Viorel Preoteasa

Mechanical Verification of Recursive Procedures Manipulating Pointers using Separation Logic

Turku Centre *for* Computer Science

TUCS Technical Report
No 753, April 2006

Mechanical Verification of Recursive Procedures Manipulating Pointers using Separation Logic

Viorel Preoteasa
Department of Computer Science
Åbo Akademi University and
Turku Centre for Computer Science
DataCity, Lemminkäisenkatu 14A
Turku 20520, Finland

TUCS Technical Report
No 753, April 2006

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 consistent Hoare total correctness rules for pointer manipulating statements according to the predicate transformer semantics. We prove the frame rule in the context of a programming language with recursive procedures with value and result parameters and local variables, where program variables and addresses can store values of any type of the theorem prover. The theory, including the proofs, is implemented in the theorem prover PVS.

Keywords: Pointer Programs. Separation Logic. Recursive Procedures. Predicate Transformers Semantics. Mechanical Verification of Programs.

TUCS Laboratory
Software Construction

1 Introduction

Separation logic [10, 7, 13] 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 [6] theorem prover and separation logic, Weber [12] implements relatively complete Hoare [4] logics for a simple while programming language extended with heap operations. Nevertheless, his implementation does not treat (recursive) procedures and local variables.

In this paper, we introduce a predicate transformer semantics for imperative programs with pointers and define separation logic constructs. Based on this semantics, we prove Hoare total correctness rules for heap operations (new, dispose, lookup, and update). Our work is implemented in the theorem prover PVS [8] and it is based on a previous formalization [1] of Refinement Calculus [2] with recursive procedures.

Even if possible, we have chosen not to implement address arithmetic [10] in our calculus. Unlike approaches presented by Reynolds [10] or Weber [12], where memory addresses are integers and can store only integer values, our work allows addresses to store values of any type available in the theorem prover.

The main contribution of this work is the formal proof of the frame rule, [5, 13], in the context of a programming language with recursive procedures with value and result parameters and local variables, where program variables and addresses can store values of any type of the theorem prover.

2 Preliminaries

We use higher-order logic [3] as the underlying logic. In this section we recall some facts about refinement calculus [2] and about fixed points in complete lattices.

Let Σ be the state space. Predicates, denoted Pred, are the functions from $\Sigma \to \mathsf{bool}$. 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.

MTran is the type of all monotonic functions from Pred to Pred. Programs are modeled as elements of MTran. If $S : \mathsf{MTran}$ and $p : \mathsf{Pred}$, then $S.p : \mathsf{Pred}$ 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 extension of $\subseteq$, $\cup$, and $\cap$, respectively. The type MTran, together with the pointwise extension of the operations on predicates, forms a complete lattice. The partial order $\sqsubseteq$ on MTran is the *refinement relation* [2]. The predicate transformer

$S \sqcap T$ models *nondeterministic* choice – the choice between executing S or T is arbitrary.

Often we 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 starting from the initial state σ. We can model a nondeterministic program by a relation on Σ, i.e. a function $R : \Sigma \to \Sigma \to \mathsf{bool}$. The state σ' belongs to $R.\sigma$ if there exists an execution of the program starting in σ and ending in σ'.

If $p, q : \mathsf{Pred}$, $R : \Sigma \to \Sigma \to \mathsf{bool}$, $f : \Sigma \to \Sigma$, then we define

$[f] : \mathsf{MTran} \mathrel{\hat{=}} \lambda q, s \bullet q(f(s))$ – the monotonic predicate transformer corresponding to the function f.

$[R] : \mathsf{MTran} \mathrel{\hat{=}} \lambda q, s \bullet \forall s' \bullet R(s)(s') \Rightarrow q(s')$ – the monotonic predicate transformer corresponding to the nondeterministic choice given by R.

$\{p\} : \mathsf{MTran} \mathrel{\hat{=}} \lambda q \bullet p \cap q$ – the assert statement.

$\mathsf{if}\ p\ \mathsf{then}\ S\ \mathsf{else}\ T\ \mathsf{endif} : \mathsf{MTran} \mathrel{\hat{=}} (\{p\}\ ;\ S) \sqcup (\{\neg p\}\ ;\ T)$ – the conditional statement.

If L is a complete lattice and $f : L \to L$ is monotonic, then the least fix-point of f, denoted $\mu\, f$, exists [11]. If $b \in \mathsf{Pred}$ and $S \in \mathsf{MTran}$, then the iterative programming construct is define by:

$$\mathsf{while}\ b\ \mathsf{do}\ S\ \mathsf{od} \mathrel{\hat{=}} (\mu X \bullet \mathsf{if}\ b\ \mathsf{then}\ S\ ;\ X\ \mathsf{else\ skip\ fi})$$

Lemma 1 (Fusion lemma) *If f and g are monotonic functions on complete lattices L and L' and $h : L \to L'$ is continuous then*

1. *if $h \circ f \leq g \circ h$ then $h.(\mu f) \leq \mu g$*
2. *if $h \circ f = g \circ h$ then $h.(\mu f) = \mu g$*

Proof. See Theorem 19.5, page 322 from [2] ∎

Lemma 2

$$\mathsf{while}\ b\ \mathsf{do}\ S\ \mathsf{od}.q = (\mu X \bullet (b \cap S.X) \cup (\neg b \cap q))$$

Proof.

$$\begin{array}{ll} & \mathsf{while}\ b\ \mathsf{do}\ S\ \mathsf{od}.q = (\mu X \bullet (b \cap S.X) \cup (\neg b \cap q)) \\ \Leftarrow & \{\text{Lemma 1 using } h.X = X.q\} \\ & h.(\{b\}\ ;\ S\ ;\ X \sqcup \{\neg b\}) = b \cap S.(h.X) \cup \neg b \cap q \end{array}$$

$\Leftrightarrow$ {Definitions}

$(b \cap S.(X.q)) \cup (\neg b \cap q) = (b \cap S.(X.q)) \cup (\neg b \cap q)$

$\Leftrightarrow$ {Equality}

true

■

Lemma 3 *If $f : L \to L$ is a monotonic function on the complete lattice L and $h : L \to$ bool satisfies:*

1. $h.x \Rightarrow h.(f.x)$ *and*
2. $(\forall i \in I \bullet h.x_i) \Rightarrow h.(\bigvee_{i \in I} x_i)$

then $h.(\mu f)$ is true.

Proof. We will use transfinite induction on ordinals to prove this lemma. We define an ordinal indexed collection of functions:

$$\begin{array}{lcl} f^0.x & \hat{=} & x, \\ f^{a+1}.x & \hat{=} & f.(f^a.x) \text{ for arbitrary ordinals } a, \\ f^a.x & \hat{=} & \bigvee_{b<a} f^b.x \text{ for nonzero limit ordinals } a. \end{array}$$

From [2], Theorem 19.3, page 321, we know that there exists an ordinal c such that $\mu f = f^c.\bot$, where $\bot$ is the smallest element of the lattice L. It can be easily proved by transfinite induction that $h.(f^a.\bot)$ is true for all ordinals a. Therefore $h.(\mu\, f)$ holds. ■

Corollary 4 *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 $\mu_L\, f = \mu_{L'}\, f$.*

3 Program variables, addresses, constants, & expressions

We assume that we have a type value that contains all program variables, program addresses, and constants. 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 represents 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 $\mathsf{T}.x$, as an arbitrary subtype of value. $\mathsf{T}.x$ represents all values that can be assigned to

x. For a type $X \subseteq \mathsf{value}$ we denote by X^{nil} the type $X \cup \{\mathsf{nil}\}$ and we define the subtypes $\mathsf{V}.X \subseteq \mathsf{variable}$, $\mathsf{A}.X \subseteq \mathsf{address}$, and $\mathsf{B}.X \subseteq \mathsf{address}^{\mathsf{nil}}$ by

$$\begin{aligned}\mathsf{V}.X &\hat{=} \{x \in \mathsf{variable} \mid \mathsf{T}.x = X\}\\ \mathsf{A}.X &\hat{=} \{x \in \mathsf{address} \mid \mathsf{T}.x = X\}\\ \mathsf{B}.X &\hat{=} (\mathsf{A}.X)^{\mathsf{nil}}\end{aligned}$$

The type $\mathsf{V}.X$ represents the program variables of type X. The elements of $\mathsf{A}.X$ are the addresses that can store elements of type X. An element of $\mathsf{B}.X$ is either nil or is an address that can store an element of type X. For example the type of the program variables of type addresses to natural numbers is defined by $\mathsf{V}.(\mathsf{B}.\mathsf{nat})$.

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

$$\begin{array}{l}\mathsf{struct\ btree}\{\\ \quad \mathsf{int\ label};\\ \quad \mathsf{btree}\ *\mathsf{left};\\ \quad \mathsf{btree}\ *\mathsf{right}\}\end{array} \tag{1}$$

In our formalism, binary trees, labeled with elements from an arbitrary type A, are modeled by a type $\mathsf{ptree}.A$. Elements of $\mathsf{ptree}.A$ are records with three components: $a : A$, and $p, q : \mathsf{B}.\mathsf{ptree}.A$. Formally the record structure on $\mathsf{ptree}.A$ is given by a bijective function $\mathsf{ptree} : A \times \mathsf{B}.(\mathsf{ptree}.A) \times \mathsf{B}.(\mathsf{ptree}.A) \to \mathsf{ptree}.A$. If $a : A$, and $p, q : \mathsf{B}.\mathsf{ptree}$, then $\mathsf{ptree}.(a, p, q)$ is the record containing the elements a, p, q. The inverse of ptree has three components $(\mathsf{label}, \mathsf{left}, \mathsf{right})$, $\mathsf{label} : \mathsf{ptree}.A \to A$ and $\mathsf{lef}, \mathsf{right} : \mathsf{ptree}.A \to \mathsf{B}.(\mathsf{ptree}.A)$. The type $\mathsf{ptree.int}$ corresponds to btree from definition (1) and the type $\mathsf{B}.(\mathsf{ptree.int})$ corresponds to $(\mathsf{btree}\ *)$ from (1).

We access and update program locations using two functions.

$$\begin{aligned}&\mathsf{val}.x : \Sigma \to \mathsf{T}.x\\ &\mathsf{set}.x : \mathsf{T}.x \to \Sigma \to \Sigma\end{aligned}$$

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 a set of axioms [1].

Program expressions of type A are functions from Σ to A. We denote by $\mathsf{Exp}.A$ 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 $\oplus : \mathsf{Exp}.A \times \mathsf{Exp}.B \to \mathsf{Exp}.C$ 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

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

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

If $e \in \mathsf{Exp}.A$, $x \in \mathsf{variable}$, and $e' \in \mathsf{Exp}.(\mathsf{T}.x)$, 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 $e \in \mathsf{Exp}.A$, as the semantic correspondent of the syntactic condition that x does not occur free in e. If $f \in \Sigma \to \Sigma$ and $e \in \mathsf{Exp}.A$, 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 $a \in \mathsf{T}.x$.

The program expressions defined so far may depend not only on the current values of the program variables, but also on the values from the stack. For example $\mathsf{del}.x \,;\, \mathsf{val}.x$ does not depend on any program variable value (changing any variable, including x, does not change the value of this expression), but it depends on the top value stored in the stack. We define the subclass of program expression which depends only on the current values of the program variables. Two states σ and σ' are val–equivalent if for all program variables x, $\mathsf{val}.x.\sigma = \mathsf{val}.x.\sigma'$. A program expression $e \in \mathsf{Exp}.A$ is called val–determined if for all σ and σ' val–equivalent we have $e.\sigma = e.\sigma'$.

3.1 Program statements and Hoare total correctness triples

In this subsection we introduce the program statements for assignment and handling local variables and we will also give the Hoare total correctness rules to work with these statements.

Let $x, y \in \mathsf{variable}$ such that $\mathsf{T}.x = \mathsf{T}.y$ and $e \in \mathsf{Exp}.(\mathsf{T}.x)$. We recall the definition of the assignment statement from [2] and the definition of local variables manipulation statements from [1].

- $x := e \mathrel{\hat{=}} [\lambda\sigma \bullet \mathsf{set}.x.(e.\sigma).\sigma]$ – assignment
- $\mathsf{add}.x \mathrel{\hat{=}} (\lambda\sigma, \sigma' \bullet \sigma = \mathsf{del}.x.\sigma')$ – add local variable

- $\mathsf{Add}.x \mathrel{\hat{=}} [\mathsf{add}.x]$ – add local variable statement
- $\mathsf{add}.x.e \mathrel{\hat{=}} (\lambda\sigma, \sigma' \bullet \exists\sigma_0 \bullet \sigma = \mathsf{del}.x.\sigma_0 \wedge \mathsf{set}.x.(e.\sigma).\sigma_0 = \sigma')$ – add and initialize local variable
- $\mathsf{Add}.x.e \mathrel{\hat{=}} [\mathsf{add}.x.e]$ – add and initialize local variable statement
- $\mathsf{Del}.x \mathrel{\hat{=}} [\mathsf{del}.x]$ – delete local variable statement
- $\mathsf{del}.x.y \mathrel{\hat{=}} (\lambda\sigma \bullet \mathsf{set}.y.(\mathsf{val}.x.\sigma).(\mathsf{del}.x.\sigma))$ – save and delete local variable
- $\mathsf{Del}.x.y \mathrel{\hat{=}} [\mathsf{del}.x.y]$ – save and delete local variable statement

See [1] for detailed explanations of these program constructs.

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 $\alpha \subseteq S.\beta$.

Theorem 5 *If x, y are lists of program variables, α and β are predicates, and e is a program expression then*

(i) $(\exists\, a \bullet \alpha.a) \{\!|\, S \,|\!\} \beta \Leftrightarrow (\forall a \bullet (\alpha.a \{\!|\, S \,|\!\} \beta))$

(ii) $(\mathsf{del}.x \,;\, \alpha) \{\!|\, \mathsf{Del}.x \,|\!\} \alpha$

(iii) $\alpha \{\!|\, \mathsf{Add}.x \,|\!\} (\mathsf{del}.x \,;\, \alpha)$

(iv) $\alpha \{\!|\, \mathsf{Add}.x.e \,|\!\} (\mathsf{del}.x \,;\, \alpha)$

(v) α *is* val*–determined* $\Rightarrow \alpha[x := e] \{\!|\, \mathsf{Add}.x.e \,|\!\} \alpha$

(vi) α *is* $\mathsf{set}.y$*–independent* $\Rightarrow (\mathsf{del}.x \,;\, \alpha) \{\!|\, \mathsf{Del}.x.y \,|\!\} \alpha$

(vii) α *is* val*–determined and* $\mathsf{set}.(x - y)$*–independent* $\Rightarrow$ $\alpha[y := x] \{\!|\, \mathsf{Del}.x.y \,|\!\} \alpha$

4 Heap operations and separation logic

So far we have introduced the mechanism of accessing and updating addresses, but we need also a mechanism for allocating and deallocating them. We introduce the type $\mathsf{allocaddr} \mathrel{\hat{=}} \mathcal{P}_{fin}(\mathsf{address})$, the finite powerset of $\mathsf{address}$; and a special program variable $\mathsf{alloc} \in \mathsf{variable}$ of type $\mathsf{allocaddr}$ ($\mathsf{T}.\mathsf{alloc} = \mathsf{allocaddr}$). The set $\mathsf{val}.\mathsf{alloc}.\sigma$ contains only those addresses allocated in state σ. The heap in a state σ is made 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: to add addresses to a state and to delete addresses from a state.

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

Based on these elements we build all heap operations and separation logic operators.

4.1 Separation logic operators

Definition 6 *If* $e, f : \mathsf{Pred}$, $r : \Sigma \to \mathsf{B}.X$, *and* $g : \Sigma \to X$, *then we define*

$$\begin{array}{lcl} \mathsf{emp}.\sigma : \mathsf{bool} & \hat{=} & (\mathsf{val.alloc}.\sigma = \emptyset) \\ (e * f).\sigma : \mathsf{bool} & \hat{=} & \exists A \subseteq \mathsf{val.alloc}.\sigma \bullet e.(\mathsf{set.alloc}.A.\sigma) \wedge f.(\mathsf{dispose}.A.\sigma) \\ (r \mapsto g).\sigma : \mathsf{bool} & \hat{=} & \mathsf{val}.(r.\sigma).\sigma = g.\sigma \wedge \mathsf{val.alloc}.\sigma = \{r.\sigma\} \\ (r \mapsto \_) : \mathsf{Pred} & \hat{=} & \exists\, g : X \bullet r \mapsto g \end{array}$$

Lemma 7 *The following relations hold*

1. $\alpha * \mathsf{emp} = \alpha$
2. $\alpha * \beta = \beta * \alpha$
3. $\alpha * (\beta * \gamma) = (\alpha * \beta) * \gamma$
4. $(\exists\, a \bullet \alpha * \beta.a) = \alpha * (\exists\, \beta)$
5. *If* γ *is* set.alloc*–independent then* $\alpha * (\beta \wedge \gamma) = (\alpha * \beta) \wedge \gamma$
6. $(\bigcup_{i \in I} p_i) * q = \bigcup_{i \in I} (p_i * q)$
7. $(\bigcap_{i \in I} p_i) * q \subseteq \bigcap_{i \in I} (p_i * q)$
8. $\mathsf{del}.x \ ; \mathsf{emp} = \mathsf{emp}$
9. $\mathsf{del}.x \ ; (\alpha * \beta) = (\mathsf{del}.x \ ; \alpha) * (\mathsf{del}.x \ ; \beta)$
10. *If* e *is* set.alloc*–independent then* $(\alpha * \beta)[x := e] = \alpha[x := e] * \beta[x := e]$
11. *If* e *is* set.alloc*–independent then* $(r \mapsto g)[x := e] = r[x := e] \mapsto g[x := e]$

In [10] a subset of program expressions called pure are defined. These are expressions which does 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 what are 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

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

The pure expressions from [10] correspond to set.alloc–independent and set address independent expressions in our framework.

We need also 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:

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

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

4.2 Specifying binary trees properties with separation logic

Let atreecons be the type of nonempty abstract binary trees with labels from a type A. We assume that nil denotes the empty tree and we take atree = atreecons $\cup \{\mathsf{nil}\}$. The abstract tree structure on atree is given by an injective function

$$\mathsf{atree} : A \to \mathsf{atree} \to \mathsf{atree} \to \mathsf{atreecons}$$

which satisfies the following induction axiom:

$$\forall P : \mathsf{atree} \to \mathsf{bool} \bullet P.\mathsf{nil} \wedge (\forall a, s, t \bullet P.s \wedge P.t \Rightarrow P.(\mathsf{atree}.a.s.t)) \Rightarrow \forall t \bullet P.t$$

Using this axiom we can prove that the function atree is also surjective and we denote by label : atreecons $\to A$ and left, right : atreecons $\to$ atree the components of atree inverse.

Abstract binary trees are very convenient to specify and prove properties involving binary trees. However imperative programming languages represent binary trees using pointers. We introduce a predicate tree : atree $\to$ B.ptree $\to$ Pred. The predicate tree.$t.p$ will be true in those states σ in which address p stores the tree t. The predicate tree.$t.p$ is defined by structural induction on t.

$$\begin{aligned}
&\mathsf{tree.nil}.p.\sigma && \hat{=}\ p = \mathsf{nil} \wedge \mathsf{emp}\\
&\mathsf{tree}.(\mathsf{atree}(a, t_1, t_2)).p && \hat{=}\ (\exists\, p_1, p_2 \bullet p \mapsto \mathsf{ptree}(a, p_1, p_2) * \mathsf{tree}.t_1.p_1 * \mathsf{tree}.t_2.p_2)
\end{aligned}$$

We extend the predicate tree to programs expressions, tree : $(\Sigma \to \mathsf{atree}) \to (\Sigma \to \mathsf{B.ptree}) \to \mathsf{Pred}$, by $\mathsf{tree}.e.f.\sigma \hat{=} \mathsf{tree}.(e.\sigma).(f.\sigma).\sigma$.

Lemma 8 *If* $a : \Sigma \to \mathsf{atree}$ *and* $p : \Sigma \to \mathsf{B.ptree}$ *then*

$$\mathsf{tree}.a.p \wedge p \neq \mathsf{nil} \subseteq (\exists\, a_1, a_2, c, t_1, t_2 \bullet (p \mapsto \mathsf{ptree}.(c, t_1, t_2)) * \mathsf{tree}.a_1.t_1 * \mathsf{tree}.a_2.t_2)$$

Lemma 9 *If* e *and* f *are two expressions of appropriate types then*

1. $\mathsf{del}.x \;;\; \mathsf{tree}.a.p = \mathsf{tree}.(\mathsf{del}.x \;;\; a).(\mathsf{del}.x \;;\; p)$

2. *If* e *is* set.alloc*–independent then*

$$(\mathsf{tree}.a.p)[x := e] = \mathsf{tree}.(a[x := e]).(p[x := e])$$

4.3 Pointer manipulation statements

We introduce in this subsection the statements for pointer manipulation and their Hoare total correctness rules.

Definition 10 *If* $X \subseteq \mathsf{value}$, $x \in \mathsf{V}.(\mathsf{B}.X)$, $e : \Sigma \to X$, $r : \Sigma \to \mathsf{B}.X$, $y \in \mathsf{V}.X$, *and* $f : X \to \mathsf{T}.y$ *then we define*

$$\begin{array}{ll}
\mathsf{New}.X.(x,e) : \mathsf{MTran} & \hat{=}\ [\lambda\sigma, \sigma' \bullet \exists a : \mathsf{A}.X \bullet \neg\mathsf{alloc}.\sigma.a \wedge \\
& \qquad \sigma' = \mathsf{set}.a.(e.\sigma).(\mathsf{set}.x.a.(\mathsf{addaddr}.a.\sigma))] \\
\mathsf{Dispose}.r : \mathsf{MTran} & \hat{=}\ \{\lambda\sigma \bullet \mathsf{alloc}.\sigma.(r.\sigma)\} \;;\; [\lambda\sigma \bullet \mathsf{dispose}.(r.\sigma).\sigma] \\
y := r \to f : \mathsf{MTran} & \hat{=}\ \{\lambda\sigma \bullet \mathsf{alloc}.\sigma.(r.\sigma)\} \;;\; [\lambda\sigma \bullet \mathsf{set}.y.(f.(\mathsf{val}.(r.\sigma).\sigma)).\sigma] \\
[r] \;:=\; e : \mathsf{MTran} & \hat{=}\ \{\lambda\sigma \bullet \mathsf{alloc}.\sigma.(r.\sigma)\} \;;\; [\lambda\sigma \bullet \mathsf{set}.(r.\sigma).(e.\sigma).\sigma]
\end{array}$$

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. This statement assumes that there is always an address of type X available for allocation. The statement $\mathsf{Dispose}.r$ deletes the address r from 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 in dispose, lookup, or update statements r is not an allocated address, then these statements do not terminate.

Next we introduce Hoare correctness rules for these statements

Lemma 11 *If* $X \subseteq \mathsf{value}$, $x \in \mathsf{V}.(\mathsf{B}.X)$, $a \in \mathsf{B}.X$, $e : \Sigma \to X$ *is* set.alloc*–independent and non-alloc independent then*

1. $\mathsf{val}.x \doteq a \wedge \mathsf{emp}\ \{\!|\ \mathsf{New}.X.(x,e)\ |\!\}\ \mathsf{val}.x \mapsto e[x := a]$

2. e *is* set.x*–independent* $\Rightarrow \mathsf{emp}\ \{\!|\ \mathsf{New}.X.(x,e)\ |\!\}\ \mathsf{val}.x \mapsto e$

3. $\mathsf{emp}\ \{\!|\ \mathsf{New}.X.(x,e)\ |\!\}\ (\exists\, a \bullet \mathsf{val}.x \mapsto e[x := a])$

Lemma 12 *Let* $r : \Sigma \to \mathsf{address}^{\mathsf{nil}}$ *and* $\alpha : \Sigma \to \mathsf{Pred}$, *if* r *is* set.alloc*–independent then*

$$r \mapsto \_\ \{\!|\ \mathsf{Dispose}.r\ |\!\}\ \mathsf{emp}$$

Lemma 13 *If* $a : \mathsf{T}.x$, $r : \Sigma \to \mathsf{B}.X$ *is* set.alloc*–independent,* $f : X \to \mathsf{T}.x$, *and* $e : \Sigma \to X$ *is* set.alloc*–independent then*

1. $\mathsf{val}.x \doteq a \wedge (r \mapsto e)$ $\{\!|\, x := r \rightarrow f \,|\!\}$ $\mathsf{val}.x \doteq f \circ e[x := a] \wedge (r \mapsto e)[x := a]$

2. *e, r, and e are* $\mathsf{set}.x$*–independent* $\Rightarrow$
$r \mapsto e$ $\{\!|\, x := r \rightarrow f \,|\!\}$ $\mathsf{val}.x \doteq f \circ e \wedge r \mapsto e$

Lemma 14 *If $r : \Sigma \rightarrow \mathsf{B}.X$ and $e : \Sigma \rightarrow X$ are set address independent then*

$$r \mapsto \_ \ \{\!|\, [r] := e \,|\!\} \ r \mapsto e$$

5 Recursive procedures

In this subsection we recall some facts about recursive procedures from [1] and we introduce a modified version of the recursive procedure correctness theorem.

A procedure with parameters from A or simply a procedure over A, is an element from $A \rightarrow \mathsf{MTran}$. We define the type $\mathsf{Proc}.A = A \rightarrow \mathsf{MTran}$, the type of all procedures over A. 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 $a : A$ is the program $P.a$.

Every monotonic function F from $\mathsf{Proc}.A$ to $\mathsf{Proc}.A$ defines a recursive procedure $P \in \mathsf{Proc}.A$, $P = \mu\, F$, where $\mu\, F$ is the least fixpoint of F. For example, the recursive procedures that disposes a tree from memory is defined by

```
procedure DisposeTree(value-result t : B.ptree)
    local x : B.ptree
    if val.t ≠ nil then
        x := val.t → left ;
        DisposeTree.x ;
        x := val.t → right ;
        DisposeTree.x ;
        Dispose(val.t) ;
        t := nil
    endif
```
(2)

The procedure DisposeTree can be called by passing a program variable u of type B.ptree. The procedure call DisposeTree.u disposes the tree stored in u and sets u to nil. The type of the parameters of the procedure DisposeTree is $A = \mathsf{V}.(\mathsf{B.ptree})$. We use the notation (2) as an abbreviation for the following formal definition of the procedure DisposeTree.

$$\mathsf{DisposeTree} = \mu\, \mathsf{body\text{–}dt}$$

where $\mathsf{body\text{–}dt} : \mathsf{Proc}.A \to \mathsf{Proc}.A$ is given by

$$\begin{array}{l}\mathsf{body\text{–}dt}.P = (\lambda u : A \bullet \\ \quad \mathsf{Add}.t.(\mathsf{val}.u)\ ;\ \mathsf{Add}.x\ ; \\ \quad \mathsf{if}\ \mathsf{val}.t \neq \mathsf{nil}\ \mathsf{then} \\ \quad\quad x := \mathsf{val}.t \to \mathsf{left}\ ; \\ \quad\quad P.x\ ; \\ \quad\quad x := \mathsf{val}.t \to \mathsf{right}\ ; \\ \quad\quad P.x\ ; \\ \quad\quad \mathsf{Dispose}.(\mathsf{val}.t)\ ; \\ \quad\quad t := \mathsf{nil} \\ \quad \mathsf{endif} \\ \quad \mathsf{Del}.x\ ;\ \mathsf{Del}.t.u)\end{array}$$

To be able to state the correctness theorem for recursive procedures we need to extend all operations on predicates, and programs to parametric predicates and procedures. For example if B is a type of specification parameters, $p, q : B \to A \to \mathsf{Pred}$, and $P, Q \in \mathsf{Proc}.A$, then we define the *procedure Hoare total correctness triple* by:

$$p\ \{\![\ P\ ]\!\}\ q \Leftrightarrow (\forall b, a \bullet p.b.a\ \{\![\ P.a\ ]\!\}\ q.b.a)$$

We assume that all operations on predicates and programs are lifted similarly.

Let W be a non-empty type and $p_w : B \to A \to \mathsf{Pred}$. If $<$ is a binary relation on W then we define

- $p_{<w} = \bigcup\{p_v \mid v < w\}$
- $p = \bigvee\{p_w \mid w \in W\}$

Theorem 15 *If L is a complete sublattice of $\mathsf{Proc}.A$, for all $w \in W$, $p_w : B \to A \to \mathsf{Pred}$, $q : B \to A \to \mathsf{Pred}$ and $body : L \to L$ is monotonic, then the following Hoare rule is true*

$$\frac{(\forall w : W,\ P : L \bullet p_{<w}\ \{\![\ P\ ]\!\}\ q \Rightarrow p_w\ \{\![\ body.P\ ]\!\}\ q)}{p\ \{\![\ \mu_L\, body\ ]\!\}\ q}$$

Proof. The proof of the similar result from [1] (where $L = \mathsf{Proc}.A$) can be adapted to this case too. ∎

6 Frame rule and recursive procedures

The specification of the procedure DisposeTree is:

$$(\forall a \bullet \mathsf{tree}.u.a\ \{\![\ \mathsf{DisposeTree}.u\ ]\!\}\ \mathsf{emp} \wedge u = \mathsf{nil}) \tag{3}$$

This Hoare total correctness triple asserts that if the heap contains only a tree with the root in u, after calling $\mathsf{DisposeTree}.u$ the heap is empty and the value of u is nil. However, we cannot use this property in contexts where the heap contains other addresses in addition to the ones specified by $\mathsf{tree}.u.a$. For example, in the recursive definition of $\mathsf{DisposeTree}$, the right subtree is still in the heap while we dispose the left subtree. We would like to derive a property like:

$$(\forall a \bullet \alpha * \mathsf{tree}.u.a \ \{\!|\ \mathsf{DisposeTree}.u \ |\!\}\ \alpha \wedge u = \mathsf{nil}) \tag{4}$$

for all predicates α which does not contain u free. This can be achieved using the frame rule.

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 a nonempty type $X \subseteq A \to \mathsf{Pred}$. The type X denotes those formulas we could add to a Hoare triple when using the frame rule, and they are in general formulas which does not contain free variables modified by the procedure. For procedure $\mathsf{DisposeTree}$ the set X is $\{\alpha : \mathsf{V}.(\mathsf{B.ptree}) \to \mathsf{Pred} \mid (\forall u \bullet \alpha.u \text{ is } \mathsf{set}.u\text{–independent})\}$. We denote by

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

If we are able to prove that procedure $\mathsf{DisposeTree}$ belongs to $\mathsf{Proc}_X.A$ and satisfies (3) then we can use (4) when proving correctness of programs calling $\mathsf{DisposeTree}$. The definition of $\mathsf{Proc}_X.A$ is a generalization to procedures of the concept "local predicate transformers which modifies a set V" of program variables from [13].

Lemma 16 *$\mathsf{Proc}_X.A$ is a complete sublattice of $\mathsf{Proc}.A$.*

Proof. We need to prove that $\mathsf{Proc}_X.A$ is closed under arbitrary meets and joins. Let $P_i \in \mathsf{Proc}_X.A$ for all $i \in I$, then

$$\begin{array}{ll} & \mathsf{Proc}_X.A.(\bigsqcup_{i\in I} P_i) \\ = & \{\text{Definition}\} \\ & (\forall \alpha : X,\ \forall q \bullet \alpha * (\bigsqcup_{i\in I} P_i).q \subseteq (\bigsqcup_{i\in I} P_i).(\alpha * q)) \\ = & \{\text{Lemma 7}\} \\ & (\forall \alpha : X,\ \forall q \bullet \bigcup_{i\in I}(\alpha * P_i.q) \subseteq \bigcup_{i\in I} P_i.(\alpha * q)) \\ \Leftarrow & \{\text{Complete lattice properties}\} \\ & (\forall i \in I \bullet \forall \alpha : X,\ \forall q \bullet \alpha * P_i.q \subseteq P_i.(\alpha * q)) \\ = & \{\text{Definition}\} \end{array}$$

$(\forall i \in I \bullet \mathsf{Proc}_X.A.P_i)$

For arbitrary intersections we have a similar proof. ∎

Theorem 17 *If for all $w \in W$, $p_w : B \to A \to \mathsf{Pred}$, $q : B \to A \to \mathsf{Pred}$ and $body : \mathsf{Proc}.A \to \mathsf{Proc}.A$ is monotonic, then the following Hoare rule is true*

$$\frac{(\forall w : W,\ P : \mathsf{Proc}_X.A \bullet p_{<w}\ \{\!|\ P\ |\!\}\ q \Rightarrow p_w\ \{\!|\ body.P\ |\!\}\ q) \quad \wedge \quad (\forall P : \mathsf{Proc}_X.A \bullet \mathsf{Proc}_X.A.(body.P))}{(p\ \{\!|\ \mu\, body\ |\!\}\ q) \quad \wedge \quad \mathsf{Proc}_X.A.(\mu\, body)}$$

Proof. Using Theorem 15, Corollary 4, and Lemma 16 ∎ This theorem lets us, when proving a recursive procedure, to assume a stronger property (like (4)), and to prove a weaker property (like (3)). When using the procedure correctness statement in proving other programs, we can also use a stronger property (like (4)).

7 Frame rule

In this section we prove the frame rule for the program statements we introduced so far.

Definition 18 *If $f : \Sigma \to \Sigma$ then we call a relation $R \in \mathsf{Rel}$ f–*independent *if for all $\sigma, \sigma' \in \Sigma$, $R.\sigma.\sigma' \Rightarrow R.(f.\sigma).(f.\sigma')$. The relation R is* set.alloc*–independent if it is* set.alloc.A*–independent for all $A \subseteq \mathcal{P}_{fin}.$*address*.

The relation R is called address preserving *if for all $\sigma, \sigma' \in \Sigma$, $R.\sigma.\sigma' \Rightarrow$ $\mathsf{val.alloc}.\sigma = \mathsf{val.alloc}.\sigma'$.*

A function $f : \Sigma \to \Sigma$ is called set.alloc–independent *(*address preserving*) if the relation $(\lambda\sigma, \sigma' \bullet f.\sigma = \sigma')$ is.*

Lemma 19 *The relations* add.x *and* add.$x.e$ *and the functions* del.x *and* del.$x.y$ *are* set.alloc*–independent and address preserving.*

Definition 20 *A predicate transformer S is called* $*$–super-junctive *if for all predicates $\alpha, \beta \in \mathsf{Pred}$, $S.\alpha * S.\beta \subseteq S.(\alpha * \beta)$.*

Lemma 21 *If $R \in \mathsf{Rel}$ $(f : \Sigma \to \Sigma)$ is* set.alloc*–independent and address preserving then $[R]$ $([f])$ is $*$–super-junctive.*

Corollary 22 Add.x, Add.$x.e$, Del.x, *and* Del.$x.y$ *are $*$–super-junctive.*

Theorem 23 (Frame rule for parameters and local variables)

1. *if α is* set.y–indep *and $(\forall q \bullet (\mathsf{del}.x\ ;\ \alpha) * S.q \subseteq S.((\mathsf{del}.x\ ;\ \alpha) * q))$ then*

$$(\forall q \bullet \alpha * (\mathsf{Add}.x.e\ ;\ S\ ;\ \mathsf{Del}.x.y).q \subseteq (\mathsf{Add}.x.e\ ;\ S\ ;\ \mathsf{Del}.x.y).(\alpha * q))$$

2. *if* $(\forall q \bullet (\mathsf{del}.x \,;\, \alpha) * S.q \subseteq S.((\mathsf{del}.x \,;\, \alpha) * q))$ *then*

$$(\forall q \bullet \alpha * (\mathsf{Add}.x \,;\, S \,;\, \mathsf{Del}.x).q \subseteq (\mathsf{Add}.x \,;\, S \,;\, \mathsf{Del}.x).(\alpha * q))$$

Proof. Using Lemma 21 and Lemma 5. ∎

Lemma 24 *If* $x \in V(B(X))$, $e \in \Sigma \to X$ *such that* α *is* set.x*–independent and is non alloc independent, and* e *is* set.alloc*–independent then*

$$\alpha * \mathsf{New}(X)(x, e).q \subseteq \mathsf{New}(X)(x, e).(\alpha * q)$$

Lemma 25 *If* r *is* set.alloc*–independent then*

$$\alpha * \mathsf{Dispose}(r).q \subseteq \mathsf{Dispose}(r).(\alpha * q)$$

Lemma 26 *If* $r \in \Sigma \to \mathsf{B}.B$ *and* $f : X \to \mathsf{T}.y$ *such that* r *is* set.alloc*–independent and* α *is* set.y*–independent then*

$$\alpha * (y := r \to f).q \subseteq (y := [r] \to f).(\alpha * q)$$

Lemma 27 *If* $r \in \Sigma \to \mathsf{B}.X$ *and* $e : \Sigma \to X$ *such that* r *is* set.alloc*–independent,* e *is* set.alloc*–independent, and* α *is non alloc independent then*

$$\alpha * ([r] := e).q \subseteq ([r] := e).(\alpha * q)$$

Lemma 28 *If* b *is* set.alloc*–independent and* $(\forall q \bullet \alpha * S.q \subseteq S.(\alpha * q))$ *then*

$$\forall q \bullet \alpha * (\mathsf{while}\ b\ \mathsf{do}\ S\ \mathsf{od}.q) \subseteq \mathsf{while}\ b\ \mathsf{do}\ S\ \mathsf{od}.(\alpha * q)$$

Proof.

$\alpha * (\mathsf{while}\ b\ \mathsf{do}\ S\ \mathsf{od}.q) \subseteq \mathsf{while}\ b\ \mathsf{do}\ S\ \mathsf{od}.(\alpha * q)$

$\Leftrightarrow$ {Lemma 2}

$\alpha * (\mu X \bullet b \wedge S.X \vee \neg b \wedge q) \subseteq (\mu X \bullet b \wedge S.X \vee \neg b \wedge (\alpha * q))$

$\Leftarrow$ {Lemma 1 using $h.X = \alpha * X$}

$\alpha * (b \wedge S.X \vee \neg b \wedge q) \subseteq b \wedge S.(\alpha * X) \vee \neg b \wedge (\alpha * q)$

- Subderivation

  $\alpha * (b \wedge S.X \vee \neg b \wedge q)$

  $=$ {Lemma 7}

  $\alpha * (b \wedge S.X) \vee \alpha * (\neg b \wedge q)$

  $=$ {b is set.alloc–independent and Lemma 7}

  $b \wedge (\alpha * S.X) \vee \neg b \wedge (\alpha * q)$

$= \{\text{Assumption}\}$

$b \wedge S.(\alpha * X) \vee \neg b \wedge (\alpha * q)$

$= \{\text{subderivation}\}$

true

■

Although we work at the semantic level we can define the subclass of programs, denoted Prog, built using the program constructs presented in this paper where Add and Del statements are only used in pairs, like in the definition of the procedure DisposeTree. For a program $S \in \mathsf{Prog}$ we define by induction on the program structure the set of variables modified by S, in a usual manner.

Theorem 29 (Frame rule) *If $S \in \mathsf{Prog}$, V is the set of variables modified by P, $\alpha, q \in \mathsf{Pred}$, and $(\forall x \in V \bullet \alpha$ is $\mathsf{set}.x$–independent), then*

$$\alpha * S.q \subseteq S.(\alpha * q)$$

Proof. Using Lemmas 23, 24, 25, 26, 27, 28, and similar results which are true for assignment statement and sequential composition of programs. ■

8 Disposing a binary tree from memory

In this section we outline the correctness proof of the procedure DisposeTree (2). Let $X = \{\alpha : A \to \mathsf{Pred} \mid \alpha.u$ is $\mathsf{set}.u$–$\mathsf{indep}\}$

Lemma 30 *The procedure* DisposeTree *is an element of* $\mathsf{Proc}_X.A$ *and*

$$(\forall a, u \bullet \mathsf{tree}.(\mathsf{val}.u, a)\ \{\!|\ \mathsf{DisposeTree}(u)\ |\!\}\ \mathsf{emp} \wedge \mathsf{val}.u \doteq \mathsf{nil}) \qquad (5)$$

Proof. We apply Theorem 17 for body–dt with

- $W = \mathsf{atree}$
- $<$ on W given by $a < b$ iff a is a subtree of b.
- $p_w = (\lambda a \bullet \lambda u \bullet \mathsf{tree}.(\mathsf{val}.u, a) \wedge a < w)$.
- $q = (\lambda a \bullet \lambda u \bullet \mathsf{emp} \wedge \mathsf{val}.u \doteq \mathsf{nil})$

If $\mathsf{Proc}_X.A.P$ then $\mathsf{Proc}_X.A.(\mathsf{body–dt}.P)$ follows from Theorem 29.
For $w \in \mathsf{nat}$, and $P : \mathsf{Proc}_X.A.P$ we assume

$$(\forall u, a, \alpha : \mathsf{set}.u\text{–}\mathsf{indep} \bullet \alpha * \mathsf{tree}.(\mathsf{val}.u, a) \wedge a < w\ \{\!|\ P.u\ |\!\}\ \alpha \wedge \mathsf{val}.u \doteq \mathsf{nil}) \quad (6)$$

and we prove

$$(\forall u, a \bullet \mathsf{tree}.(\mathsf{val}.u, a) \wedge a \doteq w \ \{\![\ \mathsf{body\text{-}dt}.P.u \ ]\!\} \ \mathsf{emp} \wedge \mathsf{val}.u \doteq \mathsf{nil}) \tag{7}$$

By expanding the definition of body-dt, (7) becomes:

$$\begin{array}{l}
\mathsf{tree}.(\mathsf{val}.u, a) \wedge a \doteq w \\
\{\![\\
\quad \mathsf{Add}.t.(\mathsf{val}.u) \ ; \ \mathsf{Add}.x \ ; \\
\quad \mathsf{if} \ \mathsf{val}.t \neq \mathsf{nil} \ \mathsf{then} \\
\qquad x \ := \ \mathsf{val}.t \rightarrow \mathsf{left} \ ; \\
\qquad P.x \ ; \\
\qquad x \ := \ \mathsf{val}.t \rightarrow \mathsf{right} \ ; \\
\qquad P.x \ ; \\
\qquad \mathsf{Dispose}.(\mathsf{val}.t) \ ; \\
\qquad t \ := \ \mathsf{nil} \\
\quad \mathsf{endif} \ ; \\
\quad \mathsf{Del}.x \ ; \ \mathsf{Del}.t.u \\
]\!\} \\
\mathsf{emp} \wedge \mathsf{val}.u \doteq \mathsf{nil}
\end{array} \tag{8}$$

We proved (8) in PVS using (6) and the rules presented in this paper. ∎

9 Conclusions and future work

Based on earlier work on local variables and recursive procedures [1], 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 recursive procedures with value and value–result parameters and local variables. All results were carried out in the theorem prover PVS.

We have also mechanically verified a more complex example [9] of a collection of mutually recursive procedures which build the abstract syntax trees of expressions generated by a LL(1) grammar. In this example we have used the procedure presented in this paper 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. We can also use in programs different datatypes: strings, integers, abstract trees, pointer represented trees.

The program constructs introduced in this paper cover an important subclass of programs that can be written in an imperative programming language like C++. As future work we want to extend the "real world" features of programming languages by for example adding pointer arithmetic. We plan also to introduce the separation implication operator and to investigate more challenging examples.

References

[1] R.J. Back and V. Preoteasa. An algebraic treatment of procedure refinement to support mechanical verification. *Formal Aspects of Computing*, 17:69 – 90, May 2005.

[2] R.J. Back and J. von Wright. *Refinement Calculus. A systematic Introduction*. Springer, 1998.

[3] A. Church. A formulation of the simple theory of types. *J. Symbolic logic*, 5:56–68, 1940.

[4] C.A.R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–580, 1969.

[5] S.S. Ishtiaq and P.W. O'Hearn. Bi as an assertion language for mutable data structures. In *POPL '01: Proceedings of the 28th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 14–26, New York, NY, USA, 2001. ACM Press.

[6] T. Nipkow, L.C. Paulson, and M. Wenzel. *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*, volume 2283 of *LNCS*. Springer, 2002.

[7] P.W. O'Hearn, 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.*, volume 2142 of *Lecture Notes In Computer Science*, pages 1–19, London, UK, 2001. Springer-Verlag.

[8] S. Owre, N. Shankar, J.M. Rushby, and D.W.J. Stringer-Clavert. PVS language reference. Technical report, Computer Science Laboratory, SRI International, dec 2001.

[9] V. Preoteasa. Mechanical verification of mutually recursive procedures for parsing expressions generated by a LL(1) grammar using separation logic. In Preparation. 2006.

[10] J. Reynolds. Intuitionistic reasoning about shared mutable data structure. In *Millenial Perspectives in Computer Science*, 2000.

[11] A. Tarski. A lattice-theoretical fixpoint theorem and its applications. *Pacific J. Math.*, 5:285–309, 1955.

[12] Tjark Weber. Towards mechanized program verification with separation logic. In Jerzy Marcinkowski and Andrzej Tarlecki, editors, *Computer Science Logic – 18th International Workshop, CSL 2004, 13th Annual*

Conference of the EACSL, Karpacz, Poland, September 2004, Proceedings, volume 3210 of *Lecture Notes in Computer Science*, pages 250–264. Springer, September 2004.

[13] H. Yang and 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*, volume 2303 of *Lecture Notes In Computer Science*, pages 402–416, London, UK, 2002. Springer-Verlag.

Lemminkäisenkatu 14 A, 20520 Turku, Finland | www.tucs.fi

University of Turku
- Department of Information Technology
- Department of Mathematics

Åbo Akademi University
- Department of Computer Science
- Institute for Advanced Management Systems Research

Turku School of Economics and Business Administration
- Institute of Information Systems Sciences

ISBN 952-12-1697-2
ISSN 1239-1891