Viorel Preoteasa

Mechanical Verification of Mutually Recursive Procedures for Parsing Expressions using Separation Logic

Turku Centre for Computer Science

TUCS Technical Report
No 771, May 2006

Mechanical Verification of Mutually Recursive Procedures for Parsing Expressions using Separation Logic

Viorel Preoteasa
Åbo Akademi University, Department of Computer Science
Turku Centre for Computer Science
Joukahaisenkatu 3-5, 20520 Turku, Finland

TUCS Technical Report
No 771, May 2006

Abstract

This paper adds support for mutually recursive procedures on top of a predicate transformer semantics of imperative programs with pointers implemented in PVS theorem prover. We define and prove correct a collection of mutually recursive procedures which constructs the parsing tree of an expression generated by a context free grammar. We use separation logic to specify and verify these procedures; the parsing tree is represented in memory using pointers and the specification predicates are defined using separation logic.

TUCS Laboratory
Software Construction

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 [3] has introduced a logic for reasoning about programs with pointers. Based on Burstall's ideas Reynolds [14] describes the separation logic, a more general and abstract logic for reasoning about correctness of pointer programs. This logic combines ideas from [10, 17, 6]. Most approaches of reasoning about pointer programs treat the heap globally, even if programs modify only a small and well defined part of it. O'Hearn [17] has introduced a frame rule in separation logic which has enabled local reasoning about program with pointers.

Mutually recursive procedures are also a very important programming concept which is used for example in programs written in an object oriented language. Reasoning about procedures has been also treated extensively in literature. Nipkow [8] has introduced a Hoare total correctness rule for parameter less mutually recursive procedures.

To be effective, verification of programs with pointers and procedures should have theorem prover support. Weber [16] has introduced a mechanization of separation logic in the theorem prover Isabelle/HOL [9]. He proved soundness and completeness for some Hoare logics extended with heap operations, but his programming language does not contain procedures.

We continue on our earlier work on program variable model and recursive procedures [1] and on a mechanization of separation logic [12]. The contributions of the paper are an abstract Hoare total correctness rule for mutually recursive procedures and the verification of a collection of mutually recursive procedures which build the abstract syntax tree for an arithmetic expression generated by a context free grammar. Our rule for mutually recursive procedures is a generalization of rules from [8, 1, 12] and can be specialized in a rule combining the frame rule [17] and the rule for mutually recursive procedures [8], but allowing procedures with value and result parameters and local variables. We work with a predicate transformer semantics as used in refinement calculus [2] and based on this we define total correctness Hoare triples. We have implemented this theory in the theorem prover PVS [11].

The overview of the paper is as follows. In Section 2 we present the predicate transformer semantics of our language. Section 3 outlines the heap operations and the separation logic which were introduced in [12]. The abstract recursion refinement and total correctness rules are introduced in Section 4. In Section 5 we introduce procedures and we prove a new Hoare total correctness frame rule for mutually recursive procedures with parameters (value and value-result), local variables, and access to global variables. Section 6 introduces mutually recursive procedures to build the abstract syntax tree of an arithmetic expression and outlines their correctness proof. Concluding remarks and future work are presented in Section 7.

2 Preliminaries

We use higher-order logic [4] as the underlying logic. In this section we recall some facts about refinement calculus [2] and about fixed points in complete lattices. We assume that basic facts about complete lattices [5], well founded sets, and well founded induction [7] are known.

2.1 Predicate transformers and refinement

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 predicate transformers, i.e. monotonic functions from Pred to Pred. Programs are modeled as elements of MTran. If $S \in \mathsf{MTran}$ and $p \in \mathsf{Pred}$, then $S.p \in \mathsf{Pred}$ 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].

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 $\alpha \subseteq S.\beta$.

2.2 Program variables, addresses, constants, and expressions

Let value be a nonempty type and let $\mathsf{variable}$, $\mathsf{address}$, $\mathsf{constant} \subseteq \mathsf{value}$ be the types of program variables, program addresses and constants respectively. We assume that $\mathsf{variable}$, $\mathsf{address}$, $\mathsf{constant}$ are pairwise disjoint and nonempty. We take $\mathsf{location} = \mathsf{variable} \cup \mathsf{address}$ and $\mathsf{nil} \in \mathsf{constant}$ an arbitrary element. The element nil represents the null address. Basic programming types like integer numbers, int, are subtypes of $\mathsf{constant}$.

For all $x \in \mathsf{location}$, we introduce the type of x, denoted $\mathsf{T}.x$, as a subtype of value. $\mathsf{T}.x$ represents all values that can be assigned to x. For a type $X \subseteq \mathsf{value}$ we define the subtypes $\mathsf{Vars}.X \subseteq \mathsf{variable}$, $\mathsf{Addrs}.X \subseteq \mathsf{address}$, and

$\mathsf{AddrsNil}.X \subseteq \mathsf{address} \cup \{\mathsf{nil}\}$ by

$$\begin{array}{lcl} \mathsf{Vars}.X & \hat{=} & \{x \in \mathsf{variable} \mid \mathsf{T}.x = X\} \\ \mathsf{Addrs}.X & \hat{=} & \{x \in \mathsf{address} \mid \mathsf{T}.x = X\} \\ \mathsf{AddrsNil}.X & \hat{=} & \mathsf{Addrs}.X \cup \{\mathsf{nil}\} \end{array}$$

For example the program variables of type addresses to integer numbers are defined by $\mathsf{Vars}.(\mathsf{AddrsNil}.\mathsf{int})$.

We access and update program locations using two functions [2, 1].

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

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 and procedure parameters are modeled using four statements that intuitively corresponds to stack operations:

- $\mathsf{Add}.x$ – adds the value of x to the stack
- $\mathsf{Add}.x.e$ – adds the value of x to the stack, and sets x to the value of e.
- $\mathsf{Del}.x$ – deletes the top value from the stack and assigns it to x.
- $\mathsf{Del}.x.y$ – saves the current value of x to y and then deletes the top value from the stack and assigns it to x.

The formal definitions of these program constructs and detailed explanations of their usage in modeling local variables and procedure value and value-result parameters are given in [1].

Program expressions of type A, denoted $\mathsf{E}.A$, are functions from Σ to 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 $\oplus : \mathsf{E}.A \times \mathsf{E}.B \to \mathsf{E}.C$ is defined by $e \oplus e' \hat{=} (\lambda\sigma \bullet e.\sigma \oplus e'.\sigma)$. To avoid confusion, we denote by $(e \doteq e')$ the expression $(\lambda\sigma \bullet e.\sigma = e'.\sigma)$. If $e \in \mathsf{E}.A$, $x \in \mathsf{variable}$, and $e' \in \mathsf{E}.(\mathsf{T}.x)$, then we define $e[x := e'] = (\lambda\sigma \bullet e.(\mathsf{set}.x.(e'.\sigma).\sigma))$, the substitution of e' for x in e. For a parametric boolean expression (predicate) $\alpha : A \to \Sigma \to \mathsf{bool}$, we define the boolean expressions

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

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

3 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 $\mathsf{allocaddr} \mathrel{\hat{=}} \mathcal{P}(\mathsf{address})$, the powerset of $\mathsf{address}$; 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 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.

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

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

Definition 1 *If* $e, f : \mathsf{Pred}$*,* $r : \Sigma \to \mathsf{AddrsNil}.X$*, and* $g : \Sigma \to X$*, then we define*

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

In [12] we proved some properties of this separation logic operators. We recall here two properties that we need in proving the rule for mutually recursive procedures.

Lemma 2 *The following relations hold*

1. $\alpha * \mathsf{emp} = \alpha$
2. $(\bigcup_{i \in I} p_i) * q = \bigcup_{i \in I} (p_i * q)$

In [13] 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 $\mathsf{set.alloc}$–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 [13] 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).

We introduce here only the statement for allocating a new address. All other pointer operations are defined in [12].

Definition 3 *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* $\mathsf{New}.X.(x, e) \in \mathsf{MTran}$ *by*

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

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.

Next lemma introduces the Hoare correctness and frame rule for New.

Lemma 4 *If* $X \subseteq \mathsf{value}$, $x \in \mathsf{Vars}.(\mathsf{AddrsNil}.X)$, $e \in \mathsf{E}.X$ *is* set.alloc*–independent,* set.x*–independent and non–alloc independent, and* $\alpha \in \mathsf{Pred}$ *is* set.x*–independent and non–alloc independent, then*

$$\alpha \, \{\!|\, \mathsf{New}.X.(x, e) \,|\!\}\, \alpha * \mathsf{val}.x \mapsto e$$

3.1 Specifying binary trees with separation logic

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

```
struct btree{
    int label;
    btree *left;
    btree *right}
```
(1)

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 $p, q \in \mathsf{AddrsNil.ptree}.A$. Formally the record structure on ptree.A is given by a bijective function $\mathsf{ptree} : A \times \mathsf{AddrsNil}.(\mathsf{ptree}.A) \times$

$\mathsf{AddrsNil}.(\mathsf{ptree}.A) \rightarrow \mathsf{ptree}.A$. If $a \in A$, and $p, q \in \mathsf{AddrsNil}.\mathsf{ptree}$, then $\mathsf{ptree}.(a, p, q)$ is the record containing the elements a, p, q. The inverse of ptree has three components $(\mathsf{label}, \mathsf{left}, \mathsf{right})$, $\mathsf{label} : \mathsf{ptree}.A \rightarrow A$ and $\mathsf{lef}, \mathsf{right} : \mathsf{ptree}.A \rightarrow \mathsf{AddrsNil}.(\mathsf{ptree}.A)$. The type $\mathsf{ptree}.\mathsf{int}$ corresponds to btree from definition (1) and the type $\mathsf{AddrsNil}.(\mathsf{ptree}.\mathsf{int})$ corresponds to $(\mathsf{btree}\ *)$ from (1).

Let $\mathsf{atreecons}$ 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 $\mathsf{atree} = \mathsf{atreecons} \cup \{\mathsf{nil}\}$. The abstract tree structure on atree is given by an injective function

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

which satisfies the following induction axiom:

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

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

For every $t \in \mathsf{atree}$ and $p \in \mathsf{AddrsNil}.\mathsf{ptree}$ let $\mathsf{tree}.t.p$ be the predicate which is true in those states σ in which address p stores the abstract tree t. The predicate $\mathsf{tree}.t.p$ is defined by structural induction on t.

$$\begin{aligned}
\mathsf{tree}.\mathsf{nil}.p &\ \hat{=}\ p \doteq \mathsf{nil} \wedge \mathsf{emp} \\
\mathsf{tree}.(\mathsf{atree}(a, t, s)).p &\ \hat{=}\ (\exists\, q, r \bullet p \mapsto \mathsf{ptree}(a, q, r) * \mathsf{tree}.t.q * \mathsf{tree}.s.r)
\end{aligned}$$

4 Abstract recursion

In this section $\langle L, \leq \rangle$ denotes a complete lattice. If $f : L \rightarrow L$ is monotonic, then by Knaster–Tarski Theorem [15] f has a least fixpoint denoted $\mu\, f$

Theorem 5 *If $f : L \rightarrow L$ is monotonic and $(x_w,\ w \in W)$ is a family of elements from L indexed by the well-founded set $(W, <)$ then*

$$(\forall w \in W \bullet x_w \leq f(x_{<w})) \Rightarrow x \leq \mu f$$

where $x_{<w} = \bigvee_{v<w} x_v$ and $x = \bigvee_w x_w$.

Proof. By well founded induction on W. ∎

Lemma 6 *If $f : L \rightarrow L$ is monotonic, $L' \subseteq L$ is a sublattice, and $f.L' \subseteq L'$, then $\mu_L\, f = \mu_{L'}\, f$.*

Proof. By using Theorem 19.3, page 321 from [2] ■

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 i–th 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 $(f.x)_i \hat{=} f_i.x_i$.

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 7 *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* $\hat{f}_i.x.a_i = f_i.(\bigvee_{b\in A} x.b)$*, 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 $(\forall a, c \in A \bullet (\mu\, \hat{f}).a = (\mu\, \hat{f}).c)$:

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

We have

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

and

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

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

4.1 The complete lattice of programs

Definition 8 *We call the structure* $\langle L,\ \le,\ \vee,\ \wedge,\ \odot,\ \mathsf{skip}\rangle$ *a* program lattice *if*

- $\langle L,\ \le,\ \vee,\ \wedge\rangle$ *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 9 *The complete lattice of monotonic predicate transformers* $\langle \mathsf{MTran},\ \sqsubseteq,\ \sqcup,\ \sqcap,\ ;\ ,\ \mathsf{skip}\rangle$ *is a lattice of programs.*

Definition 10 *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 11 *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 12 *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 13 *The complete lattice of predicates*

$$\langle \mathsf{Pred}, \subseteq, \cup, \cap, \_ . \_, \{\_\}, \{[\_]\} \rangle$$

is an assertion lattice for MTran*, where*

- $\{p\}.q = p \wedge q$ *and*
- $\{[p]\}.q = (\mathsf{if}\ p \subseteq q\ \mathsf{then\ true\ else\ false\ fi})$

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

Lemma 14 *1.* $(\!| p |\!)_{\odot} (\bigvee_i q_i) = \bigvee_i (\!| p |\!)_{\odot} q_i$

2. $p \leq q \Rightarrow (\!| p |\!) \leq (\!| q |\!)$

We are able to state and prove now the most general recursion refinement rule.

Theorem 15 (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*

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

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

Proof. Using Theorem 5 with $x_w = (\!| p_w |\!) \odot S$. ∎

Theorem 16 *If $S \in L$ and $p \in K$ then*

1. $p \; \{\!| \, S \, |\!\} \; q \Leftrightarrow (\!| p |\!) \odot \| q \| \subseteq S$.

Proof. We prove this relation by proving separately the two implications. Assume $p \; \{\!| \, S \, |\!\} \; q$, then

$$
\begin{array}{ll}
 & (\!| p |\!) \odot \| q \| \\
\leq & \{\text{Lemma 14}\} \\
 & (\!| S.q |\!) \odot \| q \| \\
\leq & \{\text{definition}\} \\
 & S
\end{array}
$$

For the second implication we assume $(\!| p |\!) \odot \| q \| \subseteq S$ and we have:

$$
\begin{array}{ll}
 & p \\
= & \{\text{definition}\} \\
 & \mathsf{skip}_{\odot} p \\
\leq & \{\text{definition}\} \\
 & (\!| \| q \|_{\odot} q |\!)_{\odot} p \\
= & \{\text{definition}\} \\
 & (\!| p |\!)_{\odot} (\| q \|_{\odot} q) \\
= & \{\text{definition}\} \\
 & ((\!| p |\!) \odot \| q \|)_{\odot} q \\
\leq & \{\text{assumption}\} \\
 & S_{\odot} q
\end{array}
$$

∎

4.2 Lifting the program lattice structure to Cartesian product and function type

If L is a program lattice and A is a nonempty 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 $\prod_i L_i$.

Next we introduce a version of the Theorem 15 (Recursion Refinement) which is more suitable to refine mutually recursive programs as we will see when we introduce procedures. 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 $w \in W$, $p_w \in K$ 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

$$
\begin{array}{llllll}
p & \hat{=} & \bigvee_{w \in W} p_w & (p_{<s})_j & \hat{=} & \bigvee_{(v,j) < s} p_{v,j} \\
(q_s)_j & \hat{=} & \bigvee_{(v,j) \leq s} p_{v,j} & q_{<s} & \hat{=} & \bigvee_{t \leq s} q_t \\
q & \hat{=} & \bigvee_{s \in W \times I} q_s & & &
\end{array}
\tag{2}
$$

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

1. $p = q$
2. $q_{<s} = p_{<s}$
3. $s \leq t \Rightarrow p_{<s} \leq p_{<t}$

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

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

Proof.

$$
\begin{array}{ll}
& (\!|p|\!) \odot S \leq \mu\, F \\
= & \{\text{Lemma 17 } (p = q)\} \\
& (\!|q|\!) \odot S \leq \mu\, F
\end{array}
$$

$\Leftarrow$ {Theorem 15 with $W \times I$ and q_s instead of W and p_w}

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

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

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

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

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

$\Leftarrow$ {Lemma 17 and F monotonic}

$(\forall v \in W \bullet \forall j \in I \bullet (\!| p_{v,j} |\!) \odot S_j \leq F_j.(p_{<(v,j)} \odot S))$

■

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

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

Proof.

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

$=$ {Theorem 16}

$(\!| p |\!) \odot \| r \| \subseteq \mu F$

$\Leftarrow$ {Theorem 18}

$\forall w \in W \bullet \forall i \in I \bullet (\!| p_{w,i} |\!) \odot \| r_i \| \leq F_i.((\!| p_{<(w,i)} |\!) \odot \| r \|)$

$=$ {complete lattice properties}

$\forall w \in W \bullet \forall i \in I \bullet \forall S \in L \bullet (\!| p_{<(w,i)} |\!) \odot \| r \| \leq S \Rightarrow (\!| p_{w,i} |\!) \odot \| r_i \| \leq F_i.S$

$=$ {Theorem 16}

$\forall w \in W \bullet \forall i \in I \bullet \forall S \in L \bullet p_{<(w,i)}\ \{\!| S |\!\}\ r \Rightarrow p_{w,i}\ \{\!| F_i.S |\!\}\ r_i$

■

When working with Hoare statements $\alpha\ \{\!| S |\!\}\ \beta$ very often we need specification variables, variables which occur only in α and β but not in S. For detailed discussions of these specification variables see [1]. 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 parameters. Intuitively, the Hoare triple $\alpha\ \{\!| S |\!\}\ \beta$ is true if

$$(\forall a \in A \bullet \alpha.a \leq S.(\beta.a)) \tag{3}$$

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 (3). We cannot however construct an assertion lattice structure on $A \to K$ for L.

Next we extend the Theorem 19 to the case when predicates may refer 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 (2).

Theorem 20 (Hoare mutual recursion and specification variables) *Under the above assumptions if* $r \in K'$ *and* $F : L \to L$ *is monotonic then*

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

Proof. We assume

$$\Big(\forall w \in W,\ \forall i \in I,\ \forall S \in L \bullet p_{<(w,i)} \;\{\!|\, S \,|\!\}\; r \Rightarrow p_{w,i} \;\{\!|\, F_i.S \,|\!\}\; r_i\Big) \qquad (4)$$

and we prove $p \;\{\!|\, \mu\, F \,|\!\}\; r$. We recall the definition of $\hat{F} : L' \to L'$ from Theorem 7, for each $\alpha \in K'$, $a \in A$, $\hat{F}.\alpha.a = F.(\bigvee_{b \in A} \alpha.b)$. From Theorem 7 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

$$\Big(\forall w \in W,\ \forall i \in I,\ \forall S \in L' \bullet p_{<(w,i)} \;\{\!|\, S \,|\!\}\; r \Rightarrow p_{w,i} \;\{\!|\, \hat{F}_i.S \,|\!\}\; r_i\Big) \qquad (5)$$

All we need to prove now is that (4) implies (5). For $w \in W$, $i \in I$, and $S \in L'$ we have the derivation:

$p_{w,i} \;\{\!|\, \hat{F}_i.S \,|\!\}\; r_i$

$\Leftrightarrow$ {Definitions}

$\forall a \in A \bullet p_{w,i}.a \;\{\!|\, \hat{F}_i.S.a \,|\!\}\; r_i.a$

$\Leftrightarrow$ {Definition}

$\forall a \in A \bullet p_{w,i}.a \;\{\!|\, F_i.(\bigvee_{b \in A} S.b) \,|\!\}\; r_i.a$

$\Leftrightarrow$ {Definition}

$p_{w,i} \;\{\!|\, F_i.(\bigvee_{b \in A} S.b) \,|\!\}\; r_i$

$\Leftarrow$ {Assumption (4)}

$p_{<(w,i)} \{\!| \bigvee_{b \in A} S.b |\!\} r$

$\Leftarrow$ {Definitions and complete lattice properties}

$p_{<(w,i)} \{\!| S |\!\} r$

■

The difference between Theorem 19 and Theorem 20 is the fact that in the former we have an assertion lattice for L, but in the later we only have a special case of predicate lattice for L.

5 Mutually recursive procedures and frame rule

In this section we introduce mutually recursive procedures with parameters and local variables and we apply the general results from the previous section to obtain a powerful Hoare total correctness rule for mutually recursive procedures. This rule combines an extension to procedures with parameters of the Hoare rule from [8] with the frame rule for pointer programs [17].

A procedure with parameters from A or simply a procedure over A, is an element from $\mathsf{Proc}.A = A \to \mathsf{MTran}$. 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 programs $P.a$.

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

In [12] we have introduced a recursive procedure for disposing a binary tree from memory $\mathsf{DisposeTree} \in \mathsf{Proc}.(\mathsf{Vars}.(\mathsf{AddrsNil}.\mathsf{ptree}))$. The call $\mathsf{DisposeTree}.u$ disposes the tree stored in program variable u and sets u to nil. We denote by $A = \mathsf{Vars}.(\mathsf{AddrsNil}.\mathsf{ptree})$ the type of $\mathsf{DisposeTree}$ parameters.

The specification of the procedure $\mathsf{DisposeTree}$ is:

$$(\forall a \bullet \mathsf{tree}.u.a \ \{\!| \mathsf{DisposeTree}.u |\!\} \ \mathsf{emp} \wedge u = \mathsf{nil}) \tag{6}$$

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:

$$(\forall a \bullet \alpha * \mathsf{tree}.u.a \ \{\!| \mathsf{DisposeTree}.u |\!\} \ \alpha \wedge u = \mathsf{nil}) \tag{7}$$

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

Let A be a non-empty type of procedure parameters and $X \subseteq A \rightarrow \mathsf{Pred}$ a nonempty type such that X is closed under arbitrary unions, separation conjuction, and $\mathsf{emp} \in X$. 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 $\{\alpha : \mathsf{Vars}.(\mathsf{AddrsNil}.\mathsf{ptree}) \rightarrow \mathsf{Pred} \mid (\forall u \bullet \alpha.u \text{ is } \mathsf{set}.u\text{–independent})\}$. 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 (6) then we can use (7) when proving correctness of programs calling $\mathsf{DisposeTree}$. The definition of $\mathsf{Proc}_X.A$ is a generalization of the concept "local predicate transformers which modifies a set V" of program variables from [17].

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

$$
\begin{aligned}
& (\textstyle\bigsqcup_i P_i) \in \mathsf{Proc}_X.A \\
= \;& \{\text{Definition}\} \\
& (\forall \alpha \in X \bullet \forall q \bullet \alpha * (\textstyle\bigsqcup_i P_i).q \subseteq (\textstyle\bigsqcup_i P_i).(\alpha * q)) \\
= \;& \{\text{Lemma 2}\} \\
& (\forall \alpha \in X \bullet \forall q \bullet \textstyle\bigcup_i (\alpha * P_i.q) \subseteq \textstyle\bigcup_i P_i.(\alpha * q)) \\
\Leftarrow \;& \{\text{Complete lattice properties}\} \\
& (\forall i \in I \bullet \forall \alpha \in X \bullet \forall q \bullet \alpha * P_i.q \subseteq P_i.(\alpha * q)) \\
= \;& \{\text{Definition}\} \\
& (\forall i \in I \bullet P_i \in \mathsf{Proc}_X.A.)
\end{aligned}
$$

For arbitrary intersections we have a similar proof. The facts that $\mathsf{skip} \in \mathsf{Proc}_X.A$ and $\mathsf{Proc}_X.A$ is closed under sequential composition follows directly form the definition of $\mathsf{Proc}_X.A$. ■

Before introducing the correctness rule for mutually recursive procedures we need to define some new concepts and prove some facts about them. We define the *separation assertion statement*, denoted $(\!|p|\!) \in \mathsf{Proc}_X.A$ by

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

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

$$\llbracket p \rrbracket.q = \bigcup\{\alpha \in X \mid p * \alpha \subseteq q\}$$

Theorem 22 *The structure* $\langle A \to \mathsf{Pred}, \subseteq, \wedge, \vee, \_ * \_, \{\!|\_|\!\}, \llbracket\_\rrbracket\rangle$ *is an assertion lattice for* $\mathsf{Proc}_X.A$.

Proof. The facts $\{\!|\_|\!\}$ is an abstract assert statement, and $\{\!|p|\!\} \in \mathsf{Proc}_X.A$ follows from Lemma 2.

We prove that $\llbracket p \rrbracket$ is an element of $\mathsf{Proc}_X.A$, i.e. for all $\alpha \in X$ and $\mathsf{q} : A \to \mathsf{Pred}$, $\alpha * \llbracket p \rrbracket.q \subseteq \llbracket p \rrbracket.(\alpha * q)$. If $X_{p,q} \subseteq X$ given by:

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

then

$$\alpha \in X \wedge \beta \in X_{p,q} \Rightarrow \alpha * \beta \in X_{p,\alpha * q} \tag{8}$$

$$\begin{aligned}
& \alpha * \llbracket p \rrbracket.q \subseteq \llbracket p \rrbracket.(\alpha * q) \\
= \ & \{\text{definition}\} \\
& \alpha * \textstyle\bigcup X_{p,q} \subseteq \bigcup X_{p,\alpha * q} \\
= \ & \{\text{Lemma 2}\} \\
& \textstyle\bigcup_{\beta \in X_{p,q}} \alpha * \beta \subseteq \bigcup X_{p,\alpha * q} \\
\Leftarrow \ & \{\text{complete lattice properties}\} \\
& \forall \beta \in X_{p,q} \bullet \alpha * \beta \subseteq \textstyle\bigcup X_{p,\alpha * q} \\
\Leftarrow \ & \{\text{complete lattice properties}\} \\
& \forall \beta \in X_{p,q} \bullet \alpha * \beta \in X_{p,\alpha * q} \\
= \ & \{\text{relation (8)}\} \\
& \mathit{true}
\end{aligned}$$

The proof of $\{\!|S.p|\!\} \,;\, \llbracket p \rrbracket \sqsubseteq S$ is given by:

$$\begin{aligned}
& (\{\!|S.p|\!\} \,;\, \llbracket p \rrbracket).q \\
= \ & \{\text{definition}\} \\
& (S.p) * \textstyle\bigcup X_{p,q} \\
= \ & \{\text{Lemma 2}\} \\
& \textstyle\bigcup_{\beta \in X_{p,q}} (S.p) * \beta \\
= \ & \{\text{definition of } \mathsf{Proc}_X.A\} \\
& \textstyle\bigcup_{\beta \in X_{p,q}} S.(p * \beta)
\end{aligned}$$

$$\subseteq \{\text{definition of } X_{p,q}\}$$
$$\bigcup_{\beta \in X_{p,q}} S.q$$
$$= \{\text{complete lattice properties}\}$$
$$S.q$$

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

$$(\!|\, \|p\|.p \,|\!).q$$
$$= \{\text{definition}\}$$
$$(\bigcup X_{p,p}) * q$$
$$\geq \{\mathsf{emp} \in X_{p,p}\}$$
$$\mathsf{emp} * q$$
$$= \{\text{Lemma 2}\}$$
$$q$$

■

We can give now the Hoare total correctness rule for mutually recursive procedures. Let W, I 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 specification parameters. For every $i \in I$, $X_i \subseteq (A_i \to \mathsf{Pred})$ such that X_i is closed under arbitrary unions, separation conjuction, and $\mathsf{emp} \in X_i$.

Theorem 23 *If for all $w \in W$ and $i \in I$, $p_{w,i} : B_i \to A_i \to \mathsf{Pred}$, $q_i : B_i \to A_i \to \mathsf{Pred}$ and $body : \prod_i \mathsf{Proc}.A_i \to \prod_i \mathsf{Proc}.A_i$ is monotonic, then the following Hoare rule is true*

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

Proof. This theorem follows from Theorem 20, Lemma 21, Theorem 22, and Lemma 6 ■

6 Parsing an arithmetical expression

In this section we will prove correctness of a collection of recursive procedures which compute the parsing tree of an expression generated by a context free grammar:

We assume that we have a type $\mathsf{string} \subseteq \mathsf{constant}$ of strings with characters from an alphabet $\mathsf{alph} \subseteq \mathsf{string}$. If $X \subseteq \mathsf{alph}$ then $X^* \subseteq \mathsf{string}$ denotes the strings with elements from X. We assume that $\mathsf{nil} \in \mathsf{string}$ is the empty string and we denote by $\cdot$ the string concatenation, $\mathsf{car}.a$ the first character of the string a, $\mathsf{cdr}.a$ 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 ($\mathsf{letter} \subseteq \mathsf{alph}$), special symbols (“$+$”, “$*$”, “(”, “)” $\in \mathsf{alph}$) and non terminal symbols ($\langle E \rangle$, $\langle T \rangle$, $\langle F \rangle$, $\langle L \rangle \in \mathsf{alph}$). We denote by $\mathsf{terminal}$ and $\mathsf{non\text{-}term}$ the types of terminal and non-terminal symbols of the alphabet.

The grammar that generates arithmetic expressions is given by:

$$\begin{array}{lcl} \langle E \rangle & ::= & \langle T \rangle \;\mid\; \langle T \rangle \cdot \text{“}+\text{”} \cdot \langle E \rangle \\ \langle T \rangle & ::= & \langle F \rangle \;\mid\; \langle F \rangle \cdot \text{“}*\text{”} \cdot \langle T \rangle \\ \langle F \rangle & ::= & \langle L \rangle \;\mid\; \text{“(”} \cdot \langle E \rangle \cdot \text{“)”} \\ \langle L \rangle & ::= & \text{“}a\text{”} \;\mid\; \text{“}b\text{”} \;\mid\; \text{“}c\text{”} \;\mid\; \ldots \quad \text{“}a\text{”}, \text{“}b\text{”}, \text{“}c\text{”}, \ldots \in \mathsf{letter} \end{array}$$

with $\langle E \rangle$ the start symbol. We denote by $\mathsf{prod} \subseteq \mathsf{Rel.string}$ the set of these grammar productions.

To define the language generated by this grammar we introduce the one step derivation relation $\Longrightarrow \subseteq \mathsf{string} \times \mathsf{string}$ and the derivation relation $\overset{*}{\Longrightarrow} \subseteq \mathsf{string} \times \mathsf{string}$ given by

$$\begin{array}{ll} a \Longrightarrow b & \hat{=} (\exists (X, c) : \mathsf{prod} \bullet \exists d, e : \mathsf{string} \bullet a = d \cdot X \cdot e \wedge b = d \cdot c \cdot e) \\ a \overset{*}{\Longrightarrow} b & \hat{=} \text{ the reflexive and transitive closure of } \Longrightarrow \end{array}$$

For a nonterminal symbol of the grammar $N \in \mathsf{non\text{-}term}$ we define the language generated by N, $\mathsf{Lang}_N \subseteq \mathsf{terminal}^*$ by

$$\mathsf{Lang}_N \;\hat{=}\; \{a \in \mathsf{terminal}^* \mid N \overset{*}{\Longrightarrow} a\}$$

Lemma 24 $\mathsf{Lang}_F \subseteq \mathsf{Lang}_T \subseteq \mathsf{Lang}_E$.

We define a predicate on strings $\mathsf{paransize} : \mathsf{string} \to \mathsf{int}$ which counts the difference between the number of open parenthesis and the number of close ones.

$$\begin{array}{lll} \mathsf{paransize.nil} & = 0 & \\ \mathsf{paransize.}(\text{“(”} \cdot a) & = \mathsf{paransize}.a + 1 & \\ \mathsf{paransize.}(\text{“)”} \cdot a) & = \mathsf{paransize}.a - 1 & \\ \mathsf{paransize.}(x \cdot a) & = \mathsf{paransize}.a & \text{if } x \in \mathsf{letter} \end{array}$$

Lemma 25 *If* $a \in \mathsf{Lang}_E$ *then* $\mathsf{paransize}.a = 0$ *and* $(\forall b \leq a \bullet \mathsf{paransize}.b \geq 0)$.

Next we introduce the pointer representation of the abstract syntax tree associated to a string generated by the grammar. For all non-terminal symbols $N \in \{\langle E \rangle, \langle T \rangle, \langle F \rangle\}$ and all $t \in \mathsf{AddrsNil.ptree}$, $a : \mathsf{terminal}^*$ we introduce the predicate $\mathsf{tree}_N(t, a) : \mathsf{Pred}$ which is true on those states where

$e \in \mathsf{Lang}_N$ 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.

$$
\begin{array}{lcl}
\mathsf{tree}_E(t, \mathsf{nil}) & \hat{=} & t \doteq \mathsf{nil} \wedge \mathsf{emp} \\
\mathsf{tree}_E(t, a) & \hat{=} & \mathsf{tree}_T(t, a) \vee (\exists\, b, c, t_1, t_2 \bullet a \doteq b \cdot \text{“}+\text{”} \cdot c \wedge \mathsf{tree}_T(t_1, b) \\
 & & \quad * \, \mathsf{tree}_E(t_2, c) * (t \mapsto \mathsf{ptree}.(\text{“}+\text{”}, t_1, t_2))) \\
\mathsf{tree}_T(t, \mathsf{nil}) & \hat{=} & t = \mathsf{nil} \wedge \mathsf{emp} \\
\mathsf{tree}_T(t, a) & \hat{=} & \mathsf{tree}_F(t, a) \vee (\exists\, b, c, t_1, t_2 \bullet a \doteq b \cdot \text{“}*\text{”} \cdot c \wedge \mathsf{tree}_F(t_1, b) \\
 & & \quad * \, \mathsf{tree}_T(t_2, c) * (t \mapsto \mathsf{ptree}.(\text{“}*\text{”}, t_1, t_2))) \\
\mathsf{tree}_F(t, \mathsf{nil}) & \hat{=} & t = \mathsf{nil} \wedge \mathsf{emp} \\
\mathsf{tree}_F(t, a) & \hat{=} & \mathsf{letter}.a \wedge t \mapsto \mathsf{ptree}.(a, \mathsf{nil}, \mathsf{nil}) \\
 & & \quad \vee (\exists\, b \bullet (a \doteq \text{“(”} \cdot b \cdot \text{“)”}) \wedge \mathsf{tree}_E(t, b))
\end{array}
$$

Lemma 26 *For all* $N \in \{\langle E\rangle,\ \langle T\rangle,\ \langle F\rangle\}$, $t \in \mathsf{Addrs.ptree}$, $a \in \mathsf{terminal}^*$, *if* $\mathsf{tree}_N(t, a)$ *then* $\mathsf{Lang}_N.a$

Lemma 27 *For all* $t \in \mathsf{AddrsNil.ptree}$, *and* $e \in \mathsf{string}$, *if* $\mathsf{tree}_E(t, e)$ *then there exists* $f \in \mathsf{atree[alph]}$ *such that* $\mathsf{tree}.t.f$.

For every nonterminal $N \in \mathsf{non\text{-}term}$ we introduce a procedure $\mathsf{parse}_N \in \mathsf{Proc}.A$ where $A = \mathsf{Vars.string} \times \mathsf{Vars.(AddrsNil.ptree)}$. The procedure call $\mathsf{parse}_N.(x, p)$ builds in p the abstract syntax tree of some maximal string a such that $a \leq x$ and $N \stackrel{*}{\Longrightarrow} a$. The procedures parse_E, parse_T, and parse_F are given by the least fixpoint of $\mathsf{body\text{--}parse} : (\mathsf{Proc}.A)^3 \to (\mathsf{Proc}.A)^3$.

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

where

```
    body–parse_E.T.E.(x, p)
=
    Add.(s, t).(val.x, val.p) ; Add.(t1, t2) ;
    T.(s, t1) ;
    if val.t1 ≠ nil ∧ val.s ≠ nil ∧ car.(val.s) ≐ “+” then
        s := cdr.(val.s) ; E.(s, t2) ;
        if val.t2 ≠ nil then
            New(t, ptree(“+”, t1, t2))
        else
            t := val.t1 ; s := “+” · val.s
    else
        t := val.t1
    Del.(t1, t2) ; Del.(s, t).(x, p)
```

$\text{body--parse}_T.F.T.(x, p)$

$=$

$\text{Add}.(s, t).(\text{val}.x, \text{val}.p)$; $\text{Add}.(t_1, t_2)$;
$F.(s, t_1)$;
if $\text{val}.t_1 \not\doteq \text{nil} \wedge \text{val}.s \not\doteq \text{nil} \wedge \text{car}.(\text{val}.s) \doteq$ "$*$" then
  $s := \text{cdr}.(\text{val}.s)$; $T.(s, t_2)$;
  if $\text{val}.t_2 \not\doteq \text{nil}$ then
    $\text{New}(t,\ \text{ptree}(\text{"}*\text{"},\ t_1,\ t_2))$
  else
    $t := \text{val}.t_1$; $s :=$ "$*$" $\cdot\ \text{val}.s$
else
  $t := \text{val}.t_1$
$\text{Del}.(t_1, t_2)$; $\text{Del}.(s, t).(x, p)$

$\text{body--parse}_F.E.(x, p)$

$=$

$\text{Add}.(s, t).(\text{val}.x, \text{val}.p)$; $\text{Add}.r$;
if $\text{val}.s \doteq \text{nil}$ then
  $t := \text{nil}$
else
  if $\text{car}.(\text{val}.s) =$ "(" then
    $r := \text{cdr}.(\text{val}.s)$; $E.(r,\ t)$;
    if($\text{val}.t \not\doteq \text{nil} \wedge \text{val}.r \not\doteq \text{nil} \wedge \text{car}.(\text{val}.r) \doteq$ ")") then
      $s := \text{cdr}.(\text{val}.r)$
    else
      $\text{DisposeTree}(t)$
  else
    if $\text{letter}(\text{car}.(\text{val}.s))$ then
      $\text{New}(t,\ \text{tree}(\text{car}.(\text{val}.s), \text{nil}, \text{nil}))$;
      $s := \text{cdr}.(\text{val}.s)$
    else
      $t := \text{nil}$
$\text{Del}.r$; $\text{Del}.(s, t).(x, p)$

For $N \in \{\langle E\rangle,\ \langle T\rangle,\ \langle F\rangle\}$, $a, b \in \text{string}$, and $t \in \text{AddrsNil.ptree}$ we define the post condition $\text{post}_N(a, b, t) \in \text{Pred}$ for the procedure parse_N by

$$\text{post}_N(a, b, t) = \exists\, d \bullet a \doteq d \cdot b \wedge \text{tree}_N(t, d) \wedge (\forall x \bullet x \leq b \wedge x \not\doteq \text{nil} \Rightarrow \neg \text{Lang}_N.(d \cdot x))$$

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

If x is a list of program variables then we denote by $\mathsf{SepPred}.x$ the predicates which are $\mathsf{set}.x$–independent and non–alloc independent. We assume that $a \in \mathsf{string}$, $u \in \mathsf{Vars.string}$, $v \in \mathsf{Vars}.(\mathsf{AddrsNil.ptree})$, and $\alpha \in \mathsf{SepPred}.(u, v)$. Then the correctness of the parse procedure $N \in \mathsf{non\text{-}term}$ is given by the following Hoare triple.

$$\forall a, v, u, \alpha \bullet \mathsf{val}.u \doteq a \wedge \alpha \; \{\!| \; \mathsf{parse}_N.(u, v) \; |\!\} \; \alpha * \mathsf{post}_N(a, \mathsf{val}.u, \mathsf{val}.v) \tag{9}$$

Let $\leq$ be a binary relation on $W = \mathsf{string}$ given by $a \leq b \Leftrightarrow$ the length of a is smaller than the length 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

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

For every $N \in I$ let

$$\begin{array}{lcl} p_{w,N} & = & (\lambda a \bullet \lambda u, v \bullet \mathsf{val}.u \doteq a \wedge \mathsf{val}.u \doteq w) \\ q_N & = & (\lambda a \bullet \lambda u, v \bullet \mathsf{post}_N(a, \mathsf{val}.u, \mathsf{val}.v)) \\ X_N & = & \{\alpha : A \to \mathsf{Pred} \mid \forall u, v \bullet \alpha.(u, v) \in \mathsf{SepPred}.(u, v)\} \end{array}$$

Using theorem 23 the correctness triples (9) for the parse procedures are true if

$$\begin{array}{l} \quad (\forall a, u, v, \alpha \bullet \alpha \wedge \mathsf{val}.u \doteq a \wedge \mathsf{val}.u \doteq w \\ \qquad \{\!| \; \mathsf{T}.(u, v) \; |\!\} \; \alpha * \mathsf{post}_T(a, \; \mathsf{val}.u, \; \mathsf{val}.v)) \\ \quad \wedge \\ \quad (\forall a, u, v, \alpha \bullet \alpha \wedge \mathsf{val}.u \doteq a \wedge \mathsf{val}.u < w \\ \qquad \{\!| \; \mathsf{E}.(u, v) \; |\!\} \; \alpha * \mathsf{post}_E(a, \; \mathsf{val}.u, \; \mathsf{val}.v)) \\ \Rightarrow \\ \quad (\forall a, u, v \bullet \mathsf{emp} \wedge \mathsf{val}.u \doteq a \wedge \mathsf{val}.u \doteq w \\ \qquad \{\!| \; \mathsf{body\text{–}parse}_E.\mathsf{T}.\mathsf{E}.(u, v) \; |\!\} \; \mathsf{post}_E(a, \; \mathsf{val}.u, \; \mathsf{val}.v)) \end{array} \tag{10}$$

and

$$\begin{array}{l} \quad (\forall a, u, v, \alpha \bullet \alpha \wedge \mathsf{val}.u \doteq a \wedge \mathsf{val}.u \doteq w \\ \qquad \{\!| \; \mathsf{F}.(u, v) \; |\!\} \; \alpha * \mathsf{post}_F(a, \; \mathsf{val}.u, \; \mathsf{val}.v)) \\ \quad \wedge \\ \quad (\forall a, u, v, \alpha \bullet \alpha \wedge \mathsf{val}.u \doteq a \wedge \mathsf{val}.u < w \\ \qquad \{\!| \; \mathsf{T}.(u, v) \; |\!\} \; \alpha * \mathsf{post}_T(a, \; \mathsf{val}.u, \; \mathsf{val}.v)) \\ \Rightarrow \\ \quad (\forall a, u, v \bullet \mathsf{emp} \wedge \mathsf{val}.u \doteq a \wedge \mathsf{val}.u \doteq w \\ \qquad \{\!| \; \mathsf{body\text{–}parse}_T.\mathsf{F}.\mathsf{T}.(u, v) \; |\!\} \; \mathsf{post}_T(a, \; \mathsf{val}.u, \; \mathsf{val}.v)) \end{array}$$

and

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

$$\Rightarrow$$

$$(\forall a, u, v \bullet \mathsf{emp} \wedge \mathsf{val}.u \doteq a \wedge \mathsf{val}.u \doteq w$$
$$\{\!|\ \mathsf{body{-}parse}_F.\mathsf{E}.(u,v)\ |\!\}\ \mathsf{post}_F(a,\ \mathsf{val}.u,\ \mathsf{val}.v))$$

If we would use in this case a straightforward generalization of the rule for single recursive procedures (something derived directly from Theorem 15), then in (10) the first hypothesis would been

$$(\forall a, u, v, \alpha \bullet \alpha \wedge \mathsf{val}.u \doteq a < w\ \{\!|\ \mathsf{T}.(u,v)\ |\!\}\ \alpha * \mathsf{post}_T(a,\ \mathsf{val}.u,\ \mathsf{val}.v))$$

which is too week to prove the conclusion of (10). This is so because when calling recursively parse_T in parse_E the term $(\mathsf{val}.u)$ which ensures the termination was not decreased yet. Moreover, no variable changes its value in parse_E before calling parse_T, so we cannot define a termination function which would be decreased before calling parse_T.

We will only show the proof for the procedure parse_F. The correctness proofs for procedures parse_E and parse_T can be done similarly. We introduce some results that will be need in the correctness proof of parse_F.

Lemma 28 *Let $a, b, x \in \mathsf{string}$ then*

- *(i)* $a \in \mathsf{Lang}_F \wedge x \neq \mathsf{nil} \Rightarrow a \cdot x \notin \mathsf{Lang}_F$
- *(ii)* $a \in \mathsf{Lang}_E \Rightarrow \forall x \leq a \bullet \text{“(”} \cdot x \notin \mathsf{Lang}_F$
- *(iii)* $a \in \mathsf{Lang}_E \wedge (\forall x \leq b \bullet x \neq \mathsf{nil} \Rightarrow a \cdot x \notin \mathsf{Lang}_E) \wedge \mathsf{car}.b \neq \text{“)”}$
  $\Rightarrow (\forall x \leq b \bullet \text{“(”} \cdot a \cdot x \notin \mathsf{Lang}_F)$

Corollary 29 *The following propositions are true*

- *(i)* $\mathsf{post}_E(a, \text{“)”} \cdot b, t) \wedge t \neq \mathsf{nil} \leq \mathsf{post}_F(\text{“(”} \cdot a, b, t)$
- *(ii)* $\mathsf{post}_E(a, b, t) \wedge (t \doteq \mathsf{nil} \vee \mathsf{car}.b \neq \text{“)”}) \leq (\exists\, u \bullet \mathsf{post}_F(\text{“(”} \cdot a, \text{“(”} \cdot a, \mathsf{nil}) * \mathsf{tree}(u, t))$
- *(iii)* $\mathsf{letter}.a \wedge (t \mapsto \mathsf{ptree}(a, \mathsf{nil}, \mathsf{nil})) \Rightarrow \mathsf{post}_F(a \cdot x, x, t)$

We assume

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

and we prove

$$\mathsf{emp} \wedge \mathsf{val}.u \doteq a \wedge \mathsf{val}.u \doteq w\ \{\!|\ \mathsf{body{-}parse}_F.\mathsf{E}.(u,v)\ |\!\}\ \mathsf{post}_F(a,\ \mathsf{val}.u,\ \mathsf{val}.v) \qquad (11)$$

Be expanding the definition of body–parse$_F$ we have to prove:

```
emp ∧ val.u ≐ a ∧ val.u ≐ w
{|
    Add.(s, t).(val.u, val.v) ; Add.r ;
    if val.s ≐ nil then
        t := nil
    else
        if car.(val.s) ≐ “(” then
            r := cdr.(val.s) ;
            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, ptree(car.(val.s), nil, nil)) ;
                s := cdr.(val.s)
            else
                t := nil
            fi
        fi ;
    fi ;
    Del.r ; Del.(s, t).(u, v)
|}
post_F(a, val.u, val.v)                                                  (12)
```

The proof is give by

1. $\{\mathsf{emp} \wedge \mathsf{val}.u \doteq a \wedge \mathsf{val}.u \doteq w\}$
2. $\mathsf{Add}.(s,t).(\mathsf{val}.u, \mathsf{val}.v)$;
3. $\{\mathsf{emp} \wedge \mathsf{val}.s \doteq a \wedge \mathsf{val}.s \doteq w\}$
4. $\mathsf{Add}.r$;
5. $\{\mathsf{emp} \wedge \mathsf{val}.s \doteq a \wedge \mathsf{val}.s \doteq w\}$
6. if $\mathsf{val}.s \doteq \mathsf{nil}$ then
7. $\quad\{\mathsf{emp} \wedge \mathsf{val}.s \doteq \mathsf{nil}\}$
8. $\quad t := \mathsf{nil}$
9. $\quad\{\mathsf{post}_F(a,\ \mathsf{val}.s,\ \mathsf{val}.t)\}$
10. else

11. $\{\mathsf{emp} \wedge \mathsf{val}.s \doteq a \wedge \mathsf{val}.s \doteq w\}$

12. if $\mathsf{car}.(\mathsf{val}.s) \doteq$ "(" then

13. $\quad\{\mathsf{emp} \wedge \mathsf{val}.s \doteq a \wedge \mathsf{val}.s \not\doteq \mathsf{nil}$
$\qquad \wedge\, \mathsf{val}.s \doteq w \wedge \mathsf{car}.(\mathsf{val}.s) \doteq \text{“(”}\}$

14. $\quad r := \mathsf{cdr}.(\mathsf{val}.s)$;

15. $\quad\{\mathsf{emp} \wedge \mathsf{val}.s \doteq a \wedge \mathsf{car}.(\mathsf{val}.s) \doteq \text{“(”}$
$\qquad \wedge\, \mathsf{val}.r \doteq \mathsf{cdr}.a \wedge \mathsf{val}.r < w\}$

16. $\quad \mathsf{E}.(r, t)$;

17. $\quad\{\mathsf{val}.s \doteq a \wedge \mathsf{car}.(\mathsf{val}.s) \doteq \text{“(”}$
$\qquad \wedge\, \mathsf{post}_E(\mathsf{cdr}.a, \mathsf{val}.r, \mathsf{val}.t)\}$

18. $\quad$ if $\mathsf{val}.t \not\doteq \mathsf{nil} \wedge \mathsf{val}.r \not\doteq \mathsf{nil} \wedge \mathsf{car}.(\mathsf{val}.r) \doteq \text{“)”}$ then

19. $\qquad\{\mathsf{val}.s \doteq a \wedge \mathsf{car}.(\mathsf{val}.s) \doteq \text{“(”}$
$\qquad\quad \wedge\, \mathsf{post}_E(\mathsf{cdr}.a, \mathsf{val}.r, \mathsf{val}.t) \wedge t \not\doteq \mathsf{nil} \wedge \mathsf{car}.(\mathsf{val}.r) \doteq \text{“)”}\}$

20. $\qquad\{\mathsf{post}_F(a,\ \mathsf{cdr}.(\mathsf{val}.r),\ \mathsf{val}.t)\}$

21. $\qquad s := \mathsf{cdr}.(\mathsf{val}.r)$

22. $\qquad\{\mathsf{post}_F(a,\ \mathsf{val}.s,\ \mathsf{val}.t)\}$

23. $\quad$ else

24. $\qquad\{\mathsf{val}.s \doteq a \wedge \mathsf{car}.(\mathsf{val}.s) \doteq \text{“(”}$
$\qquad\quad \wedge\, \mathsf{post}_E(\mathsf{cdr}.a, \mathsf{val}.r, \mathsf{val}.t) \wedge (t \doteq \mathsf{nil} \vee \mathsf{car}.(\mathsf{val}.r) \not\doteq \text{“)”})\}$

25. $\qquad\{\exists\, u \bullet \mathsf{post}_F(a,\ \mathsf{val}.s,\ \mathsf{nil}) * \mathsf{tree}.(\mathsf{val}.t).u\}$

26. $\qquad\{\mathsf{post}_F(a,\ \mathsf{val}.s,\ \mathsf{nil}) * \mathsf{tree}.(\mathsf{val}.t).u\}$

27. $\qquad \mathsf{DisposeTree}.t$

28. $\qquad\{\mathsf{post}_F(a,\ \mathsf{val}.s,\ \mathsf{nil}) \wedge \mathsf{val}.t \doteq \mathsf{nil}\}$

29. $\qquad\{\mathsf{post}_F(a,\ \mathsf{val}.s,\ \mathsf{val}.t)\}$

30. $\quad$ fi

31. $\quad\{\mathsf{post}_F(a,\ \mathsf{val}.s,\ \mathsf{val}.t)\}$

32. else

33. $\quad\{\mathsf{emp} \wedge \mathsf{val}.s \doteq a \wedge \mathsf{val}.s \doteq w \wedge \mathsf{car}.(\mathsf{val}.s) \not\doteq \text{“(”}\}$

34. $\quad$ if $\mathsf{letter}.(\mathsf{car}.(\mathsf{val}.s))$ then

35. $\qquad\{\mathsf{emp} \wedge \mathsf{val}.s \doteq a \wedge \mathsf{val}.s \doteq w$
$\qquad\quad \wedge\, \mathsf{car}.(\mathsf{val}.s) \not\doteq \text{“(”} \wedge \mathsf{letter}.(\mathsf{car}.(\mathsf{val}.s))\}$

36. $\qquad\{\mathsf{emp} \wedge \mathsf{val}.s \doteq a \wedge \mathsf{letter}.(\mathsf{car}.(\mathsf{val}.s))$

$\qquad\qquad \mathsf{New}(t, \mathsf{ptree}(\mathsf{car}.(\mathsf{val}.s), \mathsf{nil}, \mathsf{nil}))$;

$\qquad\qquad \{\mathsf{val}.s \doteq a \wedge \mathsf{letter}.(\mathsf{car}.(\mathsf{val}.s))$

$\qquad\qquad\quad \wedge (\mathsf{val}.t \mapsto \mathsf{ptree}(\mathsf{car}.(\mathsf{val}.s), \mathsf{nil}, \mathsf{nil}))$

$\qquad\qquad \{\mathsf{post}_F(a,\ \mathsf{cdr}.(\mathsf{val}.s),\ \mathsf{val}.t)\}$

$\qquad\qquad s := \mathsf{cdr}.(\mathsf{val}.s)$

$\qquad\qquad \{\mathsf{post}_F(a,\ \mathsf{val}.s,\ \mathsf{val}.t)\}$

$\qquad$ else

$\qquad\qquad \{\mathsf{emp} \wedge \mathsf{val}.s \doteq a \wedge \mathsf{val}.s \doteq w$

$\qquad\qquad\quad \wedge\ \mathsf{car}.(\mathsf{val}.s) \neq \text{“(”} \wedge \neg \mathsf{letter}.(\mathsf{car}.(\mathsf{val}.s))\}$

$\qquad\qquad t := \mathsf{nil}$

$\qquad\qquad \{\mathsf{post}_F(a,\ \mathsf{val}.s,\ \mathsf{val}.t)\}$

$\qquad$ fi

$\qquad \{\mathsf{post}_F(a,\ \mathsf{val}.s,\ \mathsf{val}.t)\}$

$\quad$ fi ;

$\quad \{\mathsf{post}_F(a,\ \mathsf{val}.s,\ \mathsf{val}.t)\}$

fi ;

$\{\mathsf{post}_F(a,\ \mathsf{val}.s,\ \mathsf{val}.t)\}$

$\mathsf{Del}.r$;

$\{\mathsf{post}_F(a,\ \mathsf{val}.s,\ \mathsf{val}.t)\}$

$\mathsf{Del}.(s, t).(u, v)$

$\{\mathsf{post}_F(a,\ \mathsf{val}.u,\ \mathsf{val}.v)\}$

7 Conclusions, future work

We have introduced abstract recursion refinement and Hoare total correctness rules. Using the abstract recursion Hoare rule we have proved a Hoare total correctness frame rule for mutually recursive procedures manipulating pointers. Our procedures can have value and value-result parameters, local variables and access to global variables.

We have also proved correctness of a nontrivial example of mutually recursive procedures which build the abstract syntax tree of an expression generated by a context free grammar.

Our theory was implemented in the PVS theorem prover.

The program variables we use can have types of any cardinal up to an arbitrary fixed cardinal γ. The cardinal of all programs is strictly greater than γ which prevents us from having higher order procedures. In future work we intent to show how we can overcome this problem.

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] R. M. Burstall. Some techniques for proving correctness of programs which alter data structures. *Machine Intelligence*, 7:23–50, 1972.

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

[5] B.A. Davey and H.A. Priestley. *Introduction to lattices and order*. Cambridge University Press, New York, second edition, 2002.

[6] S.S. Ishtiaq and 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*, pages 14–26. ACM Press, 2001.

[7] P.T. Johnstone. *Notes on logic and set theory*. Cambridge University Press, New York, NY, USA, 1987.

[8] T. Nipkow. Hoare logics for recursive procedures and unbounded nondeterminism. In J. Bradfield, editor, *Computer Science Logic (CSL 2002)*, volume 2471 of *LNCS*, pages 103–119. Springer, 2002.

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

[10] P. O'Hearn, J. Reynolds, and H. Yang. Local reasoning about programs that alter data structures. In *Computer science logic (Paris, 2001)*, volume 2142 of *Lecture Notes in Comput. Sci.*, pages 1–19. Springer, Berlin, 2001.

[11] 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.

[12] V. Preoteasa. Mechanical verification of recursive procedures manipulating pointers using separation logic. In *Formal Methods '06*. Springer-Verlag, August 2006.

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

[14] J. Reynolds. Separation logic: A logic for shared mutable data structures. In *17th Annual IEEE Symposium on Logic in Computer Science*. IEEE, July 2002.

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

[16] T. 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.

[17] 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.

TURKU

CENTRE *for*

COMPUTER

SCIENCE

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