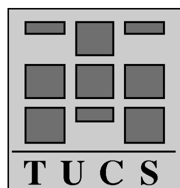
Reasoning about Recursive Procedures with Parameters

Ralph-Johan Back
Viorel Preoteasa



Turku Centre for Computer Science
TUCS Technical Report No 500
January 2003

ISBN 952-12-1104-0
ISSN 1239-1891

Abstract

In this paper we extend the model of program variables from the Refinement Calculus [1] in order to be able to reason more algebraically about recursive procedures with parameters and local variables. We extend the meaning of variable substitution or freeness from the syntax to the semantics of program expressions. We give a predicate transformer semantics to recursive procedures with parameters and prove a refinement rule for introduction of recursive procedure calls. We also prove a Hoare total correctness rule for our recursive procedures. These rules have no side conditions and are easier to apply to programs than the ones in the literature. The theory is built having in mind mechanical verification support using theorem provers like PVS [18] or HOL [10].

Keywords: Refinement, Recursive procedures, Semantics, Hoare rules

TUCS Laboratory
Software Construction Laboratory

1 Introduction

When giving a semantics for an imperative programming language, suitable for mechanical verification, we should deal with the fact that program variables have different types. Many computational models [13, 20, 21, 16, 3, 9], some used in mechanical verification, are based on states represented as tuples, with one component for each program variable. When we access or update a specific program variable we should access or update the corresponding component in the tuple. The problem becomes even more complicated when we have local variables. Then we should add or delete components to the tuple. This extra calculus makes the reasoning about the correctness of a program more complicated.

A more intuitive approach is given in [1], where the state is no longer given as a tuple. Instead, two functions, $\mathsf{val}.x$ and $\mathsf{set}.x$, are introduced for each program variable x. The function $\mathsf{val}.x$ is defined from states to the type of x and $\mathsf{val}.x.\sigma$ is the value of the program variable x in the state σ. The function $\mathsf{set}.x.a$ is defined from states to states and sets the program variable x to a in a given state. These functions should satisfy some behavioral axioms, for example, one axiom says that changing the value of one program variable leaves the rest of the program variables unchanged. All program constructs that deal with program variables are defined using set and val. A drawback of this approach is that, as in the case of tuples or frames, the introduction of new program variables is done by changing the state space.

We propose a cleaner solution which handles the introduction of local variables without changing the state space. We replace the way local variables are introduced in [1]. Our main goal is to be able to reason about recursive procedures so that at any recursive call the procedure parameters are saved (in a stack) and the procedures work with these parameters as if they were new. More generally, we want to be able to save any program variable x at some point during the program execution, work with x as if it were new, and then restore the old value of x. We want to do this as many times as we need.

Last but not least we want to avoid explicitly dealing with stack-like structures in our calculus. We also want to avoid any additional calculus (for tuples or frames) except for the predicate transformers and program expressions one. We only want to have program constructs that satisfy some specific desired properties and give us enough power to reason about recursive procedures with parameters and local variables, without using a stack or any additional calculus.

The contribution of the paper is to extend the axiomatic model of program variables from [1] with one additional program construct $\mathsf{del}.x.\sigma$, which

deletes the program variable x from the state σ. We give a predicate transformer semantics [6, 7] for mutually recursive procedures with value and value–result parameters and local variables. We also introduce a refinement rule and a Hoare [14] total correctness rule for these procedures. These rules have no side conditions and are much easier to apply than the ones in the literature.

The overview of the paper is as follows. We discuss related work in Section 2. Section 3 contains some basic definitions and facts about the Refinement Calculus. In Section 4, we introduce the primitive functions that we use to manipulate program variables. The behavior of these functions and their interaction is given by a set of axioms. We show that the axioms are consistent, by giving a model in which they are valid. In Section 5 we give a semantic notion of program expressions. We define substitution and freeness in the context of these program expressions. We also prove some facts relating substitution and freeness. Using the primitive functions defined in Section 4 we define in Section 6 some program statements and prove some properties about them. In Section 7 we give a least fix point semantics for recursive procedures with parameters and local variables. We give an example of a correctness proof for a recursive procedure. Section 8 presents a refinement rule for introduction of recursive procedure calls. Based on this rule we prove, in Section 9, a Hoare total correctness rule for recursive procedure calls. Section 10 contains concluding remarks.

2 Related work

Back and von Wright [1] represent a program variable x as a pair of functions (f, g) – f for getting the value of x in a state, and g for setting x to some value in some state. The sentence $\mathsf{var}\ x_1, \ldots, x_n$ indicates that the program variables $x_1, \ldots, x_n$ satisfy certain assumptions. A program refinement using $x_1, \ldots, x_n$ is done under the assumptions $\mathsf{var}\ x_1, \ldots, x_n$. As a consequence, one should know a priori what program variables are needed in order to include them in the assumptions. A solution to this problem would be to start with an infinite set of program variables and then use as many as needed. However, because any program variable has at least one assumption associated with it, we would need to state an infinite number of assumptions, which is impossible in a mechanical verification setting.

In [1], there are some restrictions on the formal and actual parameters of a recursive procedure. If a formal value parameter of a procedure has the same name as an actual reference parameter, then the formal parameter should be renamed. The problem becomes even more complicated when

recursive calls are unfolded. In this case some variables should be renamed, and for this purpose, new program variable names are needed. Moreover, a procedure call first changes the state space (adds local variables), does some computation, and then restores the state space. If there is a recursive call, then this call is made in the new (extended) state space, and the semantics should accommodate this fact.

In [2], Back and von Wright give an improved version of their program variable model [1]. They do not need to change the state space any more when adding or deleting local variables. Their approach is similar to ours, but they use a different set of primitive functions, and their focus is on refining parallel composition of action systems, while our focus is on recursive procedures with parameters.

Staples [19, 20, 21] models the state space as a cartesian product over a set of variables V of the dependent types $\tau(v)$, $v \in V$. When entering local blocks or calling procedures, the set V changes. In order to give a predicate transformer semantics the author needs to define the statements over all possible state spaces. Because the state space changes, the semantics and refinement rules of (recursive) procedures are complicated. Another limitation is that procedures cannot access global variables.

Hesselink [13] gives a predicate transformer semantics for parameterless recursive procedures with local variables and access to global variables. Based on this semantics, a Hoare total correctness rule is proven. Hesselink, similarly to Staples, uses a set (frame) F of program variables, and the state space is the product over F of the types of the program variables. A rich logic that connects (changing of) frames to predicate transformers, is developed in order to give proper semantics to recursive procedures.

Kleymann [15] gives an operational semantics for an imperative deterministic language with recursive procedures. He gives a complete set of Hoare total correctness rules with respect to the operational semantics. His approach is simple, but it is limited to handling procedures without parameters and local variables. Based on the operational semantics the author also gives a predicate transformer semantics. The latter is obtained easily since the state space does not change (there are no local variables), and the language is deterministic.

In [23] von Oheimb also gives an operational semantics for an imperative deterministic language with recursive procedures, but his procedures can have value and result parameters and local variables. He gives then a (relatively) complete set of Hoare partial correctness rules with respect to the operational semantics. The state space has two components, one for global and one for local variables. When procedure calls occur the state space changes. The correctness rule for recursive procedures is very simple and intuitive, but the

rules for procedure parameters and local variables are not. Changing the state space does not add the usual complication that we meet in a predicate transformer semantics [19, 20, 21, 13]. However, in his approach all program variables have the same type, which is an unrealistic assumption.

Gries and Levin [11] give Hoare proof rules for multiple assignments and procedures. The language has one-dimensional arrays and records. Procedures have reference and value parameters and access to global program variables. The rule for procedures is given only for the case when there is no aliasing among the global program variables and reference arguments. The rule is also adapted to result parameters instead of reference parameters. This is possible mainly because of the no-aliasing assumption. Finally, they treat a very special case of aliasing. The paper does not treat recursive procedures. The authors give an axiomatic semantics, so there is no need to talk about the state representation.

Harel, Kozen, and Tiuryn [12] model recursivity by using a program variable of type finite stack. The values can be stored in the last-in-first-out manner. When proving correctness for recursive procedures one needs to reason directly about the stack.

3 Preliminaries

We use higher–order logic [4] as the underlying logic. In this section we recall some facts about the Refinement Calculus hierarchy [1]. We also use some basic facts about complete lattices and fixpoints [5].

3.1 States, functions, predicates, relations

A *state space* is a type Σ of higher–order logic. We call an element $\sigma \in \Sigma$ a *program state* or simply a *state*. A *state transformer* is a function $f : \Sigma \to \Sigma$ that maps states to states. We denote by $\mathsf{Func}.\Sigma$ all state transformers. We use the notation $f.\sigma$ for *function application*, $f\ ;\ g$ for *forward functional composition*, i.e. $(f\ ;\ g).\sigma = g.(f.\sigma)$ and lambda notation for functions $(\lambda x \bullet x + 3)$. We denote the identity function on Σ with id. We have that $\langle \mathsf{Func}.\Sigma,\ ;\ , \mathsf{id}\rangle$ is a monoid.

We denote by bool the boolean algebra with two elements. For a type X, $\mathsf{Pred}.X$ are the *predicates* on X, i.e. the functions from X to bool. We extend pointwise all operations on bool to operations on $\mathsf{Pred}.X$. For example $\cup$ is defined on $p, q \in \mathsf{Pred}.X$ by $(p \cup q).x = p.x \cup q.x$. We have that $\langle \mathsf{Pred}.X, \cup, \cap, \neg, \mathsf{false}, \mathsf{true}\rangle$ is a boolean algebra.

A *state relation* is a binary relation on Σ, i.e. a function of type $\Sigma \to \Sigma \to \mathsf{bool}$ ($\Sigma \to \mathsf{Pred}.\Sigma$). We denote by $\mathsf{Rel}.\Sigma$ all binary relations on Σ. We think of a relation R as a non-deterministic state transformer, transforming a state σ to some state σ' such that $R.\sigma.\sigma'$ holds. We again extend pointwise the operations from $\mathsf{Pred}.\Sigma$ to operations on $\mathsf{Rel}.\Sigma$. We also have that $\langle \mathsf{Rel}.\Sigma, \cup, \cap, \neg, \mathsf{false}, \mathsf{true}\rangle$ is a boolean algebra. We denote by $R\ ;\ R'$ the composition of the relations R and R', i.e. $(R\ ;\ R').\sigma.\sigma'$ is true iff there exists σ'' such that $R.\sigma.\sigma''$ and $R'.\sigma''.\sigma'$ are true. The structure $\langle \mathsf{Rel}.\Sigma,\ ;\ , \mathsf{Id}\rangle$ is a monoid, where Id is the identity relation.

We can map predicates and functions to relations. If $p \in \mathsf{Pred}.\Sigma$ then we define $|p| \in \mathsf{Rel}.\Sigma$ by

$$|p|.\sigma.\sigma' = (p.\sigma) \cap (\sigma = \sigma').$$

The function $(\lambda x \bullet |x|)$ is a lattice homomorphism (preserves $\cup$ and $\cap$, false), but is not a boolean algebra homomorphism. It does not commute with $\neg$ and true. We also have

$$|\mathsf{true}| = \mathsf{Id} \text{ and } |p \cap q| = |p| \cap |q| = |p|\ ;\ |q|.$$

If $f \in \mathsf{Func}.\Sigma$ then we define $|f| \in \mathsf{Rel}.\Sigma$ by:

$$|f|.\sigma.\sigma' = (f.\sigma = \sigma')$$

$(\lambda x \bullet |x|)$ is a monoid homomorphism from $\langle \mathsf{Func}.\Sigma,\ ;\ , \mathsf{id}\rangle$ to $\langle \mathsf{Rel}.\Sigma,\ ;\ , \mathsf{Id}\rangle$.

3.2 Predicate transformers, refinement, monotonicity

A *predicate transformer* is a function that maps predicates over Σ to predicates over Σ. We denote by $\mathsf{PTran}.\Sigma$ the type of predicate transformers $\mathsf{Pred}.\Sigma \to \mathsf{Pred}.\Sigma$. The intuition is that $S \in \mathsf{PTran}.\Sigma$ maps postconditions over Σ to preconditions over Σ. The predicate transformers have a *weakest precondition* interpretation, i.e. $S.q$ is a predicate that characterizes all states from which the execution of S is guaranteed to terminate in a final state where postcondition q holds.

Programs are modeled by monotonic predicate transformers. We denote by $\mathsf{MTran}.\Sigma$ the monotonic predicate transformers on Σ. As in the cases of predicates and relations we extend pointwise the operators from $\mathsf{Pred}.\Sigma$ to $\mathsf{MTran}.\Sigma$. We have that $\langle \mathsf{MTran}.\Sigma, \sqsubseteq, \sqcup, \sqcap\rangle$ is a complete lattice because the monotonic predicate transformers are closed under arbitrary meets and joins.

We call a predicate transformer S *conjunctive* if for all predicates p and q, we have $S.(p \wedge q) = S.p \wedge S.q$

We denote by $S\ ;\ T$ the functional composition of predicate transformers. $\langle \mathsf{MTran}.\Sigma,\ ;\ ,\ \mathsf{skip}\rangle$ is a monoid where skip is the identity predicate transformer (the identity function on $\mathsf{Pred}.\Sigma$).

All the above operations defined on predicate transformers are interpreted as operations on programs.

$S \sqsubseteq T$	– the refinement relation.
$\bigsqcap_{i\in I} S_i$	– the demonic choice.
$\bigsqcup_{i\in I} S_i$	– the angelic choice.
$S\ ;\ T$	– the sequential composition.
skip	– the program that does nothing.

In addition to these program constructs, we give the definition of some others:

$$
\begin{array}{lcll}
\{p\}.q & = & p \cap q & \text{(assertion)} \\
[p].q & = & \neg p \cup q & \text{(assumption or guard)} \\
\{R\}.q & = & (\lambda\sigma \bullet (\exists \sigma' \bullet R.\sigma.\sigma' \wedge q.\sigma')) & \text{(angelic update)} \\
[R].q & = & (\lambda\sigma \bullet (\forall \sigma' \bullet R.\sigma.\sigma' \Rightarrow q.\sigma')) & \text{(demonic update)} \\
[f].q & = & [|f|] = \{|f|\} & \text{(functional update)}
\end{array}
$$

where $p,\ q$ are predicates, R is a state relation, and f is a state function.

Lemma 1 *If p, q are predicates, f is a function, R, Q are relations and S_i, T are predicate transformers then*

$$
\begin{array}{ll}
\{p \wedge q\} = \{p\}\ ;\ \{q\} & [p \wedge q] = [p]\ ;\ [q] \\
\{R\ ;\ Q\} = \{R\}\ ;\ \{Q\} & [R\ ;\ Q] = [R]\ ;\ [Q] \\
(\bigsqcap_{i\in I} S_i)\ ;\ T = \bigsqcap_{i\in I}(S_i\ ;\ T) & (\bigsqcup_{i\in I} S_i)\ ;\ T = \bigsqcup_{i\in I}(S_i\ ;\ T) \\
[R].(p \cap q) = [R].p \cap [R].q & \{R\}.(p \cup q) = \{R\}.p \cup \{R\}.q \\
\{p\} = \{|p|\} & [p] = [|p|] \\
\{|f|\} = [|f|] & [\mathsf{Id}] = \mathsf{skip}
\end{array}
$$

The conditional program construct is defined as follows:

$$
\mathsf{if}\ p\ \mathsf{then}\ S\ \mathsf{else}\ T\ \mathsf{fi} = \{p\}\ ;\ S \sqcup \{\neg p\}\ ;\ T
$$

3.3 Least fixpoints

In this subsection we recall some properties about fixpoints in complete lattices.

Theorem 2 (Knaster–Tarski [22]) *Assume that L is a complete lattice and $f : L \to L$ is a monotonic function, then f has a least fixpoint (denoted μf).*

If S is a monotonic predicate transformer and b is a predicate then the function $F : \mathsf{MTran}.\Sigma \to \mathsf{MTran}.\Sigma$ defined by

$$F(X) = \mathsf{if}\ b\ \mathsf{then}\ S\ ;\ X\ \mathsf{else\ skip\ fi}$$

is monotonic. We define the while statement by

$$\mathsf{while}\ b\ \mathsf{do}\ S\ \mathsf{od} = \mu F$$

μF is an element of $\mathsf{MTran}.\Sigma$, so the while construct introduces only monotonic predicate transformers.

The following fixpoint properties will be used when we give the semantics of recursive procedures.

Lemma 3 *Assume that L is a complete lattice and the functions $f, g : L \to L$ are monotonic such that $f \leq g$ (pointwise). Then $\mu f \leq \mu g$.*

Corollary 4 *Assume that L is a complete lattice and the function $f : L \to L \to L$ is monotonic in both arguments. Then $(\lambda x \bullet \mu(f.x))$ is monotonic.*

3.4 Notations

We denote predicates by p, q, functions by f, g, predicate transformers by S, T, and states by σ, σ'. We will use Func, Pred, Rel, PTran and MTran instead of $\mathsf{Func}.\Sigma$, $\mathsf{Pred}.\Sigma$, $\mathsf{Rel}.\Sigma$, $\mathsf{PTran}.\Sigma$ and $\mathsf{MTran}.\Sigma$, since Σ is fixed.

4 Program variables axioms

We denote by Var the type of *program variables*. For any program variable $x \in \mathsf{Var}$, $\Gamma.x$ is its type. Var and $\Gamma.x$ are nonempty types in higher order logic. For a given program variable x, $\mathsf{SameType}.x$ is a predicate on program variables that is true on all variables that have the same type as x.

We denote by nat the type of natural numbers and by NatVar the program variables of type nat. We assume that NatVar have enough elements for our examples.

For any program variable x we introduce the following functions:

$\mathsf{val}.x : \Sigma \to \Gamma.x$	(the *value* of x)
$\mathsf{set}.x : \Gamma.x \to \Sigma \to \Sigma$	(the *update* of x)
$\mathsf{del}.x : \Sigma \to \Sigma$	(the *delete* of local x)

The types of the functions val and set are dependent on the first argument. We could implement them in PVS, for example, using the dependent type mechanism.

If $x \in \mathsf{Var}$, $\sigma \in \Sigma$, and $a \in \Gamma.x$, then $\mathsf{val}.x.\sigma$ is the current value of program variable x in state σ, and $\mathsf{set}.x.a.\sigma$ is the state obtained from σ by setting variable x to value a in state σ.

To be able to explain $\mathsf{del}.x$ we need to describe informally the state space Σ. All elements $\sigma \in \Sigma$ are tuples with one component for each program variable x. The component corresponding to x in σ is an infinite list with elements from $\Gamma.x$. Now $\mathsf{del}.x.\sigma$ is the state σ' in which we have removed the head of the list corresponding to x.

The functions $\mathsf{val}.x$ and $\mathsf{set}.x$ are the same as the ones defined in [1]. We assume that val, set and del satisfy the following axioms:

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

The axioms (a) – (e) are the same as the ones in [1].

Axiom (f) says that any state can be reached by deleting the program variable x from some state. Axiom (g) states that if x and y are distinct program variables, then the value of y remains unchanged after deleting x. Axiom (h) states that deleting x after setting it to some value is the same as deleting x directly. Finally axiom (i) says that it is not important in which order we set x to some value and delete y if x and y distinct program variables.

4.1 A model

In order to show that the axioms are consistent, we will give a model in which they are satisfied. This model is also useful for a better understanding of the meaning of val, set and del. Let

$$\prod \{x \in \mathsf{Var} \mid (\Gamma.x)^{\omega}\}$$

Any component $s \in (\Gamma.x)^\omega$ of a state $\sigma \in \Sigma$ acts as a stack in which the program variable x is stored. The first (top) element of s is the current value of x, the value returned by $\mathsf{val}.x.\sigma$. The function $\mathsf{set}.x.a$ changes the top value of s to a and $\mathsf{del}.x$ removes the top element of s.

If $\sigma \in \Sigma$, then we write it as $(a_i^x)_{i\in\omega,\, x\in\mathsf{Var}}$ or just $(a_i^x)_{i,\,x}$, $a_i^x \in \Gamma.x$ Formally we define val, set and del as:

$$\begin{array}{lcl} \mathsf{val}.y.\sigma & = & a_0^y \\ \mathsf{del}.y.\sigma & = & (b_i^x)_{i,\,x},\ \text{where } (\forall i \bullet b_i^y = a_{i+1}^y) \text{ and } (\forall i,\, x \bullet x \neq y \Rightarrow b_i^x = a_i^x) \\ \mathsf{set}.y.a.\sigma & = & (b_i^x)_{i,\,x},\ \text{where } b_0^y = a \text{ and } (\forall i,\, x \bullet i \neq 0 \vee x \neq y \Rightarrow b_i^x = a_i^x) \end{array}$$

With these definitions it is easy to prove that all axioms (a) – (i) are satisfied.

4.2 List of program variables

We often need to have multiple assignments in programs. Procedures can also have more than one parameter. In order to define such program constructs we need to introduce lists of program variables.

We denote by $\mathsf{VarList}$ the set Var^*, of all finite lists with elements from Var. We use the same notation $x,\ y,\ z,\ \ldots$ to denotes both program variables and lists of program variables. For $x,\ y \in \mathsf{VarList}$ we denote by $x \cdot y$ the concatenation of the two lists and by ϵ the empty list. We consider $\mathsf{Var} \subseteq \mathsf{VarList}$, and we denote by $(x,\ y,\ z,\ \ldots)$ the list with the elements $x,\ y,\ z,\ \ldots \in \mathsf{Var}$.

If $\langle A^*,\ \cdot\rangle$ is monoid of lists over A with $\cdot$ the concatenation of lists then we extend the concatenation to subsets $B,\ C$ of A by $B{\cdot}C = \{b{\cdot}c \mid b \in B \wedge c \in C\}$

For $x \in \mathsf{VarList}$ we define $\Gamma.x \subseteq (\bigoplus\{y \in \mathsf{Var} \mid \Gamma.y\})^*$ the type of the list of program variables x by induction on x, where $\bigoplus$ denotes the disjoint union. If $y \in \mathsf{Var}$ and $z \in \mathsf{VarList}$, then

$$\begin{array}{ll} \Gamma.\epsilon & = \{\epsilon\} \\ \Gamma.(y \cdot z) & = \Gamma.y \cdot \Gamma.z \end{array}$$

We also extend the functions val, set and del to lists of program variables.

$$\begin{array}{lcllcl} \mathsf{val}.\epsilon.\sigma & = & \epsilon, & \mathsf{val}.(y \cdot z).\sigma & = & \mathsf{val}.y.\sigma \cdot \mathsf{val}.z.\sigma \\ \mathsf{del}.\epsilon & = & \mathsf{id}, & \mathsf{del}.(y \cdot z) & = & \mathsf{del}.y\ ;\ \mathsf{del}.z \\ \mathsf{set}.\epsilon.\epsilon & = & \mathsf{id}, & \mathsf{set}.(y \cdot z).(a \cdot b) & = & \mathsf{set}.y.a\ ;\ \mathsf{set}.z.b \end{array}$$

Lemma 5 *If $x,\, y \in \mathsf{VarList}$, $a \in \Gamma.x$, and $b \in \Gamma.y$ then*

(i) $\mathsf{del}.(x \cdot y) = \mathsf{del}.x\ ;\ \mathsf{del}.y$

(ii) $\mathsf{set}.(x \cdot y).(a \cdot b) = \mathsf{set}.x.a\ ;\ \mathsf{set}.y.b$

(iii) $\mathsf{val}.(x \cdot y).s = \mathsf{val}.x.s \cdot \mathsf{val}.y.s$

We usually need lists of program variables in which each program variable occurs at most once. The predicate $\mathsf{var}.x$ is true if and only if all variables from x are distinct. We denote by $x \cap y$ the set of program variables that occur in both x and y

Lemma 6 *If* $\mathsf{var}.(x \cdot y)$ *then* $\mathsf{var}.x$*,* $\mathsf{var}.y$*, and* $x \cap y = \emptyset$*.*

The axioms of the program variables can be easily generalized to properties about lists of program variables.

Lemma 7 *If* x*,* $y \in \mathsf{VarList}$*,* $\sigma \in \Sigma$ *and* a*,* b *are of appropriate types, then*

(a) $\mathsf{var}.x \Rightarrow \mathsf{val}.x.(\mathsf{set}.x.a.\sigma) = a$

(b) $x \cap y = \emptyset \Rightarrow \mathsf{val}.y.(\mathsf{set}.x.a.\sigma) = \mathsf{val}.y.\sigma$

(c) $\mathsf{set}.x.a \,;\, \mathsf{set}.x.b = \mathsf{set}.x.b$

(d) $x \cap y = \emptyset \Rightarrow \mathsf{set}.x.a \,;\, \mathsf{set}.y.b = \mathsf{set}.y.b \,;\, \mathsf{set}.x.a$

(e) $\mathsf{set}.x.(\mathsf{val}.x.\sigma).\sigma = \sigma$

(f) $\mathsf{del}.x$ is surjective

(g) $x \cap y = \emptyset \Rightarrow \mathsf{del}.x \,;\, \mathsf{val}.y = \mathsf{val}.y$

(h) $\mathsf{set}.x.a \,;\, \mathsf{del}.x = \mathsf{del}.x$

(i) $x \cap y = \emptyset \Rightarrow \mathsf{set}.x.a \,;\, \mathsf{del}.y = \mathsf{del}.y \,;\, \mathsf{set}.x.a$

Proof. The proof can be done by induction on x and y and is based on the fact that val, del, set commute for distinct program variables. ■

Unless specified, we will use x, y, z to denote lists of program variables.

5 Program expressions

A *syntactic program expression* is made of program variables and constants using some operators. So if x and y are program variables then $x + 2 \cdot y$ is a syntactic program expression.

The value of a program expression e in a state σ, denoted $\mathsf{val}.e.\sigma$, is defined by structural induction on e:

- if e is a variable x then $\mathsf{val}.e.\sigma = \mathsf{val}.x.\sigma$
- if e is a constant c then $\mathsf{val}.e.\sigma = c$
- if e is $op(e_1, \ldots, e_n)$ for some operator op and some expressions e_i then $\mathsf{val}.e.\sigma = op(\mathsf{val}.e_1.\sigma, \ldots, \mathsf{val}.e_n)$

The semantics of a program expression of type A is a function from Σ to A. In general, we will consider a *program expression* of some type A as being any function from Σ to A. An expression of type bool is called *boolean program expression*. The boolean expressions coincide with the predicates on Σ. We denote by NatExp the type of *natural program expressions*, i.e. $\Sigma \to \mathsf{nat}$.

If $e_1,\ e_2,\ \ldots, e_n$ are program expressions of types $A_1,\ A_2,\ \ldots, A_n$, respectively, then we denote by $(e_1,\ e_2,\ \ldots, e_n)$ the program expression

$$(\lambda\sigma \bullet (e_1.\sigma,\ e_2.\sigma,\ \ldots, e_n.\sigma))$$

of type $A_1 \cdot A_2 \cdot \ldots \cdot A_n$.

Lemma 8 *If* $S, T \in \mathsf{MTran}$ *and* $e : \Sigma \to A$ *is a program expression where* $A \neq \emptyset$ *then*

(*i*) $\bigsqcup_{a\in A}\{e = a\} = \mathsf{skip}$

(*ii*) $(\forall a \in A \bullet \{e = a\}\ ;\ S \sqsubseteq \{e = a\}\ ;\ T) \Leftrightarrow S \sqsubseteq T$

(*iii*) $(\forall a \in A \bullet \{e = a\}\ ;\ S \sqsubseteq T) \Leftrightarrow S \sqsubseteq T$

Proof. We prove (*i*). Let $q \in \mathsf{Pred}$ and $\sigma \in \Sigma$, then

$(\bigsqcup_{a\in A}\{e = a\}).q.\sigma$

$=$ {Definitions}

$\bigvee_{a\in A}(\{e = a\}.q.\sigma)$

$=$ {Definition of assume}

$\bigvee_{a\in A}(e.\sigma = a \wedge q.\sigma)$

$=$ {Distributivity}

$q.\sigma \wedge \bigvee_{a\in A}(e.\sigma = a)$

$=$ $\{A \neq \emptyset\}$

$q.\sigma$

Therefore $\bigsqcup_{a\in A}\{e = a\} = \mathsf{skip}$

The relations (*ii*) and (*iii*) can be proved using the first one and the fact that the sequential composition of programs distributes over any arbitrary nonempty join at left (Lemma 1). ■

Next we define the fact that a variable does not occur free in an expression and the substitution of an expression e' for a variable x in an expression e.

5.1 Substitution

Let $e : \Sigma \to A$, $x \in \mathsf{VarList}$, and $e' : \Sigma \to \Gamma.x$. We define $e[x := e'] : \Sigma \to A$, the *substitution of e' for x in e* as:

$$e[x := e'].\sigma = e.(\mathsf{set}.x.(e'.\sigma).\sigma)$$

Lemma 9 *If If $x, y \in \mathsf{VarList}$ then*

$$(i) \qquad \mathsf{var}.x \Rightarrow (\mathsf{val}.x)[x := e] = e$$

$$(ii) \qquad op(e_1,\ e_2)[x := e] = op(e_1[x := e],\ e_2[x := e])$$

$$(iii) \qquad x \cap y = \emptyset \Rightarrow (\mathsf{val}.y)[x := e] = \mathsf{val}.y$$

5.2 Freeness

Next we introduce the notion of x-freeness for an expression $e : \Sigma \to A$, as the semantic corespondent of the syntactic condition that the program variables x do not occur free in e.

Definition 10 *Let $e : \Sigma \to A$ be an expression, $x \in \mathsf{VarList}$ and $f : \Sigma \to \Sigma$ be a function. We say that e is f-*free *if $e = f\ ;\ e$. We say that e is x-*free *if*

$$(\forall f : \Sigma \to \Sigma \bullet (\forall y \in \mathsf{Var} \bullet y \cap x = \emptyset \Rightarrow \mathsf{val}.y = f\ ;\ \mathsf{val}.y) \Rightarrow e \textit{ is } f\textit{-free}).$$

*We say that e is $\mathsf{set}.x$-*free *if e is $\mathsf{set}.x.a$-free for all $a \in \Gamma.x$.*

The idea behind the definition of x-free is that if a function $f : \Sigma \to \Sigma$ does not change any program variable y (except possibly x), then f does not change the value of e either.

Lemma 11 *If e is x-free then e is $\mathsf{del}.x$-free and $\mathsf{set}.x$-free.*

Lemma 12 *If e is a syntactic program expression that does not contains x free then $\mathsf{val}.e$ is x-free.*

Proof. Assume that e does not contains x free. We prove that $\mathsf{val}.e$ is x-free by structural induction on e. Let $f : \Sigma \to \Sigma$ such that $(\forall y \bullet x \cap y = \emptyset \Rightarrow f\ ;\ \mathsf{val}.y = \mathsf{val}.y)$.

If e is a constant $a \in A$, then $\mathsf{val}.e.\sigma = a = \mathsf{val}.e.(f.\sigma) = (f\ ;\ \mathsf{val}.e).\sigma$.

If e is a variable y such that $y \cap x = \emptyset$ then

$$\mathsf{val}.e.\sigma = \mathsf{val}.y.\sigma = (f\ ;\ \mathsf{val}.y).\sigma = (f\ ;\ \mathsf{val}.e).\sigma.$$

If e is $op(e_1,\ e_2)$ and $\mathsf{val}.e_1$, $\mathsf{val}.e_2$ are x-free then

$$\begin{aligned} \mathsf{val}.e.\sigma &= op(\mathsf{val}.e_1.\sigma,\ \mathsf{val}.e_2.\sigma) = op((f\ ;\ \mathsf{val}.e_1).\sigma,\ (f\ ;\ \mathsf{val}.e_2).\sigma) = \\ &= op(\mathsf{val}.e_1\ ;\ \mathsf{val}.e_2).(f.\sigma) = \mathsf{val}.e.(f.\sigma) = (f\ ;\ \mathsf{val}.e).\sigma. \end{aligned}$$

∎

Lemma 13 *If e is* $\mathsf{set}.x$*-free then* $e = e[x := e']$

Lemma 14 *If e' is x-free (*$\mathsf{del}.x$*-free,* $\mathsf{set}.x$*-free), then* $e[x := e']$ *is x-free (*$\mathsf{del}.x$*-free,* $\mathsf{set}.x$*-free).*

5.3 A state equivalence

The way program expressions are defined so far allow them to depend not only on the current values of the program variables but also on their values from the stack. We define a subclass of program expressions that dependent only on the current values of program variables.

Two states σ and σ' are val-*equivalent*, denoted by $\sigma \sim \sigma'$, if for all program variables x, $(\mathsf{val}.x.\sigma = \mathsf{val}.x.\sigma')$. We call a program expression e, val-*determined* if for all σ and σ' we have $\sigma \sim \sigma' \Rightarrow e.\sigma = e.\sigma'$.

The following lemma will be used to prove an assignment like rule for a new program construct we will define in the next section.

Lemma 15 *If e is a* val*-determined program expression, then*

$$e.\sigma = e.(\mathsf{set}.x.(\mathsf{val}.x.\sigma).(\mathsf{del}.x.\sigma))$$

Lemma 16 *If e is a syntactic program expression, then the program expression* $\mathsf{val}.e$ *is* val*-determined.*

Lemma 17 *If e is* val*–determined and* $\mathsf{set}.x$*–free then e is x-free.*

Proof. Let f be such that for all $y \in \mathsf{Var}$, $y \cap x = \emptyset \Rightarrow \mathsf{val}.y = f\ ;\ \mathsf{val}.y$. We have to prove that $e = f\ ;\ e$.

$$\begin{aligned}
& (f\ ;\ e).\sigma \\
= & \ \{\text{forward functional composition}\} \\
& e.(f.\sigma) \\
= & \ \{e \text{ is } \mathsf{set}.x\text{–free}\} \\
& e.(\mathsf{set}.x.(\mathsf{val}.x.\sigma).(f.\sigma)) \\
= & \ \{e \text{ is } \mathsf{val}\text{–determined and } \mathsf{set}.x.(\mathsf{val}.x.\sigma).(f.\sigma) \sim \sigma\} \\
& e.\sigma
\end{aligned}$$

■

6 Program statements

In this section we introduce some program constructs and prove some properties about them and their compositions.

The program constructs we introduce in this section are monotonic predicate transformers based on functions or demonic updates. We prove the results about these constructs at the lowest possible level. For example if we prove an equality between programs based on functions and demonic updates, we prove it at the level of relations. Then we use the fact that the demonic update is a monoid homomorphism from relations to monotonic predicate transformers to lift the properties to monotonic predicate transformers.

Let $x \in \mathsf{VarList}$ and e be a program expression of type $\Gamma.x$. Then we recall the definition of $x := e : \Sigma \to \Sigma$ (the multiple assignment) and $(x := x' \,|\, b) : \Sigma \to \Sigma \to \mathsf{bool}$ (the relational assignment) from [1].

$$\begin{aligned} (x := e).\sigma &= \mathsf{set}.x.(e.\sigma).\sigma \\ (x := x' \,|\, b).\sigma.\sigma' &= (\exists a \in \Gamma.x \bullet b[x' := a] \wedge \mathsf{set}.x.a.\sigma = \sigma') \end{aligned}$$

All properties from [1] regarding the assignment statement are true in our framework also.

Definition 18 *If $x \in \mathsf{VarList}$ and $e : \Sigma \to \Gamma.x$, then we define:*

$$\begin{aligned} \mathsf{add}.x.\sigma.\sigma' &= (\sigma = \mathsf{del}.x.\sigma') && \text{(add local variables)} \\ \mathsf{add}.x.e.\sigma.\sigma' &= (\sigma = \mathsf{del}.x.\sigma') \wedge (\mathsf{val}.x.\sigma' = e.\sigma) && \text{(add \& initialize local variables)} \end{aligned}$$

To be able to work with procedures that have result parameters we need to introduce a new delete function that deletes a list of variables from the stack and saves its value into another list of variables:

$$\mathsf{del}.x.y.\sigma = \mathsf{set}.y.(\mathsf{val}.x.\sigma).(\mathsf{del}.x.\sigma) \quad \text{(save \& delete local variables)}$$

where x and y are lists of program variables of the same type.

Next, when there is no confusion, we use the same notation $\mathsf{add}.x$ for both the relation and the demonic update $[\mathsf{add}.x]$. The same is true for $\mathsf{add}.x.e$, $\mathsf{del}.x$, $\mathsf{del}.x.y$ and $x := e$.

Lemma 19 *If $\mathsf{var}.x$, then the relations $\mathsf{add}.x$ and $\mathsf{add}.x.e$ are total, i.e. for each $\sigma \in \Sigma$, there exists $\sigma' \in \Sigma$ such that σ and σ' are related by $\mathsf{add}.x$ (respectively $\mathsf{add}.x.e$).*

Proof. For $\mathsf{add}.x$ this is trivial because $\mathsf{add}.x$ is the inverse of $\mathsf{del}.x$, and $\mathsf{del}.x$ is a surjective function. For $\mathsf{add}.x.e$ we have to prove for all σ that there exists σ_0 such that $\mathsf{add}.x.e.\sigma.\sigma_0$ is true.

$$
\begin{aligned}
& \exists \sigma_0 \bullet \mathsf{add}.x.e.\sigma.\sigma_0 \\
= \ & \{\text{definition of } \mathsf{add}.x.e\} \\
& (\exists \sigma_0 \bullet \sigma = \mathsf{del}.x.\sigma_0 \wedge \mathsf{val}.x.\sigma_0 = e.\sigma) \\
= \ & \{\mathsf{del}.x\text{–surjective}\} \\
& (\exists \sigma_1,\ \sigma_0 \bullet \sigma = \mathsf{del}.x.\sigma_1 \wedge \sigma = \mathsf{del}.x.\sigma_0 \wedge \mathsf{val}.x.\sigma_0 = e.\sigma) \\
\Leftarrow \ & \{\text{introduction of } \exists\} \\
& (\exists \sigma_1 \bullet \sigma = \mathsf{del}.x.\sigma_1 \wedge \sigma = \mathsf{del}.x.(\mathsf{set}.x.(e.\sigma).\sigma_1) \wedge \\
& \qquad \mathsf{val}.x.(\mathsf{set}.x.(e.\sigma).\sigma_1) = e.\sigma) \\
= \ & \{\text{Lemma 7 (a) and (h)}\} \\
& (\exists \sigma_1 \bullet \sigma = \mathsf{del}.x.\sigma_1) \\
= \ & \{\mathsf{del}.x\text{–surjective}\} \\
& \mathit{true}
\end{aligned}
$$

■

Lemma 20 *If* $x \in \mathsf{VarList}$ *then*

(i) $\quad \mathsf{add}.x \ ; \mathsf{del}.x = \mathsf{skip}$

(ii) $\quad \mathsf{var}.x \Rightarrow \mathsf{add}.x.e \ ; \mathsf{del}.x = \mathsf{skip}$

Proof. Using Lemma 19. ■

Lemma 21 *If* $x, x' \in \mathsf{VarList}$ *have the same type,* e *is program expression of type* $\Gamma.x$*, and* b *is a boolean expression such that* $(\forall \sigma \exists a \bullet b[x' := a].\sigma)$*, then*

(i) $\quad x := e \ ; \mathsf{del}.x = \mathsf{del}.x$

(ii) $\quad (x := x' \,|\, b) \ ; \mathsf{del}.x = \mathsf{del}.x$

Proof. We prove only (i). It is enough to prove it at the function level.

$$
\begin{aligned}
& (x := e \ ; \mathsf{del}.x).\sigma \\
= \ & \{\text{forward functional composition}\} \\
& \mathsf{del}.x.((x := e).\sigma) \\
= \ & \{\text{assignment definition}\} \\
& \mathsf{del}.x.(\mathsf{set}.x.(e.\sigma).\sigma) \\
= \ & \{\text{Lemma 7 (h)}\} \\
& \mathsf{del}.x.\sigma
\end{aligned}
$$

■

Lemma 22 *If* $x \cap y = \emptyset$ *and* e *is* $\mathsf{del}.x$*-free then* $y := e \;;\; \mathsf{del}.x = \mathsf{del}.x \;;\; y := e$

Proof. We prove it at the functional level.

$$
\begin{array}{ll}
 & (y := e \;;\; \mathsf{del}.x).\sigma \\
= & \{\text{forward functional composition}\} \\
 & \mathsf{del}.x.((y := e).\sigma) \\
= & \{\text{assignment definition}\} \\
 & \mathsf{del}.x.(\mathsf{set}.y.(e.\sigma).\sigma) \\
= & \{\text{Lemma 7 (i)}\} \\
 & \mathsf{set}.y.(e.\sigma).(\mathsf{del}.x.\sigma) \\
= & \{e \text{ is } \mathsf{del}.x\text{-free}\} \\
 & \mathsf{set}.y.(e.(\mathsf{del}.x.\sigma)).(\mathsf{del}.x.\sigma) \\
= & \{\text{assignment definition}\} \\
 & (y := e).(\mathsf{del}.x.\sigma) \\
= & \{\text{forward functional composition}\} \\
 & (\mathsf{del}.x \;;\; y := e).\sigma
\end{array}
$$

■

Lemma 23 *If* x*,* q*, and* e *are of appropriate types then*

(*i*) q *is* $\mathsf{del}.x$*–free* $\Rightarrow [\mathsf{add}.x].q = q$

(*ii*) $\mathsf{var}.x$ *and* q *is* $\mathsf{del}.x$*–free* $\Rightarrow [\mathsf{add}.x.e].q = q$

(*iii*) $\mathsf{var}.x$ *and* q *is* val*–determined* $\Rightarrow [\mathsf{add}.x.e].q = q[x := e]$

Proof. We prove only (*iii*). The others are similar. For (*iii*) we have to show for all $\sigma \in \Sigma$ that

$$q[x := e].\sigma \Leftrightarrow (\forall \sigma' \bullet \mathsf{add}.x.e.\sigma.\sigma' \Rightarrow q.\sigma') \tag{1}$$

We prove 1 by proving the two implications separately.

First assume $q[x := e].\sigma$ and $\mathsf{add}.x.e.\sigma.\sigma'$ for some $\sigma' \in \Sigma$. Using ate add definition we obtain $\sigma = \mathsf{del}.\sigma'$ and $\mathsf{val}.x.\sigma' = e.\sigma$. We show that $q.\sigma'$ is true.

$$
\begin{array}{ll}
 & q.\sigma' \\
= & \{\text{Lemma 15}\} \\
 & q.(\mathsf{set}.x.(\mathsf{val}.x.\sigma').(\mathsf{del}.x.\sigma')) \\
= & \{\text{assumptions}\}
\end{array}
$$

$$q.(\mathsf{set}.x.(e.\sigma).\sigma)$$
$$= \{\text{substitution definition}\}$$
$$q[x := e].\sigma$$
$$= \{\text{assumptions}\}$$
$$true$$

Assume now that $(\forall \sigma' \bullet \mathsf{add}.x.e.\sigma.\sigma' \Rightarrow q.\sigma')$. We have by Lemma 19 that exists σ_0 such that $\mathsf{add}.x.e.\sigma.\sigma_0$. Using a similar argument as before we can prove that $q[x := e].\sigma$ is true. ∎

Lemma 24 *If* $\mathsf{var}.x$ *and* p *is a* val*-determined boolean expression then*

$$\{p[x := e]\} \ ; \mathsf{add}.x.e = \mathsf{add}.x.e \ ; \{p\}$$

Proof. Let $q \in \mathsf{Pred}$.

$$(\{p[x := e]\} \ ; \mathsf{add}.x.e).q$$
$$= \{\text{definition of sequential composition}\}$$
$$\{p[x := e]\}.([\mathsf{add}.x.e].q)$$
$$= \{\text{assertion definition}\}$$
$$p[x := e] \cap [\mathsf{add}.x.e].q$$
$$= \{\text{Lemma 23}\}$$
$$[\mathsf{add}.x.e].p \cap [\mathsf{add}.x.e].q$$
$$= \{\text{Lemma 1}\}$$
$$[\mathsf{add}.x.e].(p \cap q)$$
$$= \{\text{assertion definition}\}$$
$$[\mathsf{add}.x.e].(\{p\}.q)$$
$$= \{\text{definition of sequential composition}\}$$
$$(\mathsf{add}.x.e \ ; \{p\}).q$$

∎

Lemma 25 *If* p *is a predicate then*

(*i*) $\{\mathsf{del}.x; p\} \ ; \mathsf{del}.x = \mathsf{del}.x \ ; \{p\}$

(*ii*) $\{p\} \ ; \mathsf{add}.x = \mathsf{add}.x \ ; \{\mathsf{del}.x; p\}$

(*iii*) $\{p\} \ ; \mathsf{add}.x.e = \mathsf{add}.x.e \ ; \{\mathsf{del}.x; p\}$

Next lemma, about $\mathsf{del}.x.y$, will be used to get the values of a procedure actual result parameters.

Lemma 26 *If* $x, y \in \mathsf{VarList}$ *such that* $\mathsf{var}.x$*,* e *is a* $\mathsf{del}.x$*-free expression, and* y *has the same type as* x*, then*

$$x := e \ ; \ \mathsf{del}.x.y = \mathsf{del}.x \ ; \ y := e.$$

Proof. We have:

$$\begin{aligned}
& (x := e \ ; \ \mathsf{del}.x.y).\sigma \\
= \ & \{\text{forward functional composition}\} \\
& (\mathsf{del}.x.y).(\mathsf{set}.x.(e.\sigma).\sigma) \\
= \ & \{\text{definition of } \mathsf{del}.x.y\} \\
& \mathsf{set}.y.(\mathsf{val}.x.(\mathsf{set}.x.(e.\sigma).\sigma)).(\mathsf{del}.x.(\mathsf{set}.x.(e.\sigma).\sigma)) \\
= \ & \{\text{Lemma 7 (a) and (h)}\} \\
& \mathsf{set}.y.(e.\sigma).(\mathsf{del}.x.\sigma) \\
= \ & \{e \text{ is } \mathsf{del}.x\text{-free}\} \\
& \mathsf{set}.y.(e.(\mathsf{del}.x.\sigma)).(\mathsf{del}.x.\sigma) \\
= \ & \{\text{assignment definition}\} \\
& (y := e).(\mathsf{del}.x.\sigma) \\
= \ & \{\text{forward functional composition}\} \\
& (\mathsf{del}.x \ ; \ y := e).\sigma
\end{aligned}$$

■

Example 27 *If* $k, n, c, x, y \in \mathsf{NatVar}$ *such that* $\mathsf{var}.(k, n, c, x, y)$ *and* e*,* f*,* g*,* h *are program natural expressions such that* h *is* $\mathsf{del}.(k, n, c, x, y)$*–free, then we have the following derivation for all* $u \in \mathsf{NatVar}$

$$\begin{aligned}
& \mathsf{add}.(k, n, c).e \ ; \ \mathsf{add}.(x, y) \ ; \\
& x \ := \ f \ ; \ y \ := \ g \ ; \ c \ := \ h \ ; \\
& \mathsf{del}.(x, y) \ ; \ \mathsf{del}.(k, n) \ ; \ \mathsf{del}.c.u \\
= \ & \{\text{Lemma 22}\} \\
& \mathsf{add}.(k, n, c).e \ ; \ \mathsf{add}.(x, y) \ ; \\
& x \ := \ f \ ; \ y \ := \ g \ ; \\
& \mathsf{del}.(x, y) \ ; \ \mathsf{del}.(k, n) \ ; \\
& c \ := \ h \ ; \ \mathsf{del}.c.u \\
= \ & \{\text{Lemma 21, Lemma 5 and Lemma 20}\} \\
& \mathsf{add}.(k, n, c).e \ ; \ \mathsf{del}.(k, n) \\
& c \ := \ h \ ; \ \mathsf{del}.c.u \\
= \ & \{\text{Lemma 26, Lemma 5 and Lemma 20}\} \\
& u \ := \ h
\end{aligned}$$

7 Procedures

In this section we show how recursive procedures can be modeled using the program construct add and del. We define procedures that have value and/or result parameters. These procedures could also have local variables.

Definition 28 *We call a* procedure with parameters from A *or simply a* procedure over A *an element from* $A \to \mathsf{MTran}$. *We denote by* $\mathsf{Proc}.A$ *the type of all procedures over* A.

The set A is the range of the procedure's actual parameters. For example, a procedure with a value parameter x and a result parameter y, both of type nat, has $A = \mathsf{NatExp} \times \mathsf{NatVar}$. A call to a procedure $P \in \mathsf{Proc}.A$ with the actual parameter $a \in A$ is the program $P.a$.

We again extend pointwise all operations on programs to procedures over A. We have that the structure $(\mathsf{Proc}.A, \sqsubseteq, \sqcup, \sqcap)$ is a complete lattice and $(\mathsf{Proc}.A, ;, \lambda a \bullet \mathsf{skip})$ is a monoid such that ; commutes over arbitrary meets and joins at right. We call $\sqsubseteq$ the procedure refinement relation, $\sqcap$ the demonic choice, $\sqcup$ the angelic choice, and ; the sequential composition of procedures.

Next we show how we model the call by value and call by result in this setting. We also give semantics for recursive procedures.

7.1 Nonrecursive procedures

A general nonrecursive procedure declaration is:

$$\begin{array}{l} \mathsf{procedure}\ name(\mathsf{val}\ x;\ \mathsf{res}\ y) : \\ \qquad body \end{array} \tag{2}$$

where *body* is a program that does not contain any recursive call. The meaning of this procedure declaration is that *name* is a procedure that has the list x as value parameters and the list y as result parameters. When a call is made to *name* the caller should provide a program expression e of type $\Gamma.x$ and a list of program variable z with $\Gamma.z = \Gamma.y$ as actual parameters. The intuition of the call is that the formal parameters of the procedure get the values given by e and $\mathsf{val}.z$, it executes *body* and then saves the values of the formal parameters y to z.

The procedure declaration (2) is an abbreviation for the following formal definition:

$$name = (\lambda e, z \bullet \mathsf{add}.(x \cdot y).(e \cdot \mathsf{val}.z)\ ;\ body\ ;\ \mathsf{del}.x\ ;\ \mathsf{del}.y.z)$$

The variables e and z are chosen such that they do not occur free in x, y and *body*.

Using this approach we can have local variables as well. We can have any number of local variables added in the body of the procedure. If w are the local variables, then *body* is $\mathsf{add}.w$; $body_0$; $\mathsf{del}.w$.

7.2 Recursive procedures

The semantics of a recursive procedure over A is the least fixpoint of some monotonic function on $\mathsf{Proc}.A$ given by the procedure declaration.

If $body : \mathsf{Proc}.A \to \mathsf{Proc}.A$ is a monotonic function then we define the recursive procedure given by *body* as $\mu\, body$.

We can also give semantics for mutually recursive procedures. It is sufficient to show this for two procedures. If we have two recursive procedures, over A and B respectively, then the bodies of the two procedures have these types:

$$body_1 : (\mathsf{Proc}.A) \times (\mathsf{Proc}.B) \to (\mathsf{Proc}.A)$$

and

$$body_2 : (\mathsf{Proc}.A) \times (\mathsf{Proc}.B) \to (\mathsf{Proc}.B)$$

Where $body_1$ and $body_2$ are monotonic in both arguments. We define the procedures $(p_1,\ p_2) = \mu\, body \in (\mathsf{Proc}.A) \times (\mathsf{Proc}.B)$, where

$$body = (body_1,\ body_2) : \mathsf{Proc}.A \times \mathsf{Proc}.B \to \mathsf{Proc}.A \times \mathsf{Proc}.B.$$

If we use only the program constructs defined in this paper to built the *body* of a procedure then, using Corollary 4, *body* is a monotonic function on $\mathsf{Proc}.A$.

7.3 An example

Next we will show an example of how this definition works in practice. We give a recursive procedure that computes the binomial coefficient

$$\binom{n}{k} = \frac{n!}{k! \cdot (n-k)!}$$

using the recursive formula

$$\binom{n}{k} = \binom{n-1}{k-1} + \binom{n-1}{k}$$

when $0 < k < n$.

If k, n, c, x, y are program variables of type nat such that $\mathsf{var}.(k, n, c, x, y)$ then let comb be the procedure defined by:

```
procedure comb (val k, n; res c):
    local x, y
    if k = 0 ∨ k = n then
        c := 1
    else
        comb(k − 1, n − 1, x) ;
        comb(k, n − 1, y) ;
        c := x + y
    fi
```

We take $A = \mathsf{NatExp} \times \mathsf{NatExp} \times \mathsf{NatVar}$ and define

$$
\begin{array}{ll}
 & \mathit{body}.S.(e, f, u) \\
= & \\
 & \mathsf{add}.(k, n, c).(e, f, \mathsf{val}.u) \ ; \ \mathsf{add}.(x, y) \ ; \\
 & \mathsf{if}\ k = 0 \vee k = n\ \mathsf{then} \\
 & \quad c := 1 \\
 & \mathsf{else} \\
 & \quad S.(k-1,\ n-1,\ x) \ ; \ S.(k,\ n-1,\ y) \ ; \ c := x + y \\
 & \mathsf{fi} \ ; \\
 & \mathsf{del}.(x, y) \ ; \ \mathsf{del}.(k, n) \ ; \ \mathsf{del}.c.u
\end{array}
\tag{3}
$$

Then body is a monotonic function from $\mathsf{Proc}.A$ to $\mathsf{Proc}.A$.

If $\mathsf{comb} = \mu\ \mathit{body}$, then we can prove for all $u \in \mathsf{NatVar}$ and $e, f \in \mathsf{NatExp}$ that

$$
\{e \leq f\} \ ; \ u := \begin{pmatrix} f \\ e \end{pmatrix} = \{e \leq f\} \ ; \ \mathsf{comb}(e, f, u). \tag{4}
$$

To prove (4) it is enough (by Lemma 8) to show

$$
\begin{array}{ll}
 & \{a \leq b \wedge e = a \wedge f = b\} \ ; \ u := \begin{pmatrix} f \\ e \end{pmatrix} \\
= & \\
 & \{a \leq b \wedge e = a \wedge f = b\} \ ; \ \mathsf{comb}(e,\ f,\ u)
\end{array}
\tag{5}
$$

for all $a, b \in \mathsf{nat}$, $u \in \mathsf{NatVar}$ and $f, g : \Sigma \to \mathsf{nat}$. We split the proof of (5) in two cases: $(i) - a > b$ and $(ii) - a \leq b$

The case (i) is trivial because $\{\mathsf{false}\} \ ; \ S = \{\mathsf{false}\}$.

We prove (ii) by induction on b. We have the case $b = 0$ and the case $b > 0$ with (5) true for $b - 1$. We split again the case $b > 0$ in $a = 0 \vee a = b$ and $0 < a < b$. The cases $b = 0$ and $a = 0 \vee a = b$ are similar. We prove the

case $0 < a < b$ assuming that (5) is true for $b-1$ and all $a \in \mathsf{nat}$, $u \in \mathsf{NatVar}$ and $f, g : \Sigma \to \mathsf{nat}$. We have

$$\{a \leq b \wedge e = a \wedge f = b\} \ ; \ \mathsf{comb}(e, f, u)$$

$= \{\text{Assumptions}\}$

$$\{0 < a < b \wedge e = a \wedge f = b\} \ ; \ \mathsf{comb}(e, f, u)$$

$= \{\mathsf{comb} = \mu \ \mathsf{body}\}$

$\{0 < a < b \wedge e = a \wedge f = b\}$;
$\mathsf{add}.(k, n, c).(e, f, \mathsf{val}.u)$; $\mathsf{add}.(x, y)$;
if $k = 0 \vee k = n$ then
  $c := 1$
else
  $\mathsf{comb}(k-1, n-1, x)$; $\mathsf{comb}(k, n-1, y)$; $c := x + y$
fi ;
$\mathsf{del}.(x, y)$; $\mathsf{del}.(k, n)$; $\mathsf{del}.c.u$

$= \{\text{Lemma 24}\}$

$\{0 < a < b \wedge e = a \wedge f = b\}$
$\mathsf{add}.(k, n, c).(e, f, \mathsf{val}.u)$; $\mathsf{add}.(x, y)$;
$\{0 < a < b \wedge k = a \wedge n = b\}$;
if $k = 0 \vee k = n$ then
  $c := 1$
else
  $\mathsf{comb}(k-1, n-1, x)$; $\mathsf{comb}(k, n-1, y)$; $c := x + y$
fi ;
$\mathsf{del}.(x, y)$; $\mathsf{del}.(k, n)$; $\mathsf{del}.c.u$

$= \{\text{by induction hypothesis using refinement [1]}\}$

$\{0 < a < b \wedge e = a \wedge f = b\}$
$\mathsf{add}.(k, n, c).(e, f, \mathsf{val}.u)$; $\mathsf{add}.(x, y)$;
$x := \begin{pmatrix} b-1 \\ a-1 \end{pmatrix}$; $y := \begin{pmatrix} b-1 \\ a \end{pmatrix}$; $c := \begin{pmatrix} b \\ a \end{pmatrix}$;
$\mathsf{del}.(x, y)$; $\mathsf{del}.(k, n)$; $\mathsf{del}.c.u$

$= \{\text{Example 27}\}$

$$\{0 < a < b \wedge e = a \wedge f = b\} \ ; \ u := \begin{pmatrix} b \\ a \end{pmatrix}$$

$= \{\text{Refinement [1]}\}$

$$\{0 < a < b \wedge e = a \wedge f = b\} \ ; \ u := \begin{pmatrix} f \\ e \end{pmatrix}$$

This concludes the proof of (4).

8 Refinement rule for introduction of recursive procedure calls

So far we have defined the refinement of procedures and we gave semantics for recursive procedures. In order to express properties about procedures over A, we now need to lift the predicates to predicates that depends on A.

We call the type $A \to \mathsf{Pred}$ the *parametric predicate type* over A. The order relation (meet, join) on parametric predicates is the pointwise extension of the order relation (meet, join) on predicates. For $p : A \to \mathsf{Pred}$ we define *assert* p (denoted $\{p\}$) as the procedure

$$\{p\} = (\lambda a \bullet \{p.a\})$$

In the same way we can define parametric relations and we can lift all operations over predicates, relations, and programs to operations over parametric predicates, parametric relations and procedures. All properties are trivially preserved.

We recall some facts about recursion from [1], page 334, but we use procedures and parametric predicates instead of monotonic predicate transformers and predicates.

Let $P = \{p_w \mid w \in W\}$ be a collection of parametric predicates (over A) that are indexed by the well-founded set W such that $v \leq w \Rightarrow p_v \subseteq p_w$. We refer to this collection as a collection of *ranked parametric predicates*. We define

$$p = (\bigcup_{w \in W} p_w) \text{ and } p_{<w} = (\bigcup_{v<w} p_v)$$

Now we are able to give the theorem for recursion introduction.

Theorem 29 *If body* : $\mathsf{Proc}.A \to \mathsf{Proc}.A$ *is a monotonic function on procedures over* A, $\{p_w \mid w \in W\}$ *is a collection of ranked parametric predicates, and* P *is a procedure over* A, *then*

$$(\forall w \in W \bullet \{p_w\} \; ; \; P \sqsubseteq \mathit{body}.(\{p_{<w}\} \; ; \; P)) \Rightarrow \{p\} \; ; \; P \sqsubseteq \mu \; \mathit{body}$$

Proof. The proof is by well-founded induction on W and is essentially the same as the proof of Theorem 20.1 in [1] because the parametric predicates and procedures over A satisfy the properties of predicates and predicate transformers used in the proof. ■

We will show how this theorem can be applied to obtain by refinement the recursive procedure defined in the previous section. We take

- $W = \mathsf{nat}$,

- $p_w = (\lambda(e,\ f,\ u) \bullet e \le f \wedge f \le w)$,
- $P = \left(\lambda(e,\ f,\ u) \bullet u := \binom{f}{e}\right)$, and
- *body* given by (3)

To prove that

$$\{e \le f\} \ ;\ u := \binom{f}{e} \sqsubseteq \mathsf{comb}(e,\ f,\ u) \tag{6}$$

we have to show for all $w \in \mathsf{nat}$ that

$$\{p_w\} \ ;\ P \sqsubseteq \mathit{body}.(\{p_{<w}\} \ ;\ P)).$$

This is true if, by Lemma 8, for all $e, f \in \mathsf{NatExp}$, $u \in \mathsf{NatVar}$, and $a, b \in \mathsf{nat}$

$$\begin{array}{ll} & \{e = a \wedge f = b \wedge e \le f \le w\} \ ;\ u := \binom{f}{e} \\ \sqsubseteq & \\ & \mathsf{add}.(k, n, c).(e, f, \mathsf{val}.u) \ ;\ \mathsf{add}.(x, y) \ ; \\ & \mathsf{if}\ k = 0 \vee k = n\ \mathsf{then} \\ & \quad c := 1 \\ & \mathsf{else} \\ & \quad \{k-1 \le n-1 < w\} \ ;\ x := \binom{n-1}{k-1} \ ; \\ & \quad \{k \le n-1 < w\} \ ;\ y := \binom{n-1}{k} \ ; \\ & \quad c := x + y \\ & \mathsf{fi} \ ; \\ & \mathsf{del}.(x, y) \ ;\ \mathsf{del}.(k, n) \ ;\ \mathsf{del}.c.u \end{array} \tag{7}$$

We prove this using refinement and equality rules proved in this paper for the program constructs we have introduced.

$$\begin{array}{ll} & \{e = a \wedge f = b \wedge e \le f \le w\} \ ;\ u := \binom{f}{e} \\ = & \{\text{Refinement}\} \\ & \{e = a \wedge f = b \wedge e \le f \le w\} \ ;\ u := \binom{b}{a} \\ = & \{\text{Lemma 20 and Lemma 21}\} \end{array}$$

$$
\begin{array}{ll}
& \{e = a \wedge f = b \wedge e \leq f \leq w\} \ ; \\
& \mathsf{add}.(k, n, c).(e, f, \mathsf{val}.u) \ ; \ \mathsf{add}.(x, y) \ ; \\
& [x := x_0 \mid \mathsf{true}] \ ; \ [y := y_0 \mid \mathsf{true}] \ ; \\
& \mathsf{del}.(x, y) \ ; \ \mathsf{del}.(k, n) \ ; \ \mathsf{del}.c \ ; \\
& u := \begin{pmatrix} b \\ a \end{pmatrix} \\
= & \{\text{Lemma 22}\} \\
& \{e = a \wedge f = b \wedge e \leq f \leq w\} \ ; \\
& \mathsf{add}.(k, n, c).(e, f, \mathsf{val}.u) \ ; \ \mathsf{add}.(x, y) \ ; \\
& [x := x_0 \mid \mathsf{true}] \ ; \ [y := y_0 \mid \mathsf{true}] \ ; \ c := \begin{pmatrix} b \\ a \end{pmatrix} \ ; \\
& \mathsf{del}.(x, y) \ ; \ \mathsf{del}.(k, n) \ ; \ \mathsf{del}.c.u \\
= & \{\text{Lemma 24}\} \\
& \mathsf{add}.(k, n, c).(e, f, \mathsf{val}.u) \ ; \ \mathsf{add}.(x, y) \ ; \\
& \{k = a \wedge n = b \wedge k \leq n \leq w\} \ ; \\
& [x := x_0 \mid \mathsf{true}] \ ; \ [y := y_0 \mid \mathsf{true}] \ ; \ c := \begin{pmatrix} b \\ a \end{pmatrix} \ ; \\
& \mathsf{del}.(x, y) \ ; \ \mathsf{del}.(k, n) \ ; \ \mathsf{del}.c.u \\
= & \{\text{Refinement}\} \\
& \mathsf{add}.(k, n, c).(e, f, \mathsf{val}.u) \ ; \ \mathsf{add}.(x, y) \ ; \\
& \mathsf{if} \ k = 0 \vee k = n \ \mathsf{then} \\
& \quad \{(k = 0 \vee k = n) \wedge k = a \wedge n = b \wedge k \leq n \leq w\} \ ; \\
& \quad [x := x_0 \mid \mathsf{true}] \ ; \ [y := y_0 \mid \mathsf{true}] \ ; \ c := \begin{pmatrix} b \\ a \end{pmatrix} \ ; \\
& \mathsf{else} \\
& \quad \{k \neq 0 \wedge k \neq n \wedge k = a \wedge n = b \wedge k \leq n \leq w\} \ ; \\
& \quad [x := x_0 \mid \mathsf{true}] \ ; \ [y := y_0 \mid \mathsf{true}] \ ; \ c := \begin{pmatrix} b \\ a \end{pmatrix} \ ; \\
& \mathsf{fi} \ ; \\
& \mathsf{del}.(x, y) \ ; \ \mathsf{del}.(k, n) \ ; \ \mathsf{del}.c.u \\
\sqsubseteq & \{\text{Refinement}\}
\end{array}
$$

add.$(k, n, c).(e, f, \mathsf{val}.u)$; add.(x, y) ;
if $k = 0 \vee k = n$ then
  $c := 1$
else
  $\{k \neq 0 \wedge k \neq n \wedge k = a \wedge n = b \wedge k \leq n \leq w\}$;
  $x := \binom{b-1}{a-1}$; $y := \binom{b-1}{a}$;
  $c := \binom{b-1}{a-1} + \binom{b-1}{a}$
fi ;
del.(x, y) ; del.(k, n) ; del.$c.u$

$\sqsubseteq$ {Refinement}

add.$(k, n, c).(e, f, \mathsf{val}.u)$; add.(x, y) ;
if $k = 0 \vee k = n$ then
  $c := 1$
else
  $\{k - 1 \leq n - 1 < w\}$; $x := \binom{n-1}{k-1}$;
  $\{k \leq n - 1 < w\}$; $y := \binom{n-1}{k}$;
  $c := x + y$
fi ;
del.(x, y) ; del.(k, n) ; del.$c.u$

The proof of (6) using Theorem 29 is simpler than the proof of (4). We don not need induction anymore as the recursion introduction theorem has the induction built in. The refinement relation (6) is weaker than (4) but it is strong enough in practice.

9 Hoare total correctness rule for recursive procedures with parameters

In this section we will give a Hoare rule for proving the correctness of recursive procedures based on the refinement rule given by Theorem 29. To be able to do so we need to define a Hoare triple in general and for procedures in particular. If p and q are predicates and S is a program, then a *Hoare triple* is denoted $p \;\{\!|S|\!\}\; q$ and is true if and only if $p \subseteq S.q$. For a predicate q we will also need the notation $\hat{q}$ for the relation $(\lambda \sigma \bullet q)$

Lemma 30 *$p \;\{\!|S|\!\}\; q$ is true if and only if $\{p\}$; $[\hat{q}] \sqsubseteq S$.*

Proof. See Lemma 17.4 from [1]. ∎

We extend the Hoare triple notion to procedures. If p, q are parametric predicates over A and P is a procedure over A then a *parametric Hoare triple* is denoted $p \; \{\!| P |\!\} \; q$ and is true if and only if for all $a \in A$ the Hoare triple $p.a \; \{\!| P.a |\!\} \; q.a$ is true. In this case we also lift the predicate q to the parametric relation over A, $\hat{q} = (\lambda a \, \sigma \bullet b.a)$. Lemma 30 can be trivially extended to parametric Hoare triples.

Lemma 31 *$p \; \{\!| P |\!\} \; q$ is true if and only if $\{p\} \; ; \; [\hat{q}] \sqsubseteq P$.*

Theorem 32 *If $\{p_w \mid w \in W\}$ is a collection of ranked parametric predicates, q is a parametric predicate over the set A and $body : \mathsf{Proc}.A \to \mathsf{Proc}.A$, then the following parametric Hoare rule is true*

$$\frac{(\forall w,\ P \bullet p_{<w} \; \{\!| P |\!\} \; q \Rightarrow p_w \; \{\!| body.P |\!\} \; q)}{p \; \{\!| \mu \, body |\!\} \; q}$$

Proof.

$$\begin{array}{cl}
 & p \; \{\!| \mu \, body |\!\} \; q \\
= & \{\text{Lemma 31}\} \\
 & \{p\} \; ; \; [\hat{q}] \sqsubseteq \mu \, body \\
\Leftarrow & \{\text{Theorem 29}\} \\
 & (\forall w \bullet \{p_w\} \; ; \; [\hat{q}] \sqsubseteq body.(\{p_{<w}\} \; ; \; [\hat{q}]) \\
= & \{\text{Lattice properties}\} \\
 & (\forall w,\ P \bullet \{p_{<w}\} \; ; \; [\hat{q}] \sqsubseteq P \Rightarrow \{p_w\} \; ; \; [\hat{q}] \sqsubseteq body.P \\
= & \{\text{Lemma 31}\} \\
 & (\forall w,\ P \bullet p_{<w} \; \{\!| P |\!\} \; q \Rightarrow p_w \; \{\!| body.P |\!\} \; q)
\end{array}$$

∎

Similarly we can also derive correctness rules for add and del based on lemmas we have proved about them.

For two lists $x,\ y$ of program variables, we denote by $x - y$ the list x from which we delete the variables occurring in y.

Lemma 33 *If x, y are lists of program variables, p is a predicate, and e is a program expression then*

- (*i*) $(\mathsf{del}.x; p) \; \{\!| \mathsf{del}.x |\!\} \; p$
- (*ii*) p *is* $\mathsf{del}.x$*–free* $\Rightarrow (\mathsf{del}.x; p) \; \{\!| \mathsf{del}.x.y |\!\} \; p$
- (*iii*) p *is* val*–determined and* $\mathsf{set}.(x - y)$*–free* $\Rightarrow p[y := x] \; \{\!| \mathsf{del}.x.y |\!\} \; p$
- (*iv*) $p \; \{\!| \mathsf{add}.x |\!\} \; (\mathsf{del}.x; p)$
- (*v*) $p \; \{\!| \mathsf{add}.x.e |\!\} \; (\mathsf{del}.x; p)$
- (*vi*) p *is* val*–determined and* $\mathsf{var}.x \Rightarrow p[x := e] \; \{\!| \mathsf{add}.x.e |\!\} \; p$

Proof. The proofs of (*i*), (*ii*), (*iv*), and (*v*) are strait forward consequences of the fact that $\mathsf{add}.x$, $\mathsf{add}.x.e$, $\mathsf{del}.x$, and $\mathsf{del}.x.y$ are total relations. The relation (*vi*) is a direct consequence of Lemma 23.

We prove (*iii*) by proving $p[y := x] = [\mathsf{del}.x.y].p$. We assume that p is val–determined and $\mathsf{set}.(x-y)$–free, then

$$
\begin{array}{cl}
 & [\mathsf{del}.x.y].p.\sigma \\
= & \{\text{functional update definition}\} \\
 & p.(\mathsf{del}.x.y.\sigma) \\
= & \{\mathsf{del}.x.y \text{ definition}\} \\
 & p.(\mathsf{set}.y.(\mathsf{val}.x.\sigma).(\mathsf{del}.x.\sigma)) \\
= & \{p \text{ is } \mathsf{set}.(x-y)\text{–free}\} \\
 & p.(\mathsf{set}.(x-y).(\mathsf{val}.(x-y).\sigma).(\mathsf{set}.y.(\mathsf{val}.x.\sigma).(\mathsf{del}.x.\sigma))) \\
= & \{p \text{ is } \mathsf{val}\text{–determined and} \\
 & \quad (\mathsf{set}.(x-y).(\mathsf{val}.(x-y).\sigma).(\mathsf{set}.y.(\mathsf{val}.x.\sigma).(\mathsf{del}.x.\sigma))) \\
 & \quad \sim \\
 & \quad \mathsf{set}.y.(\mathsf{val}.x.\sigma).\sigma\} \\
 & p.(\mathsf{set}.y.(\mathsf{val}.x.\sigma).\sigma) \\
= & \{\text{substitution definition}\} \\
 & p[y := x]
\end{array}
$$

■

Lemma 34 *If* $S \in \mathsf{MTran}$*,* e *is a program expression of type* A*, and* $p, q \in \mathsf{Pred}$*, then*

$$(\forall a \in A \bullet (e = a) \wedge p \; \{\!|S|\!\} \; q) \Leftrightarrow p \; \{\!|S|\!\} \; q$$

Lemma 35 *If* S *a conjunctive predicate transformer, and* p, q, p', q' *are predicates then*

$$p \; \{\!|S|\!\} \; q \wedge p' \; \{\!|S|\!\} \; q' \Rightarrow (p \wedge p') \; \{\!|S|\!\} \; (q \wedge q')$$

Lemma 36 *If* e, f *are program expressions of appropiate types then*

(*i*) e *is* $\mathsf{del}.x$*–free* $\Rightarrow \mathsf{del}.x; e = e$

(*ii*) $(\mathsf{del}.x; e)[x := f] = \mathsf{del}.x; e$

(*iii*) $x \cap y = \emptyset \Rightarrow (\mathsf{del}.x; e)[y := f] = \mathsf{del}.x; (e[y := f])$

Using these rules we prove the correctness of the procedure for computing the binomial coefficient using the same arguments we used, but reformulated in the context of Hoare proof rules.

We assume for all $q \in \mathsf{Pred}$, and $a, b, d \in \mathsf{nat}$ that

$$
\begin{array}{l}
(\lambda e, f, u \bullet q \wedge u = d \wedge e = a \wedge f = b \wedge e \le f < w) \\
\{\!|P|\!\} \\
\left(\lambda e, f, u \bullet q[u := d] = a \wedge u = \binom{b}{a}\right)
\end{array}
\tag{8}
$$

The predicate q in (8) specify that a procedure call to comb does not change any program variable except u. We need this fact when we prove the correctness of this procedure. We have to show that the recursive call $\mathsf{comb}.(k-1, n-1, x)$ does not change k and n.

We prove for all q, a, b, d, e, f, and u that

$q \wedge \mathsf{val}.u = d \wedge e = a \wedge f = b \wedge e \le f \le w$

$\{\!|$

$\quad \mathsf{add}.(k, n, c).(e, f, \mathsf{val}.u)$; $\mathsf{add}.(x, y)$;

$\quad$ if $k = 0 \vee k = n$ then

$\quad\quad c := 1$

$\quad$ else

$\quad\quad P.(k-1, n-1, x)$; $P.(k, n-1, y)$; $c := x + y$

$\quad$ fi ;

$\quad \mathsf{del}.(x, y)$; $\mathsf{del}.(k, n)$; $\mathsf{del}.c.u$

$|\!\}$

$q[u := d] \wedge \mathsf{val}.u = \binom{b}{a}$

$\Leftarrow$ {Lemma 33 and Lemma 35}

$\mathsf{del}.(k, n, c); q \wedge \mathsf{del}.(k, n, c); \mathsf{val}.u = d \wedge$
$k = a \wedge n = b \wedge k \le n \le w$

$\{\!|$

$\quad \mathsf{add}.(x, y)$;

$\quad$ if $k = 0 \vee k = n$ then

$\quad\quad c := 1$

$\quad$ else

$\quad\quad P.(k-1, n-1, x)$; $P.(k, n-1, y)$; $c := x + y$

$\quad$ fi ;

$\quad \mathsf{del}.(x, y)$; $\mathsf{del}.(k, n)$

$|\!\}$

$\mathsf{del}.c; (q[u := d]) \wedge \mathsf{val}.c = \binom{b}{a}$

$\Leftarrow$ {Lemma 33 and $v = (x, y, k, n, c)$}

$\mathsf{del}.v; q \wedge \mathsf{del}.v; \mathsf{val}.u = d \wedge k = a \wedge n = b \wedge k \leq n \leq w$

$\{\!|$

  $\mathsf{if}\ k = 0 \vee k = n\ \mathsf{then}$

    $c := 1$

  else

    $P.(k-1, n-1, x)\ ;\ P.(k, n-1, y)\ ;\ c := x + y$

  $\mathsf{fi}\ ;$

$|\!\}$

$\mathsf{del}.v; (q[u := d]) \wedge \mathsf{val}.v = \binom{b}{a}$

$\Leftarrow$ {if statement corectness: case $k \neq 0 \wedge k \neq n$}

$\mathsf{del}.v; q \wedge \mathsf{del}.v; \mathsf{val}.u = d \wedge k = a \wedge n = b \wedge 0 < k < n \leq w$

$\{\!|\ P.(k-1, n-1, x)\ ;\ P.(k, n-1, y)\ ;\ c := x + y\ |\!\}$

$\mathsf{del}.v; (q[u := d]) \wedge \mathsf{val}.v = \binom{b}{a}$

$\Leftarrow$ {Lemma 34, for all $d' \in \Gamma.x$}

$(\mathsf{del}.v; q \wedge \mathsf{del}.v; \mathsf{val}.u = d \wedge k = a \wedge n = b \wedge 0 < k < n \leq w) \wedge$
$\mathsf{val}.x = d' \wedge k - 1 = a - 1 \wedge n - 1 = b - 1 \wedge k - 1 \leq n - 1 < w$

$\{\!|\ P.(k-1, n-1, x)\ ;\ P.(k, n-1, y)\ |\!\}$

$\mathsf{del}.v; (q[u := d]) \wedge \mathsf{val}.x + \mathsf{val}.y = \binom{b}{a}$

$\Leftarrow$ {Lemma 36 and Assumption (8)}

$(\mathsf{del}.v; q \wedge \mathsf{del}.v; \mathsf{val}.u = d \wedge k = a \wedge n = b \wedge 0 < k < n \leq w)[x := d'] \wedge$
$\mathsf{val}.x = \binom{b-1}{a-1}$

$\{\!|\ P.(k, n-1, y)\ |\!\}$

$\mathsf{del}.v; (q[u := d]) \wedge \mathsf{val}.x + \mathsf{val}.y = \binom{b}{a}$

$\Leftarrow$ {Assumption (8) and weakening the precondition}

$\mathsf{del}.v; q \wedge \mathsf{del}.v; \mathsf{val}.u = d \wedge 0 < a < b \wedge$
$\mathsf{val}.x = \binom{b-1}{a-1} \wedge \mathsf{val}.y = \binom{b-1}{a}$

$\Rightarrow$

$\mathsf{del}.v; (q[u := d]) \wedge \mathsf{val}.x + \mathsf{val}.y = \binom{b}{a}$

$=$ {logic}

true

Using Theorem 32 we obtain for all $a, b, d \in \mathsf{nat}$, $e, f \in \mathsf{NatExp}$, $u \in \mathsf{NatVar}$, and $q \in \mathsf{Pred}$ that

$$(q \wedge u = d \wedge e = a \wedge f = b \wedge e \leq f)\ \{|\mathsf{comb}(e, f, u)|\}\ \left(q[u := d] \wedge u = \binom{b}{a}\right)$$

10 Conclusions

We have introduced new program constructs for adding and deleting program variables and have used them to give a predicate transformer semantics for recursive procedures with parameters and local variables. We proved some properties of these constructs and showed how one can prove the correctness of a recursive procedure using this semantics. We have also given a refinement rule for introduction of recursive procedure calls and, based on this, we proved a Hoare correctness rule for recursive procedures with parameters and local variables. We do not need to change the state space in our approach to accommodate local variables or procedure parameters. Because of this our calculus is simpler and more algebraic than the ones in the literature.

Having only value and value-result parameters does not seem to be a major drawback. According to [8], in the absence of aliasing, call by reference is equivalent to call by value result.

Many procedure proof rules in the literature are mixing the procedure call with the procedure parameters. We have separated these concerns [17], and obtained as a result much simpler rules. We have rules for recursive procedures in which the parameters are not involved at all. We have different rules that deal with parameters (local variables) and they are almost as simple as the assignment rules.

References

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

[2] R.J. Back and J. von Wright. Compositional action system refinement. In J. Derrick, E. Boiten, J. Woodcock, and J. von Wright, editors, *Electronic Notes in Theoretical Computer Science*, volume 70. Elsevier Science Publishers, 2002.

[3] O. Celiku and J. von Wright. Theorem prover support for precondition and correctness calculation. In *4th International Conference on For-*

mal Engineering Methods, volume 2495 of *Lecture Notes in Computer Science*, pages 299–310. Springer-Verlag, October 2002.

[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] E.W. Dijkstra. Guarded commands, nondeterminacy and formal derivation of programs. *Comm. ACM*, 18(8):453–457, 1975.

[7] E.W. Dijkstra. *A discipline of programming*. Prentice-Hall Inc., Englewood Cliffs, N.J., 1976. With a foreword by C. A. R. Hoare, Prentice-Hall Series in Automatic Computation.

[8] J.E. Donahue. *Complementary definitions of programming language semantics*. Springer-Verlag, Berlin, 1976. Lecture Notes in Computer Science, Vol. 42.

[9] M.J.C. Gordon. Mechanizing programming logics in higher-order logic. In Birtwistle, G.M. and Subrahmanyam, P.A., editors, *Current Trends in Hardware Verification and Automatic Theorem Proving (Proceedings of the Workshop on Hardware Verification)*, pages 387–439, Banff, Canada, 1988. Springer-Verlag, Berlin.

[10] M.J.C. Gordon and T.F. Melham, editors. *Introduction to HOL*. Cambridge University Press, Cambridge, 1993. A theorem proving environment for higher order logic, Appendix B by R. J. Boulton.

[11] D. Gries and G. Levin. Assignment and procedure call proof rules. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 2(4):564–579, 1980.

[12] D. Harel, D. Kozen, and J. Tiuryn. *Dynamic logic*. MIT Press, Cambridge, MA, 2000.

[13] W.M. Hesselink. Predicate transformers for recursive procedures with local variables. *Formal Aspect of Computing*, 11:616–336, 1999.

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

[15] T. Kleymann. Hoare logic and auxiliary variables. *Formal Aspect of Computing*, 11:541–566, 1999.

[16] L. Laibinis. *Mechanised Formal Reasoning About Modular Programs.* PhD dissertation, Turku Centre for Computer Science, April 2000.

[17] C. Morgan. Procedures, parameters, and abstraction: separate concerns. *Sci. Comput. Programming*, 11(1):17–27, 1988.

[18] S. Owre, N. Shankar, J.M. Rushby, and D.W.J. Stringer-Clavert. *PVS Language Reference*, 1999.

[19] M. Staples. *A Mechanised Theory of Refinement*. PhD dissertation, Computer Laboratory, University of Cambridge, November 1998.

[20] M. Staples. Representing WP semantics in Isabelle/ZF. In *Theorem proving in higher order logics (Nice, 1999)*, volume 1690 of *Lecture Notes in Comput. Sci.*, pages 239–254. Springer, Berlin, 1999.

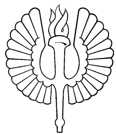
[21] M. Staples. Interfaces for refining recursion and procedures. *Formal Aspect of Computing*, 12:372–391, 2000.

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

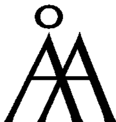
[23] D. von Oheimb. Hoare logic for mutual recursion and local variables. In C. Pandu Rangan, V. Raman, and R. Ramanujam, editors, *Foundations of Software Technology and Theoretical Computer Science*, volume 1738 of *LNCS*, pages 168–180. Springer, 1999.

Turku Centre for Computer Science
Lemminkäisenkatu 14
FIN-20520 Turku
Finland

http://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 Science**