigma')]$$

- *Delete local variable* statement

$$\mathsf{Del}.x \mathrel{\hat{=}} [\mathsf{del}.x]$$

- *Save and delete local variable* statement

$$\mathsf{Del}.x.y \mathrel{\hat{=}} [(\lambda\sigma \bullet \mathsf{set}.y.(\mathsf{val}.x.\sigma).(\mathsf{del}.x.\sigma))]$$

As mentioned earlier, the program statements $\mathsf{Add}.x$, $\mathsf{Del}.x$ and their variants correspond intuitively to stack operations (adding the value of x to the stack and deleting the top value from the stack and assigning it to x).

The presentation in this section follows the PVS formalization closely. We use the PVS dependent type mechanism to represent the types $\mathsf{T}.x$ and the functions $\mathsf{val}.x$ and $\mathsf{set}.x$. Since in PVS interpreted dependent types can only be subtypes of a given type, we have to work with a type Value containing all variables, addresses, and constants as subtypes. The assumptions stated here about these types are made explicitly in the PVS theories. The axiom (a)–(i) are also stated as such in PVS.

5. Separation logic

This section introduces a model for pointer programs, based on separation logic [19]. Compared to other approaches that use separation logic and in which programs are given an operational semantics, our work uses a predicate transformer semantics for programs. Our theory is implemented in the theorem prover PVS and it is suitable for program verification. Originally, the frame rule [23] was proved for a simple imperative language without (recursive) procedures. Here, the frame rule is proved in the context of a language that allows mutually recursive procedures with value and result parameters and local variables.

So far, we have introduced in Section 4 the mechanism of accessing and updating addresses, but we also need a mechanism for allocating and deallocating them. For this, we introduce the type $\mathsf{AllocAddr} \mathrel{\hat{=}} \mathcal{P}_f.\mathsf{Address}$, the finite power-set of Address, and a special program variable $\mathsf{alloc} \in \mathsf{Variable}$ of type $\mathsf{AllocAddr}$ ($\mathsf{T.alloc} = \mathsf{AllocAddr}$). The set $\mathsf{val.alloc}.\sigma$ contains only those addresses allocated in state σ. The heap in a state σ is made up of the allocated addresses in σ and their values.

For $A, B \in \mathsf{AllocAddr}$, we denote by $A - B$ the set difference of A and B. We introduce two more functions: for adding addresses to a state and for deleting addresses from a state. These functions are:

$$\begin{aligned} \mathsf{addaddr}.A.\sigma &\mathrel{\hat{=}} \mathsf{set.alloc}.(\mathsf{val.alloc}.\sigma \cup A).\sigma \\ \mathsf{dispose}.A.\sigma &\mathrel{\hat{=}} \mathsf{set.alloc}.(\mathsf{val.alloc}.\sigma - A).\sigma \end{aligned}$$

Next, we introduce the separation logic predicates. The predicate emp holds for a state where the set of allocated addresses is empty. If α, β are predicates, then the *separation conjuction* $\alpha * \beta$ holds in a state where the heap can be divided into two disjoint parts, such that α and β hold for the two parts, respectively. The predicate *singleton heap*, $r \mapsto g$, holds in a state where the only allocated address is r and the value stored in r is g. Formally, we have:

Definition 5. If $\alpha, \beta \in \mathsf{Pred}$, $r : \Sigma \to \mathsf{AddrsNil}.X$, and $g : \Sigma \to X$, then we define

$$\begin{aligned} \mathsf{emp}.\sigma : \mathsf{Bool} &\mathrel{\hat{=}} (\mathsf{val.alloc}.\sigma = \emptyset) \\ (\alpha * \beta).\sigma : \mathsf{Bool} &\mathrel{\hat{=}} \exists A \subseteq \mathsf{val.alloc}.\sigma \bullet \alpha.(\mathsf{set.alloc}.A.\sigma) \wedge \beta.(\mathsf{dispose}.A.\sigma) \\ (r \mapsto g).\sigma : \mathsf{Bool} &\mathrel{\hat{=}} \mathsf{val}.(r.\sigma).\sigma = g.\sigma \wedge \mathsf{val.alloc}.\sigma = \{r.\sigma\} \end{aligned}$$

Lemma 6. *The following relations hold*

(i) $\alpha * \mathsf{emp} = \alpha$
(ii) $\alpha * \beta = \beta * \alpha$
(iii) $\alpha * (\beta * \gamma) = (\alpha * \beta) * \gamma$
(iv) $(\exists a \bullet \alpha * \beta.a) = \alpha * (\exists \beta)$
(v) $(\bigcup_{i\in I} p_i) * q = \bigcup_{i\in I}(p_i * q)$
(vi) $(\bigcap_{i\in I} p_i) * q \subseteq \bigcap_{i\in I}(p_i * q)$

Reynolds defines a subset of program expressions called pure [19]. These are expressions that do not depend on the heap, and are the usual program expressions built from program variables, constants and normal (non-separation logic) operators. In our framework, we use two different concepts corresponding to pure expressions. If an expression is set.alloc-independent, then its value does not depend on the allocated addresses. An expression e is called *set address independent*, if e does not depend on the value of any (allocated or not) address. Formally, we have:

$$(\forall u : \mathsf{Address}, a : \mathsf{T}.u \bullet e \text{ is } \mathsf{set}.u.a\text{-independent}).$$

The pure expressions from [19] correspond to set.alloc-independent and set address independent expressions in our framework.

We also need another subclass of program expressions. An expression e is called *non-alloc independent* if e does not depend on the values of non-allocated addresses, that is:

$$\forall \sigma \bullet \forall u \notin \mathsf{val.alloc}.\sigma \bullet \forall a \in \mathsf{T}.u \bullet e.(\mathsf{set}.u.a.\sigma) = e.\sigma.$$

These expressions include all expressions obtained from program variables and constants, by employing all operators (including separation logic operators).

Next, we introduce the pointer manipulation statements.

Definition 7. If $X \subseteq \mathsf{Value}$, $x \in \mathsf{Vars}.(\mathsf{AddrsNil}.X)$, $e : \Sigma \to X$, $r : \Sigma \to \mathsf{AddrsNil}.X$, $y \in \mathsf{Vars}.X$, and $f : X \to \mathsf{T}.y$, then we define:

$$\mathsf{New}.X.(x, e) : \mathsf{Prog} \quad \hat{=} \quad [\lambda\sigma, \sigma' \bullet \exists a : \mathsf{Addrs}.X \bullet \neg\mathsf{alloc}.\sigma.a \wedge \sigma' = \mathsf{set}.a.(e.\sigma).(\mathsf{set}.x.a.(\mathsf{addaddr}.a.\sigma))]$$

$$\mathsf{Dispose}.r : \mathsf{Prog} \mathrel{\hat{=}} \{\lambda\sigma \bullet \mathsf{alloc}.\sigma.(r.\sigma)\} \ ; \ [\lambda\sigma \bullet \mathsf{dispose}.(r.\sigma).\sigma]$$

$$y := r \to f : \mathsf{Prog} \mathrel{\hat{=}} \{\lambda\sigma \bullet \mathsf{alloc}.\sigma.(r.\sigma)\} \ ; \ [\lambda\sigma \bullet \mathsf{set}.y.(f.(\mathsf{val}.(r.\sigma).\sigma)).\sigma]$$

$$[r] \ := \ e : \mathsf{Prog} \mathrel{\hat{=}} \{\lambda\sigma \bullet \mathsf{alloc}.\sigma.(r.\sigma)\} \ ; \ [\lambda\sigma \bullet \mathsf{set}.(r.\sigma).(e.\sigma).\sigma]$$

The statement $\mathsf{New}.X.(x, e)$ allocates a new address a of type X, sets the value of x to a, and sets the value of a to e. The new address to be allocated is chosen arbitrarily form all available addresses, and this fact, similarly to [23,22], is essential in proving the frame rule for the New statement.

The statement $\mathsf{Dispose}.r$ deletes the address r from the allocated addresses. The lookup statement, $y := r \to f$, assigns to y the value of the field f of the record stored at address r. The update statement, $[r] := e$, sets the value of address r to e. If r is not an allocated address in dispose, lookup, or update statements, then these statements do not terminate.

In the traditional work in separation logic the interpretation of Hoare triples includes the guarantee that the programs will not go wrong with respect to memory dereferencing and memory disposal only. However there is no guarantee that a program which is proved correct will not fail due to insufficient memory when new addresses are allocated. Reynolds, Yang, O'Hearn and Weber [19,23,22] restrict the heap structure to make sure that whenever there is a memory allocation statement, there is a free address that can be allocated. In [19,23] this is done by assuming that there is an infinite number of available addresses from which only a finite number are allocated. In Weber's approach [22] the program states are restricted to those states in which there are always an infinite number of free addresses. In both of these approaches one would need to prove that the execution of a program preserves these properties. In our approach there is no restriction on the structure of the state. The assumption that there are always available addresses is embedded in the address allocation statement. Therefore we do not need to prove any invariant property about the state (that after the execution of a program we always have available addresses). In the case when there are no addresses available our programs satisfy any postcondition. The approaches discussed above, including ours, lead to the same proof rule for the memory allocation statement and none of them would guarantee that a "correct" program would not fail due to insufficient memory.

In [16,17] we proved Hoare total correctness rules for these pointer manipulating statements.

6. Mutually recursive procedures

In this section we introduce mutually recursive procedures with parameters and local variables and we give a non-trivial example of a collection of mutually recursive procedures for parsing expressions.

A procedure with parameters from A or simply a procedure over A, is an element from $\mathsf{Proc}.A = A \to \mathsf{Prog}$. The type A is the range of the procedure's actual parameters. A call to a procedure $P \in \mathsf{Proc}.A$ with the actual parameter $a \in A$ is the program $P.a$.

A general non-recursive procedure declaration is:

$$\begin{array}{l}\mathsf{procedure}\ \mathit{name}(\mathsf{value}\ x;\ \mathsf{value\text{—}result}\ y) : \\ \quad \mathit{body}\end{array} \tag{4}$$

where *body* is a program that does not contain any recursive call. The meaning of this procedure declaration is that *name* is a procedure with the list x standing for value parameters and the list y for value–result parameters. When a call is made to *name*, the caller should provide a program expression e of type $\mathsf{T}.x$ and a list of program variables z, with $\mathsf{T}.z = \mathsf{T}.y$ as actual parameters. The intuition behind the call is that first, the formal parameters of the procedure get the values given by e and $\mathsf{val}.z$, then *body* is executed, and finally, the values of the formal parameters y are saved to z.

The procedure declaration (4) is an abbreviation of the following formal definition:

$$\mathit{name} = (\lambda e, z \bullet \mathsf{Add}.(x, y).(e, \mathsf{val}.z) \ ; \ \mathit{body} \ ; \ \mathsf{Del}.x \ ; \ \mathsf{Del}.y.z)$$

By using this approach, any number of local variables can be added to the procedure body. If w are the local variables, then

$$\mathit{body} = \mathsf{Add}.w \ ; \ \mathit{body}_0 \ ; \ \mathsf{Del}.w.$$

If I is a non-empty index set, and A_i, $i \in I$, is a collection of procedure parameter types, then every monotonic function F from the Cartesian product $\prod_i \mathsf{Proc}.A_i$ to $\prod_i \mathsf{Proc}.A_i$ defines a tuple $P \in \prod_i A_i$ of mutually recursive procedures: $P = \mu F$.

For example, we define two mutually recursive procedures that compute whether a given natural number is even or odd.

$$\begin{array}{l}\mathsf{procedure\ even}(\mathsf{value}\ n : \mathsf{Nat},\ \mathsf{value\text{—}result}\ b : \mathsf{Bool}) \\ \quad \mathsf{odd}(\mathsf{val}.n, b) \ ; \ b := \neg\, \mathsf{val}.b \\ \\ \mathsf{procedure\ odd}(\mathsf{value}\ n : \mathsf{Nat},\ \mathsf{value\text{—}result}\ b : \mathsf{Bool}) \\ \quad \mathsf{if}\ \mathsf{val}.n \doteq 0\ \mathsf{then}\ b := \mathsf{false}\ \mathsf{else}\ \mathsf{even}(\mathsf{val}.n - 1, b)\ \mathsf{fi}\end{array} \tag{5}$$

The procedures even and odd can be called by passing an expression e of type Nat and a variable u of type Bool. The procedure call even.(e, u) assigns true to u if the expression e is even, and false otherwise. The type of the parameters of the procedures even and odd is $A = \text{Expr.Nat} \times \text{Vars.Bool}$.

The procedure declarations (5) are abbreviations for the following formal definition.

$$(\text{even},\ \text{odd}) = \mu\ (\lambda S, T \bullet \text{body—even}.T,\ \text{body—odd}.S)$$

where body—even, body—odd : Proc.$A \to$ Proc.A are given by

$$\begin{aligned}
&\text{body—even}.T.(e, u) = \\
&\quad \text{Add}.(n, b).(e, \text{val}.u)\ ;\ T(\text{val}.n, b)\ ;\ b := \neg\ \text{val}.b\ ;\ \text{Del}.n\ ;\ \text{Del}.b.u \\
&\text{body—odd}.S.(e, u) = \\
&\quad \text{Add}.(n, b).(e, \text{val}.u)\ ; \\
&\quad \text{if val}.n \doteq 0 \text{ then } b := \text{false else } S(\text{val}.n - 1, b) \text{ fi}\ ; \\
&\quad \text{Del}.n\ ;\ \text{Del}.b.u
\end{aligned}$$

6.1. Example. Mutually recursive procedures for parsing expressions

In this section a more complex example of a collection of mutually recursive procedures for parsing expressions is introduced.

We assume that we have a type String $\subseteq$ Constant of strings with characters from an alphabet Alph $\subseteq$ String. If $X \subseteq$ Alph, then $X^* \subseteq$ String denotes the strings with elements from X. We also assume that nil $\in$ String is the empty string and we denote by $\cdot$ the string concatenation, car.a the first character of the string a, cdr.a the string obtained from a by removing the first character, and by $a \leq b$ the fact that the string a is a prefix of string b.

The alphabet contains terminal symbols: letters (Letter $\subseteq$ Alph), special symbols ("+", "$*$", "(", ")" $\in$ Alph), and non-terminal symbols ($\langle E\rangle, \langle T\rangle, \langle F\rangle, \langle L\rangle \in$ Alph). We denote by Terminal and NonTerm the types of terminal and non-terminal symbols of the alphabet.

The context free grammar that generates arithmetic expressions is given by:

$$\begin{aligned}
&\langle E\rangle ::= \langle T\rangle \mid \langle T\rangle \cdot \text{"+"} \cdot \langle E\rangle \\
&\langle T\rangle ::= \langle F\rangle \mid \langle F\rangle \cdot \text{"}*\text{"} \cdot \langle T\rangle \\
&\langle F\rangle ::= \langle L\rangle \mid \text{"("} \cdot \langle E\rangle \cdot \text{")"} \\
&\langle L\rangle ::= \text{"}a\text{"} \mid \text{"}b\text{"} \mid \text{"}c\text{"} \mid \ldots \quad \text{"}a\text{", "}b\text{", "}c\text{", } \ldots \in \text{Letter},
\end{aligned}$$

with $\langle E\rangle$ the start symbol.

We denote by $\overset{*}{\Longrightarrow} \subseteq$ (NonTerm $\to$ Terminal* $\to$ Bool), the derivation relation of the grammar given above. $N \overset{*}{\Longrightarrow} a$ is true if a is a word generated by the grammar rules starting from the non-terminal symbol N. The language generated by N, $\text{Lang}_N \subseteq$ Terminal*, is given by $\text{Lang}_N \hat{=} \{a \in \text{Terminal}^* \mid N \overset{*}{\Longrightarrow} a\}$.

For every non-terminal $N \in$ NonTerm, we introduce a procedure $\text{parse}_N \in$ Proc.A, where $A =$ Vars.String $\times$ Vars.(AddrsNil.Ptree). The procedure call $\text{parse}_N.(x, p)$ builds in p the abstract syntax tree of some maximal string a such that $a \leq x$ and $a \in \text{Lang}_N$.

```
 procedure parse_E(value-result s, t)
local t_1, t_2
parse_T(s, t_1) ;
if val.t_1 ≠̇ nil ∧ val.s ≠̇ nil ∧ car.(val.s) ≐ "+" then
    s := cdr.(val.s) ; parse_E(s, t_2) ;
    if val.t_2 ≠̇ nil then
        New(t, ptree("+", t_1, t_2))
    else
        t := val.t_1 ; s := "+" · val.s
    fi
else
    t := val.t_1
fi
```

The definition of the procedure parse_T is similar to the definition of parse_E, except that the constant "+" is replaced by "$*$", and the calls to parse_T and parse_E are replaced by calls to parse_F and parse_T, respectively.

```
 procedure parse_F(value-result s, t)
local r
if val.s ≐ nil then
    t := nil
else
    if car.(val.s) = "(" then
```

```
        r := cdr.(val.s) ; parse_E(r, t) ;
        if (val.t ≠̇ nil ∧ val.r ≠̇ nil ∧ car.(val.r) ≐ ")") then
            s := cdr.(val.r)
        else
            DisposeTree(t)
        fi
    else
        if Letter(car.(val.s)) then
            New(t, tree(car.(val.s), nil, nil)) ; s := cdr.(val.s)
        else
            t := nil
        fi
    fi
fi
```

The procedure DisposeTree used in parse_F is defined in [16,17]. All we need to know here is that the call to DisposeTree(t), where $t \in$ Vars.(AddrsNil.Ptree) disposes the tree stored in program variable t and sets t to nil.

The procedures parse_E, parse_T, and parse_F are given by the least fixpoint of body–parse : $(\mathsf{Proc}.A)^3 \to (\mathsf{Proc}.A)^3$.

$$\mathsf{body\text{–}parse}.(E, T, F) = (\mathsf{body}_E.T.E,\ \mathsf{body}_T.F.T,\ \mathsf{body}_F.E)$$

where body_E, body_T, and body_F are given by the above procedure definitions.

Next, the necessary predicates and formulas for specifying the parse procedures are introduced. For all non-terminal symbols $N \in \{\langle E\rangle, \langle T\rangle, \langle F\rangle\}$ and all $t \in$ AddrsNil.Ptree, $a \in \mathsf{Terminal}^*$, the predicate $\mathsf{tree}_N(t, a) \in$ Pred is true in those states where $a \in \mathsf{Lang}_N$ and t is the address of a pointer representation of the abstract syntax tree corresponding to the string a. The definitions are by total induction on the length of the string a.

$$\begin{aligned}
\mathsf{tree}_N(t, \mathsf{nil}) &\ \hat{=}\ t = \mathsf{nil} \wedge \mathsf{emp}, \qquad \forall N \in \mathsf{NonTerm}\\
\mathsf{tree}_E(t, a) &\ \hat{=}\ \mathsf{tree}_T(t, a) \vee (\exists\, b, c, t_1, t_2 \bullet a \doteq b \cdot \text{“+”} \cdot c \wedge \mathsf{tree}_T(t_1, b)\\
&\qquad * \mathsf{tree}_E(t_2, c) * (t \mapsto \mathsf{ptree}.(\text{“+”}, t_1, t_2)))\\
\mathsf{tree}_T(t, a) &\ \hat{=}\ \mathsf{tree}_F(t, a) \vee (\exists\, b, c, t_1, t_2 \bullet a \doteq b \cdot \text{“}*\text{”} \cdot c \wedge \mathsf{tree}_F(t_1, b)\\
&\qquad * \mathsf{tree}_T(t_2, c) * (t \mapsto \mathsf{ptree}.(\text{“}*\text{”}, t_1, t_2)))\\
\mathsf{tree}_F(t, a) &\ \hat{=}\ \mathsf{Letter}.a \wedge t \mapsto \mathsf{ptree}.(a, \mathsf{nil}, \mathsf{nil})\\
&\qquad \vee (\exists\, b \bullet (a \doteq \text{“(”} \cdot b \cdot \text{“)”}) \wedge \mathsf{tree}_E(t, b))
\end{aligned}$$

Lemma 8. *For all* $N \in \{\langle E\rangle, \langle T\rangle, \langle F\rangle\}$, $t \in$ Addrs.Ptree, *and* $a \in \mathsf{Terminal}^*$, *if* $\mathsf{tree}_N(t, a)$, *then* $\mathsf{Lang}_N.a$.

For $N \in \{\langle E\rangle, \langle T\rangle, \langle F\rangle\}$, we define the postcondition $\mathsf{post}_N(a, b, t) \in$ Pred for the procedure $\mathsf{parse}_N(b, t)$ by

$$\mathsf{post}_N(a, b, t) = \exists\, d \bullet a \doteq c \cdot b \wedge \mathsf{tree}_N(t, c) \wedge (\forall x \bullet x \leq b \wedge x \neq̇ \mathsf{nil} \Rightarrow \neg \mathsf{Lang}_N.(c \cdot x))$$

The predicate $\mathsf{post}_N(a, b, t)$ states that the initial string a can be split into $c \cdot b$, where c is maximal such that $\mathsf{tree}_N(t, c)$.

If x is a list of program variables, then SepPred.x denotes the predicates that are set.x-independent and non-alloc independent. If $a \in$ String, $u \in$ Vars.String, $v \in$ Vars.(AddrsNil.Ptree), and $\alpha \in$ SepPred.(u, v), then the correctness of the procedure $N \in \{\langle E\rangle, \langle T\rangle, \langle F\rangle\}$ is given by the following Hoare triple:

$$\forall a, v, u, \alpha \bullet \mathsf{val}.u \doteq a \wedge \alpha\ \{\!|\ \mathsf{parse}_N.(u, v)\ |\!\}\ \alpha * \mathsf{post}_N(a, \mathsf{val}.u, \mathsf{val}.v) \tag{6}$$

If the heap contains some addresses specified by α and the value of u is a, then after the execution of parse_N, the heap still contains the addresses specified by α, but in addition it contains also some new addresses which store the parsing tree of the expression a.

The next two sections will gradually introduce more and more powerful theorems that can be used to prove the correctness of mutually recursive procedures manipulating pointers.

7. Abstract recursion

This section introduces the concept of program lattice as a generalization of monotonic predicate transformers. We prove general refinement and total correctness rules for mutually recursive programs (procedures). The Hoare rule for mutually recursive procedures is introduced in a number of steps. First we prove a refinement rule for mutually recursive procedures as a straightforward generalization from recursion to mutual recursion. Since this rule is difficult to use in practice, a new rule for refinement of mutually recursive procedures is derived. In the next step a Hoare rule for mutually recursive procedures is proved. The final result of this section is the Hoare rule for mutually recursive procedures with auxiliary variables in specifications. At each step the new rule is proved based on the previous rule.

Definition 9. We call the structure $\langle L, \leq, \vee, \wedge, \odot, \mathsf{skip}\rangle$ a *program lattice* if

- $\langle L, \leq, \vee, \wedge\rangle$ is a complete lattice

- $\langle L,\ \odot,\ \mathsf{skip}\rangle$ is a monoid
- $(\bigvee_i S_i) \odot T = \bigvee_i (S_i \odot T)$

Theorem 10. *The complete lattice of monotonic predicate transformers* $\langle \mathsf{Prog}, \sqsubseteq,\ \sqcup, \sqcap,\ ;\ , \mathsf{skip}\rangle$ *is a lattice of programs.*

Definition 11. A structure $\langle K, \leq, \vee, \wedge, {}_\odot\rangle$ is a *predicate lattice* for L if K is a complete lattice, and $\_ {}_\odot \_ : L \to K \to K$ is such that

- $(S \odot T)_\odot p = S_\odot (T_\odot p)$
- $(\bigvee_i S_i)_\odot p = \bigvee_i (S_i{}_\odot p)$
- $p \leq q \Rightarrow S_\odot p \leq S_\odot q$
- $\mathsf{skip}_\odot p = p$

We call the elements of K *predicates for* L or simply *predicates*.

Definition 12. If L is a program lattice and K is a predicate lattice for L, then an *abstract Hoare total correctness triple*, denoted $p\ \{\!|\ S\ |\!\}\ q$, $p, q \in K, S \in L$, is true if and only if $p \leq S_\odot q$.

Definition 13. A structure $\langle K, \leq, \vee, \wedge, {}_\odot, (\!|\_|\!), [\![\_]\!]\rangle$ is an *assertion lattice* for L if $\langle K, \leq, \vee, \wedge, {}_\odot\rangle$ is a predicate lattice for L and $(\!|\_|\!),\ [\![\_]\!] : K \to L$ are such that

- $(\!|\bigvee_i p_i|\!) = \bigvee_i (\!|p_i|\!)$
- $(\!|p|\!)_\odot q = (\!|q|\!)_\odot p$
- $(\!|S_\odot p|\!) \odot [\![p]\!] \leq S$ and
- $\mathsf{skip} \leq (\!|[\![p]\!]_\odot p|\!)$.

The statements $(\!|p|\!)$ and $[\![p]\!]$ are called *abstract assert statement* and *abstract postcondition statement*, respectively.

Theorem 14. *The complete lattice of predicates* $\langle \mathsf{Pred}, \subseteq, \cup, \cap, \_.\_, \{\_\}, \{\![\_]\!\}\rangle$ *is an assertion lattice for* Prog.

Usually, K and L are the lattices of all predicates and monotonic predicate transformers, respectively. However, in many situations, we will also work with other lattices and operations, thus it is useful to state and prove some results at this abstract level.

Next, unless otherwise specified, we assume that L is a program lattice and K is an assertion lattice for L.

Lemma 15. *If* $S \in L$ *and* $p,\ q_i \in K$, *then*

(i) $(\!|p|\!)_\odot (\bigvee_i q_i) = \bigvee_i (\!|p|\!)_\odot q_i$
(ii) $p \leq q \Rightarrow (\!|p|\!) \leq (\!|q|\!)$
(iii) $p\ \{\!|\ S\ |\!\}\ q \Leftrightarrow (\!|p|\!) \odot [\![q]\!] \leq S$.

Now, we are able to state and prove the most general recursion refinement rule.

Theorem 16 (*Recursion Refinement*). *If* $p_w \in K$ *is a family of elements indexed by the well-founded set* $\langle W, <\rangle$, $S \in L$, *and* $F : L \to L$ *is monotonic, then*

$$(\forall w \in W \bullet (\!|p_w|\!) \odot S \leq F.((\!|p_{<w}|\!) \odot S)) \Rightarrow (\!|p|\!) \odot S \leq \mu F, \tag{7}$$

where $p_{<w} = \bigvee_{v<w} p_v$ *and* $p = \bigvee_w p_w$.

Proof. We can easily prove by well-founded induction on W that the assumption of (7) implies $(\forall w \bullet (\!|p_w|\!) \odot S \leq \mu F)$. From this the conclusion of (7) follows immediately. □

If L is a program lattice and A is a non-empty set, then $A \to L$ with the pointwise extension of all operations from L to $A \to L$ is a program lattice. If K is a predicate (assertion) lattice for L, then $A \to K$ is a predicate (assertion) lattice for $A \to L$. Similarly, if for every $i \in I$, L_i is a program lattice, then $\prod_i L_i$, with the component-wise extension of operations from $(L_i)_{i\in I}$ to $\prod_i L_i$, is a program lattice. If for every $i \in I$, K_i is a predicate (assertion) lattice for L_i then $\prod_i K_i$ is a predicate (assertion) lattice for $\prod_i L_i$.

The specifications of the procedures even and odd, introduced in the previous section, are:

$$\mathsf{even\text{-}spec}.(e, u) \mathrel{\hat{=}} u := e \bmod 2 \doteq 0 \quad \text{and} \quad \mathsf{odd\text{-}spec}.(e, u) \mathrel{\hat{=}} u := e \bmod 2 \doteq 1$$

We want to prove that the specifications even-spec and odd-spec are refined by their mutually recursive implementations, even and odd:

$$\mathsf{even\text{-}spec}.(e, u) \sqsubseteq \mathsf{even}.(e, u) \quad \text{and} \quad \mathsf{odd\text{-}spec}.(e, u) \sqsubseteq \mathsf{odd}.(e, u) \tag{8}$$

Using Theorem 16 for $W = \mathsf{Nat}$, $p_w = ((\lambda e, u \bullet e \doteq w), (\lambda e, u \bullet e \doteq w))$, $S = (\mathsf{even\text{-}spec}, \mathsf{odd\text{-}spec})$, and $F = (\lambda S, T \bullet \mathsf{body\text{—}even}.T, \mathsf{body\text{—}odd}.S)$, the example refinement (8) is true if we prove

$$\begin{array}{l}(\forall w \bullet \{\lambda e, u \bullet e \doteq w\} \ ; \ \mathsf{even\text{-}spec} \sqsubseteq \\ \quad \mathsf{body\text{—}even}(\{\lambda e, u \bullet e < w\} \ ; \ \mathsf{even\text{-}spec}, \ \{\lambda e, u \bullet e < w\} \ ; \ \mathsf{odd\text{-}spec})) \\ (\forall w \bullet \{\lambda e, u \bullet e \doteq w\} \ ; \ \mathsf{odd\text{-}spec} \sqsubseteq \\ \quad \mathsf{body\text{—}odd}(\{\lambda e, u \bullet e < w\} \ ; \ \mathsf{even\text{-}spec}, \ \{\lambda e, u \bullet e < w\} \ ; \ \mathsf{odd\text{-}spec}))\end{array}$$

However, we cannot prove the first refinement, since in the procedure even, the call to the procedure odd is done without decreasing the termination function e.

Next, we introduce a version of Theorem 16 (Recursion Refinement), which is more suitable to refine mutually recursive programs. We assume that for every $i \in I$, L_i is a program lattice and K_i is an assertion lattice for L_i. We denote $L = \prod_i L_i$ and $K = \prod_i K_i$. Moreover, we assume for every $w \in W$ that $p_w \in K$ and $\langle W \times I, \ < \rangle$ is well-founded. We denote $p_{w,i} = (p_w)_i$ and for every $s \in W \times I$ we define $p, p_{<s}, q_s, q_{<s}, q \in K$ by

$$\begin{array}{ll} p \mathrel{\hat{=}} \bigvee\{p_w \mid w \in W\}, & (p_{<s})_j \mathrel{\hat{=}} \bigvee\{p_{v,j} \mid (v,j) < s\}, \\ (q_s)_j \mathrel{\hat{=}} \bigvee\{p_{v,j} \mid (v,j) \leq s\}, & q_{<s} \mathrel{\hat{=}} \bigvee\{q_t \mid t < s\}, \\ q \mathrel{\hat{=}} \bigvee\{q_s \mid s \in W \times I\}. & \end{array} \tag{9}$$

Lemma 17. *If $s, t \in W \times I$, then*

(i) $p = q$
(ii) $q_{<s} = p_{<s}$
(iii) $s \leq t \Rightarrow p_{<s} \leq p_{<t}$

Theorem 18 (*Mutual Recursion Refinement*). *Under the above assumptions, if $F : L \to L$ is monotonic, then*

$$\left(\forall w \in W \bullet \forall i \in I \bullet (\!|p_{w,i}|\!) \odot S_i \leq (F.((\!|p_{<(w,i)}|\!) \odot S))_i\right) \Rightarrow (\!|p|\!) \odot S \leq \mu F$$

Proof. $(\!|p|\!) \odot S \leq \mu F$

$= \{$Lemma 17 $(p = q)\}$

$(\!|q|\!) \odot S \leq \mu F$

$\Leftarrow \{$Theorem 16 with $W \times I$ and q_s instead of W and $p_w\}$

$\left(\forall w \in W \bullet \forall i \in I \bullet (\!|q_{w,i}|\!) \odot S \leq F.(q_{<(w,i)} \odot S)\right)$

$= \{$Definition of $\leq, \odot, (\!|\_|\!)$ on tuples and Lemma 17$\}$

$\left(\forall w \in W \bullet \forall i, j \in I \bullet (\!|(q_{w,i})_j|\!) \odot S_j \leq (F.(p_{<(w,i)} \odot S))_j\right)$

$= \{$Definition of $(q_{w,i})_j$ and complete lattice properties$\}$

$\left(\forall w, v \in W \bullet \forall i, j \in I \bullet (v,j) \leq (w,i) \Rightarrow (\!|p_{v,j}|\!) \odot S_j \leq (F.(p_{<(w,i)} \odot S))_j\right)$

$\Leftarrow \{$Lemma 17 and F monotonic$\}$

$\left(\forall v \in W \bullet \forall j \in I \bullet (\!|p_{v,j}|\!) \odot S_j \leq (F.(p_{<(v,j)} \odot S))_j\right)$ □

Theorem 18 is inspired from the Hoare total correctness rule introduced by Nipkow [10]. The idea of this rule is to require that the termination function is decreased eventually, in a sequence of recursive calls, and not necessarily before each call.

Using Theorem 18, the proof obligation of the procedure even becomes:

$$\begin{array}{l}(\forall w \bullet \{\lambda e, u \bullet e \doteq w\} \ ; \ \mathsf{even\text{-}spec} \sqsubseteq \\ \quad \mathsf{body\text{—}even}(\{\lambda e, u \bullet e < w\} \ ; \ \mathsf{even\text{-}spec}, \ \{\lambda e, u \bullet e \leq w\} \ ; \ \mathsf{odd\text{-}spec}))\end{array}$$

where $W = \mathsf{Nat}$, $I = \{1, 2\}$, and the order on $W \times I$ is given by

$$(v,j) < (w,i) \Leftrightarrow v < w \vee (v = w \wedge j > i).$$

The difference from the relations obtained with the first theorem is the proof obligation of the procedure even, where we are not required to decrease the termination function e before calling the procedure odd.

Theorem 19 (*Hoare Mutual Recursion*). *Under the above assumptions, if $r \in K$ and $F : L \to L$ is monotonic, then*

$$\left(\forall w \in W \bullet \forall i \in I \bullet \forall S \in L \bullet p_{<(w,i)} \ \{\!|\ S \ |\!\}\ r \Rightarrow p_{w,i} \ \{\!|\ (F.S)_i \ |\!\}\ r_i\right) \Rightarrow p \ \{\!|\ \mu F \ |\!\}\ r$$

Proof. $p \ \{\!|\ \mu F \ |\!\}\ r$

$= \{$Lemma 15$\}$

$(\!|p|\!) \odot [\![r]\!] \leq \mu F$

$\Leftarrow \{$Theorem 18$\}$

$$\begin{aligned}
&\left(\forall w \in W \bullet \forall i \in I \bullet (\!|p_{w,i}|\!) \odot [\![r_i]\!] \leq (F.((\!|p_{<(w,i)}|\!) \odot [\![r]\!]))_i\right)\\
=\ &\{\text{Complete lattice properties}\}\\
&\left(\forall w \in W \bullet \forall i \in I \bullet \forall S \in L \bullet (\!|p_{<(w,i)}|\!) \odot [\![r]\!] \leq S \Rightarrow (\!|p_{w,i}|\!) \odot [\![r_i]\!] \leq (F.S)_i\right)\\
=\ &\{\text{Lemma 15}\}\\
&\left(\forall w \in W \bullet \forall i \in I \bullet \forall S \in L \bullet p_{<(w,i)}\ \{\!|\ S\ |\!\}\ r \Rightarrow p_{w,i}\ \{\!|\ (F.S)_i\ |\!\}\ r_i\right) \quad \square
\end{aligned}$$

Although this theorem can be formulated in the context of a program lattice L and a predicate lattice K, the proof uses the fact that K is an assertion lattice for L.

When working with Hoare statements, $\alpha\ \{\!|\ S\ |\!\}\ \beta$, we very often need specification variables that occur only in α and β, but not in S. A detailed discussion of this problem could be found in [2]. However, here we mention that we add support for specification variables by considering $S \in L, \alpha, \beta : A \to K$, where K is an assertion lattice for L and A is a non-empty set of specification values. Intuitively, the Hoare triple $\alpha\ \{\!|\ S\ |\!\}\ \beta$ is true if

$$(\forall a \in A \bullet \alpha.a \leq S.(\beta.a)) \tag{10}$$

Formally, if L is a program lattice, K is an assertion lattice for L, and A is a non-empty set, then $A \to K$ is a predicate lattice for L, where the operations on K are pointwise extended to $A \to K$, and $_\odot : L \to K \to K$ is extended to $_\odot : L \to (A \to K) \to (A \to K)$ by

$$(S_\odot \alpha).a \mathrel{\hat{=}} S_\odot(\alpha.a).$$

It is easy to see that, if $\alpha, \beta : A \to K$ and $S \in L$, then $\alpha\ \{\!|\ S\ |\!\}\ \beta$ is equivalent to definition (10). We cannot however construct an assertion lattice structure on $A \to K$ for L.

Next, we extend Theorem 19 to the case when predicates may refer to some specification variables. We assume that for each $i \in I$, L_i is a program lattice, K_i is an assertion lattice for L_i, and A_i is a non-empty set of specification values. We denote $L = \prod_i L_i$, $A = \prod_i A_i$, $K_i' = A_i \to K_i$, $L_i' = A_i \to L_i$, $K' = \prod_i K_i'$, and $L' = \prod_i L_i'$. If W is a non-empty set, $\langle W \times I, < \rangle$ is well-founded, and $p_w \in K'$, then for every $s \in W \times I$, we define $p, p_{<s}, q_s, q_{<s}, q \in K'$ as in (9).

Theorem 20 (*Hoare Mutual Recursion & Specification Variables*). *Under the above assumptions, if $r \in K'$ and $F : L \to L$ is monotonic, then*

$$\left(\forall w \in W \bullet \forall i \in I \bullet \forall S \in L \bullet p_{<(w,i)}\ \{\!|\ S\ |\!\}\ r \Rightarrow p_{w,i}\ \{\!|\ (F.S)_i\ |\!\}\ r_i\right) \Rightarrow p\ \{\!|\ \mu F\ |\!\}\ r$$

Proof. We assume

$$\left(\forall w \in W \bullet \forall i \in I \bullet \forall S \in L \bullet p_{<(w,i)}\ \{\!|\ S\ |\!\}\ r \Rightarrow p_{w,i}\ \{\!|\ (F.S)_i\ |\!\}\ r_i\right) \tag{11}$$

and we prove $p\ \{\!|\ \mu F\ |\!\}\ r$. We recall the definition of $\hat{F} : L' \to L'$ from Theorem 2, for each $\alpha \in K'$, $a \in A$, $\hat{F}.\alpha.a = F.(\bigvee_{b \in A} \alpha.b)$. From Theorem 2, it follows that $p\ \{\!|\ \mu F\ |\!\}\ r \Leftrightarrow p\ \{\!|\ \mu \hat{F}\ |\!\}\ r$.

By applying Theorem 19 for p_w, r, and $\hat{F}$, we obtain $p\ \{\!|\ \mu \hat{F}\ |\!\}\ r$ if

$$\left(\forall w \in W \bullet \forall i \in I \bullet \forall S \in L' \bullet p_{<(w,i)}\ \{\!|\ S\ |\!\}\ r \Rightarrow p_{w,i}\ \{\!|\ (\hat{F}.S)_i\ |\!\}\ r_i\right) \tag{12}$$

All we need to prove now is that (11) implies (12). For $w \in W$, $i \in I$, and $S \in L'$, the following derivation is true:

$$\begin{aligned}
&p_{w,i}\ \{\!|\ (\hat{F}.S)_i\ |\!\}\ r_i\\
\Leftrightarrow\ &\{\text{Definitions}\}\\
&\left(\forall a \in A_i \bullet p_{w,i}.a\ \{\!|\ (\hat{F}.S)_i.a\ |\!\}\ r_i.a\right)\\
\Leftrightarrow\ &\{\text{Definition}\}\\
&\left(\forall a \in A_i \bullet p_{w,i}.a\ \{\!|\ (F.(\textstyle\bigvee_{b \in A} S.b))_i\ |\!\}\ r_i.a\right)\\
\Leftrightarrow\ &\{\text{Definition}\}\\
&p_{w,i}\ \{\!|\ (F.(\textstyle\bigvee_{b \in A} S.b))_i\ |\!\}\ r_i\\
\Leftarrow\ &\{\text{Assumption (11)}\}\\
&p_{<(w,i)}\ \{\!|\ \textstyle\bigvee_{b \in A} S.b\ |\!\}\ r\\
\Leftarrow\ &\{\text{Definitions and complete lattice properties}\}\\
&p_{<(w,i)}\ \{\!|\ S\ |\!\}\ r \quad \square
\end{aligned}$$

Theorem 20 is similar to the rule for mutually recursive procedures from [10]. However, our rule can be applied to procedures with parameters and local variables, and it would be used in the next section to derive a new rule, more suitable for procedures manipulating pointers.

Theorem 20 can be used to prove the parsing procedures introduced before.

Let $\leq_s$ be a binary relation on $W = \mathsf{String}$ given by

$$a \leq_s b \Leftrightarrow a \text{ is a suffix of } b.$$

If $I = \{\langle E\rangle,\ \langle T\rangle,\ \langle F\rangle\}$ and $\langle E\rangle > \langle T\rangle > \langle F\rangle$, then we define the well founded order $<$ on $W \times I$ by

$$(a, N) < (b, N') \Leftrightarrow a <_s b \vee (a = b \wedge N < N').$$

For every $N \in I$, let us define:

$$p_{w,N} = (\lambda a \bullet \lambda u, v \bullet \mathsf{val}.u \doteq a \wedge \mathsf{val}.u \doteq w)$$

Using Theorem 20, the correctness triples (6) for the parse procedures are true, if

$$\begin{array}{ll} & (\forall a, u, v, \alpha \bullet \alpha \wedge \mathsf{val}.u \doteq a \leq_s w \ \{\!|\, T.(u, v) \,|\!\}\ \alpha * \mathsf{post}_T(a,\ \mathsf{val}.u, \mathsf{val}.v)) \\ \wedge & (\forall a, u, v, \alpha \bullet \alpha \wedge \mathsf{val}.u \doteq a <_s w \ \{\!|\, E.(u, v) \,|\!\}\ \alpha * \mathsf{post}_E(a,\ \mathsf{val}.u, \mathsf{val}.v)) \\ \Rightarrow & \\ & (\forall a, u, v, \alpha \bullet \alpha \wedge \mathsf{val}.u \doteq a \doteq w \ \{\!|\, \mathsf{body}_E.T.E.(u, v) \,|\!\}\ \alpha * \mathsf{post}_E(a,\ \mathsf{val}.u, \mathsf{val}.v)) \end{array}$$

and

$$\begin{array}{ll} & (\forall a, u, v, \alpha \bullet \alpha \wedge \mathsf{val}.u \doteq a \leq_s w \ \{\!|\, F.(u, v) \,|\!\}\ \alpha * \mathsf{post}_F(a,\ \mathsf{val}.u, \mathsf{val}.v)) \\ \wedge & (\forall a, u, v, \alpha \bullet \alpha \wedge \mathsf{val}.u \doteq a <_s w \ \{\!|\, T.(u, v) \,|\!\}\ \alpha * \mathsf{post}_T(a,\ \mathsf{val}.u, \mathsf{val}.v)) \\ \Rightarrow & \\ & (\forall a, u, v, \alpha \bullet \alpha \wedge \mathsf{val}.u \doteq a \doteq w \ \{\!|\, \mathsf{body}_T.F.T.(u, v) \,|\!\}\ \alpha * \mathsf{post}_T(a,\ \mathsf{val}.u, \mathsf{val}.v)) \end{array}$$

and

$$\begin{array}{ll} & (\forall a, u, v, \alpha \bullet \alpha \wedge \mathsf{val}.u \doteq a <_s w \ \{\!|\, E.(u, v) \,|\!\}\ \alpha * \mathsf{post}_E(a,\ \mathsf{val}.u, \mathsf{val}.v)) \\ \Rightarrow & \\ & (\forall a, u, v, \alpha \bullet \alpha \wedge \mathsf{val}.u \doteq a \doteq w \ \{\!|\, \mathsf{body}_F.E.(u, v) \,|\!\}\ \alpha * \mathsf{post}_F(a,\ \mathsf{val}.u, \mathsf{val}.v)) \end{array}$$

Similarly to the procedure even which calls directly procedure odd without any computation, procedure parse_E calls parse_T. We can prove the correctness of parse_E because we can assume that parse_T is correct when starting from a state where $\mathsf{val}.u \leq_s w$. If we would use Theorem 16 instead of Theorem 18 as the basis for the Hoare mutual recursion theorem (Theorem 19), then we would have the same problem as we had with the first attempt to prove the procedures even and odd.

The correctness of the parsing procedures can be proved by proving the proof obligations presented above. However, this would require some extra work that could be avoided. In the first proof obligation, assuming that the procedures T and E are correct we need to prove:

$$(\forall a, u, v, \alpha \bullet \alpha \wedge \mathsf{val}.u \doteq a \doteq w \ \{\!|\, \mathsf{body}_E.T.E.(u, v) \,|\!\}\ \alpha * \mathsf{post}_E(a,\ \mathsf{val}.u, \mathsf{val}.v)) \tag{13}$$

where α ranges over predicates which do not contain the variables u, v. The formula α describes the part of the heap which is not modified by $\mathsf{body}_E.T.E$. If possible we would like to prove some form of (13) where we are not concerned about the part of the heap which is not modified by $\mathsf{body}_E.T.E$. In the next section we introduce a rule which would enable proving only:

$$(\forall a, u, v \bullet \mathsf{emp} \wedge \mathsf{val}.u \doteq a \doteq w \ \{\!|\, \mathsf{body}_E.T.E.(u, v) \,|\!\}\ \mathsf{post}_E(a,\ \mathsf{val}.u, \mathsf{val}.v)) \tag{14}$$

instead of (13).

8. Recursive procedures & frame rule

In this section we introduce a new powerful Hoare total correctness rule for mutually recursive procedures. This rule combines an extension to procedures with parameters of the Hoare rule from [10] with the frame rule for pointer programs [23].

We introduce a new theorem that can be used when proving the correctness of recursive procedures manipulating pointers. We assume that we have a non-empty type A of procedure parameters and $X : A \to \mathcal{P}.\mathsf{Pred}$, such that for all $a \in A$, $X.a$ is closed under arbitrary union, separation conjuction, and $\mathsf{emp} \in X.a$. The type $X.a$ denotes those formulas that could be added to a Hoare triple when using the frame rule, and they are in general formulas which do not contain free variables modified by the procedure call. We define:

$$\mathsf{Proc}_X.A = \{P \in \mathsf{Proc}.A \mid \forall a \in A,\ \forall \alpha \in X.a,\ \forall q \in \mathsf{Pred} \bullet \alpha * (P.a).q \subseteq (P.a).(\alpha * q)\}$$

In [23] the concept "local predicate transformers that modify a set V" of program variables is introduced to define the class of predicate transformers that modify only variables from V and satisfy the frame property. $\mathsf{Proc}_X.A$ is a generalization of local predicate transformers to procedures with parameters. The elements of $\mathsf{Proc}_X.A$ are the local predicate transformers when $A = \{\bullet\}$ and $X =$ the set of predicates which do not contain free variables from V.

Lemma 21. $\mathsf{Proc}_X.A$ *is a program sublattice of* $\mathsf{Proc}.A$.

Proof. We need to prove that $\mathsf{Proc}_X.A$ is closed under arbitrary meets, joins, sequential composition and $\mathsf{skip} \in \mathsf{Proc}_X.A$. Let $P_i \in \mathsf{Proc}_X.A$ for all $i \in I$. Then we have that:

$$
\begin{array}{ll}
& (\bigsqcup_i P_i) \in \mathsf{Proc}_X.A \\
= & \{\text{Definition}\} \\
& (\forall a \in A,\ \forall \alpha \in X.a,\ \forall q \in \mathsf{Pred} \bullet \alpha * (\bigsqcup_i P_i).a.q \subseteq (\bigsqcup_i P_i).a.(\alpha * q)) \\
= & \{\text{Lemma 6}\} \\
& (\forall a \in A,\ \forall \alpha \in X.a,\ \forall q \in \mathsf{Pred} \bullet \bigcup_i (\alpha * P_i.a.q) \subseteq \bigcup_i P_i.a.(\alpha * q)) \\
\Leftarrow & \{\text{Complete lattice properties}\} \\
& (\forall i \in I,\ \forall a \in A,\ \forall \alpha \in X.a,\ \forall q \in \mathsf{Pred} \bullet \alpha * P_i.a.q \subseteq P_i.a.(\alpha * q)) \\
= & \{\text{Definition}\} \\
& (\forall i \in I \bullet P_i \in \mathsf{Proc}_X.A.)
\end{array}
$$

For arbitrary intersections, we can give a similar proof. The facts that $\mathsf{skip} \in \mathsf{Proc}_X.A$ and $\mathsf{Proc}_X.A$ is closed under sequential composition follow directly from the definition of $\mathsf{Proc}_X.A$. □

Theorem 22. *If for all* $w \in W$, *we have* $p_w : B \to A \to \mathsf{Pred}$, $q : B \to A \to \mathsf{Pred}$, *and body* : $\mathsf{Proc}.A \to \mathsf{Proc}.A$ *is monotonic, then the following Hoare rule is true:*

$$
\begin{array}{l}
\quad (\forall w \in W \bullet \forall P \in \mathsf{Proc}_X.A \bullet p_{<w}\ \{\!|\ P\ |\!\}\ q \Rightarrow p_w\ \{\!|\ body.P\ |\!\}\ q) \\
\quad \wedge\ (\forall P \in \mathsf{Proc}_X.A \bullet body.P \in \mathsf{Proc}_X.A) \\
\Rightarrow \\
\quad p\ \{\!|\ \mu\ body\ |\!\}\ q \quad \wedge \quad \mu\ body \in \mathsf{Proc}_X.A.
\end{array}
$$

The conclusion of this theorem that $\mu\ body$ is from $\mathsf{Proc}_X.A$ and not only from $\mathsf{Proc}.A$ is the key element of this theorem. The recursive procedure $\mu\ body$ satisfies the frame rule if its body does.

When proving the correctness of a recursive procedure, Theorem 22 allows us to assume stronger properties (like (13)), and in fact prove a weaker property (like (14)). If we use the procedure correctness statement in proving other programs, we can also use a stronger property (like (14)).

We would like to prove this theorem using Theorem 20 for program lattice $\mathsf{Proc}_X.A$, since $\mu\ body \in \mathsf{Proc}_X.A$ by Lemma 1. However, $\{\!|p|\!\} \notin \mathsf{Proc}_X.A$, so we cannot use Theorem 20 for $\langle A \to \mathsf{Pred}, \ldots, \{\!|\_|\!\}, [\![\_]\!] \rangle$ as an assertion lattice for $\mathsf{Proc}_X.A$.

We define the *separation assertion statement*, denoted $(\!|p|\!) \in \mathsf{Proc}_X.A$ by

$$(\!|p|\!).a.q = p.a * q$$

and the *separation postcondition statement*, denoted $[\![p]\!] \in \mathsf{Proc}_X.A$, by:

$$[\![p]\!].a.q = \bigcup \{\alpha \in X.a \mid p.a * \alpha \subseteq q\}$$

Theorem 23. *The structure* $\langle A \to \mathsf{Pred},\ \subseteq, \wedge, \vee,\ \_\cdot\_,\ (\!|\_|\!),\ [\![\_]\!] \rangle$ *is an assertion lattice for* $\mathsf{Proc}_X.A$.

Proof. The facts that $(\!|\_|\!)$ is an abstract assert statement, and $(\!|p|\!) \in \mathsf{Proc}_X.A$ follow from Lemma 6.

We prove that $[\![p]\!]$ is an element of $\mathsf{Proc}_X.A$, that is, for all $a \in A$, $\alpha \in X.a$ and $q \in \mathsf{Pred}$, $\alpha * [\![p]\!].a.q \subseteq [\![p]\!].a.(\alpha * q)$. If $X_{a,p,q} \subseteq X.a$ is given by:

$$X_{a,p,q} = \{\alpha \in X.a \mid p.a * \alpha \subseteq q\},$$

then

$$\alpha \in X.a \wedge \beta \in X_{a,p,q} \Rightarrow \alpha * \beta \in X_{a,p,\alpha * q} \tag{15}$$

$$
\begin{array}{ll}
& \alpha * [\![p]\!].a.q \subseteq [\![p]\!].a.(\alpha * q) \\
= & \{\text{Definition}\} \\
& \alpha * \bigcup X_{a,p,q} \subseteq \bigcup X_{a,p,\alpha * q} \\
= & \{\text{Lemma 6}\} \\
& \bigcup_{\beta \in X_{a,p,q}} \alpha * \beta \subseteq \bigcup X_{a,p,\alpha * q} \\
\Leftarrow & \{\text{Complete lattice properties}\} \\
& \forall \beta \in X_{a,p,q} \bullet \alpha * \beta \subseteq \bigcup X_{a,p,\alpha * q} \\
\Leftarrow & \{\text{Complete lattice properties}\}
\end{array}
$$

$$
\begin{array}{ll}
& \forall \beta \in X_{a,p,q} \bullet \alpha * \beta \in X_{a,p,\alpha * q} \\
= & \{\text{Relation (15)}\} \\
& \mathit{true}
\end{array}
$$

The proof of $(\!|S.p|\!)\ ;\ [\![p]\!] \sqsubseteq S$ is given below:

$$
\begin{array}{ll}
& ((\!|S.p|\!)\ ;\ [\![p]\!]).a.q \\
= & \{\text{Definition}\} \\
& (S.p).a * \bigcup X_{a,p,q} \\
= & \{\text{Lemma 6}\} \\
& \bigcup_{\beta \in X_{a,p,q}} (S.p).a * \beta \\
\subseteq & \{\text{Definition of } \mathsf{Proc}_X.A\} \\
& \bigcup_{\beta \in X_{a,p,q}} S.a.(p.a * \beta) \\
\subseteq & \{\text{Definition of } X_{a,p,q}\} \\
& \bigcup_{\beta \in X_{a,p,q}} S.a.q \\
= & \{\text{Complete lattice properties}\} \\
& S.a.q
\end{array}
$$

Finally, $\mathsf{skip} \sqsubseteq (\!|\, [\![p]\!].p \,|\!)$ is proved by:

$$
\begin{array}{ll}
& (\!|\, [\![p]\!].p \,|\!).a.q \\
= & \{\text{Definition}\} \\
& (\bigcup X_{a,p,p.a}) * q \\
\geq & \{\mathsf{emp} \in X_{a,p,p.a}\} \\
& \mathsf{emp} * q \\
= & \{\text{Lemma 6}\} \\
& q \quad \square
\end{array}
$$

Now, the proof of Theorem 22 follows from Theorem 20 applied to $\mathsf{Proc}_X.A$ and $\langle A \to \mathsf{Pred}, \ldots, (\!|\_|\!), [\![\_]\!] \rangle$, indeed.

We can also give the Hoare total correctness rule for mutually recursive procedures. Let W, I be sets such that $\langle W \times I, < \rangle$ is well founded. For each $i \in I$, A_i is a type of procedure parameters and B_i is a type of auxiliary values. For every $i \in I$, $X_i : A_i \to \mathcal{P}.\mathsf{Pred}$, such that for all $a \in A$, $X_i.a$ is closed under arbitrary unions, separation conjuction, and $\mathsf{emp} \in X_i.a$.

Theorem 24. *If for all $w \in W$, $p_w : \prod_i (B_i \to A_i \to \mathsf{Pred})$, $q : \prod_i (B_i \to A_i \to \mathsf{Pred})$, and $body : \prod_i \mathsf{Proc}.A_i \to \prod_i \mathsf{Proc}.A_i$ is monotonic, then the following Hoare rule is true*

$$
\begin{array}{ll}
& (\forall w \in W \bullet \forall i \in I \bullet \forall P \in \prod_i \mathsf{Proc}_{X_i}.A_i \bullet \\
& \quad p_{<(w,i)} \ \{\!|\ P\ |\!\}\ q \Rightarrow p_{w,i} \ \{\!|\ (body.P)_i\ |\!\}\ q_i) \wedge \\
& \quad (\forall P \in \prod_i \mathsf{Proc}_{X_i}.A_i \bullet body.P \in \prod_i \mathsf{Proc}_{X_i}.A_i)) \\
\Rightarrow & \\
& p \ \{\!|\ \mu\, body\ |\!\}\ q \ \wedge \ \mu\, body \in \prod_i \mathsf{Proc}_{X_i}.A_i.
\end{array}
$$

Proof. This theorem follows directly by applying Theorem 20 for the program lattice $\prod_i \mathsf{Proc}_{X_i}.A_i$ and by using Lemma 1. □

Similarly to Theorem 22, the fact that $(\mu\, body)_i$ is from $\mathsf{Proc}_{X_i}.A_i$ and not only from $\mathsf{Proc}.A_i$ is the key element of Theorem 24. This enables the use of the frame rule for $(\mu\, body)_i$.

The frame rule for pointer programs allows us to use the Hoare total correctness triple $\alpha * \beta\ \{\!|\ S\ |\!\}\ \alpha * \gamma$, if we prove $\beta\ \{\!|\ S\ |\!\}\ \gamma$ and α does not contain free variables modified by S. This is especially useful for procedures. We prove that a procedure P is correct assuming that the heap contains only the addresses that are relevant to P, and later we could use the correctness of procedure P in contexts where the heap contains some other non-interfering addresses. We are not aware of any other proof of the frame rule in the context of a language with mutually recursive procedures with parameters. Theorem 24 is the main new result which allows proving the frame rule for programs using mutually recursive procedures. The proof details of the frame rule can be found in [16,17].

Using Theorem 24, the proof obligations of the parsing procedures become simpler. If for all $N \in \mathsf{NonTerm}$:

$$X_N.(u, v) = \mathsf{SepPred}.(u, v)$$

then, for example, the proof obligation for parse_E becomes:

$$\begin{array}{ll} & (\forall a, u, v, \alpha \bullet \alpha \wedge \mathsf{val}.u \doteq a \doteq w \ \{\!|\, T.(u, v) \,|\!\}\ \alpha * \mathsf{post}_T(a,\ \mathsf{val}.u, \mathsf{val}.v)) \\ \wedge & (\forall a, u, v, \alpha \bullet \alpha \wedge \mathsf{val}.u \doteq a <_s w \ \{\!|\, E.(u, v) \,|\!\}\ \alpha * \mathsf{post}_E(a,\ \mathsf{val}.u, \mathsf{val}.v)) \\ \Rightarrow & \\ & (\forall a, u, v \bullet \mathsf{emp} \wedge \mathsf{val}.u \doteq a \doteq w \ \{\!|\, \mathsf{body}_E.T.E.(u, v) \,|\!\}\ \mathsf{post}_E(a,\ \mathsf{val}.u, \mathsf{val}.v)) \end{array}$$

We have proved the correctness of these procedures in PVS. An outline of this proof is given in [17].

9. Conclusions and future work

In this study, we have introduced a predicate transformer semantics for imperative programs implemented in the theorem prover PVS. We have treated mutually recursive procedures with parameters and local variables, and pointers.

We have introduced a model for pointers in which we treat addresses similarly to program variables. In this model, the heap is defined by the set of all allocated addresses and their values. We have mechanically verified separation logic properties and Hoare total correctness rules for heap operations. We have proved a frame rule that can be applied to mutually recursive procedures with value and value–result parameters, and local variables.

We have mechanically verified a complex example of a collection of mutually recursive procedures that build the abstract syntax trees of expressions generated by a context free grammar. In this example, we have used the procedure DisposeTree for disposing a binary tree. This shows the flexibility of our approach: we can use general procedures like DisposeTree in specific situations when the type of the tree labels are strings.

The program constructs introduced in this paper cover an important subclass of programs that can be written in an imperative programming language. We can add more features that are present in real programming languages. Extending this approach to pointer arithmetic is very simple. All we need is to assume that we have some address arithmetic $(+ : \mathsf{Address} \times \mathsf{Int} \to \mathsf{Address})$ which satisfies

$$a + 0 = a, \qquad (a + i) + j = a + (i + j), \qquad a + i = a + j \Rightarrow i = j$$

and to extend the allocation statement with the possibility of allocating a consecutive range of addresses. The statement $\mathsf{New}(e_1, \ldots, e_n)$ will allocate an address a, such that $a,\ a + 1,\ \ldots,\ a + n - 1$ are free. The values $e_1, \ldots, e_n$ will be stored at the addresses $a, \ldots, a + n - 1$.

For a given infinite cardinal γ, we can have program variable types of cardinals up to γ. The cardinal of all programs (and of procedures of a given type) is strictly greater than γ, which would prevent us from having higher-order procedures. However, in practice, we are interested only in procedures which can be defined using the program constructs introduced here, and these are only an infinite countable number. Therefore, we can introduce program variables of type procedures, and then pass them as parameters to other procedures.

Extending the language to support higher-order procedures and pointer arithmetic seem straightforward, however verifying some examples using higher-order procedures may be more challenging, and we plan to investigate it in future work.

Our implementation uses the dependent type mechanism of PVS. However, in PVS, dependent types can only be subtypes of a given type. This restriction does not allow us to use directly the PVS basic types as program variable types. We plan to investigate this problem further and improve the representation of the program semantics, such that it will be possible to use directly the theorem prover types as program types.

In this paper we did not treat the problem of completeness. Our guess is that techniques used for proving completeness for Hoare logics of imperative programs could be adapted to our work. However, it seems that the frame rule for mutually recursive procedures from this paper is not necessary to prove the completeness. This is so because one could always prove a more general specification about procedures. For example one could always state and prove the more general property (13) instead of (14). The frame rules for the pointer manipulating statements are required for completeness. The frame rule for mutually recursive procedure is useful when proving actual programs in practice because it enables local reasoning.

Other work on procedures and object oriented programs [15] is using a more powerful adaptation rule for proving programs with procedures. As noted in [15], an adaptation rule for programs manipulating objects (pointers) is more difficult to derive. The work from [15] is limited to procedures which only allocates addresses and never releases them. We plan to investigate in future work the completeness of our rules, and if it is possible to introduce a separation logic adaptation rule which would work for all kinds of programs (also those that releases addresses).

Acknowledgments

We thank Carsten Varming and Lars Birkedal for the discussion about their work on higher-order separation logic and the anonymous referees for their useful comments and suggestions which led to an improvement of this paper.

References

[1] R.J. Back, V. Preoteasa, Reasoning about recursive procedures with parameters, in: Proceedings of the 2003 Workshop on Mechanized Reasoning about Languages with Variable Binding, ACM Press, 2003, pp. 1–7.

[2] R.J. Back, V. Preoteasa, An algebraic treatment of procedure refinement to support mechanical verification, Formal Aspects of Computing 17 (May) (2005) 69–90.
[3] R.J. Back, J. von Wright, Refinement Calculus. A Systematic Introduction, Springer, 1998.
[4] L. Birkedal, Y. Yang, Relational parametricity and separation logic, Logical Methods in Computer Science 4 (2008).
[5] R.M. Burstall, Some techniques for proving correctness of programs which alter data structures, Machine Intelligence 7 (1972) 23–50.
[6] A. Church, A formulation of the simple theory of types, Journal of Symbolic Logic 5 (1940) 56–68.
[7] B.A. Davey, H.A. Priestley, Introduction to Lattices and Order, second edition, Cambridge University Press, New York, 2002.
[8] C.A.R. Hoare, An axiomatic basis for computer programming, Communications of the ACM 12 (10) (1969) 576–580.
[9] S.S. Ishtiaq, P.W. O'Hearn, Bi as an assertion language for mutable data structures, in: Proceedings of the 28th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, ACM Press, 2001, pp. 14–26.
[10] T. Nipkow, Hoare logics for recursive procedures and unbounded nondeterminism, in: J. Bradfield (Ed.), Computer Science Logic, CSL 2002, in: LNCS, vol. 2471, Springer, 2002, pp. 103–119.
[11] T. Nipkow, L.C. Paulson, M. Wenzel, Isabelle/HOL — A Proof Assistant for Higher-Order Logic, in: LNCS, vol. 2283, Springer, 2002.
[12] P.W. O'Hearn, J.C. Reynolds, H. Yang, Local reasoning about programs that alter data structures, in: CSL'01: Proceedings of the 15th International Workshop on Computer Science Logic, in: Lecture Notes in Computer Science, vol. 2142, Springer-Verlag, London, UK, 2001, pp. 1–19.
[13] S. Owre, N. Shankar, J.M. Rushby, D.W.J. Stringer-Clavert, PVS language reference. Technical report, Computer Science Laboratory, SRI International, Dec 2001.
[14] M.J. Parkinson, Local Reasoning for Java, Ph.D. Thesis, University of Cambridge, Computer Laboratory, Nov 2005.
[15] Cees Pierik, Frank S. de Boer, A proof outline logic for object-oriented programming, in: Formal Methods for Components and Objects, Theoretical Computer Science 343 (3) (2005) 413–442.
[16] V. Preoteasa, Mechanical verification of recursive procedures manipulating pointers using separation logic, in: J. Misra, T. Nipkow, E. Sekerinski (Eds.), FM 2006: Formal Methods, in: LNCS, vol. 4085, Springer-Verlag, August 2006, pp. 508–523.
[17] V. Preoteasa, Program Variables – The Core of Mechanical Reasoning about Imperative Programs. Ph.D. Thesis, Turku Centre for Computer Science, Nov 2006.
[18] J. Reynolds, Intuitionistic reasoning about shared mutable data structure, in: Millenial Perspectives in Computer Science, 2000.
[19] J. Reynolds, Separation logic: A logic for shared mutable data structures, in: 17th Annual IEEE Symposium on Logic in Computer Science, IEEE, July 2002.
[20] A. Tarski, A lattice-theoretical fixpoint theorem and its applications, Pacific Journal of Mathematics 5 (1955) 285–309.
[21] Carsten Varming, Lars Birkedal, Higher-order separation logic in isabelle/holcf, Electronic Notes in Theoretical Computer Science 218 (2008) 371–389.
[22] T. Weber, Towards mechanized program verification with separation logic, in: Jerzy Marcinkowski, Andrzej Tarlecki (Eds.), Computer Science Logic, 18th International Workshop, CSL 2004, 13th Annual Conference of the EACSL, Karpacz, Poland, September 2004, Proceedings, in: Lecture Notes in Computer Science, vol. 3210, Springer, September 2004, pp. 250–264.
[23] H. Yang, P.W. O'Hearn, A semantic basis for local reasoning, in: FoSSaCS '02: Proceedings of the 5th International Conference on Foundations of Software Science and Computation Structures, in: Lecture Notes in Computer Science, vol. 2303, Springer-Verlag, London, UK, 2002, pp. 402–416.