Under consideration for publication in Formal Aspects of Computing

An Algebraic Treatment of Procedure Refinement to Support Mechanical Verification[1]

Ralph-Johan Back and Viorel Preoteasa

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

Abstract. We introduce a new algebraic model for program variables, suitable for reasoning about recursive procedures with parameters and local variables in a mechanical verification setting. We give a predicate transformer semantics to recursive procedures and prove refinement rules for introducing recursive procedure calls, procedure parameters, and local variables. We also prove, based on the refinement rules, Hoare total correctness rules for recursive procedures, and parameters. We introduce a special form of Hoare specification statement which alone is enough to fully specify a procedure. Moreover, we prove that this Hoare specification statement is equivalent to a refinement specification. We implemented this theory in the PVS theorem prover.

Keywords: Refinement, Recursive procedures, Semantics, Hoare logic, Mechanical verification.

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 [Hes99, Sta99, Sta00, Lai00, CvW02, Gor88], some of which are used in mechanical verification, are based on states represented as tuples, with one component for each program variable. Accessing or updating a specific program variable in a mechanical verification setting requires accessing or updating the corresponding component in the tuple. The problem becomes even more complicated when working with local variables. Then we should add or delete components from the tuple. This extra calculus makes the reasoning about the correctness of a program more complicated.

[1] This work is based on an earlier work: Reasoning about recursive procedures with parameters. In Proceedings of the Workshop on Mechanized Reasoning about Languages with Variable Binding, 2003, Uppsala, Sweden, ACM Press
Correspondence and offprint requests to: Ralph-Johan Back and Viorel Preoteasa

An intuitive alternative approach is given in [BvW98]: 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 are required to 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 the 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 [BvW98]. 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. This should be possible as often as needed.

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 a new axiomatic model of program variables suitable for implementations in a theorem prover. We give a predicate transformer semantics [Dij75, Dij76] for mutually recursive procedures with value and value–result parameters and local variables. We introduce new refinement rules and Hoare [Hoa69] total correctness rules for local variables, procedure parameters and recursive procedures. The Hoare rules are proved consistent with respect to the predicate transformer semantics. Most of these rules have no side conditions hence they are easier to apply than the ones in the literature. When the rules have side conditions they are straightforward to verify and the verification can be done automatically. We introduce a special form of Hoare specification statement which fully specifies a program without the help of any (syntactic) side conditions. We also prove a theorem that connects these specification statements to refinement specifications. We have implemented this theory and proved all the results of this paper in the PVS theorem prover [OSRSC01]. The implementation follows very closely the theory presented here. However, throughout the paper, we present the significant technical details of the implementation.

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 [Bac78, Bac80, Mor87, Mor90b, Mor90a, BvW98]. In Section 4, we introduce the primitive functions that we use to manipulate program variables. In Section 5 we give a semantic notion of program expressions. Using the primitive functions defined in Section 4 we define in Section 6 some program statements for introducing local variables and prove some properties about them. In Section 7 we give a least fixpoint semantics for recursive procedures with parameters and local variables and prove a refinement rule for introducing recursive procedure calls. Based on the refinement rule we prove, in Section 8, a Hoare total correctness rule for recursive procedures, and parameters. We discuss refinement and Hoare specification statements in Section 9 in both the refinement calculus and Hoare logic and prove a theorem connecting the two. Section 10 contains concluding remarks.

2. Related work

Back and von Wright [BvW98] represent a program variable x as a pair of functions ($\mathsf{val}.x$, $\mathsf{set}.x$) – $\mathsf{val}.x$ for getting the value of x in a state, and $\mathsf{set}.x$ 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 in advance 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 every 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.

Reynolds [Rey81b] introduces the more general concept of *acceptors*, which are functions mapping values into state transformers. A program variable is modeled as a pair of an acceptor and an expression. The acceptor and expression used in such a pair correspond to $\mathsf{set}.x$ and $\mathsf{val}.x$ of Back and von Wright [BvW98]. However, no assumptions about the behavior of the program variables are made.

In [BvW98] name collisions (between actual and formal parameters) are handled by renaming. This usually requires changing the state space at procedure calls, and any nested (or recursive) calls are then made in the new (extended) state space. In an improved version of this variable model [BvW03] changing the state space is no longer necessary. The latter approach of handling local variables 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 [Sta98, Sta99, Sta00] 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 types of variables may change so the state space may change as a result. 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 [Hes99] 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 proved. 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) frames to predicate transformers, is developed in order to give proper semantics to recursive procedures.

Kleymann [Kle99] 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 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 [Kle98] Kleymann introduces recursive procedures with local variables. Similarly to our approach the author uses a dependent type technique to represent program variables of various types. However, the author does not define a predicate transformer semantics for the language. Contrasting with our approach, adding a local variable in [Kle98] seems to require a change in the state space. In turn, this change may add complexity when giving a predicate transformer semantics for the language.

In [vO99] von Oheimb gives an operational semantics for an imperative deterministic language with recursive procedures, and 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. 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. However, in his approach all program variables have the same type, which is an unrealistic assumption.

Reynolds [Rey81a] gives an axiomatic semantics for recursive procedures. The main procedure mechanism is call-by-name and uses the copy rule. This mechanism is also used to define procedures with value, result, and reference parameters. Since identifier collision might occur, renaming of program variables may be needed. However, the renaming of variables leads to the introduction of new variables, which in turn might cause the state space to change.

In [Rey81a] recursive procedures are handled by extending the logic with new primitives that express when two expressions do not interfere. A distinction between environments and states of computation is made. Environments map identifiers to meanings; in particular, an environment may map two different identifiers to the same meaning. When a procedure is specified, the new non-interference primitives might be needed in order to specify that some variables do not interfere with each other. Such a specification would then be satisfied only in the environments in which those variables are mapped into distinct meanings. Lack of interference is expressed more easily in our formalism, since having a type of all program variables means that two program variables x and y do not interfere as long as they are different ($x \neq y$).

Gries and Levin [GL80] 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.

3. Preliminaries

We use higher–order logic [Chu40] as the underlying logic. In this section we recall some facts about the refinement calculus [BvW98].

A *state space* is a type State of higher–order logic. We call an element $\sigma \in \mathsf{State}$ a *program state* or simply a *state*. A *state transformer* is a function $f : \mathsf{State} \to \mathsf{State}$ that maps states to states. We denote by $\mathsf{Func.State}$ 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 State with id.

We denote by bool the Boolean algebra with two elements. For a type X, $\langle \mathsf{Pred}.X, \cup, \cap, \neg, \mathsf{false}, \mathsf{true}\rangle$ is the Boolean algebra of predicates over (subsets of) X.

A *state relation* is a binary relation on State, i.e. a function of type $\mathsf{State} \to \mathsf{State} \to \mathsf{bool}$. We denote by $\mathsf{Rel.State}$ all binary relations on State. 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 extend pointwise the operations from $\mathsf{Pred.State}$ to operations on $\mathsf{Rel.State}$. We denote by $R\ ;\ R'$ the composition of the relations R and R', and by R^{-1} the *inverse* of the relation R. We call a relation R *total* if for all x there exists y such that $R.x.y$. If $f \in \mathsf{Func.State}$ then we define $|f| \in \mathsf{Rel.State}$ by: $|f|.\sigma.\sigma' = (f.\sigma = \sigma')$. The relation $|f|$ is total.

Programs, denoted by $\mathsf{MTran.State}$, are modeled by *monotonic predicate transformers*, i.e. monotonic functions from $\mathsf{Pred.State}$ to $\mathsf{Pred.State}$. $\langle \mathsf{MTran.State}, \sqsubseteq, \sqcup, \sqcap\rangle$ is a complete lattice where $\sqsubseteq, \sqcup, \sqcap$ are the pointwise extensions of $\subseteq, \cup, \cap$ on predicates. The complete lattice operations on predicate transformers are interpreted as operations on programs: $S \sqsubseteq T$ – the refinement relation, $\bigsqcap_{i\in I} S_i$ – the demonic choice, and $\bigsqcup_{i \in I} S_i$ – the angelic choice.

We denote by $S\ ;\ T$ the functional composition of predicate transformers, the sequential composition of programs, and by skip the identity predicate transformer.

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

$$\begin{array}{lcll}
\{p\}.q & = & p \cap q & \text{(assertion)}\\
{[p]}.q & = & \neg p \cup q & \text{(assumption or guard)}\\
{[R]}.q.\sigma & = & (\forall \sigma' \bullet R.\sigma.\sigma' \Rightarrow q.\sigma') & \text{(demonic update)}\\
{[f]}.q & = & [|f|] = \{|f|\} & \text{(functional update)}\\
\mathsf{if}\ p\ \mathsf{then}\ S\ \mathsf{else}\ T\ \mathsf{fi} & = & [p]\ ;\ S \sqcap [\neg p]\ ;\ T & \text{(conditional program)}
\end{array}$$

where p, q are predicates, R is a state relation, f is a state function, and S, T are predicate transformers.

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

Many properties [BvW98] relating functions, relations and predicate transformers hold. In the next lemma we list some that we will use later.

Lemma 1. If p, q are predicates, f is a function, R, Q are relations and S is a predicate transformer then the following hold

(*i*) if $R \subseteq |f|^{-1}$ and R is total then $R\ ;\ |f| = \mathsf{Id}$.

(*ii*) $\{p \wedge q\} = \{p\}\ ;\ \{q\}$

(*iii*) $[R]$ is conjunctive

(*iv*) $R \subseteq Q \Rightarrow [Q] \sqsubseteq [R]$

(*v*) S is conjunctive $\Rightarrow \{S.q\}\ ;\ S = S\ ;\ \{q\}$

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, and MTran instead of $\mathsf{Func.State}$, $\mathsf{Pred.State}$, $\mathsf{Rel.State}$, and $\mathsf{MTran.State}$, since State is fixed.

All these constructs are implemented in PVS exactly as they are presented here.

4. Program variables axioms

We denote by Var the type of *program variables*. If x is a program variable ($x \in \mathsf{Var}$) then $\mathsf{T}.x$ denotes its type. Both Var and $\mathsf{T}.x$ are assumed to be nonempty types in higher order logic.

We next introduce functions for accessing and updating the value of program variables.

$$\begin{array}{ll} \mathsf{val}.x : \mathsf{State} \to \mathsf{T}.x & \text{(the } \textit{value} \text{ of } x) \\ \mathsf{set}.x : \mathsf{T}.x \to \mathsf{State} \to \mathsf{State} & (\textit{updating } x) \end{array}$$

As expected, for $x \in \mathsf{Var}$, $\sigma \in \mathsf{State}$, and $a \in \mathsf{T}.x$, $\mathsf{val}.x.\sigma$ is the value of program variable x in state σ, and $\mathsf{set}.x.a.\sigma$ is the state obtained from σ by setting the value of variable x to a.

Local variables are modeled using two statements (add and del) which intuitively correspond to stack operations – adding a variable to the stack and deleting it from the stack. In particular, these statements make it possible to save a program variable x at any point during execution, work with x as if it were new, and restore the old value of x. Of the two statements, only del is a primitive one in our calculus, whereas add will be defined in Section 6 as the relational inverse of del.

$$\mathsf{del}.x : \mathsf{State} \to \mathsf{State} \qquad (\textit{deleting} \text{ local } x)$$

A consequence of this treatment of program variables is that a program variable x cannot change its type in a local scope. However, we overcome this problem by choosing a semantic map which translates a syntactic program into a predicate transformer by for example prefixing all variable names with their type names, so two (syntactic) program variables with the same name but different types would be mapped into different program variables.

The behavior of the primitives val, set and del is described using the following axioms. Note that although the intuition behind del is described in terms of a stack, the axioms do not make any assumptions about the structure of the program state.

(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 assumptions in [BvW98].

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 and delete two distinct program variables x and y.

The types of the functions val, set, and del depend on their first argument ($\mathsf{T}.x$), which is why we chose to implement them in PVS using the dependent type mechanism. This, in particular, means that all $\mathsf{T}.x$ should have the same super-type, which is implemented as a generic (nonempty) type ProgType. The program variables theory then depends on ProgType, i.e., is a parameterized theory with ProgType as parameter. The benefit to using this approach is that the theories we develop do not make any assumptions about how ProgType is to be instantiated.

When performing concrete verifications we need to instantiate ProgType accordingly. This can be easily done by considering the disjoint union of the program variable (values) types that we need. If, say, we need variables of types bool and nat, then their disjoint union bool + nat, denoted as MyProgType, is used to instantiate ProgType. Moreover, we denote by Nat the subtype of MyProgType corresponding to the nat component of the disjoint union, and lift all operations on nat to corresponding ones on Nat. In order to use a program variable x with values from the natural numbers, we axiomatize: $\mathsf{T}.x = \mathsf{Nat}$. However, working with the types of the program variables becomes as simple as working with the PVS basic types once suitable rewrite rules are introduced; type conditions would then be discharged automatically just like they would if we were using the basic types.

Although building an algebra of program data-types requires some work, the mechanism is flexible and can be used to define any collection of data-types, such as arrays and pointers. The fact that we do not know beforehand which types may be used in a refinement is not a limitation in our calculus because we can add new types to our program data-type algebra without invalidating results that have already been proved.

4.1. A model

In order to show that the axioms are consistent, we 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.

For a type A we denote by $\langle A^*,\ \cdot,\ \varepsilon\rangle$ the free monoid of finite lists over A with "$\cdot$" the concatenation of lists and ε the empty list. We assume $A \subset A^*$ and for $a_1, \ldots, a_n \in A$ we denote by $(a_1, \ldots, a_n) \in A^*$ the list of $a_1, \ldots, a_n$.

Let ProgType be the disjoint union of all program variables types and $\mathsf{Stack} = \mathsf{ProgType}^*$. We take

$$\mathsf{State} = \left(\prod_{x \in \mathsf{Var}} \mathsf{T}.x\right) \times \mathsf{Stack}$$

A state $\sigma \in \mathsf{State}$ has two components (v, s). The component v is a tuple with one component for each program variable. The component corresponding to x in v, denoted v_x, represents the value of x in the state σ. The component s of σ is the stack of the computation.

Formally we define val, set and del as:

$$\begin{array}{lcl}
\mathsf{val}.x.(v,s) & = & v_x \\
\mathsf{set}.x.a.(v,s) & = & (v',s) \text{ where } v'_y = \mathsf{if}\ x = y\ \mathsf{then}\ a\ \mathsf{else}\ v_y\ \mathsf{fi} \\
\mathsf{del}.x.(v,s) & = & \left\{ \begin{array}{ll} \mathsf{set}.x.a.(v,s') & \text{if } s = a \cdot s' \wedge a \in \mathsf{T}.x \\ (v,s) & \text{if } s = \varepsilon \vee (s = a \cdot s' \wedge a \notin \mathsf{T}.x) \end{array} \right.
\end{array}$$

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

In this model the statement add.x (defined as the inverse of del.x) saves the value of x in the stack and then makes the value of x undefined.

In [BvW03] an approach similar to ours for treating program variables is followed: the behavior of the program variables is given by a set of functions (add, del, and val) that satisfy some axioms, and a model is provided. Our axioms are also valid in the model provided in [BvW03], but the axioms from [BvW03] are not valid in our model.

4.2. Lists 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. Although in theory lists of program variables do not require a special treatment, when working with a theorem prover this problem should be considered carefully.

We denote by VarList the set Var^*. We use the same notation x, y, z, ... to denote both program variables and lists of program variables. Unless specified, x, y, z denote lists of program variables.

For $x \in \mathsf{VarList}$ we define $\mathsf{T}.x \subseteq \mathsf{ProgType}^*$ the type of the list of program variables $x = (x_1, \ldots, x_n)$ by

$$\mathsf{T}.(x_1, \ldots, x_n) = \{(a_1, \ldots, a_n) \mid (\forall i \bullet a_i \in \mathsf{T}.x_i)\}$$

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

$$\begin{array}{lcl}
\mathsf{val}.x.\sigma & = & (\mathsf{val}.x_1.\sigma, \ldots, \mathsf{val}.x_n.\sigma) \\
\mathsf{del}.x & = & \mathsf{del}.x_1 \ ; \ \ldots \ ; \ \mathsf{del}.x_n \\
\mathsf{set}.x.a & = & \mathsf{set}.x_1.a_1 \ ; \ \ldots \ ; \ \mathsf{set}.x_n.a_n
\end{array}$$

where $a = (a_1, \ldots, a_n) \in \mathsf{T}.x$.

The predicate var.x is true if and only if all variables in x are distinct. We denote by $x \cap y$ the set of program variables that occur in both x and y and by $x - y$ the list x from which the variables occurring in y have been deleted.

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

Lemma 2. If $x,\ y \in \mathsf{VarList}$, $\sigma \in \mathsf{State}$ and a, b are of appropriate types, then

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

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

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

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

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

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

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

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

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

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

5. Program expressions

A *syntactic program expression* is built from program variables and constants using some operators. So if x and y are program variables then $x + 2 * y$ 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 $\mathsf{val}.e.\sigma = op(\mathsf{val}.e_1.\sigma, \ldots, \mathsf{val}.e_n.\sigma)$

The semantics of a syntactic program expression of type A is a function from State to A. In general, we will consider a *program expression* of some type A as being any function from State to A. An expression of type bool is called *Boolean program expression.* The Boolean expressions coincide with the predicates on State. We denote by NatExp the type of *natural program expressions*, i.e. State $\rightarrow$ 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 $A_1 \cdot A_2 \cdot \ldots \cdot A_n$.

We denote by $e \doteq e'$ the program expression $(\lambda\sigma \bullet e.\sigma = e'.\sigma)$, the states of computation in which e and e' are equal. When there is no confusion, we sometimes use the notation x for the program expression $\mathsf{val}.x$. For example we use $x + 1$ for the program expression $(\lambda\sigma \bullet \mathsf{val}.x.\sigma + 1)$.

Theorem 3. If $S, T \in \mathsf{MTran}$ and $e : \mathsf{State} \rightarrow A$ is a program expression where $A \neq \emptyset$ then

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

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

Proof. The relation (i) follows easily by expanding the definitions and using the fact that A is a non-empty type. The relation (ii) can be proved using (i) and the fact that the sequential composition of programs left-distributes over arbitrary nonempty joins. □

Definition 4. Let $e : \mathsf{State} \rightarrow A$, $x \in \mathsf{VarList}$, and $e' : \mathsf{State} \rightarrow \mathsf{T}.x$. We define $e[x := e'] : \mathsf{State} \rightarrow A$, the *substitution of* e' *for* x *in* e as: $e[x := e'].\sigma = e.(\mathsf{set}.x.(e'.\sigma).\sigma)$

Lemma 5. If $x, y \in \mathsf{VarList}$ and $f : \mathsf{State} \rightarrow \mathsf{State}$ then (*i*) $\mathsf{var}.x \Rightarrow (\mathsf{val}.x)[x := e] = e$

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

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

Next we introduce the notion of x-independence for an expression $e : \mathsf{State} \to A$, as the semantic correspondent of the syntactic condition that the program variables x do not occur free in e.

Definition 6. Let $e : \mathsf{State} \to A$ be an expression, $x \in \mathsf{VarList}$ and $f : \mathsf{State} \to \mathsf{State}$ be a function. We say that e is *f–independent* if $e = f \,;\, e$. We say that e is *x–independent* if

$$(\forall f : \mathsf{State} \to \mathsf{State} \bullet (\forall y \in \mathsf{Var} \bullet y \cap x = \emptyset \Rightarrow \mathsf{val}.y = f \,;\, \mathsf{val}.y) \Rightarrow e \text{ is } f\text{–independent}).$$

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

The idea behind the definition of x–independent is that if a function $f : \mathsf{State} \to \mathsf{State}$ does not change any program variable y (except possibly x), then f does not change the value of e either.

Lemma 7. If e is a syntactic program expression that does not contain x free then $\mathsf{val}.e$ is x–independent.

The proof can be done by structural induction on e.

Lemma 8. If e, f are program expressions of appropriate types then

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

(*ii*) $x \cap y = \emptyset \wedge f$ is $\mathsf{del}.x$–independent $\Rightarrow (\mathsf{del}.x; e)[y := f] = \mathsf{del}.x; (e[y := f])$

The way program expressions are defined so far allows them to depend not only on the current values of the program variables but also on their values from the stack. For example $\mathsf{del}.x \,;\, \mathsf{val}.x$ is a program expression that depends on the previous value of x. We define a subclass of program expressions that depend 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 fact that $\mathsf{del}.x \,;\, \mathsf{val}.x$ is val-determined cannot be proved using the axioms since this fact is not true in the model we provided and all provable sentences must be true in the model.

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

Lemma 9. If $x \in \mathsf{VarList}$ and $f : \mathsf{State} \to \mathsf{State}$ is a function such that $\mathsf{val}.y$ is f–independent for all $y \in \mathsf{Var}$, $y \cap x = \emptyset$, then $\mathsf{set}.x.a.\sigma \sim \mathsf{set}.x.a.(f.\sigma)$.

Lemma 10. If e is a syntactic program expression, then the program expression $\mathsf{val}.e$ is val-determined.

6. Program statements

In this section we introduce some program constructs and prove some properties about them and their compositions. These are monotonic predicate transformers based on functions or demonic updates. When proving an equality between programs based on functions and demonic updates, we work at the relations level. Since the demonic update is a monoid homomorphism from relations to monotonic predicate transformers the results can then be easily lifted to monotonic predicate transformers.

Let $x \in \mathsf{VarList}$, e be a program expression of type $\mathsf{T}.x$ and $b : \mathsf{T}.x \to \mathsf{Pred}$. We recall the definition of $x := e : \mathsf{State} \to \mathsf{State}$ (the multiple assignment) and $(x \mid b) : \mathsf{State} \to \mathsf{State} \to \mathsf{bool}$ (the nondeterministic assignment) from [BvW98].

$$\begin{array}{rcl} (x := e).\sigma &=& \mathsf{set}.x.(e.\sigma).\sigma \\ (x \mid b).\sigma.\sigma' &=& (\exists a \in \mathsf{T}.x \bullet b.a.\sigma \wedge \mathsf{set}.x.a.\sigma = \sigma') \end{array}$$

Sometimes we use the notation $(x := a \mid b.a)$ for $(x \mid b)$. All properties from [BvW98] regarding the assignment statements also hold in our framework.

Definition 11. If $x \in \mathsf{VarList}$ and $e : \mathsf{State} \to \mathsf{T}.x$, then we define:

- Add local variables:
  $\mathsf{add}.x.\sigma.\sigma' = (\sigma = \mathsf{del}.x.\sigma')$
- Add and initialize local variables:
  $\mathsf{add}.x.e.\sigma.\sigma' = (\exists \sigma_0 \bullet \sigma = \mathsf{del}.x.\sigma_0 \wedge \mathsf{set}.x.(e.\sigma).\sigma_0 = \sigma')$
- Save and delete local variables:
  $\mathsf{del}.x.y.\sigma = \mathsf{set}.y.(\mathsf{val}.x.\sigma).(\mathsf{del}.x.\sigma)$

We use the function $\mathsf{del}.x.y$ to implement procedures with value–result parameters. $\mathsf{del}.x.y$ deletes a variable from the stack and saves its value into another variable.

We denote by $\mathsf{Add}.x$, $\mathsf{Add}.x.e$, $\mathsf{Del}.x$, and $\mathsf{Del}.x.y$ the predicate transformers $[\mathsf{add}.x]$, $[\mathsf{add}.x.e]$, $[\mathsf{del}.x]$, and $[\mathsf{del}.x.y]$, respectively.

Theorem 12. If $x, y \in \mathsf{VarList}$ then

(*i*) $\mathsf{Add}.x \mathbin{;} \mathsf{Del}.x = \mathsf{skip}$

(*ii*) $\mathsf{Add}.x.e \mathbin{;} \mathsf{Del}.x = \mathsf{skip}$

Proof. Using Lemma 1.(*i*). □

Lemma 13. If x, y, p, and e are of appropriate types then

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

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

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

(*iv*) p is val–determined $\Rightarrow \mathsf{Add}.x.e.p = p[x := e]$

(*v*) p is $\mathsf{set}.y$–independent $\Rightarrow \mathsf{Del}.x.y.p = \mathsf{del}.x \mathbin{;} p$

(*vi*) p is val–determined and $\mathsf{set}.(x - y)$–independent $\Rightarrow \mathsf{Del}.x.y.p = p[y := x]$

Proof. (*i*) and (*v*) follow directly from the definitions.

The equalities (*ii*) and (*iii*) follow from Lemma 12 and (*i*).
We prove (*iv*):

$$
\begin{aligned}
& \mathsf{Add}.x.e.p.\sigma \\
= {} & \{\text{Definition}\} \\
& (\forall \sigma' \bullet \mathsf{add}.x.e.\sigma.\sigma' \Rightarrow p.\sigma') \\
= {} & \{\text{Definition}\} \\
& (\forall \sigma' \bullet (\exists \sigma'' \bullet \sigma = \mathsf{del}.x.\sigma'' \wedge \mathsf{set}.x.(e.\sigma).\sigma'' = \sigma') \Rightarrow p.\sigma') \\
= {} & \{\text{Quantifier properties}\} \\
& (\forall \sigma',\ \sigma'' \bullet \sigma = \mathsf{del}.x.\sigma'' \wedge \mathsf{set}.x.(e.\sigma).\sigma'' = \sigma' \Rightarrow p.\sigma') \\
= {} & \{\text{Quantifier properties}\} \\
& (\forall \sigma'' \bullet \sigma = \mathsf{del}.x.\sigma'' \Rightarrow p.(\mathsf{set}.x.(e.\sigma).\sigma'')) \\
= {} & \{p \text{ is } \mathsf{val}\text{–determined and } \mathsf{set}.x.(e.\sigma).\sigma'' \sim \mathsf{set}.x.(e.\sigma).(\mathsf{del}.x.\sigma'') \text{ by Lemma 9}\} \\
& (\forall \sigma'' \bullet \sigma = \mathsf{del}.x.\sigma'' \Rightarrow p.(\mathsf{set}.x.(e.\sigma).(\mathsf{del}.x.\sigma''))) \\
= {} & \{\text{Quantifier properties}\} \\
& (\exists \sigma'' \sigma = \mathsf{del}.x.\sigma'') \Rightarrow p.(\mathsf{set}.x.(e.\sigma).\sigma \\
= {} & \{\mathsf{del}.x \text{ is surjective}\} \\
& p.(\mathsf{set}.x.(e.\sigma).\sigma \\
= {} & \{\mathsf{del}.x \text{ is surjective}\} \\
& p[x := e].\sigma
\end{aligned}
$$

We prove (*vi*):

$$
\begin{aligned}
& \mathsf{Del}.x.y.p.\sigma \\
= \; & \{\text{Definition}\} \\
& p.(\mathsf{set}.y.(\mathsf{val}.x.\sigma).(\mathsf{del}.x.\sigma)) \\
= \; & \{p \text{ is } \mathsf{set}.(x-y)\text{–independent}\} \\
& p.(\mathsf{set}.(x-y).a.(\mathsf{set}.y.(\mathsf{val}.x.\sigma).(\mathsf{del}.x.\sigma))) \\
= \; & \{\text{Definition of } \mathsf{set} \text{ for lists of program variables}\} \\
& p.(\mathsf{set}.(x-y,\ y).(a,\ \mathsf{val}.x.\sigma).(\mathsf{del}.x.\sigma)) \\
= \; & \{\text{Lemma 9}\} \\
& p.(\mathsf{set}.(x-y,\ y).(a,\ \mathsf{val}.x.\sigma).\sigma) \\
= \; & \{p \text{ is } \mathsf{set}.(x-y)\text{–independent}\} \\
& p.(\mathsf{set}.y.(\mathsf{val}.x.\sigma).\sigma) \\
= \; & \{\text{Definition}\} \\
& p[y := \mathsf{val}.x].\sigma
\end{aligned}
$$

□

Theorem 14. 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*) $\{p\}\ ;\ \mathsf{Add}.x.e = \mathsf{Add}.x.e\ ;\ \{\mathsf{del}.x; p\}$

(*iv*) p is val-determined $\Rightarrow \{p[x := e]\}\ ;\ \mathsf{Add}.x.e = \mathsf{Add}.x.e\ ;\ \{p\}$

Proof. Using Theorem 13 and Lemma 1.(v) □

Theorem 15. If $x, y \in \mathsf{VarList}$ have the same type, e is program expression of type $\mathsf{T}.x$, and b is a Boolean expression, then

(*i*) $x := e\ ;\ \mathsf{Del}.x = \mathsf{Del}.x$

(*ii*) $x \cap y = \emptyset \wedge e$ is $\mathsf{del}.x$–independent $\Rightarrow y := e\ ;\ \mathsf{Del}.x = \mathsf{Del}.x\ ;\ y := e$

(*iii*) $\mathsf{Del}.x \sqsubseteq [x \mid b]\ ;\ \mathsf{Del}.x$

(*iv*) $\mathsf{var}.x \wedge e$ is $\mathsf{del}.x$–independent $\Rightarrow x := e\ ;\ \mathsf{Del}.x.y = \mathsf{Del}.x\ ;\ y := e$

Proof. The properties (i) and (ii) are straightforward consequences of Lemma 2. The properties (iii) and (iv) can be easily proved by expanding the definitions and using Lemma 2. □

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 value–result parameters. These procedures could also have local variables.

Definition 16. We call an element from $A \rightarrow \mathsf{MTran}$ a *procedure with parameters from* A or simply a *procedure over* A. 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 value–result parameter y, both of type Nat, has $A = \mathsf{NatExp} \times \mathsf{NatVar}$, where NatVar is the type of all program variables of type Nat. 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. 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 ";" distributes over arbitrary meets and joins from the 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 value–result in this setting. We also give semantics for recursive procedures.

7.1. Non-recursive procedures

A general non-recursive procedure declaration is:

$$\begin{array}{l} \textsf{procedure } \mathit{name}(\textsf{val } x;\ \textsf{val-res } y) : \\ \quad \mathit{body} \end{array} \tag{1}$$

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

The procedure declaration (1) is an abbreviation of the following formal definition:

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

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

Using this approach any number of local variables can be added in the procedure body. If w are the local variables, then *body* is $\mathsf{Add}.w\ ;\ \mathit{body}_0\ ;\ \mathsf{Del}.w$.

Our procedures can access global variables as well. For example if we have the procedure

```
procedure p1 (val-res $a$) :
    $a := a + 1$;
    $x := x + 1$
```

then the program $x := 2\ ;\ b := 3\ ;\ \mathsf{p1}(b)$ has the same effect as $x := 3\ ;\ b := 4$. However if instead of procedure p1 we use the following procedure

```
procedure p2 (val-res $a$) :
    local $x$;
    p1($a$)
```

then $x := 2\ ;\ b := 3\ ;\ \mathsf{p2}(b)$ has the same effect as $x := 2\ ;\ b := 4$, since the call to p1 in procedure p2 will update the local variable x, and not the global one. Therefore our calculus implements dynamic scoping of program variables, i.e. program variables are bounded to the scope where procedures are called and not to where they are defined. However to implement static scoping all we have to do is to choose a suitable semantic mapping of programs to predicate transformers which maps syntactic program variables' names into program variables according to scoping rules of the language to be translated. This kind of map would translate for example a syntactic global program variable x to program variable gl_x.

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 $\mathit{body} : \mathsf{Proc}.A \to \mathsf{Proc}.A$ is a monotonic function then we define the recursive procedure given by *body* as $\mu\,\mathit{body}$. The least fixpoint of *body* always exists by the Knaster–Tarski [Tar55] least fixpoint theorem for monotonic endo-functions on a complete lattice. We introduced in our PVS implementation a couple of theories in which we developed the theory of complete lattices necessary for stating and proving the least fixpoint theorem.

When a recursive procedure is defined, one should prove the monotonicity of the function whose least fixpoint defines the procedure. We have introduced rewriting rules into our PVS theories to automatically discharge these facts.

7.3. An example of a recursive procedure definition

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 the definition of comb is given by:

```
procedure comb (val $k, n$; val-res $c$) :
    local $x, y$
    if $k = 0 \vee k = n$ then
        $c := 1$
    else                                                    (2)
        comb($k-1,\ n-1,\ x$) ;
        comb($k,\ n-1,\ y$) ;
        $c := x + y$
    fi
```

Definition (2) corresponds to following formal definition: Let A be $\mathsf{NatExp} \times \mathsf{NatExp} \times \mathsf{NatVar}$ and define

```
    $body.S.(e, f, u)$
=
    Add.$(k, n, c).(e, f, \mathsf{val}.u)$ ; Add.$(x, y)$ ;
    if $k \doteq 0 \vee k \doteq n$ then
        $c := 1$                                            (3)
    else
        $S.(k-1,\ n-1,\ x)$ ; $S.(k,\ n-1,\ y)$ ; $c := x + y$
    fi ;
    Del.$(x, y)$ ; Del.$(k, n)$ ; Del.$c.u$
```

Then *body* is a monotonic function from $\mathsf{Proc}.A$ to $\mathsf{Proc}.A$. We define comb as $\mu\ body$.

7.4. Refinement rule for introduction of recursive procedure calls

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

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. 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 17. (Recursion Introduction) 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 parametric predicates, and P is a procedure over A, then

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

The proof can be done by well-founded induction on W.

The recursion introduction theorem is a standard result in the refinement calculus. However, we are not aware of other versions that are as simple as this one – in particular, that have no side conditions but can be applied to recursive procedures with parameters and local variables. Similar theorems with no side conditions can only be applied to parameter-less procedures [Mor87, BvW98], whereas those that handle procedures with parameters and local variables have (syntactic) side conditions and in general mix the recursive mechanism with the treatment of the procedure parameters [BvW98, Sta99, Sta00].

7.5. An example of a recursive procedure refinement

We will show how this theorem can be applied to obtain by refinement the recursive procedure defined in subsection 7.3. We take

- $W = \mathsf{Nat}$,
- $p_w = (\lambda(e, \, f, \, u) \bullet e \leq f \wedge f \doteq w)$,
- $P = \left(\lambda(e, \, f, \, u) \bullet u := \begin{pmatrix} f \\ e \end{pmatrix}\right)$, and
- *body* given by (3)

To prove that

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

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 Theorem 3, for all $e, f \in \mathsf{NatExp}$, $u \in \mathsf{NatVar}$, and $a, b \in \mathsf{Nat}$

$$\{e \doteq a \wedge f \doteq b \wedge e \leq f \doteq w\} \; ; \; u := \begin{pmatrix} f \\ e \end{pmatrix} \sqsubseteq S \tag{5}$$

where

```
S =  $\mathsf{Add}.(k, n, c).(e, f, \mathsf{val}.u)$ ; $\mathsf{Add}.(x, y)$ ;
     if $k = 0 \vee k = n$ then
         $c := 1$
     else
         $\{k - 1 \leq n - 1 < w\}$ ; $x := \begin{pmatrix} n-1 \\ k-1 \end{pmatrix}$ ;
         $\{k \leq n - 1 < w\}$ ; $y := \begin{pmatrix} n-1 \\ k \end{pmatrix}$ ;
         $c := x + y$
     fi ;
     $\mathsf{Del}.(x, y)$ ; $\mathsf{Del}.(k, n)$ ; $\mathsf{Del}.c.u$
```

We prove (5) using refinement and equality rules proved in this paper for the program constructs we have introduced.

$$\begin{aligned} & \{e \doteq a \wedge f \doteq b \wedge e \leq f \doteq w\} \; ; \; u := \begin{pmatrix} f \\ e \end{pmatrix} \\ = \quad & \{\text{Refinement [BvW98]}\} \end{aligned}$$

$$
\begin{array}{ll}
 & \{e \doteq a \land f \doteq b \land e \leq f \doteq w\} \; ; \; u := \begin{pmatrix} b \\ a \end{pmatrix} \\
\sqsubseteq & \{\text{Theorem 12 and Theorem 15}\} \\
 & \{e \doteq a \land f \doteq b \land e \leq f \doteq w\} \; ; \\
 & \mathsf{Add}.(k, n, c).(e, f, \mathsf{val}.u) \; ; \; \mathsf{Add}.(x, y) \; ; \\
 & [x, y := s, t \mid \mathsf{true}] \\
 & \mathsf{Del}.(x, y) \; ; \; \mathsf{Del}.(k, n) \; ; \; \mathsf{Del}.c \; ; \\
 & u := \begin{pmatrix} b \\ a \end{pmatrix} \\
= & \{\text{Theorem 14 and Theorem 15}\} \\
 & \mathsf{Add}.(k, n, c).(e, f, \mathsf{val}.u) \; ; \; \mathsf{Add}.(x, y) \; ; \\
 & \{k \doteq a \land n \doteq b \land k \leq n \doteq w\} \; ; \\
 & [x, y := s, t \mid \mathsf{true}] \; ; \; c := \begin{pmatrix} b \\ a \end{pmatrix} \; ; \\
 & \mathsf{Del}.(x, y) \; ; \; \mathsf{Del}.(k, n) \; ; \; \mathsf{Del}.c.u \\
\sqsubseteq & \{\text{Refinement [BvW98]}\} \\
 & S
\end{array}
$$

8. Hoare total correctness rules

If p and q are predicates and S is a program, then a *Hoare triple*, denoted $p \; \{\!|S|\!\} \; q$, is true if and only if $p \subseteq S.q$.

Lemma 18. (Consequence rule) If S is a monotonic predicate transformer, and p, q, p', q' are predicates such that $p' \Rightarrow p$ and $q' \Rightarrow q$ then

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

Lemma 19. (Conjunctivity rule) If S is a conjunctive predicate transformer, and p, q, p', q' are predicates then

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

For a predicate p we will denote by $\tilde{p}$ the predicate transformer given by:

$$\tilde{p}.q = \begin{cases} \mathsf{true} \text{ if } p \subseteq q \\ \mathsf{false} \text{ otherwise} \end{cases}$$

Lemma 20. If S is a monotonic predicate transformer then $p \; \{\!|S|\!\} \; q$ is true if and only if $\{p\} \; ; \; \tilde{q} \sqsubseteq S$.

We will use this lemma to translate refinement results into Hoare total correctness rules.

We extend the Hoare triple notion to procedures. If p, q are parametric predicates over A and P is a procedure over A then the *parametric Hoare triple* $p \; \{\!|P|\!\} \; q$ 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 procedure over A, $\tilde{q} = (\lambda a \bullet \tilde{q.a})$. Lemma 20 can be trivially extended to parametric Hoare triples.

Lemma 21. $p \; \{\!|P|\!\} \; q$ is true if and only if $\{p\} \; ; \; \tilde{q} \sqsubseteq P$.

The following theorem introduces rules for handling procedure parameters and local variables.

Theorem 22. 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*) $p \; \{\!|\mathsf{Add}.x|\!\} \; (\mathsf{del}.x \; ; \; p)$

(*iii*) $p \; \{\!|\mathsf{Add}.x.e|\!\} \; (\mathsf{del}.x \; ; \; p)$

(*iv*) p is val–determined $\Rightarrow p[x := e]\ \{\!|\mathsf{Add}.x.e|\!\}\ p$

(*v*) p is $\mathsf{set}.y$–independent $\Rightarrow (\mathsf{del}.x\ ;\ p)\ \{\!|\mathsf{Del}.x.y|\!\}\ p$

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

Proof. Using Lemma 13. □

To prove correctness for recursive procedures we need to introduce the following theorem. We assume that $\langle W, <\rangle$ is a well-founded set.

Theorem 23. If for all $w \in W$, $p_w : A \to \mathsf{Pred}$, $q : A \to \mathsf{Pred}$ and $\mathit{body} : \mathsf{Proc}.A \to \mathsf{Proc}.A$ is monotonic, then the following parametric Hoare rule is true

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

Proof.

$$\begin{array}{ll}
 & p\ \{\!|\mu\,\mathit{body}|\!\}\ q \\
= & \{\text{Lemma 21}\} \\
 & \{p\}\ ;\ \tilde{q} \sqsubseteq \mu\,\mathit{body} \\
\Leftarrow & \{\text{Theorem 17}\} \\
 & (\forall w \bullet \{p_w\}\ ;\ \tilde{q} \sqsubseteq \mathit{body}.(\{p_{<w}\}\ ;\ \tilde{q}) \\
= & \{\text{Lattice properties}\} \\
 & (\forall w,\ P \bullet \{p_{<w}\}\ ;\ \tilde{q} \sqsubseteq P \Rightarrow \{p_w\}\ ;\ \tilde{q} \sqsubseteq \mathit{body}.P \\
= & \{\text{Lemma 21}\} \\
 & (\forall w,\ P \bullet p_{<w}\ \{\!|P|\!\}\ q \Rightarrow p_w\ \{\!|\mathit{body}.P|\!\}\ q)
\end{array}$$

□

9. Specification statements

In [Rey81a] Reynolds introduces the concept of universal specifications – a formula is a universal specification if it is true in all environments. An environment maps identifiers to program variables or expressions. Reynolds defines universal specifications by using noninterference properties. A variable identifier x does not interfere with an expression identifier e, denoted $x \sharp\ e$, in an environment if no assignment of x changes the value of e. For example the specification $y \le z\ \{\!|x := 3|\!\}\ y \le z$ becomes universal only under the assumption $x \sharp\ (y \le z)$.

We introduced the type of all program variables. The noninterference properties can be expressed in our case by requiring that e is x–independent. In our approach there is a distinction between program variables and variables of the logic, while Reynolds treats both the same. A program variable is an element of Var and can be a constant of the logic. We define a universal specification by using variables of the logic ranging over program variables and by stating that some of these variables must be distinct (as elements of Var, and not as their values).

We call a *refinement specification* a formula of the form

$$\{p\}\ ;\ [x \mid b] \sqsubseteq S \qquad (6)$$

which means that the *specification statement* $\{p\}\ ;\ [x \mid b]$ is refined by the program S. The goal is to find an executable program S that satisfies (6). Specification statements were originally introduced by Back [Bac78, Bac80], who called them non-deterministic assignments. Later, Morgan introduced a variant of this, called specification statement [Mor88b]. In this presentation the specification statement is equivalent to the one introduced by Morgan and the one from [BvW98]. We can obtain Back's non-deterministic assignment statements by replacing p with $p \wedge (\exists a \bullet b.a)$ in (6).

A *procedure refinement specification* has the form

$$(\lambda e, x \bullet \{p.x.e\}\ ;\ [x \mid b.x.e]) \sqsubseteq P$$

where e is a program expression standing for the actual parameters of the procedure value parameters, and

x stands for the actual value–result parameters. This specification has to be valid even if the variables x are not distinct or the expression e interferes with x. For example the specification for computing the quotient and the remainder of the integer division of the expression e by f can be given as:

$$(\lambda e, f, x, y \bullet \{f > 0\} \ ; \ [x, y := a, b \mid e \doteq f \cdot a + b \wedge 0 \leq b < f]) \sqsubseteq P \tag{7}$$

We do not have to mention anything else in addition to the above statement. If procedure P is implemented according to the specification (7) then P can be used with any parameters e, f, x and y. It does not matter if e and f contains the program variables x and y. If $x = y$, i. e. x and y denote the same program variable, the procedure still performs some meaningful computation, i.e. computes the remainder of the division in y. However it might be easier to implement the procedure P if one requires (7) to be true only for distinct program variables x and y.

Hoare total correctness specifications (*Hoare specifications*) have the form $p \ \{\!|S|\!\} \ q$. Unfortunately this is not enough to specify that the program S performs some meaningful computation. One can provide a program S that trivially establishes q. For example the specification $0 \leq n \ \{\!|S|\!\} \ x \doteq n!$ can be trivially satisfied by the program $S = (x := 1 \ ; \ n := 0)$. A solution to this problem is to use auxiliary variables (variables that do not occur in S) to specify how the computation should be done. Using auxiliary variables the specification for computing the factorial becomes

$$0 \leq n \wedge n \doteq a \ \{\!|S|\!\} \ x \doteq a! \tag{8}$$

provided that the program S does not contain a. However this means that in addition to (8) one would also have to state that the program S must not contain references to a. Moreover we still have the problem of adaptability [Kle99] of our specification, specifically we want to prove that $a = x + 1 \ \{\!|S|\!\} \ a = x + 1$ and $a = x \ \{\!|S|\!\} \ a = x$ are Hoare equivalent, i.e. one can be proved using the other by Hoare rules. This fact cannot be proved in [Kle99] using the Hoare consequence rule, and the author overcomes this problem by introducing a more elaborate consequence rule. We solve this problem using the same technique as in [LvW01, Nau01], i.e. we consider the specification as $(\forall a \bullet x \doteq a \ \{\!|S|\!\} \ x \doteq a)$. This specification can be read as follows: "for all integer values a if the value of x is a at the beginning of S then after executing S the value of x is a". In contrast with other approaches our specification variables are variables of the logic ranging over values (integers in this example), and not necessarily over program variables. Using the consequence rule it is trivial to prove that the above specification is equivalent to $(\forall a \bullet x + 1 \doteq a \ \{\!|S|\!\} \ x + 1 \doteq a)$.

Similar to [Nau01], we consider the pre- and postconditions of a Hoare specification as dependent on auxiliary (specification) parameters from A, $p, q : A \to \mathsf{Pred}$. The Hoare triple $p \ \{\!|S|\!\} \ q$ is true if and only if $(\forall a : A \bullet p.a \ \{\!|S|\!\} \ q.a)$ is true, where the variable a is chosen such that it does not occur free in p, q and S. In this approach we do not have to mention that a does not occur free in S any more.

To combine procedures and auxiliary variables all we have to do is add auxiliary variables to the procedure pre- and postconditions. If we have procedure parameters from A, $P : A \to \mathsf{MTran}$, and specification parameters from B, $p, q : B \to A \to \mathsf{MTran}$, then we define the Hoare specification statement $p \ \{\!|P|\!\} \ q$ by

$$p \ \{\!|P|\!\} \ q \Leftrightarrow (\forall a, b \bullet p.b.a \ \{\!|P.a|\!\} \ q.b.a) \tag{9}$$

When a procedure is used we want to know that only some variables are changed by the call (in a desired way). This fact is specified in general using a set of program variables (frame). Only the program variables in the frame are allowed to change in an implementation. The role of the frame in our case is played by a list of program variables x. We specify that a program S changes at most the program variables x using two specification variables $p : \mathsf{Pred}$ and $a : \mathsf{T}.x$ with the statement:

$$(\lambda a, p \bullet p[x := a]) \ \{\!|S|\!\} \ (\lambda a, p \bullet p[x := a]) \tag{10}$$

This statement not only says that the value of all variables except x remain unchanged but also that the stack remains unchanged. For example if we take $p = (y \doteq b)$ in (10) where $y \cap x = \emptyset$ and b is not free in S then we have $(\lambda b \bullet y \doteq b) \ \{\!|S|\!\} \ (\lambda b \bullet y \doteq b)$, i. e. the value of y is not changed by the execution of S. When we specify a program S with a Hoare specification statement we will also specify using (10) what variable can be changed by S.

The following theorem shows how the elements we described above for writing Hoare specifications can be combined to obtain a Hoare specification statement equivalent to a refinement specification.

Theorem 24. (Refinement specification to Hoare specification) If S is a monotonic predicate transformer, $x \in \mathsf{VarList}$ such that $\mathsf{var}.x$ and $b : \mathsf{T}.x \to \mathsf{Pred}$ then

$$\begin{array}{ll} & \{p\}\ ;\ [x \mid b] \sqsubseteq S \\ \Leftrightarrow & \\ & (\lambda q, m \bullet p \wedge x \doteq m \wedge q)\ \{\!|S|\!\}\ (\lambda q, m \bullet q[x := m] \wedge b[x := m].(\mathsf{val}.x)) \end{array}$$

Proof. We prove first the implication from refinement to the Hoare statement. We assume $\{p\}\ ;\ [x \mid b] \sqsubseteq S$.

$$\begin{array}{ll}
& (p \wedge x \doteq m \wedge q)\ \{\!|S|\!\}\ (q[x := m] \wedge b[x := m].(\mathsf{val}.x)) \\
\Leftrightarrow & \{\text{Definition}\} \\
& p \wedge x \doteq m \wedge q \subseteq S.(q[x := m] \wedge b[x := m].(\mathsf{val}.x)) \\
\Leftarrow & \{\text{Using the assumption}\} \\
& p \wedge x \doteq m \wedge q \subseteq (\{p\}\ ;\ [x \mid b]).(q[x := m] \wedge b[x := m].(\mathsf{val}.x)) \\
\Leftarrow & \{\text{Definition of assert}\} \\
& p \wedge x \doteq m \wedge q \subseteq p \wedge [x \mid b].(q[x := m] \wedge b[x := m].(\mathsf{val}.x)) \\
\Leftarrow & \{\text{Monotonicity of } (\lambda r \bullet p \wedge r)\} \\
& x \doteq m \wedge q \subseteq [x \mid b].(q[x := m] \wedge b[x := m].(\mathsf{val}.x)) \\
\Leftarrow & \{\text{For some arbitrary } \sigma \in \mathsf{State}\} \\
& \mathsf{val}.x.\sigma = m \wedge q.\sigma \Rightarrow [x \mid b].(q[x := m] \wedge b[x := m].(\mathsf{val}.x)).\sigma \\
\Leftrightarrow & \{\text{Sub-derivation}\} \\
& \begin{array}{ll}
\bullet & \{\text{Assume } \mathsf{val}.x.\sigma = m \wedge q.\sigma\} \\
& [x \mid b].(q[x := m] \wedge b[x := m].(\mathsf{val}.x)).\sigma \\
\Leftrightarrow & \{\text{Definitions}\} \\
& (\forall a \bullet b.a.\sigma \Rightarrow q.(\mathsf{set}.x.m.\sigma) \wedge b.a.(\mathsf{set}.x.m.\sigma)) \\
\Leftrightarrow & \{\text{Assumption}\} \\
& (\forall a \bullet b.a.\sigma \Rightarrow q.\sigma \wedge b.a.\sigma) \\
\Leftrightarrow & \{\text{Assumption}\} \\
& \mathit{true}
\end{array} \\
& \mathit{true}
\end{array}$$

To prove the reverse implication we assume

$$(\forall q, m \bullet (p \wedge x \doteq m \wedge q).\sigma \Rightarrow S.(q[x := m] \wedge b[x := m].(\mathsf{val}.x)).\sigma) \tag{11}$$

for all q, m, and σ.

$$\begin{array}{ll}
& (\{p\}\ ;\ [x \mid b]).r.\sigma \Rightarrow S.r.\sigma \\
\Leftrightarrow & \{\text{Definitions}\} \\
& p.\sigma \wedge [x \mid b].r.\sigma \Rightarrow S.r.\sigma \\
\Leftarrow & \{\text{Using (11) with } m = \mathsf{val}.x.\sigma \text{ and } q = [x \mid b].r\} \\
& S.(([x \mid b].r)[x := \mathsf{val}.x.\sigma] \wedge b[x := \mathsf{val}.x.\sigma].(\mathsf{val}.x)).\sigma \Rightarrow S.r.\sigma \\
\Leftarrow & \{S \text{ is monotonic}\} \\
& ([x \mid b].r)[x := \mathsf{val}.x.\sigma] \cap b[x := \mathsf{val}.x.\sigma].(\mathsf{val}.x) \subseteq r \\
\Leftarrow & \{\text{For all } \sigma'\} \\
& ([x \mid b].r)[x := \mathsf{val}.x.\sigma].\sigma' \wedge b[x := \mathsf{val}.x.\sigma].(\mathsf{val}.x).\sigma' \Rightarrow r.\sigma' \\
\Leftrightarrow & \{\text{Definitions}\}
\end{array}$$

$$(\exists a \bullet (b.a.(\mathsf{set}.x.(\mathsf{val}.x.\sigma).\sigma') \Rightarrow r.(\mathsf{set}.x.a.\sigma')) \wedge$$
$$b.(\mathsf{val}.x.\sigma').(\mathsf{set}.x.(\mathsf{val}.x.\sigma).\sigma') \Rightarrow r.\sigma')$$

$\Leftarrow$ $\{a = \mathsf{val}.x.\sigma'\}$

$$(b.(\mathsf{val}.x.\sigma').(\mathsf{set}.x.(\mathsf{val}.x.\sigma).\sigma') \Rightarrow r.\sigma') \wedge$$
$$b.(\mathsf{val}.x.\sigma').(\mathsf{set}.x.(\mathsf{val}.x.\sigma).\sigma') \Rightarrow r.\sigma'$$

$\Leftrightarrow$ {Propositional logic}

true

□

The Hoare specification statement in this theorem fully expresses all the needed properties of a specification without the need for any additional (syntactic) conditions.

There exist many versions of Theorem 24 in the literature [BvW98, LvW01, Mor88b]. Most of them are proved under the syntactic condition that x are the only variables that are changed by S. But this condition is already expressed in the refinement specification. We instead embedded the fact that S may change only x in the Hoare specification. This fact also helps us in proving correctness for recursive procedures. We do not need an adaptation rule [BMW89, Old83, Kle99, Nau01] anymore. We do not use an invariance rule either, but rather we specify the invariance property (10) for a procedure and then we prove it.

Lemma 25. (Adding specification variables) If $S \in \mathsf{MTran}$, e is a program expression of type A, and $p, q \in \mathsf{Pred}$, then

$$(\lambda a \bullet e \doteq a \wedge p)\ \{\!|S|\!\}\ (\lambda a \bullet q) \Leftrightarrow p\ \{\!|S|\!\}\ q$$

Proof. Using the definition of the Hoare triple and Lemma 3. □

Lemma 26. If S is a monotonic predicate transformer then

$$\begin{array}{c} (\lambda q \bullet p \wedge q \wedge x \doteq m)\ \{\!|S|\!\}\ (\lambda q \bullet q[x := m] \wedge r) \\ \Leftrightarrow \\ (\lambda q \bullet p \wedge q \wedge x \doteq m)\ \{\!|S|\!\}\ (\lambda q \bullet p[x := m] \wedge q[x := m] \wedge r) \end{array}$$

Proof. From left to right we prove it by substituting q with $p \wedge q$. The other implication follows from the consequence rule. □

Theorem 27. (Adding specification variables) If $x, y \in \mathsf{VarList}$ so that y has the same type with x and $r : A \to \mathsf{Pred}$, then

$$\begin{array}{c} (\lambda q, m \bullet p \wedge x \doteq m \wedge q)\ \{\!|S|\!\}\ (\lambda q, m \bullet q[x := m] \wedge r.(e[x := m])) \\ \Leftrightarrow \\ (\lambda a, q, m \bullet e \doteq a \wedge p \wedge x \doteq m \wedge q)\ \{\!|S|\!\}\ (\lambda a, q, m \bullet q[x := m] \wedge r.a) \end{array}$$

Proof. $(\forall q \bullet (p \wedge x \doteq m \wedge q)\ \{\!|S|\!\}\ (q[x := m] \wedge r.(e[x := m])))$

$\Leftrightarrow$ {Lemma 25}

$(\forall a, q \bullet (e \doteq a \wedge p \wedge x \doteq m \wedge q)\ \{\!|S|\!\}\ (q[x := m] \wedge r.(e[x := m])))$

$\Leftrightarrow$ {Lemma 26}

$(\forall a, q \bullet (e \doteq a \wedge p \wedge x \doteq m \wedge q)\ \{\!|S|\!\}\ (q[x := m] \wedge e[x := m] \doteq a \wedge r.(e[x := m])))$

$\Leftrightarrow$ {Expression property}

$(\forall a, q \bullet (e \doteq a \wedge p \wedge x \doteq m \wedge q)\ \{\!|S|\!\}\ (q[x := m] \wedge e[x := m] \doteq a \wedge r.a))$

$\Leftrightarrow$ {Lemma 26}

$(\forall a, q \bullet (e \doteq a \wedge p \wedge x \doteq m \wedge q)\ \{\!|S|\!\}\ (q[x := m] \wedge r.a))$

□

We have explained the need for auxiliary variables when specifying programs and procedures. In order to prove correctness of recursive procedures using auxiliary variables, Theorem 23 is not enough; the following theorem enables us to prove it.

Theorem 28. If for all $w \in W$, $p_w : B \to A \to \mathsf{Pred}$, $q : B \to A \to \mathsf{Pred}$ and $body : \mathsf{Proc}.A \to \mathsf{Proc}.A$ is monotonic, then the following Hoare rule is true

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

Proof. Using Theorem 23. $\square$

9.1. An example of Hoare proof

Using these rules we prove the correctness of the procedure for computing the binomial coefficient using similar arguments as in the refinement case, but reformulated in the context of Hoare proof rules.

If $q : \mathsf{Pred}$ and $a, b, d : \mathsf{Nat}$ than we assume

$$\begin{array}{l}(\lambda q, d, a, b, e, f, u \bullet q \wedge u \doteq d \wedge e \doteq a \wedge f \doteq b \wedge e \leq f < w)\\ \{\!|P|\!\}\\ \left(\lambda q, d, a, b, e, f, u \bullet q[u := d] \wedge u \doteq \binom{b}{a}\right)\end{array} \tag{12}$$

The logical variables q, d, a, b denote the procedure specification parameters and e, f, u denotes the procedure parameters. The predicate q in (12) specifies 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 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

```
    $q \wedge \mathsf{val}.u \doteq d \wedge e \doteq a \wedge f \doteq b \wedge e \leq f \doteq w$
    $\{\!|$
       Add.$(k, n, c).(e, f, \mathsf{val}.u)$ ; Add.$(x, y)$ ;
       if $k \doteq 0 \vee k \doteq n$ then
          $c := 1$
       else
          $P.(k-1, n-1, x)$ ; $P.(k, n-1, y)$ ; $c := x + y$
       fi ;
       Del.$(x, y)$ ; Del.$(k, n)$ ; Del.$c.u$
    $|\!\}$
    $q[u := d] \wedge \mathsf{val}.u \doteq \binom{b}{a}$
$\Leftarrow$  {Lemma 22 and Lemma 19}
    del.$(k, n, c)$; $q \wedge$ del.$(k, n, c)$; val.$u \doteq d \wedge$
    $k \doteq a \wedge n \doteq b \wedge k \leq n \doteq w$
    $\{\!|$
       Add.$(x, y)$ ;
       if $k \doteq 0 \vee k \doteq n$ then
          $c := 1$
       else
          $P.(k-1, n-1, x)$ ; $P.(k, n-1, y)$ ; $c := x + y$
       fi ;
       Del.$(x, y)$ ; Del.$(k, n)$
    $|\!\}$
    del.$c$; $(q[u := d]) \wedge \mathsf{val}.c \doteq \binom{b}{a}$
$\Leftarrow$  {Lemma 22 and $v = (x, y, k, n, c)$}
```

$$
\begin{array}{ll}
 & \mathsf{del}.v; q \wedge \mathsf{del}.v; \mathsf{val}.u \doteq d \wedge k \doteq a \wedge n \doteq b \wedge k \leq n \doteq w \\
 & \{\!|\\
 & \quad \mathsf{if}\ k \doteq 0 \vee k \doteq n\ \mathsf{then} \\
 & \qquad c := 1 \\
 & \quad \mathsf{else} \\
 & \qquad P.(k-1, n-1, x) \ ; \ P.(k, n-1, y) \ ; \ c := x + y \\
 & \quad \mathsf{fi}\ ; \\
 & |\!\} \\
 & \mathsf{del}.v; (q[u := d]) \wedge \mathsf{val}.c \doteq \binom{b}{a} \\
\Leftarrow & \{\text{if statement correctness: case } k \neq 0 \wedge k \neq n\} \\
 & \mathsf{del}.v; q \wedge \mathsf{del}.v; \mathsf{val}.u \doteq d \wedge k \doteq a \wedge n \doteq b \wedge 0 < k < n \doteq w \\
 & \{\!|\ P.(k-1, n-1, x) \ ; \ P.(k, n-1, y) \ ; \ c := x + y\ |\!\} \\
 & \mathsf{del}.v; (q[u := d]) \wedge \mathsf{val}.c \doteq \binom{b}{a} \\
\Leftarrow & \{\text{Lemma 25, for all } d' \in \mathsf{T}.x\} \\
 & (\mathsf{del}.v; q \wedge \mathsf{del}.v; \mathsf{val}.u \doteq d \wedge k \doteq a \wedge n \doteq b \wedge 0 < k < n \doteq w) \wedge \\
 & \mathsf{val}.x \doteq d' \wedge k-1 \doteq a-1 \wedge n-1 \doteq 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 \doteq \binom{b}{a} \\
\Leftarrow & \{\text{Lemma 8 and Assumption (12)}\} \\
 & (\mathsf{del}.v; q \wedge \mathsf{del}.v; \mathsf{val}.u \doteq d \wedge k \doteq a \wedge n \doteq b \wedge 0 < k < n \doteq w)[x := d'] \wedge \\
 & \mathsf{val}.x \doteq \binom{b-1}{a-1} \\
 & \{\!|\ P.(k, n-1, y)\ |\!\} \\
 & \mathsf{del}.v; (q[u := d]) \wedge \mathsf{val}.x + \mathsf{val}.y \doteq \binom{b}{a} \\
\Leftarrow & \{\text{Assumption (12) and weakening the precondition}\} \\
 & \mathsf{del}.v; q \wedge \mathsf{del}.v; \mathsf{val}.u \doteq d \wedge 0 < a < b \wedge \\
 & \mathsf{val}.x \doteq \binom{b-1}{a-1} \wedge \mathsf{val}.y \doteq \binom{b-1}{a} \\
 & \subseteq \\
 & \mathsf{del}.v; (q[u := d]) \wedge \mathsf{val}.x + \mathsf{val}.y \doteq \binom{b}{a} \\
= & \{\text{logic}\} \\
 & \mathsf{true}
\end{array}
$$

Using Theorem 23 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

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

Using the elimination of auxiliary variables, Theorem 27, we obtain for all $e, f \in \mathsf{NatExp}$, $u \in \mathsf{NatVar}$, and $q \in \mathsf{Pred}$ that

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

and using Theorem 24 we obtain for all $e, f \in \mathsf{NatExp}$, $u \in \mathsf{NatVar}$ the refinement

$$\{e \leq f\} \ ; \left[u := o \mid o \doteq \binom{f}{e}\right] \sqsubseteq \mathsf{comb}(e, f, u)$$

10. Conclusions

We have introduced new program constructs for adding and deleting program variables and 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 recursive procedures using this semantics. We have also given a refinement rule for introduction of recursive procedure calls and, based on it, 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.

We introduced a special form of Hoare specification statement which alone is sufficient to fully specify a program or a procedure. Using this specification the Hoare consequence rule is enough for the adaptation of the procedure specification to any context. We proved that this specification statement is equivalent to a refinement specification.

Having only value and value–result parameters does not seem to be a major drawback. According to [Don76], in the absence of aliasing, call by reference is equivalent to call by value–result.

Although our calculus does not allow program variables to change their types in local scopes and implements only dynamic scoping we have showed how to handle programming languages which do not have these restrictions by choosing a suitable semantic map of syntactic programs to predicate transformers.

Many procedure proof rules in the literature are mixing the procedure call with the procedure parameters. We have separated these concerns as in [Mor88a], 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.

However, we have not investigated computability issues or the completeness of our calculus either theoretically or practically. So there might be problems for which one would need more rules in addition to the ones we have introduced.

Acknowledgments. We thank the anonymous referees for their useful comments and suggestions which lead to an improvement of the presentation of this paper.

References

[Bac78] R. J. Back. *On the correctness of refinement in program development.* PhD thesis, Department of Computer Science, University of Helsinki, 1978.

[Bac80] R. J. Back. *Correctness preserving program refinements: proof theory and applications*, volume 131 of *Mathematical Centre Tracts.* Mathematisch Centrum, Amsterdam, 1980.

[BMW89] A. Bijlsma, P. A. Matthews, and J. G. Wiltink. A sharp proof rule for procedures in wp semantics. *Acta Inform.*, 26(5):409–419, 1989.

[BvW98] R.J. Back and J. von Wright. *Refinement Calculus. A systematic Introduction.* Springer, 1998.

[BvW03] R.J. Back and J. von Wright. Compositional action system refinement. *Formal Aspects of Computing*, 15(2–3):103–117, November 2003.

[Chu40] A. Church. A formulation of the simple theory of types. *J. Symbolic logic*, 5:56–68, 1940.

[CvW02] O. Celiku and J. von Wright. Theorem prover support for precondition and correctness calculation. In *4th International Conference on Formal Engineering Methods*, volume 2495 of *Lecture Notes in Computer Science*, pages 299–310. Springer-Verlag, October 2002.

[Dij75] E.W. Dijkstra. Guarded commands, nondeterminacy and formal derivation of programs. *Comm. ACM*, 18(8):453–457, 1975.

[Dij76] 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.

[Don76] J.E. Donahue. *Complementary definitions of programming language semantics.* Springer-Verlag, Berlin, 1976. Lecture Notes in Computer Science, Vol. 42.

[GL80] D. Gries and G. Levin. Assignment and procedure call proof rules. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 2(4):564–579, 1980.

[Gor88] 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.

[Hes99] W.M. Hesselink. Predicate transformers for recursive procedures with local variables. *Formal Aspect of Computing*, 11:616–336, 1999.

[Hoa69] C.A.R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–580, 1969.

[Kle98] T. Kleymann. *Hoare Logic and VDM: Machine-Checked Soundness and Completeness Proofs.* PhD thesis, Laboratory for Foundations of Computer Science, University of Edinburgh, 1998.

[Kle99] T. Kleymann. Hoare logic and auxiliary variables. *Formal Aspect of Computing*, 11:541–566, 1999.

[Lai00] L. Laibinis. *Mechanised Formal Reasoning About Modular Programs.* PhD dissertation, Turku Centre for Computer Science, April 2000.

[LvW01] L. Laibinis and J. von Wright. Specification variables: Between the angel and the demon. Technical Report 412, TUCS - Turku Centre for Computer Science, June 2001.

[Mor87] J.M. Morris. A theoretical basis for stepwise refinement and the programming calculus. *Science of Computer Programming*, 9(3):287–306, 1987.

[Mor88a] C. Morgan. Procedures, parameters, and abstraction: separate concerns. *Sci. Comput. Programming*, 11(1):17–27, 1988.

[Mor88b] C. Morgan. The specification statement. *ACM Trans. Program. Lang. Syst.*, 10(3):403–419, 1988.

[Mor90a] Carroll Morgan. *Programming from specifications.* Prentice-Hall, Inc., 1990.

[Mor90b] J. M. Morris. Programs from specifications. In *Formal development programs and proofs*, pages 81–115. Addison-Wesley Longman Publishing Co., Inc., 1990.

[Nau01] D.A. Naumann. Calculating sharp adaptation rules. *Inform. Process. Lett.*, 77(2-4):201–208, 2001.

[Old83] E.R. Olderog. On the notion of expressiveness and the rule of adaptation. *Theoret. Comput. Sci.*, 24(3):337–347, 1983.

[OSRSC01] 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.

[Rey81a] J.C. Reynolds. *The Craft of Programming.* Prentice-Hall Inc., London, 1981.

[Rey81b] J.C. Reynolds. The essence of ALGOL. In *Algorithmic languages (Amsterdam, 1981)*, pages 345–372. North-Holland, Amsterdam, 1981.

[Sta98] M. Staples. *A Mechanised Theory of Refinement.* PhD dissertation, Computer Laboratory, University of Cambridge, November 1998.

[Sta99] 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.

[Sta00] M. Staples. Interfaces for refining recursion and procedures. *Formal Aspect of Computing*, 12:372–391, 2000.

[Tar55] A. Tarski. A lattice-theoretical fixpoint theorem and its applications. *Pacific J. Math.*, 5:285–309, 1955.

[vO99] 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.