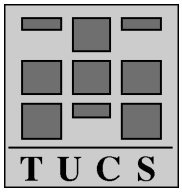
Refinement of recursive procedures with parameters in PVS

Viorel Preoteasa

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



Turku Centre for Computer Science
TUCS Technical Report No 596
March 2004
ISBN 952-12-1319-1
ISSN 1239-1891

Abstract

We present a shallow embedding in PVS of a predicate transformer semantics of an imperative language suitable for reasoning about recursive procedures with parameters and local variables. We use the PVS dependent type mechanism for implementing program variables of different types. We use an uninterpreted state space and define the program variables behavior by means of certain tree functions that are supposed to satisfy some axioms. Unlike in the implementations mentioned in the literature, we do not need to change the state space when adding local variables or procedure parameters.

1 Introduction

Refinement calculus developed by Back [1] and later by Morris [14], and Morgan [13], is a generalization of the predicate transformer semantics introduced by Dijkstra [8]. Mechanizations of programming logics, in general, and refinement calculus, in particular, have been widely studied [9, 18, 19, 10, 11, 12, 6].

An important issue when mechanizing a programming logic is how to implement the state. For the mechanization to be of any use the program variables should be able to be of different types. Many implementations [5, 12, 6] solve this problem by representing states as tuples, with one component for each program variable. However, since program variables are identified with the projection functions of the tuple, manipulating programs with many variables becomes very inefficient due to the terms becoming very large. Perhaps a more important drawback of such a representation is the need to add or remove components from the tuple whenever local variables are added or deleted. As a result, when the programming language has recursive procedures with parameters, recursive calls need to be made in a different state space; this adds much complexity to the calculus. In some mechanizations [19] this problem is solved by considering the predicates transformers over all possible state spaces. Others avoid the problem altogether by handling only parameterless procedures [11, 20].

We provide a PVS [17] implementation based on the theory developed in [2] where the state model allows the handling of recursive procedures with parameters without requiring any state changes. As a result, the procedure semantics and refinement rules become quite simple. In particular, the rule for introducing recursive procedures has no side conditions and does not involve the parameters or local variables. The rules for local variables and procedure parameters are almost as simple as the ordinary assignment rule, and their side conditions are decidable.

We consider the state space to be an uninterpreded type State and define the behavior of the program variables using tree primitive functions val(x), set(x), and del(x), where x is a program variable. The term val(x)(s) gives the value of x in the state s, set(x)(a)(s) gives the state obtained from s by setting the value of x to a, and del(x)(s) gives the state obtained from s by deleting the local variable x. The function del(x) pops a value from the computation stack and assigns it to x. The behavior of these functions is introduced by a set of axioms, which are proved consistent using the theory interpretation mechanism of PVS [16]. The program variables themselves can have different types (among those listed in a customized ProgVar datatype).

The resulting mechanized theory is very general: adding new types for program variables requires few changes (related to ProgVar); the rest can be

reused without additional proofs. For example we can easily add program variables ranging over bounded integers, floating point numbers, or have a rich algebra of data types.

In Section 2 we present some facts about PVS. Sections 3 and 4 introduce the program variables representation and the program expressions. In Section 5 the theory of complete lattices is introduced, and in Section 6 we build the complete lattice of monotonic predicate transformers. We introduce the assignment and local variables rule in Section 7. Section 8 presents the semantics for procedures and introduces the refinement rule for recursive procedures. In Section 9 we show how this theory can be used to refine a recursive procedure with parameters from its specification. Section 10 contains concluding remarks and comments on future work.

2 Preliminaries PVS

PVS is a specification and verification system whose language is based on classical, typed higher-order logic [7]. The base types include uninterpreted types that may be introduced by the user, and built-in types such as the booleans (bool), naturals (nat), integers (int), and reals (real); the type-constructors include functions, sets, tuples, disjoint unions, records, and abstract data types [15], such as lists and binary trees.

PVS specifications are organized into parameterized theories that may contain assumptions, definitions, axioms, and theorems. Definitions are guaranteed to provide conservative extension. PVS also provides a theory interpretation mechanism that can be used to prove consistency of theories that contains axioms.

The PVS theorem prover provides a collection of powerful primitive inference procedures that are applied interactively under user guidance within a sequent calculus framework. The primitive inferences include propositional and quantifier rules, induction, rewriting, and decision procedures for linear arithmetic. User-defined procedures can combine these primitive inferences to yield higher-level proof strategies.

We use some mathematical simplified notations and symbols instead of the PVS syntax. We use $\wedge$, $\vee$, $\neg$, $\forall$, $\exists$, $\Rightarrow$, $\leq$, $\rightarrow$, $\emptyset$ instead of and, or, not, forall, exists, =>, <=, ->, emptyset. We also write $\forall \mathsf{x} : \mathsf{T} \bullet \mathsf{b}$ instead of the PVS syntax forall (x : T) : b. We use a sanserif font for identifiers.

PVS language supports predicate subtypes that increase the expressivity of specifications. For a PVS type T the type of predicates (sets) over T is denoted by set[T] or pred[T] and is equal with the type of all functions from T to bool. For a predicate P : pred[T] over some type T, PVS has primitives

to create a subtype of T based on the predicate P. The new type is denoted by (P) and PVS knows that the elements of (P) are elements of T too. If we assert that some element of T has the type (P) than a type condition constrain is generated. PVS tries to automatically prove such constrains using built-in tactics; if the attempt fails, the user needs to prove them interactively.

For two PVS types T, T' the type of all function from T to T' is denoted by $\mathsf{T} \to \mathsf{T}'$, the type of the disjoint union is denoted by $[\mathsf{T} + \mathsf{T}']$, and the type of the cartesian product is denoted $[\mathsf{T}, \mathsf{T}']$. For the cartesian product type the functions $\mathsf{proj1} : [[\mathsf{T}, \mathsf{T}'] \to \mathsf{T}]$ and $\mathsf{proj2} : [[\mathsf{T}, \mathsf{T}'] \to \mathsf{T}']$ are the projection functions. For the disjoint union type the function $\mathsf{in1} : [\mathsf{T} \to [\mathsf{T} + \mathsf{T}']]$ is the inclusion function, and $\mathsf{in?1}$ is a predicate on $[\mathsf{T} + \mathsf{T}']$, which is true for the elements in the first component of the union. $\mathsf{out1} : [(\mathsf{in?1}) \to \mathsf{T}]$ is a bijective function and its inverse is $\mathsf{in1}$ co-restricted to $(\mathsf{in?1})$. Similarly the functions $\mathsf{in2?}$, $\mathsf{in2}$, and $\mathsf{out2}$ correspond to the second component of the union.

PVS also supports dependent types, i.e. functions can have the type of the result depend on their arguments or the argument types can depend on the previous arguments. However the dependent types can only be subtypes a given common type or user defined uninterpreted types. As an example of a dependent type, consider the function that computes the binomial coefficient "n choose k". Its second argument k depends on n, $\mathsf{k} \leq \mathsf{n}$:

$$\mathsf{comb}(\mathsf{n} : \mathsf{nat},\ \mathsf{k} : \mathsf{upto}(\mathsf{n})) = \ldots$$

We also use in this paper the PVS abstract data type of lists over a type T, $\mathsf{list}[\mathsf{T}]$. The empty list is $\mathsf{null} : \mathsf{list}[\mathsf{T}]$, and if $\mathsf{a} : \mathsf{T}$ and $\mathsf{x} : \mathsf{list}[\mathsf{T}]$ then $\mathsf{cons}(\mathsf{a}, \mathsf{x}) : \mathsf{list}[\mathsf{T}]$ is the list that has head a and tail x.

Instead of using the PVS style for writing recursive definitions we use a mathematical notation that can be straightforwardly translated to PVS. For example we define the sum of the elements of a list of numbers by:

$$\mathsf{sum}(\mathsf{null}) : \mathsf{nat} = 0$$
$$\mathsf{sum}(\mathsf{cons}(\mathsf{a}, \mathsf{x})) : \mathsf{nat} = \mathsf{a} + \mathsf{sum}(\mathsf{x})$$

instead of

$$\begin{array}{l}\mathsf{sum}(\mathsf{x} : \mathsf{list}[\mathsf{nat}]) : \mathsf{recursive\ nat} = \\ \quad \mathsf{cases\ x\ of} \\ \quad\quad \mathsf{null} :\ 0, \\ \quad\quad \mathsf{cons}(\mathsf{a}, \mathsf{y}) :\ \mathsf{a} + \mathsf{sum}(\mathsf{y}) \\ \quad \mathsf{endcases} \\ \quad \mathsf{measure\ length}(\mathsf{x})\end{array}$$

If $\mathsf{f} : [\mathsf{T} \to \mathsf{T}']$ and $\mathsf{X} : \mathsf{pred}[\mathsf{T}]$ then $\mathsf{image}(\mathsf{f}, \mathsf{X})$ is a predicate on T' and is true on the elements of T' that are in the image of X.

3 Program variables representation

3.1 Program variables types

Our program variables can be of type real, int, nat, or bool. The behavior of program variables is defined using the tree functions set, val, and del which, in turn, are defined using the PVS dependent type mechanism. In order to be able to define them we need to create our own types Real, Int, Nat, and Bool, so that they are all subtypes of a given type. So we define in PVS:

$$\begin{aligned}&\mathsf{ProgType} : \mathsf{type}+ = [\mathsf{real} + \mathsf{bool}]\\&\mathsf{Bool}? : [\mathsf{ProgType} \to \mathsf{bool}] = \mathsf{in}?2\\&\mathsf{Bool} : \mathsf{type}+ = (\mathsf{Bool}?)\end{aligned}$$

ProgType is the type of the disjoint union of the PVS types real and bool. Bool? is a predicate on ProgType which is true only for elements of the second component of ProgType. In a similar manner we define the predicates Nat?, Int? and Real? and the corresponding types

$$\mathsf{Nat} \subseteq \mathsf{Int} \subseteq \mathsf{Real} \subseteq \mathsf{ProgType}.$$

We also lift some operations on bool, real, int, and nat to our Bool, Real, Int, and Nat, for example:

$$\begin{aligned}&\mathsf{false} : \mathsf{Bool} = \mathsf{in2}(\mathsf{false})\\&(\mathsf{x} - \mathsf{y}) : \mathsf{Real} = \mathsf{in2}(\mathsf{out1}(\mathsf{x}) - \mathsf{out1}(\mathsf{y}))\\&\mathsf{minus\_nat} : \mathsf{judgment}\ -(\mathsf{k}, \mathsf{n} : \mathsf{Nat})\ \mathsf{has\_type}\ \mathsf{Int}\end{aligned}$$

3.2 Program variables

We define the type of all program variables as:

```
ProgVar : datatype
  begin
    b(name : string) : BoolVar?
    r(name : string) : RealVar?
    i(name : string) : IntVar?
    n(name : string) : NatVar?
  end
```

For a program variable $\mathsf{x} : \mathsf{ProgVar}$ we define the predicate $\mathsf{T?(x)}$ on ProgType to be Bool?, Real?, Int?, or Nat? depending on x being from (BoolVar?), (RealVar?), (IntVar?), or (NatVar?). We define the type of the program variable x, $\mathsf{T(x)}$ as the PVS type corresponding to the predicate T?(x). For two program variables x and y we define the predicate $\mathsf{SameType(x)(y)}$ to be true if and only if $\mathsf{T?(x)} = \mathsf{T?(y)}$. We denote by $\mathsf{elem(x)}$ some arbitrary but fixed element of type $\mathsf{T(x)}$.

3.3 Program variables axioms

Now we are able to introduce the functions for handling program variables and the axioms that define their properties. We define them in a parametric theory $\mathsf{ProgVarAxiom(State : type+)}$ whose parameter is the uninterpreted type State.

$$\mathsf{set} : [\mathsf{x} : \mathsf{ProgVar} \to [\mathsf{T(x)} \to [\mathsf{State} \to \mathsf{State}]]]$$
$$\mathsf{val} : [\mathsf{x} : \mathsf{ProgVar} \to [\mathsf{State} \to \mathsf{T(x)}]]$$
$$\mathsf{del} : [\mathsf{x} : \mathsf{ProgVar} \to [\mathsf{State} \to \mathsf{State}]]$$

The function set takes as parameters a program variable x, a value $\mathsf{a} : \mathsf{T(x)}$ and a state s and returns a state where the variables x is set to a. $\mathsf{val(x)(s)}$ returns the value of x in state s, and $\mathsf{del(x)(s)}$ deletes a value from the stack in state s and assigns it to x.

Instead of defining the type State and defining the functions set, val and del as operations on this type, we introduce the behavior of set, val and del with a collection of axioms:

$$\mathsf{var\_a} : \mathsf{axiom}\ \forall \mathsf{a} : \mathsf{T(x)} \bullet \mathsf{val(x)(set(x)(a)(s))} = \mathsf{a}$$
$$\mathsf{var\_b} : \mathsf{axiom}\ \forall \mathsf{a} : \mathsf{T(x)} \bullet \mathsf{x} \neq \mathsf{y} \Rightarrow \mathsf{val(y)(set(x)(a)(s))} = \mathsf{val(y)(s)}$$
$$\mathsf{var\_c} : \mathsf{axiom}\ \forall \mathsf{a}, \mathsf{b} : \mathsf{T(x)} \bullet \mathsf{set(x)(b)} \circ \mathsf{set(x)(a)} = \mathsf{set(x)(b)}$$
$$\mathsf{var\_d} : \mathsf{axiom}\ \forall \mathsf{a} : \mathsf{T(x)}, \mathsf{b} : \mathsf{T(y)} \bullet \mathsf{x} \neq \mathsf{y} \Rightarrow$$
$$\quad \mathsf{set.(x)(a)} \circ \mathsf{set(y)(b)} = \mathsf{set(y)(b)} \circ \mathsf{set(x)(a)}$$
$$\mathsf{var\_e} : \mathsf{axiom}\ \mathsf{set(x)(val(x)(s))(s)} = \mathsf{s}$$
$$\mathsf{var\_f} : \mathsf{axiom}\ \mathsf{del(x)}\ \text{ is surjective}$$
$$\mathsf{var\_g} : \mathsf{axiom}\ \mathsf{x} \neq \mathsf{y} \Rightarrow \mathsf{val(x)} \circ \mathsf{del(x)} = \mathsf{val(x)}$$
$$\mathsf{var\_h} : \mathsf{axiom}\ \mathsf{del(x)} \circ \mathsf{set(x)(a)} = \mathsf{del(x)}$$
$$\mathsf{var\_i} : \mathsf{axiom}\ \forall \mathsf{a} : \mathsf{T(x)} \bullet \mathsf{x} \neq \mathsf{y} \Rightarrow$$
$$\quad \mathsf{del(y)} \circ \mathsf{set(x)(a)} = \mathsf{set(x)(a)} \circ \mathsf{del(y)}$$

In order to show that these axioms are consistent we use the PVS theory interpretation mechanism [16]. In a new theory prog_var_model we define the type State as

$$\mathsf{State} = [[\mathsf{x} : \mathsf{ProgVar} \to \mathsf{T}(\mathsf{x})],\ \mathsf{list}[\mathsf{ProgType}]]$$

The first component of the tuple stores the values of the program variables and the second component is the stack of the computation. We denote by $\mathsf{stack}(\mathsf{s}) = \mathsf{proj2}(\mathsf{s})$ and define set_m, val_m, and del_m by

$$\begin{aligned}
&\mathsf{val_m}(\mathsf{x})(\mathsf{s}) = \mathsf{proj1}(\mathsf{s})(\mathsf{x})\\
&\mathsf{set_m}(\mathsf{x})(\mathsf{a})(\mathsf{s}) = (\lambda \mathsf{y} \bullet \mathsf{if}\ \mathsf{x} = \mathsf{y}\ \mathsf{then}\ \mathsf{a}\ \mathsf{else}\ \mathsf{val}(\mathsf{y})(\mathsf{s})\ \mathsf{fi},\ \mathsf{stack}(\mathsf{s}))\\
&\mathsf{del_m}(\mathsf{x})(\mathsf{s}) = (\lambda \mathsf{y} \bullet \mathsf{if}\ \mathsf{x} = \mathsf{y}\ \mathsf{then}\ \mathsf{a}_0\ \mathsf{else}\ \mathsf{val}(\mathsf{y})(\mathsf{s})\ \mathsf{fi},\ \mathsf{st}_0)
\end{aligned}$$

where

$$(\mathsf{a}_0, \mathsf{st}_0) = \begin{cases} (\mathsf{a}, \mathsf{t}) & \text{if } \mathsf{stack}(\mathsf{s}) = \mathsf{cons}(\mathsf{a}, \mathsf{t}) \wedge \mathsf{T?}(\mathsf{x})(\mathsf{a}) \\ (\mathsf{elem}(\mathsf{x}), \mathsf{stack}(\mathsf{s})) & \text{otherwise} \end{cases}$$

We import in this theory the theory of program variables axioms instantiating val, set, and del with set_m, val_m, and del_m, and all the axioms are proved valid in this model.

3.4 List of program variables

In order to reason about procedures with parameters we need to introduce lists of program variables. We introduce the type ProgVarList as list[ProgVar] and we inductively extend all elements defined on program variables to lists of program variables. For example we extend T? and val to lists of program variables with:

$$\begin{aligned}
&\mathsf{T?}(\mathsf{null})(\mathsf{null}) = \mathsf{true}\\
&\mathsf{T?}(\mathsf{cons}(\mathsf{x},\ \mathsf{y}))(\mathsf{cons}(\mathsf{a},\ \mathsf{b})) = \mathsf{T?}(\mathsf{x})(\mathsf{a}) \wedge \mathsf{T?}(\mathsf{y})(\mathsf{b})\\
&\mathsf{val}(\mathsf{null})(\mathsf{s}) = \mathsf{null}\\
&\mathsf{val}(\mathsf{cons}(\mathsf{x},\ \mathsf{y}))(\mathsf{s}) = \mathsf{cons}(\mathsf{val}(\mathsf{x})(\mathsf{s}),\ \mathsf{val}(\mathsf{y})(\mathsf{s}))
\end{aligned}$$

We also need some additional operations on lists of program variables We define a predicate $\mathsf{variable} : \mathsf{ProgVarList} \to \mathsf{bool}$ such that variable(x) is true if and only if all variables in x are distinct:

$$\begin{aligned}
&\mathsf{variable}(\mathsf{null}) = \mathsf{true}\\
&\mathsf{variable}(\mathsf{cons}(\mathsf{x},\ \mathsf{y})) = \neg\mathsf{member}(\mathsf{x},\ \mathsf{y}) \wedge \mathsf{variable}(\mathsf{y})
\end{aligned}$$

If $x, y \in \mathsf{ProgVarList}$ then the predicate $x \cap y$ on ProgVar, and the list $x - y : \mathsf{ProgVarList}$ are defined by

$$x \cap y = \lambda z : \mathsf{ProgVar} \bullet \mathsf{member}(z, x) \wedge \mathsf{member}(z, y)$$

$$x - y = \begin{cases} \mathsf{null} & \text{if } x = \mathsf{null} \\ \mathsf{cons}(u, z - y) & \text{if } x = \mathsf{cons}(u, z) \wedge \neg\, \mathsf{member}(u, y) \\ z - y & \text{if } x = \mathsf{cons}(u, z) \wedge \mathsf{member}(u, y) \end{cases}$$

All program variables axioms can be extended to lists of program variables. If x, y are lists of program variables of appropriate types then:

$\mathsf{var\_list\_a}$: lemma $\forall a : T(x) \bullet \mathsf{variable}(x) \Rightarrow \mathsf{val}(x)(\mathsf{set}(x)(a)(s)) = a$

$\mathsf{var\_list\_b}$: lemma $\forall a : T(x) \bullet x \cap y = \emptyset \Rightarrow \mathsf{val}(y)(\mathsf{set}(x)(a)(s)) = \mathsf{val}(y)(s)$

$\mathsf{var\_list\_c}$: lemma $\forall a, b : T(x) \bullet \mathsf{set}(x)(b) \circ \mathsf{set}(x)(a) = \mathsf{set}(x)(b)$

$\mathsf{var\_list\_d}$: lemma $\forall a : T(x), b : T(y) \bullet x \cap y = \emptyset \Rightarrow$
$\mathsf{set}.(x)(a) \circ \mathsf{set}(y)(b) = \mathsf{set}(y)(b) \circ \mathsf{set}(x)(a)$

$\mathsf{var\_list\_e}$: lemma $\mathsf{set}(x)(\mathsf{val}(x)(s))(s) = s$

$\mathsf{var\_list\_f}$: lemma $\mathsf{del}(x)$ is surjective

$\mathsf{var\_list\_g}$: lemma $x \cap y = \emptyset \Rightarrow \mathsf{val}(x) \circ \mathsf{del}(x) = \mathsf{val}(x)$

$\mathsf{var\_list\_h}$: lemma $\mathsf{del}(x) \circ \mathsf{set}(x)(a) = \mathsf{del}(x)$

$\mathsf{var\_list\_i}$: lemma $\forall a : T(x) \bullet x \cap y = \emptyset \Rightarrow$
$\mathsf{del}(y) \circ \mathsf{set}(x)(a) = \mathsf{set}(x)(a) \circ \mathsf{del}(y)$

The proofs of these lemmas are done by induction on x and y and using some additional lemmas.

In the rest of the implementation we mainly work with lists of program variables.

4 Program expressions

Because we give a shallow embedding of a predicate transformer semantics of imperative program constructs we need to solve some problems that would be straightforward if we had access to the syntax. We define program expressions of some type E as the functions from State to E.

$\mathsf{ProgExp} : \mathsf{type}+ = [\mathsf{State} \to E]$

We also define substitution and freeness. Assuming $x : \mathsf{ProgVar}$, $e' : [\mathsf{State} \to T(x)]$, $e : \mathsf{ProgExp}$, and $s : \mathsf{State}$ we define:

$$\mathsf{subst}(\mathsf{x}, \mathsf{e}', \mathsf{e})(\mathsf{s}) : \mathsf{ProgExp} = \mathsf{e}(\mathsf{set}(\mathsf{x})(\mathsf{e}'(\mathsf{x}))(\mathsf{s}))$$

We also need to specify when a variable does not occur free in an expression. We define a more general concept of freeness. If $\mathsf{f} : \mathsf{State} \to \mathsf{State}$ then

$$\mathsf{free}(\mathsf{f})(\mathsf{e}) : \mathsf{bool} = (\mathsf{e} \circ \mathsf{f} = \mathsf{e})$$

i.e. e is f free if the state transformer f does not change e. We also define:

$$\mathsf{freeset}(\mathsf{x})(\mathsf{e}) = \forall \mathsf{a} : \mathsf{T}(\mathsf{x}) \bullet \mathsf{free}(\mathsf{set}(\mathsf{x})(\mathsf{a})(\mathsf{e})$$

i.e. the expression e remains unchenged when the variable x changes.

The expressions defined so far depend not only on the values of the program variables, but also on the stack. We define a special form of program expressions that only depend on the current values of the program variables. If $\mathsf{s}, \mathsf{s}' : \mathsf{State}$ then

$$\mathsf{valeq}(\mathsf{s}, \mathsf{s}') : \mathsf{bool} = (\forall \mathsf{y} \bullet \mathsf{val}(\mathsf{y})(\mathsf{s}) = \mathsf{val}(\mathsf{y})(\mathsf{s}'))$$
$$\mathsf{valdet}(\mathsf{e}) : \mathsf{bool} = (\forall \mathsf{s}, \mathsf{s}' \bullet \mathsf{valeq}(\mathsf{s}, \mathsf{s}') \Rightarrow \mathsf{e}(\mathsf{s}) = \mathsf{e}(\mathsf{s}'))$$

5 Complete lattices, least fixpoints

We introduce some theories about complete lattices and use them to give semantics to recursive procedures.

We first introduce a theory for partial orders in which we define least upper bounds and greatest lower bounds.

$\mathsf{po}[\mathsf{A} : \mathsf{type}+] : \mathsf{theory}$
  $\mathsf{importing}\ \mathsf{orders}[\mathsf{A}]$
  $\mathsf{po} : \mathsf{type} = (\mathsf{partial\_order}?)$
  $\leq : \mathsf{var}\ \mathsf{po}$
  $\mathsf{p} : \mathsf{var}\ \mathsf{pred}[\mathsf{A}]$
  $\mathsf{x}, \mathsf{y} : \mathsf{var}\ \mathsf{A}$

We define the upper bound $\mathsf{ub}?(\leq)(\mathsf{p})$, and least upper bound $\mathsf{lub}?(\leq)(\mathsf{p})$ as predicates on A.

$$\mathsf{ub}?(\leq)(\mathsf{p})(\mathsf{x}) : \mathsf{bool} = (\forall \mathsf{y} : (\mathsf{p}) \bullet \mathsf{y} \leq \mathsf{x})$$
$$\mathsf{lub}?(\leq)(\mathsf{p})(\mathsf{x}) : \mathsf{bool} = \mathsf{ub}?(\leq)(\mathsf{p})(\mathsf{x}) \wedge (\forall \mathsf{y} : (\mathsf{p}) \bullet \mathsf{ub}?(\leq)(\mathsf{p})(\mathsf{y}) \Rightarrow \mathsf{x} \leq \mathsf{y})$$

Similarly we define the predicates $\mathsf{lb?}(\leq)(\mathsf{p})$ – lower bound, $\mathsf{glb?}(\leq)(\mathsf{p})$ – greatest lower bound, $\mathsf{top?}(\leq)$ – top element, and $\mathsf{bottom?}(\leq)$ – bottom element. We also prove properties of these predicates. For example we prove that if a partial order $\leq$ has least upper bounds than it has greatest lower bounds, top, and bottom. We prove that the least upper bound, greatest lower bound, top, and bottom are unique. We also define the predicates $\mathsf{exists\_glb?}$, $\mathsf{exists\_lub?}$, $\mathsf{exists\_top?}$, and $\mathsf{exists\_bottom?}$ on partial orders

$$\mathsf{exists\_lub?}(\leq) : \mathsf{bool} = (\forall \mathsf{p} : \mathsf{exists1}(\mathsf{lub?}(\leq)(\mathsf{p}))$$

We define the predicate $\mathsf{cl?}$ on partial orders over L as

$$\begin{aligned} \mathsf{cl?}(\leq) : \mathsf{bool} = \quad & \mathsf{exists\_glb?}(\leq) \wedge \mathsf{exists\_lub?}(\leq) \wedge \\ & \mathsf{exists\_bottom?}(\leq) \wedge \mathsf{exists\_top?}(\leq) \end{aligned} \tag{1}$$

and we take $\mathsf{cl} = (\mathsf{cl?})$ the corresponding type. Although it is sufficient to define that a partial order is a complete lattice if it has least upper bounds, we prefer definition (1) because we want all properties listed in (1) when we expand the definition of $\mathsf{cl?}$. When proving that some partial order is a complete lattice we use the following judgment:

$$\mathsf{lub\_is\_cl} : \mathsf{judgment}\ (\mathsf{exists\_lub?})\ \mathsf{subtype\_of}\ \mathsf{cl}$$

and prove only that the order has least upper bounds. If $\leq$: cl and $\mathsf{p} : \mathsf{set}[\mathsf{L}]$ then we define

$$\mathsf{inf}(\leq)(\mathsf{p}) : (\mathsf{glb?}(\leq)(\mathsf{p}))$$
$$\mathsf{sup}(\leq)(\mathsf{p}) : (\mathsf{lub?}(\leq)(\mathsf{p}))$$
$$\mathsf{bottom}(\leq) : (\mathsf{bottom?}(\leq))$$
$$\mathsf{top}(\leq) : (\mathsf{top?}(\leq))$$

Very often it is more convenient to work with an operation from an indexed family of elements from A to A instead of an operation from $\mathsf{set}[\mathsf{A}]$ to A:

$$\mathsf{op} : \mathsf{var}\ [\mathsf{set}[\mathsf{A}] \to \mathsf{A}]$$
$$\mathsf{f} : \mathsf{var}\ [[\mathsf{I} \to \mathsf{A}] \to \mathsf{A}]$$
$$\mathsf{family}(\mathsf{op})(\mathsf{f}) = \mathsf{op}(\mathsf{image}(\mathsf{f}, \lambda \mathsf{x} : \mathsf{A} \bullet \mathsf{true}))$$
$$\mathsf{conversion\ family}$$

We define predicates by lifting the order on the boolean algebra with two elements to predicates; then we lift the complete lattice of predicates to predicate transformers. For this purpose we lift the complete lattice order $\leq$ on L to a complete lattice order on $\mathsf{T} \to \mathsf{L}$ where T is a nonempty type.

$$\mathsf{lift}(\leq) : \mathsf{cl}[[\mathsf{T} \to \mathsf{L}]] = \lambda \mathsf{f}, \mathsf{g} \bullet \forall \mathsf{x} \bullet \mathsf{f}(\mathsf{x}) \leq \mathsf{g}(\mathsf{x})$$

We prove as a type constrain condition that $\mathsf{lift}(\leq)$ is a complete lattice partial order.

If $\leq$ is a complete lattice partial order on L then we define the predicate $\mathsf{monotonic}?(\leq)$ on $[\mathsf{L} \to \mathsf{L}]$; and the predicates fixpoint $\mathsf{fixpoint}?(\leq)(\mathsf{f})$, and least fixpoint $\mathsf{lfp}?(\mathsf{le})(\mathsf{f})$ on L given by

$$\begin{aligned}
&\mathsf{monotonic}?(\leq)(\mathsf{f}) : \mathsf{bool} = \forall \mathsf{x}, \mathsf{y} \bullet \mathsf{x} \leq \mathsf{y} \Rightarrow \mathsf{f}(\mathsf{x}) \leq \mathsf{f}(\mathsf{y}) \\
&\mathsf{fixpoint}?(\leq)(\mathsf{f})(\mathsf{x}) : \mathsf{bool} = \mathsf{f}(\mathsf{x}) = \mathsf{x} \\
&\mathsf{lfp}?(\leq)(\mathsf{f})(\mathsf{x}) : \mathsf{bool} = \mathsf{fixpoint}?(\leq)(\mathsf{f})(\mathsf{x}) \wedge \forall \mathsf{y} \bullet \\
&\qquad \mathsf{fixpoint}?(\leq)(\mathsf{f})(\mathsf{y}) \Rightarrow \mathsf{x} \leq \mathsf{y}
\end{aligned}$$

We are able now to introduce and prove the Knaster Tarski theorem about the existence of a least fixpoint of a monotonic function on a complete lattice.

$$\mathsf{KnasterTarski} : \mathsf{theorem}\ \mathsf{monotonic}?(\leq)(\mathsf{f}) \Rightarrow \exists \mathsf{x} \bullet \mathsf{lfp}?(\leq)(\mathsf{f})(\mathsf{x})$$

Using this theorem we define $\mathsf{mu}(\leq)(\mathsf{f}) : \mathsf{L}$, the least fixpoint of the monotonic function f by

$$\mathsf{mu}(\leq)(\mathsf{f}) : \mathsf{lfp}?(\leq)(\mathsf{f})$$

6 Predicates, relations, functions, and monotonic predicate transformers

We introduce a complete lattice order on bool, and define inf and sup by

$$\begin{aligned}
&\leq : \mathsf{cl}[\mathsf{bool}] = \lambda \mathsf{x}, \mathsf{y} \bullet \mathsf{x} \Rightarrow \mathsf{y} \\
&\mathsf{inf}(\mathsf{a} : [\mathsf{bool} \to \mathsf{bool}]) : (\mathsf{glb}?(\leq)(\mathsf{a})) = \neg \mathsf{a}(\mathsf{false}) \\
&\mathsf{sup}(\mathsf{a} : [\mathsf{bool} \to \mathsf{bool}]) : (\mathsf{lub}?(\leq)(\mathsf{a})) = \mathsf{a}(\mathsf{true})
\end{aligned}$$

and prove as type constrain conditions that $\leq$, inf, and sup have the postulated properties.

The complete lattice of predicates over State is obtained by lifting the complete lattice on bool to $\mathsf{State} \to \mathsf{bool}$. We also extend some operations from bool to $\mathsf{Pred}[\mathsf{bool}]$.

$$\begin{aligned}
&\leq : \mathsf{cl}[\mathsf{Pred}[\mathsf{bool}]] = \lambda \mathsf{p}, \mathsf{q} \bullet \forall \mathsf{s} \bullet \mathsf{p}(\mathsf{s}) \Rightarrow \mathsf{q}(\mathsf{s}) \\
&\wedge(\mathsf{p}, \mathsf{q} : \mathsf{Pred}[\mathsf{bool}]) : \mathsf{Pred}[\mathsf{bool}] = \lambda \mathsf{s} \bullet \mathsf{p}(\mathsf{s}) \wedge \mathsf{q}(\mathsf{s})
\end{aligned}$$

$\mathsf{union}(\mathsf{t} : \mathsf{set}[\mathsf{Pred}[\mathsf{bool}]]) : \mathsf{Pred}[\mathsf{bool}] = \lambda \mathsf{s} \bullet \exists \mathsf{p} : (\mathsf{t}) \bullet \mathsf{p}(\mathsf{s})$
$\mathsf{intersection}(\mathsf{t} : \mathsf{set}[\mathsf{Pred}[\mathsf{bool}]]) : \mathsf{Pred}[\mathsf{bool}] = \lambda \mathsf{s} \bullet \forall \mathsf{p} : (\mathsf{t}) \bullet \mathsf{p}(\mathsf{s})$

We introduce the monotonic predicate transformers MTran as a subtype of [Pred[State] → Pred[State]] and define the complete lattice partial order on MTran together with some operations:

$\mathsf{MTran?} : \mathsf{Set}[[\mathsf{Pred}[\mathsf{State}] \to \mathsf{Pred}[\mathsf{State}]]] = \mathsf{monotonic?}(\leq)$
$\mathsf{MTran} : \mathsf{type}+ = (\mathsf{MTran?})$
$\leq : \mathsf{cl}[\mathsf{MTran}] = \lambda \mathsf{S}, \mathsf{S}' \bullet \lambda \mathsf{q} \bullet \mathsf{S}(\mathsf{q}) \leq \mathsf{S}'(\mathsf{q})$
$\mathsf{union}(\mathsf{X} : \mathsf{Set}[\mathsf{MTran}]) : (\mathsf{lub?}[\mathsf{MTran}](\leq)(\mathsf{X})) =$
$\quad \lambda \mathsf{q} \bullet \mathsf{union}(\lambda \mathsf{S} : (\mathsf{X}) \bullet \mathsf{S}(\mathsf{q}))$
$\mathsf{intersection}(\mathsf{X} : \mathsf{Set}[\mathsf{MTran}]) : (\mathsf{lub?}[\mathsf{MTran}](\leq)(\mathsf{X})) =$
$\quad \lambda \mathsf{q} \bullet \mathsf{intersection}(\lambda \mathsf{S} : (\mathsf{X}) \bullet \mathsf{S}(\mathsf{q}))$
$\mathsf{Magic} : (\mathsf{top?}[\mathsf{MTran}](\leq)) = \lambda \mathsf{q} \bullet \mathsf{true}$
$\mathsf{Abort} : (\mathsf{bottom?}[\mathsf{MTran}](\leq)) = \lambda \mathsf{q} \bullet \mathsf{false}$
$\wedge(\mathsf{S}, \mathsf{T} : \mathsf{MTran}) : \mathsf{MTran} = \lambda \mathsf{q} \bullet \mathsf{S}(\mathsf{q}) \wedge \mathsf{T}(\mathsf{q})$
$\vee(\mathsf{S}, \mathsf{T} : \mathsf{MTran}) : \mathsf{MTran} = \lambda \mathsf{q} \bullet \mathsf{S}(\mathsf{q}) \vee \mathsf{T}(\mathsf{q})$

The partial order $\leq$ on MTran is the *refinement relation*. The predicate transformer $\mathsf{S} \wedge \mathsf{T}$ models *demonic* choice – the choice between executing S or T is arbitrary; $\mathsf{S} \vee \mathsf{T}$ models *angelic* choice – the choice is resolved so that the postcondition is established, if possible. The program *sequential composition* is modeled by the functional composition $\circ$ of monotonic predicate transformers.

Often we work with predicate transformers based on functions or relations. We introduce the types Func and Rel by

$\mathsf{Func} : \mathsf{type}+ = [\mathsf{State} \to \mathsf{State}]$
$\mathsf{Rel} : \mathsf{type}+ = [\mathsf{State} \to \mathsf{Pred}]$

and define the identity function $\mathsf{id} : \mathsf{Func} = \lambda \mathsf{s} : \mathsf{State} \bullet \mathsf{s}$.

If $\mathsf{p}, \mathsf{q} : \mathsf{Pred}$, $\mathsf{R} : \mathsf{Rel}$, $\mathsf{f} : \mathsf{Func}$, $\mathsf{s}, \mathsf{s}' : \mathsf{State}$, and $\mathsf{S}, \mathsf{T} : \mathsf{MTran}$ then we define

$\mathsf{f\_update}(\mathsf{f}) : \mathsf{MTran} = \lambda \mathsf{q}, \mathsf{s} \bullet \mathsf{q}(\mathsf{f}(\mathsf{s}))$
$\mathsf{d\_update}(\mathsf{R}) : \mathsf{MTran} = \lambda \mathsf{q}, \mathsf{s} \bullet \forall \mathsf{s}' \bullet \mathsf{R}(\mathsf{s})(\mathsf{s}') \Rightarrow \mathsf{q}(\mathsf{s}')$
$\mathsf{Assert}(\mathsf{p}) : \mathsf{MTran} = \lambda \mathsf{q} \bullet \mathsf{p} \wedge \mathsf{q}$
$\mathsf{Assume}(\mathsf{p}) : \mathsf{MTran} = \lambda \mathsf{q} \bullet \mathsf{p} \Rightarrow \mathsf{q}$
$\mathsf{If}(\mathsf{p})(\mathsf{S})(\mathsf{T}) : \mathsf{MTran} = (\mathsf{Assume}(\mathsf{p}) \circ \mathsf{S}) \wedge (\mathsf{Assume}(\neg \mathsf{p}) \circ \mathsf{T})$

We also prove the following property:

$$\mathsf{assert\_union} : \mathsf{lemma}\ \mathsf{Assert}(\mathsf{union}(\mathsf{X})) = \mathsf{union}(\mathsf{image}(\mathsf{Assert}, \mathsf{X}))$$

where $\mathsf{X} : \mathsf{Set}[\mathsf{Pred}]$.

7 Assignment and local variables statements

Now we are able to introduce the program statements for handling program variables. We define two versions of the assignment statement. The *deterministic multiple assignment* statement $\mathsf{Assign}(\mathsf{x}, \mathsf{e})$ where x is a list of program variables and e is a program expression of type $\mathsf{T}(\mathsf{x})$ is given by:

$$\mathsf{Assign}(\mathsf{x}, \mathsf{e}) : \mathsf{MTran} = \mathsf{f\_update}(\lambda \mathsf{s} \bullet \mathsf{set}(\mathsf{x})(\mathsf{e}(\mathsf{s}))(\mathsf{s}))$$

The *nondeterministic assignment* statement [3], $\mathsf{Assign}(\mathsf{x}, \mathsf{b})$, where x is a list of program variable and $[\mathsf{b} : \mathsf{T}(\mathsf{x}) \to [\mathsf{State} \to \mathsf{bool}]]$ is given by:

$$\mathsf{Assign}(\mathsf{x}, \mathsf{b}) : \mathsf{MTran} = \mathsf{d\_update}(\lambda \mathsf{s}, \mathsf{s}' \bullet \exists \mathsf{a} : \mathsf{T}(\mathsf{x}) \bullet \mathsf{b}(\mathsf{a})(\mathsf{s}) \wedge \mathsf{s}' = \mathsf{set}(\mathsf{x})(\mathsf{a})(\mathsf{s}))$$

The interpretation of this statement is that x is assigned a value a such that $\mathsf{b}(\mathsf{a})$ is true in the initial state.

We introduce four program constructs for handling introduction and deletion of local program variables.

$$\mathsf{Del}(\mathsf{x}) : \mathsf{MTran} = \mathsf{f\_update}(\mathsf{del}(\mathsf{x}))$$
$$\mathsf{Add}(\mathsf{x}) : \mathsf{MTran} = \mathsf{d\_update}(\lambda \mathsf{s}, \mathsf{s}' \bullet \mathsf{s} = \mathsf{del}(\mathsf{x})(\mathsf{s}'))$$
$$\mathsf{Add}(\mathsf{x}, \mathsf{e}) : \mathsf{MTran} = \mathsf{d\_update}(\lambda\, \mathsf{s}, \mathsf{s}' \bullet \mathsf{s} = \mathsf{del}(\mathsf{x})(\mathsf{s}') \wedge \mathsf{val}(\mathsf{x})(\mathsf{s}') = \mathsf{e}(\mathsf{s}))$$
$$\mathsf{Del}(\mathsf{x}, \mathsf{y}) : \mathsf{MTran} = \mathsf{f\_update}(\lambda\, \mathsf{s} \bullet \mathsf{set}(\mathsf{y})(\mathsf{val}(\mathsf{x})(\mathsf{s}))(\mathsf{del}(\mathsf{x})(\mathsf{s}))$$

where $\mathsf{e} : \mathsf{State} \to \mathsf{T}(\mathsf{x})$ and $\mathsf{y} : \mathsf{SameType}(\mathsf{x})$. The statement $\mathsf{Del}(\mathsf{x})$ deletes the local variables x, i.e., according to the model presented above, it pops up the top value from the stack and assigns it to x. The statement $\mathsf{Add}(\mathsf{x})$ is the inverse of $\mathsf{Del}(\mathsf{x})$, i.e., according to the model, it pushes the value of x into the stack and makes x undefined. $\mathsf{Add}(\mathsf{x}, \mathsf{e})$ is similar to $\mathsf{Add}(\mathsf{x})$, but it also initializes x with the value of the program expression e in the initial state. The statement $\mathsf{Del}(\mathsf{x}, \mathsf{y})$, deletes the local program variables x, but saves their values in y. We have proved the following refinement rules for the program statements we introduced.

add_del_skip : lemma $\mathsf{Add}(x) \circ \mathsf{Del}(x) = \mathsf{Skip}$

add_exp_del_skip : lemma $\forall e : [\mathsf{State} \to \mathsf{T}(x)] \bullet$
$\mathsf{variable}(x) \Rightarrow \mathsf{Add}(x, e) \circ \mathsf{Del}(x) = \mathsf{Skip}$

assign_x_del_x : lemma $\forall e : [\mathsf{State} \to \mathsf{T}(x)] \bullet$
$\mathsf{Assign}(x, e) \circ \mathsf{Del}(x) = \mathsf{Del}(x)$

assign_y_del_x : lemma $\forall e : [\mathsf{State} \to \mathsf{T}(x)] \bullet$
$x \cap y = \emptyset \wedge \mathsf{free}(\mathsf{del}(y))(e) \Rightarrow$
$\mathsf{Assign}(x, e) \circ \mathsf{Del}(y) = \mathsf{Del}(y) \circ \mathsf{Assign}(x, e)$

assign_x_del_x_y : lemma $\forall y : \mathsf{SameType}(x), e : [\mathsf{State} \to \mathsf{T}(x)] \bullet$
$\mathsf{variable}(x) \wedge \mathsf{free}(\mathsf{del}(x))(e) \Rightarrow$
$\mathsf{Assign}(x, e) \circ \mathsf{Del}(x, y) = \mathsf{Del}(x) \circ \mathsf{Assign}(y, e)$

p, q : var Pred

add_x : lemma $\mathsf{Add}(x)(q \circ \mathsf{del}(x)) = q$

add_x_free : lemma $\mathsf{free}(\mathsf{del}(x))(q) \Rightarrow \mathsf{Add}(x)(q) = q$

add_x_e : lemma $\forall e : [\mathsf{State} \to \mathsf{T}(x)] \bullet$
$\mathsf{variable}(x) \Rightarrow \mathsf{Add}(x, e)(q \circ \mathsf{del}(x)) = q$

add_x_e_free : lemma $\forall e : [\mathsf{State} \to \mathsf{T}(x)] \bullet$
$\mathsf{variable}(x) \wedge \mathsf{free}(\mathsf{del}(x))(q) \Rightarrow \mathsf{Add}(x, e)(q \circ \mathsf{del}(x)) = q$

add_x_e_subst : lemma $\forall e : [\mathsf{State} \to \mathsf{T}(x)] \bullet$
$\mathsf{variable}(x) \wedge \mathsf{valdet}(q) \Rightarrow \mathsf{Add}(x, e)(q) = \mathsf{subst}(x, e, q)$

del_x : lemma $\mathsf{Del}(x)(q) = (q \circ \mathsf{del}(x))$

del_x_free : lemma $\mathsf{free}(\mathsf{del}(x))(q) \Rightarrow \mathsf{Del}(x)(q) = q$

del_x_y : lemma $\forall y : \mathsf{SameType}(x) \bullet$
$\mathsf{freeset}(y)(q) \Rightarrow \mathsf{Del}(x, y)(q) = q \circ \mathsf{del}(x)$

del_x_y_free : lemma $\forall y : \mathsf{SameType}(x) \bullet$
$\mathsf{freeset}(y)(q) \wedge \mathsf{free}(\mathsf{del}(x))(q) \Rightarrow \mathsf{Del}(x, y)(q) = q$

del_x_y_subst : lemma $\forall y : \mathsf{SameType}(x) \bullet$
$\mathsf{valdet}(q) \wedge \mathsf{freeset}(x - y)(q) \Rightarrow \mathsf{Del}(x, y)(q) = \mathsf{subst}(y, \mathsf{val}(x), q)$

Lemma add_del_skip asserts that adding a local variable followed by deleting it is the same as skipping. Lemmas add_x_e_subst and del_x_y_subst state that under certain conditions the statements $\mathsf{Add}(x)(e)$ and $\mathsf{Del}(x)(y)$ behave as assignment statements.

All conditions of the rules presented above are decidable and one could write tactics that automatically prove them.

8 Procedures

A procedure with parameters from A or simply a procedure over A, is an element from $A \to MTran$. We define the type $Proc : type+ = A \to MTran$ the type of all procedures over A. The type A is the range of the procedure's actual parameters. For example, a procedure with a value parameter x and a result parameter y, both of type Nat, has $A = NatExp \times NatVar$. A call to a procedure $P \in Proc[A]$ with the actual parameter $a : A$ is the program $P(a)$.

We use again the lifting mechanism we used to obtain a complete lattice over predicates to lift the complete lattice structure from monotonic predicate transformers to procedures over A. We also extend sequential composition to procedures by

$$P \circ P' : Proc = \lambda x \bullet P(x) \circ P'(x)$$

Moreover we lift the algebraic structure on predicates to parametric predicates, $ParamPred : type+ = A \to Pred$ and define

$$Assert(p : PramPred) : Proc = \lambda x \bullet Assert(p(x))$$

Recursive procedures are defined as the least fixpoint of a monotonic function from Proc to Proc. To introduce the theorem for refining recursive procedures we need a well founded partial order on a set W, $<: (well\_founded?[W])$. For $w : W$ and $p : [W \to ParamPred]$ we define

$$\begin{aligned} &below(p)(w) : ParamPred \ = \\ &\quad union(\lambda x : ParamPred \bullet \exists v : W \bullet v < w \wedge p(v) = x) \end{aligned}$$

and the theorem is

$$\begin{aligned} &rec\_proc\_rule : theorem \\ &\quad (\forall w \bullet Assert(p(w)) \circ P \leq f(Assert(below(p)(w)) \circ P)) \ \Rightarrow \\ &\quad Assert(union(p)) \circ P \leq mu(\leq)(f) \end{aligned}$$

9 Example

All the elements introduced so far are sufficient to define and reason about recursive procedures with value, value-result parameters and local variables. We give a recursive procedure that computes the binomial coefficient

$$\begin{pmatrix} n \\ k \end{pmatrix} = \frac{n!}{k! \cdot (n-k)!}$$

using the recursive formula

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

when $0 < \mathsf{k} < \mathsf{n}$.

The specification of this computation is:

$\mathsf{fact}(\mathsf{n} : \mathsf{nat}) : \mathsf{recursive\ nat}\ =$
$\quad \mathsf{if}\ \mathsf{n} = 0\ \mathsf{then}\ 1\ \mathsf{else}\ \mathsf{n} * \mathsf{fact}(\mathsf{n}-1)$
$\quad \mathsf{measure}\ \mathsf{n}$
$\mathsf{comb}(\mathsf{n} : \mathsf{nat}, \mathsf{k} : \mathsf{upto}(\mathsf{n})) : \mathsf{nat} = \mathsf{fact}(\mathsf{n})/(\mathsf{fact}(\mathsf{k}) * \mathsf{fact}(\mathsf{n}-\mathsf{k}))$
$\mathsf{comp\_proc\_spec} : \mathsf{Proc}[[\mathsf{NatExp}, \mathsf{NatExp}, \mathsf{NatVar}]] =$
$\quad \lambda\,(\mathsf{e}, \mathsf{f}, \mathsf{u}) \bullet \mathsf{Assign}(\mathsf{u}, \mathsf{comb}(\mathsf{e}, \mathsf{f}))$

where u is a variable of type NatVar, e, f are variables of type NatExp, and $\mathsf{comb}(\mathsf{e}, \mathsf{f})(\mathsf{s}) : \mathsf{Nat} = \mathsf{if}\ \mathsf{f.s} \leq \mathsf{e.s}\ \mathsf{then}\ \mathsf{comb}(\mathsf{f.s}, \mathsf{e.s})\ \mathsf{else}\ 0\ \mathsf{fi}$. The type of the parameters of this specification procedure is $\mathsf{A} = [\mathsf{NatExp},\ \mathsf{NatExp},\ \mathsf{NatVar}]$.

To define the procedure for computing the binomial coefficient we need some program variables for the formal procedure parameters and for the local variables:

$\mathsf{k} : \mathsf{NatVar} = \mathsf{n}(\text{"k"})$
$\mathsf{n} : \mathsf{NatVar} = \mathsf{n}(\text{"n"})$
$\mathsf{c} : \mathsf{NatVar} = \mathsf{n}(\text{"c"})$
$\mathsf{x} : \mathsf{NatVar} = \mathsf{n}(\text{"x"})$
$\mathsf{y} : \mathsf{NatVar} = \mathsf{n}(\text{"y"})$

Assuming the following definitions

$\mathsf{eq}(\mathsf{e}, \mathsf{f} : [\mathsf{State} \to \mathsf{E}])(\mathsf{s}) : \mathsf{bool} = \mathsf{e}(\mathsf{s}) = \mathsf{f}(\mathsf{s})$
$\mathsf{conversion}\ \lambda \mathsf{x} : \mathsf{ProgVarList} \bullet \mathsf{cons}(\mathsf{x},\ \mathsf{null})$
$\&\,(\mathsf{x} : \mathsf{ProgVar}, \mathsf{y} : \mathsf{ProgVarList}) : \mathsf{ProgVarList} = \mathsf{cons}(\mathsf{x},\ \mathsf{y})$
$\mathsf{conversion\ val}$
$\mathsf{conversion}\ \lambda \mathsf{e} : [\mathsf{State} \to \mathsf{ProgType}] \bullet \lambda \mathsf{s} \bullet \mathsf{cons}(\mathsf{e}(\mathsf{s}),\ \mathsf{null})$
$\mathsf{conversion}\ \lambda \mathsf{a} : \mathsf{ProgType} \bullet \lambda \mathsf{s} \bullet \mathsf{cons}(\mathsf{a},\ \mathsf{null})$
$\mathsf{conversion}\ \lambda \mathsf{x} \bullet \lambda \mathsf{s} \bullet \mathsf{cons}(\mathsf{val}(\mathsf{x})(\mathsf{s}),\ \mathsf{null})$
$\mathsf{a} : \mathsf{var}\ [\mathsf{State} \to \mathsf{ProgType}]$
$\mathsf{b} : \mathsf{var}\ [\mathsf{State} \to \mathsf{list}[\mathsf{ProgType}]]$

```
& (a, b) : [State → list[ProgType]] = λs • cons(a(s), b(s))
```

We introduce the function that defines the procedure by taking its fixpoint:

```
comb_body : monotonic[Proc[A]](≤) = λ comb • λ (e, f, u) •
    Add (n & k & c,  e & f & u) ∘
    Add (x & y) ∘
    If  eq (k, 0) ∨ eq (k, n)  then
        Assign (c,  1)
    else
        comb (k − 1,  n − 1,  x) ∘
        comb (k,  n − 1,  y) ∘
        Assign (c,  x + y)
    endif ∘
    Del (x & y) ∘
    Del (n & k) ∘
    Del (c,  u)
```

and the procedure definition is:

```
comb_proc : Proc[A] = mu(comb_body)
```

We are able to prove now that the specification comb_proc_spec is refined by the procedure comb_proc under the assertion e ≤ f.

```
comb_refin : theorem
    Assert(λ (e, f, u) • e ≤ f) ∘  comb_proc_spec ≤ comb_proc
```

10 Conclusions, future work

We have presented a PVS implementation of a predicate transformer semantics suitable for refinement of recursive procedures with parameters and local variables. Since in our approach the state space does not change when adding local variables or using procedure parameters, the refinement rule for introduction of recursive procedure calls does not have (syntactic) side conditions and does not involve the procedure parameters or the local variables. The refinement rules for local variables and procedure parameters have decidable

side conditions and are almost as simple as the assignment rules. Moreover, the theory is general and we can plug in without additional proofs any "algebra" of program variables types.

Although mutually recursive procedures are not mentioned in this paper, they can be easily included by lifting the structure of complete lattices on procedure types to their cartesian product. More concretely, if $\mathsf{P} : \mathsf{Proc}[\mathsf{A}]$ and $\mathsf{P}' : \mathsf{Proc}[\mathsf{B}]$ are two mutually recursive procedures, then their semantics is the least fixpoint of a monotonic function from $[\mathsf{Proc}[\mathsf{A}], \mathsf{Proc}[\mathsf{B}]]$ to itself.

In future work we intend to improve the handling of program expressions so that lifting properties from the PVS basic types to program expressions becomes seemless and can exploit better the power of the theorem prover. Another interesting direction would be to extend the program variables data types with pointer structures [4].

References

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

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

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

[4] R.J. Back, F. Xiaocong, and V. Preoteasa. Reasoning about pointers in refinement calculus. In *Tenth Asia-Pacific Software Engineering Conference Software Engineering Conference*, pages 425–434. IEEE Computer Society, December 2003.

[5] M.J. Butler, J. Grundy, T. Långbacka, Ruksenas R., and J. von Wright. The refinemen calculator: Proof support for program verification. In *FMP'97 Formal Methods Pacific*, Discrete Mathematics and Theoretical Computer Science. Springer-Verlag, Wellington, New Zealand, July 1997.

[6] 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.

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

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

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

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

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

[13] C. Morgan. *Programming from specifications*. Prentice-Hall International, 1994.

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

[15] S. Owre and N. Shankar. Abstract datatypes in PVS. Technical Report SRI-CSL-93-9R, Computer Science Laboratory, SRI International, Menlo Park, CA, dec 1993. Extensively revised June 1997.

[16] S. Owre and N. Shankar. Theory interpretations in PVS. Technical Report SRI-CSL-01-01, Computer Science Laboratory, SRI International, Menlo Park, CA, April 2001.

[17] 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.

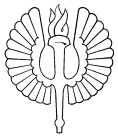
[18] M. Staples. *A Mechanised Theory of Refinement*. PhD dissertation, Computer Laboratory, University of Cambridge, November 1998.

[19] 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.

[20] X. Zhang, M. Munro, M. Harman, and L. Hu. Weakest precondition for general recursive programs formalized in coq. In *Theorem proving in higher order logics (Hampton, 2002)*, volume 2410 of *Lecture Notes in Comput. Sci.*, pages 332–347. Springer, Berlin, 2002.

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

http://www.tucs.fi



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

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

Turku School of Economics and Business Administration
- **Institute of Information Systems Science**