Reasoning about Recursive Procedures with Parameters

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

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

ABSTRACT

In this paper we extend the model of program variables from the Refinement Calculus [2] 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 [11].

Categories and Subject Descriptors

F.3.1 [**Specifying and Verifying and Reasoning about Programs**]; F.3.2 [**Semantics of Programming Languages**]

Keywords

Hoare logic, predicate transformer semantics, recursive procedures, refinement calculus

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 [12, 22, 23, 16, 4, 10], some used in mechanical verification, are based on states represented as tuples, with one component for each program variable. When accessing or updating a specific program variable we should access or update the corresponding component in the tuple. The problem becomes even more complicated when there are 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 [2], 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. 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 [2]. 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 this paper is to extend the axiomatic model of program variables from [2] with one additional program construct $\mathsf{del}.x.\sigma$, which deletes the local variable x from the state σ. We give a predicate transformer semantics [7, 8] for recursive procedures with value and value–result parameters and local variables. We also introduce a refinement rule and a Hoare [13] 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 about the Refinement Calculus. 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. We define substitution and freeness in the context of these program expressions. Using the prim-

MERλIN'03, August 26, 2003, Uppsala, Sweden

itive functions defined in Section 4 we define in Section 6 some program statements and state 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.

Because of the space limitation we had to leave out most of the proofs. A more detailed version of the paper, containing proofs for all nontrivial results, can be found in [1].

2. RELATED WORK

Back and von Wright [2] 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 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.

Reynolds [20] 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 [2]. However, no assumptions about the behavior of the program variables are made.

In [2], 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 [3], Back and von Wright give an improved version of their program variable model [2]. 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 [21, 22, 23] 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. 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 [12] 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 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 [14] 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 [14] 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 [25] 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 [19] 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 causes the state space to change.

In [19] 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 may 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 ($x \neq y$).

3. PRELIMINARIES

We use higher–order logic [5] as the underlying logic. In this section we recall some facts about the Refinement Calculus hierarchy [2]. We also use some basic facts about complete lattices and fixpoints [6].

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 use the notation $f.\sigma$ for *function application*, $f\ ;\ g$ for *forward functional composition*, i.e. $(f\ ;\ g).\sigma = g.(f.\sigma)$, and id for the identity function.

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 $\mathsf{Pred}.X$. We have that the structure $\langle \mathsf{Pred}.X, \cup, \cap, \neg, \mathsf{false}, \mathsf{true}\rangle$ 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 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 relations. We can map functions to relations. If f is a state transformer then we define $|f| \in \mathsf{Rel}.\Sigma$ by $|f|.\sigma.\sigma' = (f.\sigma = \sigma')$.

A *predicate transformer* is a function that maps predicates over Σ to predicates over Σ. *Programs* are modeled by monotonic predicate transformers, denoted by $\mathsf{MTran}.\Sigma$. 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. We denote by $S\ ;\ T$ the functional composition of predicate transformers and by skip the identity function on $\mathsf{Pred}.\Sigma$. All operations defined on predicate transformers are interpreted as operations on programs.

$S \sqsubseteq T$ – the refinement relation.
$S\ ;\ T$ – the sequential composition.
$S \sqcap T$ – the demonic choice.
$S \sqcup T$ – the angelic choice.

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

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

where p, q are predicates, R is a state relation, f is a function, and S, T are predicate transformers. We define the conditional statement by:

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

We recall Knaster–Tarski's fixpoint theorem [24] for complete lattices. We will use it to give semantics to recursive procedures.

THEOREM 1. *Assume that L is a complete lattice and $f : L \to L$ is a monotonic function, then f has a least fixpoint (denoted μf).*

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

4. PROGRAM VARIABLES

We denote by Var the type of *program variables*. For any program variable $x \in \mathsf{Var}$, $\Gamma.x$ is its type. We denote by nat the type of natural numbers and by NatVar the program variables of type nat.

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.

The functions $\mathsf{val}.x$ and $\mathsf{set}.x$ are the same as the ones defined in [2]. 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 [2].

To show that the axioms are consistent, we will give a model in which they are satisfied. Let Σ be the cartesian product $\prod\{x \in \mathsf{Var} \mid (\Gamma.x)^\omega\}$. 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 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 of s. It is easy to prove that all axioms (a) – (i) are satisfied.

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 denote 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 $x, y, z, \ldots \in \mathsf{Var}$.

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 $\Gamma.\epsilon = \{\epsilon\}$ and $\Gamma.(y \cdot z) = \Gamma.y \cdot \Gamma.z$. 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 2. *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 all variables from x are distinct. We denote by $x \cap y$ the set of program variables that occur in both x and y.

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

LEMMA 3. *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) $x \cap y = \emptyset \Rightarrow \mathsf{set}.x.a \ ; \ \mathsf{del}.y = \mathsf{del}.y \ ; \ \mathsf{set}.x.a$

5. PROGRAM EXPRESSIONS

We define a *program expression* of some type A as being any function from Σ to A. A program expression of type bool is called *boolean program expression*. 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 4. *If* $S, T \in \mathsf{MTran}$ *and* $e : \Sigma \to A$ *is a program expression, then*
$(\forall a \in A \bullet \{e = a\} \ ; \ S \sqsubseteq \{e = a\} \ ; \ T) \Leftrightarrow S \sqsubseteq T$.

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

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

We define a subclass of program expressions that depend only on the current values of the program variables. Two states σ and σ' are *val-equivalent*, denoted $\sigma \sim \sigma'$, if $(\forall x \bullet \mathsf{val}.x.\sigma = \mathsf{val}.x.\sigma')$. We call a program expression e, *val-determined* if for all σ and σ' we have $\sigma \sim \sigma' \Rightarrow e.\sigma = e.\sigma'$.

6. PROGRAM STATEMENTS

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

If $x,\ y \in \mathsf{VarList}$ have the same type and $e : \Sigma \to \Gamma.x$, then we define:

Multiple assignment:
$(x := e).\sigma = \mathsf{set}.x.(e.\sigma).\sigma$

Add local variables:
$\mathsf{add}.x.\sigma.\sigma' = (\sigma = \mathsf{del}.x.\sigma')$

Add & initialize local variables:
$\mathsf{add}.x.e.\sigma.\sigma' = (\sigma = \mathsf{del}.x.\sigma') \wedge (\mathsf{val}.x.\sigma' = e.\sigma)$

Save & delete local variables:
$\mathsf{del}.x.y.\sigma = \mathsf{set}.y.(\mathsf{val}.x.\sigma).(\mathsf{del}.x.\sigma)$

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 5. *If* $x \in \mathsf{VarList}$ *then*
(i) $\mathsf{add}.x \ ; \ \mathsf{del}.x = \mathsf{skip}$
(ii) $\mathsf{var}.x \Rightarrow \mathsf{add}.x.e \ ; \ \mathsf{del}.x = \mathsf{skip}$

LEMMA 6. *If* $x, y \in \mathsf{VarList}$ *then*
(i) $x := e \ ; \ \mathsf{del}.x = \mathsf{del}.x$
(ii) $(x \cap y = \emptyset) \wedge (e \ \textit{is}\ \mathsf{del}.x\textit{–free}) \Rightarrow$
$y := e \ ; \ \mathsf{del}.x = \mathsf{del}.x \ ; \ y := e$
(iii) $\mathsf{var}.x \wedge (e \ \textit{is}\ \mathsf{del}.x\textit{–free}) \Rightarrow$
$x := e \ ; \ \mathsf{del}.x.y = \mathsf{del}.x \ ; \ y := e$

LEMMA 7. *If* $x \in \mathsf{VarList}$ *and* p *is a boolean expression then*
(i) $(p \ \textit{is}\ \mathsf{del}.x\textit{-free}) \Rightarrow \{p\} \ ; \ \mathsf{add}.x = \mathsf{add}.x \ ; \ \{p\}$
(ii) $(p \ \textit{is}\ \mathsf{del}.x\textit{-free}) \Rightarrow \{p\} \ ; \ \mathsf{add}.x.e = \mathsf{add}.x.e \ ; \ \{p\}$
(iii) $\mathsf{var}.x \wedge (p \ \textit{is a}\ \mathsf{val}\textit{-determined}) \Rightarrow$
$\{p[x := e]\} \ ; \ \mathsf{add}.x.e = \mathsf{add}.x.e \ ; \ \{p\}$

EXAMPLE 8. *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 expressions such that* h *is* $\mathsf{del}.(k, n, c, x, y)$*–free, then we have the following derivation for all* $u \in \mathsf{NatVar}$

$$
\begin{array}{ll}
 & \mathsf{add}.(k, n, c).e \ ; \ \mathsf{add}.(x, y) \ ; \\
 & x \ := \ f \ ; \ y \ := \ g \ ; \ c \ := \ h \ ; \\
 & \mathsf{del}.(x, y) \ ; \ \mathsf{del}.(k, n) \ ; \ \mathsf{del}.c.u \\
= & \{\text{Lemma 2, Lemma 5 and Lemma 6}\} \\
 & u \ := \ h
\end{array}
$$

7. PROCEDURES

In this section we show how recursive procedures with value and/or result parameters can be modeled using the program constructs add and del.

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. 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 $(\mathsf{Proc}, \sqsubseteq, \sqcup, \sqcap)$ is a complete lattice. We call $\sqsubseteq$ the procedure refinement relation, $\sqcap$ the demonic choice, $\sqcup$ the angelic choice, and ; the sequential composition of procedures.

A general nonrecursive procedure declaration is:

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

where *body* is a program that does not contain any recursive call. The procedure declaration (1) is an abbreviation for the following formal definition:

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

where the variables e and z do not occur free in x, y and *body*.

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

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 $\mu\, body$.

EXAMPLE 9. *We give a recursive procedure that computes the binomial coefficient*

$$\left(\begin{array}{c} n \\ k \end{array} \right) = \frac{n!}{k! \cdot (n-k)!} = \left(\begin{array}{c} n-1 \\ k-1 \end{array} \right) + \left(\begin{array}{c} n-1 \\ k \end{array} \right)$$

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

$$
\begin{array}{l}
\quad body.S.(e, f, u) \\
= \\
\quad \mathsf{add}.(k, n, c).(e, f, \mathsf{val}.u)\ ;\ \mathsf{add}.(x, y)\ ; \\
\quad \mathsf{if}\ k = 0 \vee k = n\ \mathsf{then} \\
\qquad c := 1 \\
\quad \mathsf{else} \\
\qquad S.(k-1,\ n-1,\ x)\ ;\ S.(k,\ n-1,\ y)\ ;\ c := x + y \\
\quad \mathsf{fi}\ ; \\
\quad \mathsf{del}.(x, y)\ ;\ \mathsf{del}.(k, n)\ ;\ \mathsf{del}.c.u
\end{array}
\tag{2}
$$

Then *body* is a monotonic function. If $\mathsf{comb} = \mu\ body$, then we can prove

$$
\begin{array}{c}
\forall u, e, f \bullet \{e \le f\}\ ;\ u := \begin{pmatrix} f \\ e \end{pmatrix} = \\
\{e \le f\}\ ;\ \mathsf{comb}(e, f, u)
\end{array}
\tag{3}
$$

To prove (3) it is enough (by Lemma 4) to show

$$
\begin{array}{l}
\quad \{a \le b \wedge e = a \wedge f = b\}\ ;\ u := \begin{pmatrix} f \\ e \end{pmatrix} \\
= \\
\quad \{a \le b \wedge e = a \wedge f = b\}\ ;\ \mathsf{comb}(e,\ f,\ u)
\end{array}
\tag{4}
$$

for all $a, b \in \mathsf{nat}$, $u \in \mathsf{NatVar}$ and $f, g \in \mathsf{NatExp}$. We split the proof of (4) in two cases: $a > b$ and $a \le b$. The case $a > b$ is trivial because $\{\mathsf{false}\}\ ;\ S = \{\mathsf{false}\}$.

We prove the second case by induction on b. We assume $b > 0$, and (4) true for $b - 1$ and all $a \in \mathsf{nat}$, $u \in \mathsf{NatVar}$, $f, g \in \mathsf{NatExp}$. Moreover we assume that $0 < a < b$. Then we have:

$$
\begin{array}{ll}
 & \{a \le b \wedge e = a \wedge f = b\}\ ;\ \mathsf{comb}(e, f, u) \\
= & \{\text{Assumptions, } \mathsf{comb} = \mu\ \mathsf{body}, \text{ and Lemma 7}\} \\
 & \{0 < a < b \wedge e = a \wedge f = b\}\ ; \\
 & \mathsf{add}.(k, n, c).(e, f, \mathsf{val}.u)\ ;\ \mathsf{add}.(x, y)\ ; \\
 & \{0 < a < b \wedge k = a \wedge n = b\}\ ; \\
 & \mathsf{if}\ k = 0 \vee k = n\ \mathsf{then} \\
 & \quad c := 1 \\
 & \mathsf{else} \\
 & \quad \mathsf{comb}(k-1, n-1, x)\ ;\ \mathsf{comb}(k, n-1, y)\ ; \\
 & \quad c := x + y \\
 & \mathsf{fi}\ ; \\
 & \mathsf{del}.(x, y)\ ;\ \mathsf{del}.(k, n)\ ;\ \mathsf{del}.c.u \\
= & \{\text{by induction hypothesis using Refinement [2]}\} \\
 & \{0 < a < b \wedge e = a \wedge f = b\}\ ; \\
 & \mathsf{add}.(k, n, c).(e, f, \mathsf{val}.u)\ ;\ \mathsf{add}.(x, y)\ ; \\
 & x := \begin{pmatrix} b-1 \\ a-1 \end{pmatrix}\ ;\ y := \begin{pmatrix} b-1 \\ a \end{pmatrix}\ ;\ c := \begin{pmatrix} b \\ a \end{pmatrix}\ ; \\
 & \mathsf{del}.(x, y)\ ;\ \mathsf{del}.(k, n)\ ;\ \mathsf{del}.c.u \\
= & \{\text{Example 8 and Refinement [2]}\}
\end{array}
$$

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

The remaining cases are similar. This concludes the proof of (3).

8. REFINEMENT RULE FOR INTRODUCTION OF RECURSIVE PROCEDURE CALLS

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.

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 $v \le w \Rightarrow p_v \subseteq p_w$. We refer to this collection as a collection of *ranked parametric predicates*. We define

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

THEOREM 10. *(Theorem 20.1, [2]) If body* : $\mathsf{Proc}.A \to \mathsf{Proc}.A$ *is monotonic,* $\{p_w \mid w \in W\}$ *is a collection of ranked parametric predicates, and* P *is a procedure over* A*, then*

$$
\begin{array}{ll}
 & (\forall w \in W \bullet \{p_w\}\ ;\ P \sqsubseteq body.(\{p_{<w}\}\ ;\ P)) \\
\Rightarrow & \\
 & \{p\}\ ;\ P \sqsubseteq \mu\ body
\end{array}
$$

We will show how this theorem can be applied to obtain by refinement the recursive procedure defined in the previous section. We take *body* given by (2) and

$$
\begin{array}{l}
W = \mathsf{nat} \\
p_a = (\lambda(e,\ f,\ u) \bullet e \le f \wedge f = a) \\
P = \left(\lambda(e,\ f,\ u) \bullet u := \begin{pmatrix} f \\ e \end{pmatrix}\right)
\end{array}
$$

To prove that

$$
\{e \le f\}\ ;\ u := \begin{pmatrix} f \\ e \end{pmatrix} \sqsubseteq \mathsf{comb}(e,\ f,\ u)
\tag{5}
$$

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

$$
\{p_a\}\ ;\ P \sqsubseteq body.(\{p_{<a}\}\ ;\ P)).
$$

This can be proved using refinement and equality rules proved in this paper for the program constructs we have introduced. A detailed proof of this fact can be found in [1].

The proof of (5) using Theorem 10 is simpler than the proof of (3). We do not need induction anymore as the recursion introduction theorem has the induction built in. The refinement relation (5) is weaker than (3) but it is strong enough in practice.

9. HOARE TOTAL CORRECTNESS RULE FOR RECURSIVE PROCEDURES

We will give a Hoare rule for proving the correctness of recursive procedures based on the refinement rule given by Theorem 10. If p and q are predicates and S is a program, then a *Hoare triple* is denoted by $p \,\{\!|S|\!\}\, q$ and is true if and only if $p \subseteq S.q$.

If p, q are parametric predicates over A and P is a procedure over A then a *parametric Hoare triple* is denoted by $p \,\{\!|P|\!\}\, q$ and is true if and only if for all $a \in A$ the Hoare triple $p.a \,\{\!|P.a|\!\}\, q.a$ is true. We also define for a parametric predicate q the parametric relation over A, $\hat{q} = (\lambda a\, \sigma \bullet q.a)$.

LEMMA 11. *(Lemma 17.4, [2]) $p \,\{\!|P|\!\}\, q$ is true if and only if $\{p\} \,;\, [\hat{q}] \sqsubseteq P$.*

THEOREM 12. *If $\{p_w \mid w \in W\}$ and body are as in Theorem 10, and q is a parametric predicate over 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.

$$\begin{array}{ll} & p \,\{\!|\mu\, body|\!\}\, q \\ = & \{\text{Lemma 11}\} \\ & \{p\} \,;\, [\hat{q}] \sqsubseteq \mu\, body \\ \Leftarrow & \{\text{Theorem 10}\} \\ & (\forall w \bullet \{p_w\} \,;\, [\hat{q}] \sqsubseteq body.(\{p_{<w}\} \,;\, [\hat{q}]) \\ = & \{\text{Lattice properties}\} \\ & (\forall w,\ P \bullet \{p_{<w}\} \,;\, [\hat{q}] \sqsubseteq P \Rightarrow \{p_w\} \,;\, [\hat{q}] \sqsubseteq body.P \\ = & \{\text{Lemma 11}\} \\ & (\forall w,\ P \bullet p_{<w} \,\{\!|P|\!\}\, q \Rightarrow p_w \,\{\!|body.P|\!\}\, q) \end{array}$$

□

Similarly, we can also derive correctness rules for add and del. Using these rules we would be able to prove correctness of the procedure for computing the binomial coefficient using the same arguments we used, but reformulated in the context of Hoare proof rules.

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 [9], 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 and they are almost as simple as the assignment rules.

11. REFERENCES

[1] R.J. Back and V. Preoteasa. Reasoning about recursive procedures with parameters. Technical Report 500, TUCS - Turku Centre for Computer Science, January 2003.

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

[3] 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.

[4] 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.

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

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

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

[8] 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.

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

[10] 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.

[11] 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.

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

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

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

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

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

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

[22] 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.

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

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

[25] 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.