Validating and Animating Higher-Order Recursive Functions in B

Michael Leuschel[1], Dominique Cansell[2], and Michael Butler[3]

[1] Institut für Informatik, Universität Düsseldorf
leuschel@cs.uni-duesseldorf.de
[2] LORIA, Nancy, cansell@loria.fr
[3] School of Electronics and Computer Science, University of Southampton
mjb@ecs.soton.ac.uk

Abstract. ProB is an animation and model checking tool for the B Method, which can deal with many interesting specifications. Some specifications, however, contain complicated functions which cannot be represented explicitly by a tool. We present a scheme with which higher-order recursive functions can be encoded in B, and establish soundness of this scheme. We then describe a symbolic representation for such functions. This representation enables ProB to successfully animate and model check a new class of relevant specifications, where animation is especially important due to the involved nature of the specification.
Keywords: B-Method, Tool Support, Model Checking, Animation, Logic Programming, Constraints. [4]

1 Introduction

The B-method, originally devised by J.-R. Abrial [1], is a theory and methodology for formal development of computer systems. It is used by industries in a range of critical domains. B specifications are structured into *abstract machines*. The state of an abstract machines is represented by variables and correctness conditions can be expressed in the *invariant*, which is specified in predicate logic augmented with set theory and arithmetic. In addition to variables, B machines may also contain constants, which must satisfy conditions expressed in the "properties" clause. Operations of a machine are specified as *generalised substitutions*, which allow deterministic and non-deterministic state transitions to be specified.

There are two main proof activities in B: *consistency checking*, which is used to show that the operations of a machine preserve the invariant, and *refinement checking*, which is used to show that one machine is a valid refinement of another. These activities are supported by tools, such as Atelier-B, B4Free, and the B-toolkit.

In addition to the proof activities it is increasingly being realised that validation of the initial specification is important to avoid deriving a "correct" implementation of an incorrect specification. This validation can come in the

[4] This research is being carried out as part of the EU funded research project IST 511599 RODIN (Rigourous Open Development Environment for Complex Systems).

form of *animation*, e.g., to check that certain functionality is present in the specification. Another useful tool is *model checking* [6], whereby the specification can be systematically checked for certain temporal properties. In previous work [9], we have presented the ProB animator and model checker to support those activities. The tool can also be used to complement proof activities, as it supports automated consistency checking of B machines and has been recently extended for automated refinement checking [10].

Motivation The ProB tool has been successfully applied to various academic and industrial examples (e.g., a Volvo vehicle function [9]). ProB can deal with B's data structures, such as relations, functions and sequences as well as set comprehensions and lambda abstractions. ProB can also handle constants and the first step of animating or model checking a B model then consists of finding values for the constants which satisfy the PROPERTIES clause. To avoid naïve enumeration of possible values for the constants, ProB uses various mechanisms to propagate (partial) information about the possible values of the constants. Still, in the end, the constants will be represented explicitly inside ProB. This is not a problem for some models: for example, the railway model in [4] based on a requirements document from Siemens Transportation Systems, can be animated and model checked: the constants represent, amongst others, the underlying rail network topology. Some specifications, however, contain complicated functions or sets which cannot be represented explicitly. Take the following recursive function over sequences of sequences, coming from an industrial case study [12]:

```
removeDuplicates = {ss,rs | ss: seq(seq(PLACE)) & rs:seq(seq(PLACE))
    &    (ss=<> => rs=<>) &
         (card(ss)=1 => rs=ss) &
         (card(ss)>1 => ( #(s1,s2).(s1:seq(PLACE) & s1=first(ss)
                              & s2:seq(PLACE) & s2=ss(2) &
               (last(s1)= first(s2) =>
                  rs = front(s1) -> removeDuplicates(tail(ss)) )
            &   (last(s1)/=first(s2) =>
                  rs = s1 -> removeDuplicates(tail(ss)))
       ))))}
& removeDuplicates: seq(seq(PLACE)) --> seq(seq(PLACE))
```

Intuitively, the above function takes a sequence of sequences and removes duplicates (i.e., the last element of a sequence is identical to the first element of the next sequence). This is represented as a recursive function, and is part of a larger algorithm.

Validating such specifications is especially important, since they are particularly error prone and may contain crucial computational aspects of a software system. Such specifications also pose a major challenge to animation and model checking. Indeed, completely expanding these functions or sets is prohibitively expensive or even impossible. Supposing that $card(PLACE) = p$ and supposing that we set a maximum length m for sequences, the above function contains $\frac{q^{m+1}-1}{q-1}$ maplets, where $q = \frac{p^{m+1}-1}{p-1}$. With $p = 3$ and $m = 4$ this gives rise to

216, 145, 205 pairs which need to be pre-computed. Assuming that every maplet requires 25 bytes of storage on average, we obtain a memory requirement of over one Gigabyte; and with $p = 4$, $m = 4$ we exceed 315 Gigabytes.

In this paper we try to overcome this problem, and enable ProB to animate and model check such specifications by using a symbolic representation. The main contributions of the paper are as follows:

- A method and implementation to symbolically store set comprehensions and lambda abstractions within an animator and model checker, without having to expand them into an explicit form.
- A sound scheme whereby recursive higher-order functions can be encoded in B, along with proof obligations to ensure well-definedness.
- A method and implementation to animate and model check specifications containing such functions. The central idea is to extend the above symbolic form with a way to encode the recursion. Only when a recursive function is actually applied to some arguments, is the symbolic form evaluated (for that particular argument).
  We also provide some empirical results as well as some possible applications of our technique.

2 Symbolic Representation of Sets of Values

We will first study how to represent non-recursive functions (and more generally set comprehensions) symbolically.

ProB and Explicit Value Representation ProB uses an explicit representation for the values of machine variables, constants, etc. The following table shows how the basic B's data structures are encoded by the ProB Kernel:

B Type	B value	Prolog encoding
INTEGER	5	`int(5)`
BOOL	TRUE	`term(bool(1))`
element of a SET S	C	`fd(3,'S')`
pair (*)	$2 \mapsto 5$	`(int(2),int(5))`
set (POW)	$\{2, 5\}$	`[int(2), int(5)]`

For example, the set comprehension $\{x \mid x \in 1..3\}$ gets expanded into $\{1, 2, 3\}$ (or in the Prolog representation: `[int(1),int(2),int(3)]`). The lambda abstraction $\lambda x.(x \in 1..3 \mid x * x)$ gets expanded into the set of pairs $\{1 \mapsto 2, 2 \mapsto 4, 3 \mapsto 9\}$. Note that internally, a lambda abstraction is simply translated into a set comprehension and then expanded. E.g., $\lambda x.(x \in 1..3 \mid x * x)$ gets translated into $\{x, res \mid x \in 1..3 \wedge res = x * x\}$.

Symbolic Values In this paper we introduce an additional symbolic representation for set comprehensions (and thus also lambda abstractions). The idea is that under certain circumstances a set comprehension is *not* expanded out but kept symbolically. Some of the B operators inside the ProB kernel are then

extended to deal with those symbolic representations. Suppose for example that the properties of a B machine contains the predicate $s = \{x \mid x * x \in 1..99\}$ and that at some point we wish to evaluate the predicate $9 \in s$. This can simply be done by replacing x inside the set comprehension by the value 9 and then checking the predicate inside the set comprehension, i.e., in this case $9 * 9 \in 1..99$. Similarly, if the properties (or an initialisation or an operation body) contain the predicate $sqr = \lambda x.(x : INTEGER \mid x * x)$ and if at some point we need to evaluate $sqr(8)$, we need to replace the parameter x with the actual value 8 and compute the body $8 * 8$ of the lambda abstraction. Other operators, however, may require the complete set of values and hence require the expansion of the set comprehension, e.g., if we encounter $s \cap 8..10 \neq \varnothing$ we need to expand s to check if it has a value in common with 8..10.

Closures Which form should the symbolic representation take? Naively, one may think that to represent a set like $\{y \mid P\}$ with y being of type INTEGER, we could just store the arguments, the types of the arguments, and a representation `'P'` of P inside a Prolog term such as: `symbolic_set([y],[INTEGER],'P')`. A similar scheme could be used for lambda expressions, by translating them into a set comprehension first.

There is, however, one problem with this approach, which the following example illustrates:

```
MACHINE FunPlus
VARIABLES x, fun
INVARIANT x:INTEGER & fun: INTEGER --> INTEGER
INITIALISATION x:=0 || fun:= %y.(y:INTEGER|0)
OPERATIONS
  Inc = x := x+1;
  Set_fun_plus_x = fun := %y.(y:INTEGER| x+y);
  cc <-- ApplyFun(y) = PRE y:INTEGER THEN cc := fun(y) END
END
```

Assume that we start out by executing the operation `Set_fun_plus_x`. Here the variable `fun` would be given the value `symbolic_set([y,res],[INTEGER,INTEGER], 'res=x+y')`. After applying the `Inc` operation, the variable `x` is updated, which would implicitly change the function `fun` represented by our symbolic set! This is of course incorrect, as in the above B machine `Inc` does *not* change the function `fun`.

The solution lies with the *closure* concept, familiar from programming language implementation (see, e.g., [15]) in general and functional programming in particular. It is used when procedures or functions can be used as values. A closure of a function combines the source code of the function together with the current values of the global variables it refers to. This is also called *environment capture*, as the environment is packaged up together with the function.

In PROB this is achieved by compiling a set comprehension or lambda abstraction into a closure; where all references to machine variables are compiled

into the code. This is achieved by replacing a reference to a variable `x` by a special construct `value(V)` where `V` is the value of `x` at the point of construction of the lambda abstraction or set comprehension. Thus, the symbolic representation of `%y.(y:INTEGER|x+y)`, supposing that `x` has the value 2, would be: `closure([y,res], [INTEGER,INTEGER], 'res=value(int(2))+y')`. The value indicates to the ProB kernel that this is not an expression that needs to be interpreted (it is already in the internal representation).[5]

Implementation We have implemented the symbolic representation inside ProB's kernel. Several operators were extended to directly work on this symbolic representation: equality ($=$), set membership ($\in$), and function application ($f(.)$). If a symbolic representation is used with any other B operator the symbolic closure is expanded into an explicit form (where the expansion is delayed until all the free variables of the set comprehension have been given a value). We plan to extend the kernel for further operators; but for the case studies so far, extending the above operators was sufficient. The user can set a boolean preference value to indicate whether set comprehensions and lambda abstractions should be stored symbolically if possible. In future we plan to add a more fine-grained control, whereby each individual set comprehension can be treated differently (indeed, in some cases it is more efficient to expand a symbolic representation once and for all).

We have also applied the same scheme for certain other expressions which are likely to yield big sets: Cartesian products ($A \times B$), powerset constructions ($\mathbb{P}(A)$), sets of relations ($A \leftrightarrow B$) and sets of functions ($A \nrightarrow B, A \rightarrow B, \ldots$). Those expressions are usually used for typing and rarely need to be expanded out. For example, given a predicate $r \in \mathbb{P}(NAT \leftrightarrow NAT)$ and a value of 3 for $MAXINT$[6] the set $\mathbb{P}(NAT \leftrightarrow NAT)$ has a cardinality of $2^{2^{4*4}} = 2^{65536}$, and even with MAXINT of just 2 we have a cardinality of $2^{512} \approx 1.34 * 10^{154}$. Thus, $\mathbb{P}(NAT \leftrightarrow NAT)$ cannot possibly be stored explicitly, while it is relatively straightforward to check if a given relation r is a member of the set. A separate user preference indicates whether the symbolic representation should be applied for those type expressions (but there should be little need to switch off the symbolic representation for those expressions).

Correctness By storing a set comprehension $\{x_1, \ldots x_n \mid P\}$ symbolically inside an animator or model checker for B, we implicitly assume two things:

1. that the set comprehension exists,
2. that only a single value for the set comprehension exists

In this section we do not yet study recursion, i.e., we assume that the predicate P makes no reference to the set itself (i.e., we do *not* yet consider definitions

[5] Also, if the value of `x` is not yet known by the kernel then the `value` constructor will contain a Prolog variable, which will be instantiated as soon as `x` becomes known (e.g., through enumeration or through evaluation of some other predicate).

[6] `NAT` represents the implementable natural numbers from 0 to $MAXINT$.

of the form $s = \{x \mid x \in s\}$ or $s = \{x \mid x \notin s\}$). Without recursion, the set comprehension must exist, but it could of course be empty. For example, $\{x \mid x \in INT \wedge 1 = 2\}$ is equal to the empty set. Also, there can only be a single solution, otherwise we would have found values for the comprehension parameters for which P is both true and false; this cannot be.

Hence, in the non-recursive case, the correctness of our approach is unproblematic. However, some interesting real-life specifications require recursive definitions, and we tackle those in the following sections.

3 Defining Recursive Functions in B

3.1 The problems with recursive set comprehensions

In the previous section we have assumed that set comprehensions are not recursive, i.e., that the truth value of the predicate P inside $\{x_1, \ldots, x_k \mid P\}$ does not depend on the set itself. Let us examine what happens if we drop this restriction. Take for example the two cases: $st = \{x \mid x \in \mathbb{N} \wedge x \in st\}$ and $sf = \{x \mid x \in \mathbb{N} \wedge x \notin sf\}$. The predicates $x \in \mathbb{N} \wedge x \in st$ and $x \in \mathbb{N} \wedge x \in sf$ both depend on the values represented by the set comprehensions themselves. In the first case this means that there are multiple solutions for the equation (st can be any set of natural numbers) while in the second case there is no solution for sf. Assigning a single symbolic representation to st would hide non-determinism in the animator and model checker. More seriously, however, assigning a symbolic representation to sf, even though there is no solution for it, would lead to unsoundness of the animator and model checker.

Note that the set comprehensions $\{x \mid x \in \mathbb{N} \wedge x \in st\}$ and $\{x \mid x \in \mathbb{N} \wedge x \notin sf\}$ on their own are not a problem: these expressions have a single value (for any given value of the free variables st and sf). The problem only arises when treating the equations $st = \{x \mid x \in \mathbb{N} \wedge x \in st\}$ and $sf = \{x \mid x \in \mathbb{N} \wedge x \notin sf\}$: we can no longer simply assign the symbolic closure computed for the set comprehension to st and sf respectively. Our solution to this problem is as follows:

- identify cases where we can guarantee that a recursive set comprehension has a single solution,
- in those cases, provide a special treatment for recursive set comprehensions and unroll them on demand.

3.2 How to define recursive functions in B

The most common use of recursive definitions is to define constant functions (i.e., the function is a B constant defined in the PROPERTIES clause) which perform computations required by the specification. We will restrict ourselves to such cases in this paper. We will present a way to formally write down such recursive functions in B such that they are well defined in B *and* can be animated and validated by ProB. We will illustrate this using the well-known Factorial

function (see, e.g., [5]), trying to define a constant `factorial` which can be used to compute the factorial of a natural number.

Let us first attempt to use lambda abstractions. Unfortunately, in B, these are not well suited to define recursive functions. Indeed—unlike Z—B has no if-then-else expression[7] and hence no way to provide a base case for the recursion. Hence, one can use a lambda abstraction only as a *part* of the function definition:

```
PROPERTIES
  factorial: NATURAL --> NATURAL1 &
  factorial = {0|->1} \/ %x.(NATURAL1 | x* factorial(x-1))
```

While this is a possible way to define the factorial function, it is not particularly elegant and well suited for proof. Furthermore, from an implementation point of view, this style would require us to extend set union to be able to combine (recursive) symbolic closures. We have chosen not to pursue this approach. An alternative specification style is to use universal quantification to express the various cases of the function:

```
PROPERTIES
 factorial: NATURAL --> NATURAL1  &  factorial(0) = 1  &
 !x.(NATURAL1  =>  factorial(x) = x* factorial(x-1))
```

This is already more elegant, and better suited for proof using the B prover tools. However, the definition of the factorial function is "scattered" among different conjuncts of the PROPERTIES clause. It is not obvious how one could translate that into a symbolic closure representation.

Fortunately, it turns out that a set comprehension allows for an elegant specification of the function:

```
PROPERTIES
  factorial: NATURAL --> NATURAL1 &
  factorial = {x,y| x:NATURAL & y:NATURAL &
    (x=0 => y=1) &
    (x>0 => (y=x* factorial(x-1)))  }
```

The advantage over the previous scheme is that the definition of the factorial function now resides within a *single* set comprehension.[8] This will enable us to provide techniques to detect such recursive function definitions and then provide a special symbolic representation for them.

We will of course need to make sure that the recursion is well-founded and progresses towards the base case. Otherwise, we can get a specification like $f \in \mathbb{N} \to \mathbb{N} \wedge f = \{x, y \mid x \in \mathbb{N} \wedge y \in \mathbb{N} \wedge y = f(x+1)\}$ which has multiple solutions (something which we do not want; see the end of Sect. 2). In the above definition

[7] It only has an if-then-else generalised substitution which cannot be used inside a lambda expressions.

[8] The form is less convenient for proof; we return to this issue later.

of `factorial` we can find a variant (x) which ensures that the recursion must terminate.

There is, however, still one problematic issue with the above solution: the soundness of the recursive definition of the function relies on the preceding declaration $fact \in \mathbb{N} \rightarrow \mathbb{N}1$. Without it, we have in principle no guarantee that, for the recursive call $fact(x-1)$, the function is actually defined for $x-1$ and that it actually is a function (and not just a relation; see also [3]).

Thus, a more rigourous definition of the function is as follows:

```
MODEL Factorial
CONSTANTS  factorial
PROPERTIES
   factorial : NATURAL <-> NATURAL &
   factorial =
       { x,y | x: NATURAL & y: NATURAL &
           (x=0 => y=1) &
           (x>0 & x-1:dom(factorial) => #z.(x-1|->z:factorial& y=x*z))}
END
```

It turns out that from this model, we can completely prove within B4Free, the following theorems:

```
THEOREMS
   !P.(P <: NATURAL & 0 : P & succ[P] <: P => NATURAL <: P);
   dom(factorial)=NATURAL;
   factorial : NATURAL --> NATURAL1;
   factorial(0)=1 & !x.(x: NATURAL1  =>  factorial(x)=x*factorial(x-1));
   factorial = { x,y | x: NATURAL & y: NATURAL &
                       (x=0 => y=1) &
                       (x>0  =>  y=x*factorial(x-1))}
```

The first theorem establishes the induction principle over natural numbers; it can be proven by contradiction. We then proceed to prove that the domain of the factorial is the set of natural numbers and that we have defined a total function. (There is thus no need to declare factorial to be a total function; this follows mathematically from the way the function is defined.) From that we can further deduce two alternate formalisations of the factorial function: one well suited for proof with the B prover tools and one close to a functional programming style and well suited for animation. We have thus established full correctness of our initial description as a set comprehension.

The crucial proof is for the theorem `factorial: NATURAL --> NATURAL1` (i.e., `factorial` is a total function). B4free splits this into two subsidiary predicates `dom(factorial) = NATURAL` (`factorial` is total) and `factorial:NATURAL +-> NATURAL1` (`factorial` is a partial function). Both predicates are proved by induction as in [5]. To prove the totality of `factorial` we have instantiated the inductive theorem with `dom(factorial)` and to prove the partial functionality of `factorial` we have instantiated the inductive theorem with the set

`{n | n : NATURAL & 0..n <| factorial :  0..n --> NATURAL1}`.

Remark that we prove also the totality between `0..n` for convenience reasons, otherwise we need to consider (for the inductive step) two cases: `n:dom(factorial)` and `n/:dom(factorial)`.

3.3 A General Scheme

Inspired by the factorial example, we now proceed to present a general scheme for defining recursive functions in B. To ensure well-foundedness, we will require a variant which for simplicity we will assume to be of type natural. In our general scheme, a recursive function with n arguments and N cases is defined as follows:

$$\begin{array}{c} f = \{x_1, \ldots, x_n, out \mid x_1 \in T_1 \wedge \ldots \wedge x_n \in T_n \wedge P(x_1, \ldots, x_n) \wedge out \in T_{n+1} \wedge \\ (Cond_1 \;\Rightarrow\; out = Exp_1) \wedge \\ \ldots \wedge \\ (Cond_N \;\Rightarrow\; out = Exp_N)\} \end{array}$$

In this scheme each $Cond_i$ has free variables included in $\{x_1, \ldots, x_n\}$ and where Exp_i can also make reference to f. The scheme must include a variant function $V \in T_1 \times \ldots T_n \rightarrow \mathbb{N}$.

We introduce the following abbreviation:

$$ArgType \;\equiv\; x_1 \in T1 \wedge \ldots x_n \in T_n \wedge P(x_1, \ldots, x_n)$$

Below we will formalise the conditions under which this scheme ensures that f defines a total function over $T_1 \times \ldots T_n$.

We first ensure that for any input value in $T_1 \times \ldots T_n$ we have one and exactly one condition $Cond_i$ that is true:

1. $\forall(x_1, \ldots, x_n).(ArgType \Rightarrow (Cond_1 \vee \ldots Cond_N))$
2. for any $i \neq j$: $\forall(x_1, \ldots, x_n).(ArgType \Rightarrow \neg(Cond_i \wedge Cond_j))$

The variant must always be a natural number and must always be decreased by each recursive reference:

3. $\forall(x_1, \ldots, x_n).(ArgType \Rightarrow V(x_1, \ldots, x_n) \geq 0)$
4. for any $i \in 1..N$ and for any recursive call $f(e_1, \ldots, e_n)$ inside Exp_i: $\forall(x_1, \ldots, x_n).(ArgType \wedge Cond_i \Rightarrow 0 \leq V(e_1, \ldots, e_n) < V(x_1, \ldots, x_n))$.

More precisely the conditions will ensure that

A. Each $(x_1, \ldots, x_n)$ is in the domain of f:

$$\forall(x_1, \ldots, x_n).(ArgType \Rightarrow (x_1, \ldots, x_n) \in dom(f))$$

B. Each $(x_1, \ldots, x_n)$ is mapped to at most one range value:

$$\begin{array}{c} \forall(x_1, \ldots, x_n, y_1, y_2).(ArgType \;\wedge\; (x_1, \ldots, x_n) \mapsto y_1 \in f \;\wedge\; (x_1, \ldots, x_n) \mapsto \\ y_2 \in f \quad\Rightarrow\quad y_1 = y_2) \end{array}$$

The proof of A and B is by general induction over the range of variant function V, that is, assuming that the property holds for all $(x'_1, \ldots, x'_n)$ where $V(x'_1, \ldots, x'_n) < V(x_1, \ldots, x_n)$ show that it holds for $(x_1, \ldots, x_n)$.

4 Implementation: Recursive Closures

We now show how the implementation scheme of Sect. 2 can be adapted to deal with recursive set comprehensions. For this we introduce a second symbolic representation, namely for recursive closures.

The first part of our implementation is the extension in the ProB kernel of the equality Prolog predicate. This Prolog predicate gets called whenever an equality between two B objects needs to be checked. If none of the objects is a symbolic closure then the check proceeds as usual [9]. If both objects are closures, then they are both expanded and checked for equality. Let us now assume that one of the objects is a closure C, as explained in Sect. 2, while the other (say f) is not. (This situation would arise for the predicate $f = \{x_1, \ldots, x_n \mid P\}$, unless f was already given a symbolic closure by some preceding predicate.) There are now three cases for f:

- f has some value associated with it (or is partially instantiated); in this case the closure for the set comprehension gets expanded and the equality is checked as usual.
- f is unconstrained (i.e., apart from the type, nothing is known about f) but does not appear in P: in this case f is now set to be equal to the symbolic closure C.
- f is unconstrained and does appear in P: in this case we have a recursive set comprehension. There two sub-cases:
  - The equality $f = \{x_1, \ldots, x_n \mid P\}$ does not conform to the scheme outlined in Sect. 3.3. In this case the Prolog predicate needs to delay until f is completely known, at which point the set comprehension can be expanded and the equality checked.
  - Otherwise we generate a new identifier `ID` to identify the recursive function and replace all occurences of f inside the C with `rec(ID)`, yielding C'. The variable or constant f now gets as value the symbolic representation `recursive_closure(ID,`C'`)`.

The equality predicate is also the only place which can lead to the generation of a new recursive closure. For our `factorial` constant above we would thus get as symbolic representation:

```
recursive_closure(1, closure([x,y],[INTEGER,INTEGER],
      'x:NATURAL & y:NATURAL & (x=0 => y=1) &
       (x>0 => y=x*value(rec(1))(x-1))' ))
```

These recursive closures are then unrolled on demand. More precisely, when a recursive closure τ = `recursive_closure(`i`,`C`)` is examined (e.g., by the function application Prolog predicate) it is converted into the closure $C' = C[\tau/rec(i)]$, i.e. C where all $rec(i)$ have been replaced by τ.

For example, unrolling the above recursive closure yields:

```
closure([x,y],[INTEGER,INTEGER],
      'x:NATURAL & y:NATURAL & (x=0 => y=1) &
       (x>0 => y=x*value(recursive_closure(1,...))(x-1))' ))
```

If the first argument is 0 then no further unrolling is required, but if $x > 0$ the inner recursive closure will be unrolled, etc., until we reach the base case of the recursion.

Note that this way to handle recursion is related to the fix operator sometimes used in process algebras (see, e.g., [11]).

New Syntax

The introduction of a new syntax for recursive functions can provide both an effective way to animate recursive functions as well as a convenient way to prove properties with and about them. This would require extending the ProB parser and then either convert the syntax into a form suitable for proving or suitable for ProB for animating and model checking. A possible syntax for the factorial function could be:

```
FUNCTIONS
   y <-- factorial(x) = WHERE x:NAT & y:NAT THEN
                              CASE x=0 THEN y=1
                              CASE x>0 THEN y=fact(x-1)
                           VARIANT x
                        END
```

We have not yet finalised the new syntax, and in the remainder of the paper we carry out our experimentation with the set comprehension style suitable for animation.

5 Higher-Order Functional Programming Examples in B

So far we have shown that first-order recursive functions can be encoded within B, and animated and validated by ProB. As it turns out, our scheme is actually powerful enough to animate higher-order recursive functions. In other words, we have actually developed a scheme to enable higher-order functional "programming" inside B specifications. In this section we provide a few examples to illustrate this point.

Mutual Recursion Before examining higher-order functions, let us first discuss the issue of mutual recursion. The examples so far contained a single recursive function. Does our scheme also work for several mutually recursive functions? The answer to this question is affirmative: the proof obligations need to be adapted to handle multiple functions (in a straightforward fashion), but the implementation scheme is already suited to handle such functions.

Take the following artificial example, splitting the factorial function into two mutually recursive functions:

```
CONSTANTS fact1,fact2
PROPERTIES
  fact1: INT --> INT &
  fact1 = {x,y | x:NAT & y:NAT &
                (x=0 => y=1)   &    (x>0 => (y=x*fact2(x-1))) }  &
  fact2: INT --> INT  &
  fact2 = {x,y | x:NAT & y:NAT &
                (x=0 => y=1)   &    (x>0 => (y=x*fact1(x-1)))  }
```

In this case, `fact1` will be stored as a standard closure (calling `fact2`) and `fact2` will be a recursive closure with no reference to `fact1`. Note that we can also deal with the problematic example discussed in [8].

Higher-Order Functional Programming Some higher-order programming is actually already built into B: to *map* a function f over a sequence s we simply need to use the relational composition $(s; f)$, as s is a function from $1..size(s)$ to $ran(s)$. In general, however, this is not so easy. Below we show how we can specify the well-known "`foldr`" higher-order function, which takes a base value and a function f and maps it over a sequence to compute a single value. In the `FoldMul` operation we use this higher-order function to compute the product of the elements of a sequence.

```
MACHINE SeqFoldr
CONSTANTS mul, foldr
PROPERTIES
  mul: (NATURAL*NATURAL)<->NATURAL &
  mul = {i,j,res | i:NATURAL & j:NATURAL & res:NATURAL & res=i*j} &
  foldr:(((NATURAL*NATURAL)<->NATURAL)*NATURAL*seq(NATURAL))<->NATURAL &
  foldr =
    { f,base,i,res | i:seq(NATURAL) & base:NATURAL &
                     res: NATURAL & f:(NATURAL*NATURAL)-->NATURAL &
           (i=<>  => res=base) &
           (size(i)>0 =>  res =  f(first(i),foldr(f,base,tail(i))) )
    }
VARIABLES ss
INVARIANT ss: seq(NATURAL)
INITIALISATION ss  := <>
OPERATIONS
   Add(nn) = PRE nn:NATURAL  THEN ss := ss <- nn END;
   FoldMul = BEGIN ss := foldr(mul,1,ss) -> ss  END
END
```

We can easily show that `foldr` satisfies our proof obligations, if we choose $size(i)$ as the variant:

1. $\forall(f, i).(i \in seq(\mathbb{N}) \wedge f \in \mathbb{N} \times \mathbb{N} \rightarrow \mathbb{N} \Rightarrow (i = \langle\rangle \vee size(i) > 0))$
2. $\forall(f, i).(i \in seq(\mathbb{N}) \wedge f \in \mathbb{N} \times \mathbb{N} \rightarrow \mathbb{N} \Rightarrow \neg(i = \langle\rangle \wedge size(i) > 0))$
3. $\forall(f, i).(i \in seq(\mathbb{N}) \wedge f \in \mathbb{N} \times \mathbb{N} \rightarrow \mathbb{N} \Rightarrow size(i) \geq 0)$
4. $\forall(f, i).(i \in seq(\mathbb{N}) \wedge f \in \mathbb{N} \times \mathbb{N} \rightarrow \mathbb{N} \wedge size(i) > 0 \Rightarrow size(tail(i)) < size(i))$

One difference with functional programming still persists though: `foldr` is usually defined to be a polymorphic function, i.e., the type of the elements of the list is not hardcoded like in our B machine. To overcome this, one has to define `foldr` within a separate machine, whose argument is the type of the elements of the list. Unfortunately, the machine parameters are not visible inside the PROPERTIES clause; hence we actually need to make `foldr` a variable (which is not very convenient):

```
MODEL Foldr(TYP)
VARIABLES foldr
INVARIANT foldr : (((TYP* TYP)--> TYP)*TYP*seq(TYP)) --> TYP
INITIALISATION
  foldr : (foldr = { f,b,i,res |
             i:seq(TYP) & res:TYP & f:(TYP* TYP)--> TYP & b:TYP &
             (i=<>  => res=b) &
             (size(i)>0 =>  res =  f(first(i), foldr(f,b,tail(i))) )  } )
END
```

6 Empirical Results

The experiments were all run on a multiprocessor system with 4 AMD Opteron 870 Dual Core 2 GHz processors, running SUSE Linux 10.1, SICStus Prolog 3.12.7 (x86_64-linux-glibc2.3) and ProB version 1.2.4.[9]

The first experiment consisted in running the previously discussed `factorial` function. The results are presented in the upper half of Table 1. Note that ProB does check the function arguments (to see if they are a natural number) at every function application. Also, note that without our new symbolic approach, the present formalisation cannot be animated (even for small values of n). However, using the axiomatic formalisation from Sect. 3.2 (using universal quantification) it is possible, provided we limit the domain of the function. It then takes 0.85 sec to compute factorial for 0..100 using classical ProB. (For functions such as `SeqFoldr` as seen earlier, it is of course impossible to precompute the function.)

In order to measure a specification requiring a large number of recursive calls we have used the naïve recursive definition for computing the Fibonacci numbers: `fib = {x,z| x:NATURAL & z:NATURAL & (x=0 => z=1) & (x=1 => z=1) & (x>1 => (z=fib(x-1)+fib(x-2))) }`. The results are summarised in the same Table 1. For Fib(20) we have 21891 ($2 \times fib(20) - 1$, see, e.g., [13]) calls to the Fibonacci function. This corresponds to 3357 calls per second. For a programming language this would of course be very slow (even though ProB works with big integers); but for animation purposes this is actually quite reasonable (also given the fact that the typing predicates are repeatedly evaluated). However, there is definitely scope for improvement. Possibly with the use of partial evaluation [7] and more sophisticated implementation techniques, a big improvement in speed should be possible. Still, in its current form the tool can be used to animate a wide

[9] Note that neither SICStus Prolog nor ProB take advantage of multiple processors.

n	factorial(n)	Time	Function calls per sec
5	120	0.00 sec	-
10	3,628,800	0.00 sec	-
20	2,432,902,008,176,640,000	0.02 sec	1000
100	see footnote[10]	0.06 sec	1667

n	fib(n)	Time	Function calls per sec
5	8	0.01 sec	1500
10	89	0.10 sec	1770
15	987	0.67 sec	1999
20	10946	6.52 sec	3357

Table 1. Empirical Results

range of specifications with recursive functions. In particular, ProB has been successfully applied to an industrial case study (cf. Section 1) which was hitherto impossible to animate or model check, and was able to detect an error in the original specification [12].

7 Related Work and Conclusion

While there are various other animators for B and Z, to our knowledge, none of them can handle recursive functions.

In [2] authors explain how we can specify higher order expression and theorems using B and how we can prove such theorems using B tools. The second work [5] is more related to our work. The `factorial` function is also defined first like the smallest relation which satisfies `factorial`'s properties and then the proof of the functionality of `factorial` is done using B4free as in our work. This definition is not suited for animation but some algorithms to compute the `factorial` function are given. These algorithms are developed using B and the refinement. These are correct by construction. The first abstract model computes `factorial(n)` in one shot. The first refinement computes a finite subset of the `factorial` function like in dynamic programming The last refinement computes `factorial(n)` using the well known loop from `1` to `n`. Another related work is [14], which presents a framework to reconcile axiomatic and model-based specifications. As such it is related to our desire at the end of Sect. 4 to present two different views of a specification: one suitable for proving and one suitable for animation. In the context of higher-order logic and Isabelle/HOL, [8] presents a method to reason about recursive functions.

In summary, we have presented a scheme to define higher-order recursive functions in B and have shown how we can animate and model check them

[10] For factorial(100) the result computed by ProB is:
933262154439441526816992388562667004907159682643816214685929638952175999932299156089414639761565182862536979208272237582511852109168640000000000000000000000000.

using extensions to the ProB toolset. We have carried out various experiments, showing the practicality of the approach.

Acknowledgements We would like to thank Daniel Plagge and Jens Bendisposto for very useful comments and discussions. We would also like to thank the anonymous referees for their insightful suggestions and pointers to related work.

References

1. J.-R. Abrial. *The B-Book*. Cambridge University Press, 1996.
2. J.-R. Abrial, D. Cansell, and G. Laffitte. Higher-order mathematics in B. In D. Bert, J. P. Bowen, M. C. Henson, and K. Robinson, editors, *Proceedings ZB'2002*, LNCS 2272, pages 370–393, 2002.
3. J.-R. Abrial and L. Mussat. On using conditional definitions in formal theories. In D. Bert, J. P. Bowen, M. C. Henson, and K. Robinson, editors, *Proceedings ZB'2002*, LNCS 2272, pages 242–269, 2002.
4. M. Butler. A system-based approach to the formal development of embedded controllers for a railway. *Design Automation for Embedded Systems*, 6(4):355–366, 2002,.
5. D. Cansell and D. Méry. Foundations of the B method. *Computing and informatics*, 22(3–4):221–256, 2003.
6. E. M. Clarke, O. Grumberg, and D. Peled. *Model Checking*. MIT Press, 1999.
7. N. D. Jones, C. K. Gomard, and P. Sestoft. *Partial Evaluation and Automatic Program Generation*. Prentice Hall, 1993.
8. A. Krauss. Partial recursive functions in higher-order logic. In U. Furbach and N. Shankar, editors, *Proceedings IJCAR'06*, LNCS 4130, pages 589–603. Springer, 2006.
9. M. Leuschel and M. Butler. ProB: A model checker for B. In K. Araki, S. Gnesi, and D. Mandrioli, editors, *FME 2003: Formal Methods*, LNCS 2805, pages 855–874. Springer-Verlag, 2003.
10. M. Leuschel and M. Butler. Automatic refinement checking for B. In K.-K. Lau and R. Banach, editors, *Proceedings ICFEM'05*, LNCS 3785, pages 345–359. Springer-Verlag, 2005.
11. T. Massart and R. Devillers. Equality of agent expressions is preserved under an extension of the universe of actions. *Formal Aspects of Computing*, 5(1):79–88, 1993.
12. D. Plagge and M. Leuschel. Validating Z Specifications using the ProB Animator and Model Checker. In J. Davies and J. Gibbons, editors, *Integrated Formal Methods (IFM 2007)*, LNCS 4591, pages 480–500. Springer-Verlag, 2007.
13. J. S. Robertson. How many recursive calls does a recursive function make? *SIGCSE Bull.*, 31(2):60–61, 1999.
14. K. Robinson. Reconciling axiomatic and model-based specifications using the B method. In J. P. Bowen, S. Dunne, A. Galloway, and S. King, editors, *Proceedings ZB 2000*, LNCS 1878, pages 95–106. Springer, 2000.
15. P. V. Roy and S. Haridi. *Concepts, Techniques, and Models of Computer Programming*. MIT Press, 2004.