# PrologCheck – Property-Based Testing in Prolog

Cláudio Amaral[1,2], Mário Florido[1,2], and Vítor Santos Costa[1,3]

[1] DCC - Faculty of Sciences, University of Porto, Porto, Portugal
[2] LIACC - University of Porto, Porto, Portugal
[3] CRACS - University of Porto, Porto, Portugal
{coa,amf,vsc}@dcc.fc.up.pt

**Abstract.** We present PrologCheck, an automatic tool for property-based testing of programs in the logic programming language Prolog with randomised test data generation. The tool is inspired by the well known QuickCheck, originally designed for the functional programming language Haskell. It includes features that deal with specific characteristics of Prolog such as its relational nature (as opposed to Haskell) and the absence of a strong type discipline.

PrologCheck expressiveness stems from describing properties as Prolog goals. It enables the definition of custom test data generators for random testing tailored for the property to be tested. Further, it allows the use of a predicate specification language that supports types, modes and constraints on the number of successful computations. We evaluate our tool on a number of examples and apply it successfully to debug a Prolog library for AVL search trees.

## 1 Introduction

Software testing consists of executing a program on a pre-selected set of inputs and inspecting whether the outputs respect the expected results. Each input tested is called a *test case* and the set of inputs is a *test suite*. Testing tries to find counter-examples and choosing the test cases to this effect is often a difficult task. The approach used can be manual, with the tester designing test cases one by one, or it can be automated to some extent, in this case resorting to tools for case generation. Ideally, the best approach would be automatic testing.

In a property-based framework test cases are automatically generated and run from assertions about logical properties of the program. Feedback is given to the user about their evaluation. Property-based testing applications include black-box, white-box, unit, integration and system testing [3] [6,7].

Property-based testing naturally fits the logic programming paradigm. Assertions are first order formulas and thus easily encoded as program predicates. Therefore, a property based approach to testing is intuitive for the logic programmer.

In this paper we introduce PrologCheck[1], a property-based testing framework for Prolog. We further discuss two main contributions: a specification language for Prolog predicates and a translation procedure into testable properties.

[1] The PrologCheck tool is available at `www.dcc.fc.up.pt/~coa/PrologCheck.html`

In most programming languages interfaces to testing frameworks rely on boolean functions, such as equality, to determine primitive properties. PrologCheck states properties through a domain-specific language that naturally supports domain quantification. In this language primitive properties are Prolog goals which can be composed by PrologCheck property operators.

PrologCheck testing consists on repetitively calling the goal for a large number of test cases. Input to such goals is based on PrologCheck value abstraction, quantification over a domain represented by a randomised generator of terms. We implement randomised test case generation, which frees the user from choosing input manually. We include a number of predefined generators for relevant sets of terms, such as integers, and combinators to help define new generators. Thus other generation techniques [10] [16] [18] can be implemented to complement the power of built-in generators.

We also define a language of testable predicate specifications including types, modes and multiplicity, which the tester can use to encode interesting properties of the predicate under test. By specifying some aspects of a predicate in a proper specification language it is possible to generate a PrologCheck property and check it. This allows us to use PrologCheck and its predicate specification to test a number of non-trivial programs.

The rest of this paper is organised as follows. We proceed with motivating examples in Sec. 3. Section 2 encloses the presentation of related work. In Sec. 4 we introduce property definitions and their testing in PrologCheck and in Sec. 5 we discuss details about test case generation. Section 6 describes the predicate specification language and how to test the specifications. A case study of AVL trees is presented Sec. 7. We finalise with the conclusions in Sec. 8.

## 2 Related Work

There is some previous support for automated testing in the logic programming community: SWI-Prolog supports unit testing through plunit [20]; the Ciao Prolog System [13] has an assertion language integrating run-time checking and unit testing [15]. We use a property specification language but in an automatic property-based randomly generated testing context. Property specification languages for Prolog were used before [9] [15] [19] in different contexts.

Automated testing is supported in several languages and paradigms. The three most influential tools for our work were QuickCheck [5] for the functional programming language Haskell, PropEr [17] for the functional programming language Erlang and EasyCheck [4] for the functional-logic language Curry.

Easycheck is an automated tool for specification-based testing of declarative programs, which deals with logic programming features. It is written in the functional-logic programming language Curry and it is based on the view of free variables of an appropriate type as non-deterministic generators [1] and mechanisms to specify properties of non-deterministic operations by generalizing the set of combinators for non-deterministic operations of QuickCheck. In our work we focus on Prolog and, in contrast with EasyCheck, non-deterministic generators are implemented by non-deterministic Prolog programs, types are implemented by monadic logic programs [11,12], and we use a specification language for standard features of logic programming such as modes and number of answers [9].

There are several automatic testing tools for functional programming languages, namely QuickCheck, PropEr, SmallCheck [18], and G∀ST [14]. The first and most preeminent tool is QuickCheck. QuickCheck uses a domain specific language of testable specifications as does PropEr. We define a specification language in PrologCheck but with differences related to the relational nature of Prolog. As in QuickCheck, we use random testing - we choose this method compared to systematic methods due to its success in QuickCheck. QuickCheck generates test data based on Haskell types. In Erlang, types are dynamically checked and PropEr, as does as ErlangQuickCheck, guides value generation by functions, using quantified types defined by these generating functions. Prolog is an untyped language, but type information is crucial in PrologCheck test data generation as well. Similarly to the Erlang tools, we adopt the view of types defined by test case generators. Our types are intended to construct test cases that depict input instantiations. Thus we would not take advantage of the use of restricted type languages based on regular types [11,12] [21,22].

## 3 Motivating Examples

### 3.1 Append

Consider the well-known concatenation predicate.

```
app([], YS, YS).
app([X|XS], YS, [X|AS]) :- app(XS, YS, AS).
```

We specify the predicate behaviour through the predicate specification language presented in Sec. 6. The properties and predicates to be tested are in module `m`.

`app(A,B,C)` is used in a functional way in many programs, i.e., by giving it two lists as input and getting their concatenation as output. The directionality is determined by the modes of each parameter: *ground, ground, variable* to *ground, ground, ground*. The range of answers for a predicate with a (total) functional behaviour is exactly one. This behaviour is specified in PrologCheck as:

```
app of_type (A-(listOf(int)), B-(listOf(int)), C-(variable))
   where (i(g, g, v), o(g, g, g))  has_range {1,1}.
```

The property originated by this specification clause passes the tests generated by the tool.

```
   ?- prologcheck(m:prop(spec_app)).
OK: Passed 100 test(s).
```

`app/3` may be used in other situations. One can use it to create an open list bound to the third parameter by calling it with a variable in the second input parameter, which remains uninspected. The result is a list with the ground elements of the list in the first parameter and the variable in the second parameter as the tail, therefore it is neither ground nor variable. This usage also behaves as a function. We state this as specification clause `1` of predicate `app`.

```
{app, 1} of_type (A-(listOf(int)), B-(variable), C-(variable))
  where (i(g, v, v), o(g, v, ngv)) has_range {1,1}.
```

Testing reveals that the *out* part of the directionality is not satisfied.

```
 ?- prologcheck(m:prop(spec_app_1), [noshrink]).
{failed_out_modes,[[o,g,v,ngv]], [[],_10258,_10258]}
Failed: After 3 test(s).
Counterexample found: [[[],_10258,_10260]]
```

The counterexample shows that the output modes do not respect the specification when the first input parameter is the empty list. One way to solve this issue is to add the missing directionality `(i(g, v, v), o(g, v, ngv)), o(g,v,v)`. Although, the correct choice in general is to split the input types, since this is a matter between disjoint sets of terms. Multiple output directionalities are mainly intended for multiple modes of multiple answers.

```
{app, 1a} of_type (A-(listOf1(int)), B-(variable), C-(variable))
  where (i(g, v, v), o(g, v, ngv)) has_range {1,1}.

{app, 1b} of_type (A-(value([])), B-(variable), C-(variable))
  where (i(g, v, v), o(g, v, v)) has_range {1,1}.
```

## 3.2 List Reverse

Let us explore an example of list reversing predicates. The reversing procedure relates two lists and is polymorphic in the type of the list's elements. It is usually sufficient to check the behaviour for a single type of elements. Moreover, sometimes even a type with a finite number of values suffice, but we can safely overestimate the size of the type [2]. Therefore, we use the generator for integers, `int`, as the elements of the parametric list generator, `listOf(int)`.

```
rev([],[]).
rev([X|XS], YS) :- rev(XS,ZS), append(ZS, [X], YS).
```

We express the symmetry of the reversing relation in terms of its intended use: given a ground list in one input parameter retrieve a result in the other.

```
prop(d_rev) :- for_all(listOf(int), XS, (rev(XS, RX), rev(RX, XS)))
```

*Prologchecking* the property bears no surprises.

```
  ?- prologcheck(m:prop(d_rev)).
OK: Passed 100 test(s).
```

We could have mis-typed the property, making it impossible to be satisfied:

```
prop(wrong_dr) :-
  for_all(listOf(int), XS, (rev(XS,RX), rev(RX,RX))).
```

We mistakenly make the second call to `rev/2` with `RX` as the second parameter.

```
  ?- prologcheck(m:prop(wrong_dr)).
Failed: After 11 test(s).
Shrinking (6 time(s))
Counterexample found: [[0,6]]
```

A counterexample is found and shrunk to the presented counter-example `[0,6]`.

To check that the order is being reversed we can randomly choose an element (or a set of elements) and inspect its position in the parameters. Choosing random elements prevent us from checking the whole list.

```
prop(rev_i) :- plqc:for_all(
    suchThat(structure({listOf(int), int}), m:valid_index),
    {L,I}, m:prop({double_rev_i_body, L, I}) ).
valid_index({L, I}) :- length(L,X), I<X.

prop({double_rev_i_body, L, I}) :-
  m:rev(L, LR), length(L,X), Index is I+1, RevIndex is X-I,
  lists:nth(Index, L, Val), lists:nth(RevIndex, LR, Val).
```

When performing a large number of tests this method should randomly choose enough indexes to give good element coverage.

```
  ?- prologcheck(m:prop(rev_i)).
OK: Passed 100 test(s).
```

We have another implementation of reverse, using an accumulator instead of concatenation. The previous properties can be adapted to this implementation with the same results.

```
rev_acc([], LR, LR).
rev_acc([X|XS], Acc, LR) :- rev_acc(XS, [X|Acc], LR).

rev_acc(L, LR) :- rev_acc(L, [], LR).
```

Since we have two implementations of the same concept we can explore this by stating and testing a property comparing their behaviours.

```
prop(eqv_acc_app) :-
  for_all(listOf(int), L, (rev_acc(L,LR),rev(L,LR))).
```

The comparison succeeds if both have the same behaviour.

```
  ?- prologcheck(m:prop(eqv_acc_app)).
OK: Passed 100 test(s).
```

## 4 Properties

Property-based testing extends program code with *property definitions*. Properties are specifications in a suitable language and are tested automatically by generating test cases.

PrologCheck is a property-based testing framework. Given a specification it randomly generates test cases for the properties to be tested, executing them to assess their validity. A *primitive property* is a Prolog goal, hence, the whole language can be used to define properties. Properties may then be composed according to composition rules described later in the paper. This enables the specification of a wide range of properties. Next, in this section we introduce PrologCheck through the append example.

We will go through the process of using the tool, beginning by turning a logical statement of a property into a PrologCheck testable property. An example of a property of `app/3` is that, assuming the first two input parameters are lists, after its execution a variable given in the third input parameter is instantiated with a list. This property can be represented by the first order formula

$$\forall\, l_1, l_2 \in \mathit{list}.\ (l_1 +\!\!+\, l_2 \in \mathit{list})$$

where $l_1$ and $l_2$ denote lists given as input and $+\!\!+$ is interpreted as list concatenation. The primitive property in the formula, $l_1 +\!\!+\, l_2 \in \mathit{list}$, can then be represented by the goal

```
  app(L1, L2, L), (L = []; L = [_|_]).
```

The next step is optional. We explicitly parametrise the property into a first order object. The resulting property is written as a clause for the special predicate `prop/1` and parametrised accordingly.

```
prop({appLLL, L1, L2}) :- app(L1, L2, L), (L = []; L = [_|_]).
```

This is PrologCheck's predicate for labelling properties. The parametric label, `{appLLL, L1, L2}` in the example, uniquely identifies the property and holds the variables for all the input required. The symbol `appLLL` is the "*append of lists results in list*" property identifier and the variables `L1, L2` the input. The body of labelled properties is inspected by PrologCheck, making it possible to abstract long or frequently used properties.

A last step is needed to verify properties with PrologCheck. In order to enable random testing, we define a domain of parameter instantiations. Values from this domain are used as test cases.

```
prop(appL) :- for_all(listOf(int), L1, for_all(listOf(int), L2,
                    prop({appLLL, L1, L2}))))
```

This more precise definition states that the property `appL` is `appLLL` over two lists of integers. More accurately, we use `for_all/3` to represent PrologCheck's universal quantification. The first input parameter describes the *type* of terms we want to generate randomly, in this case lists of integers, `listOf(int)`, and the second input parameter names the variable they will bind to, in this case `L1` and `L2`. The third is the property we want to verify. To check the property we can call PrologCheck using the alias `prop(appL)`. It starts with the outer `for_all` quantifier, generates a random list of integers, unifies it with `L1` and repeats the process for the inner quantifier, unifying `L2` with the second generated list.

```
  ?- prologcheck(m:prop(appL)).
OK: Passed 100 test(s).
```

The `prologcheck/1` predicate is the simplest property tester in PrologCheck, taking a property as a parameter and checking it for a large number (100 is the default number) of generated test cases.

We could have mis-typed the property, making it impossible to be satisfied:

```
prop(wrong_appL) :- for_all(listOf(int), L1,
  for_all(listOf(int), L2, (app(L1, L2, L), (L=[], L=[_|_])))).
```

We mistakenly determine `L` to be both `[]` and `[_|_]`.

```
  ?- prologcheck(m:prop(wrong_dr)).
Failed: After 1 test(s).
Shrinking (1 time(s))
Counterexample found: [[],[]]
```

A counterexample is found and showed. We observe at this point that a counterexample is immediately found. There is no possible value that can satisfy the written condition.

Often we want to find concise counterexamples. To do this we use a *shrinking* predicate that tries to reduce the counterexample found. To improve the probability of finding smaller counter-examples the tool keeps track of a growing *size parameter*. This parameter starts at an initial value and is updated with each successful test. Its purpose is to control the size of produced test cases and it is used in test case generation. The definition of the actual size of a term is flexible and definable by the generating procedure.

We can define general properties or define sub-properties individually. We can, for example, separate property `appLLL` into `appLLE` and `appLLC` to state the empty list and *cons* cell separately and compose them with property operators.

```
prop({appLLE, L1, L2}) :- append(L1, L2, L), L = [].
prop({appLLC, L1, L2}) :- append(L1, L2, L), L = [_|_].
```

Property operators currently include conjunction (`Prop1 and Prop2`), disjunction (`Prop1 or Prop2`), conditional execution (`if Cond then Prop1 else Prop2)`) and quantification (`for_all(Gen, Var, Prop)`). Property labelling (`prop(Label)`) is considered an operation. PrologCheck inspects its body for the occurrence of other tool specific operations. Using property connectives one can compose labelled properties or other PrologCheck property operations.

We now define other properties of `app/3`, such as the relation of lists' lengths and the left and right identity element of concatenation.

```
prop({appLLLen, L1, L2}) :- app(L1, L2, L),
  length(L1, K1), length(L2, K2), length(L, K), K is K1 + K2.
prop({appLZ,L1,L2}) :- if L1=[] then (app(L1,L2,L), L=L2).
prop({appRZ,L1,L2}) :- if L2=[] then (app(L1,L2,L), L=L1).
```

Conjunction and disjunction is used as expected. The conditional statement `if A then B else C` performs a conditional execution on `A`. If `A` runs successfully the tool continues by executing `B` and in case it fails executing `C` instead. `A`, `B` and `C` are PrologCheck properties. In the example shown the `else` branch is omitted. This is equivalent to having the property `true` in the omitted branch. The conditional statement enables conditional properties without cut.

```
prop(appAll) :- for_all(listOf(int),L1, for_all(listOf(int),L2,
  (prop({appLLLen, L1, L2}) and prop({appLZ, L1, L2})
  and prop({appRZ, L1, L2}) and prop({appLLL, L1, L2})
  and (prop({appLLE,L1,L2}) or prop({appLLC,L1,L2}))))).
```

Primitive properties are Prolog goals. In a strongly typed language (such as Haskell) only safe properties, pure functions or predicates, are allowed. In PrologCheck the user is free to use simpler or more involved properties. This provides extra flexibility but, ultimately, the user is responsible for guaranteeing the safety of impure code in a property.

## 5 Generators

Input for testing properties is randomly generated through explicitly defined procedures: *generators*. There are differences between PrologCheck generators and the generators in a strongly typed version of the tool. In Haskell QuickCheck, or any language with strong types, generators pick values inside a preexisting type according to some criteria. In PrologCheck generators represent procedures that randomly construct elements according to the shape of the term. In fact, the generators themselves define a set by the elements they generate, with non-zero probability. Thus, they define a set of terms, here denoted as a *type*. Note that this set of terms is not necessarily composed of only ground terms, instead it exactly represents the form of an input parameter to a property.

PrologCheck has *generators* and *generator predicates*. Generators specify the input parameters of properties. One example generator is `listOf(int)`. Generator predicates are the predicates responsible for the generation of test cases. The corresponding example of a call to a generator predicate is `listOf(Type, Output, Size)` where `Type` would be bound to `int`, `Output` would be instantiated with the produced test case and `Size` would be used to control the size of produced test cases. The value is passed to the property by the PrologCheck quantification through unification.

`choose/4` and `elements/3` are examples of generator predicates. Picking an integer in an interval is probably the most common operation in generators. The `choose/4` predicate discards the size parameter and randomly chooses an integer between the inclusive range given by the first two input parameters. `elements/3` randomly chooses an element from a non-empty list of possible elements. They are implemented as follows:

```
elements(AS, A, S) :-
  length(AS, Cap), choose(1,Cap,I,S), nth(I, AS, A).

choose(Min,Max, A, _) :- Cap is Max+1, random(Min,Cap,A).
```

**Combinators.** We extend generator predicates with *generator combinators* that allow us to define more complex generators. More precisely, combinators are generator predicates that are parametrised by generators. This is shown in the `app/3` example, where the generator for lists, `listOf(int)`, is parametrised by a generator for integers. Generator predicates can have several parameters, but the two last must always be, in this order, the generated value and the size. When a generator predicate uses another generator predicate to build a value, the parameter is passed in generator form.

PrologCheck combinators enable the generation of complex data and can tune the probability distribution of the generated values to better fit the needs of the tester. Next, we present some combinators distributed with the tool.

To generate lists we provide generators for arbitrary and fixed length lists. They are parametrised by a generator for the list elements. Random size lists can be generated by `listOf/3`, which randomly chooses a list length and uses `vectorOf/4`. Non-empty lists are generated by the `listOf1/3` variation. `vectorOf/4` is a fixed length generator predicate that recurs on the integer given as the first input parameter, generating each element.

```
listOf(GenA, AS, S) :- choose(0, S, K, S),vectorOf(K, GenA, AS, S).

listOf1(GenA, AS, S) :- max_list( [1,S], Cap),
  choose(1, S, K, S), vectorOf(K, GenA, AS, S).

vectorOf(0, _GenA, [], _Size) :- !.
vectorOf(K, GenA, [A|AS], Size) :-
  call(GenA, A, Size), K1 is K-1, vectorOf(K1, GenA, AS, Size).
```

Combinators can interact and, for example, create lists of random length in an interval ($[2,5]$) and create lists whose elements are in an interval ($[0,9]$).

```
  for_all( choose(2,5),I, for_all( vectorOf(I,int),L1,
    for_all( listOf(choose(0,9)),L2, (prop({appLLC, L1, L2}))))
```

Generating specific values, ground or not, fresh variables and terms with a certain structure is possible with `value/3`, `variable/2` and `structure/3` respectively. With such generators/combinators we can describe and therefore test a different input mode.

```
  for_all( structure([listOf(int), value(v), variable]),[L1,X,L],
    app(L1, X, L))
```

If the values or part of the values to be generated have to be of a certain size, we override the size parameter with the `resize/4` combinator.

```
resize(NewSize, GenA, A, _Size) :- call(GenA, A, NewSize).
```

Resizing can contribute to better chances of fulfilling a condition, e.g., a size near zero improves the chances of generating empty lists.

```
for_all(  resize(0,listOf(int)),  L1,
  for_all( listOf(int), L2, (prop({appLZ, L1, L2})) ))
```

The `suchThat/4` combinator restricts the values of a generator. If not all generated elements for a generator are useful, wrapping it with `suchThat/4` will select the elements of the generator in the first input parameter that satisfy the predicate in the second. If a generated value is valid it is returned; if not, the size parameter is slowly increased to avoid a size without valid values. This is a dangerous combinator in the sense that it can loop indefinitely if the the valid values are too sparse. We can restrict a list generator so that it only generates non-empty lists.

```
posLen([_|_]).
...
  for_all(  suchThat(listOf(int), posLen),  L1,
    for_all(listOf(int),L2, (prop({appLLC, L1, L2}))))
```

Often, it is hard to find a good generator. Choosing from a set of generators that complement each other is a good way to generate values with a desired distribution. Grouping generators can be done in several ways. We can randomly choose from a list of generators with `oneof/3`. The list of generators given in the first input parameter must be non-empty.

```
oneof(LGenA, A, S) :- length(LGenA, Cap), choose(1,Cap,I,S),
  nth(I, LGenA, GenA), call(GenA, A, S).
```

If an uniform distribution between the generators is not suitable one can specifically state the proportions of the probabilities to choose each generator. The first input parameter of `frequency/3` is a list of pairs {*weight*,*generator*} representing such proportions. The input list must be non-empty. A frequency-index list is created with the correct proportions and a generator is then randomly chosen from that list to be called.

```
frequency(FGL, A, S) :- checkFreqWeights(FGL, FIL, Cap),
  choose(1,Cap,I,S), nth(I, FIL, GenA), call(GenA, A, S).
```

We can use both combinators to randomly choose generators for each test case.

```
Gen1 = resize(0,listOf(int))
Gen2 = suchThat(listOf(int), posLen)
...
  for_all( frequency([{4,listOf(int)}, {1,Gen2}]),  L1,
    for_all( oneof([Gen1,Gen2],L2, (prop({appLLC, L1, L2}))))
```

**Shrinking.** When a test fails the tool may try to simplify the failing input to a smaller and easier to understand counterexample. Shrinking is a process by which a *shrinker* predicate returns a possibly empty list of smaller elements than the one given as input.

Similarly to generator predicates, shrinkers are calls to the corresponding generator. To trigger shrinking a generator is called with the value to shrink, the flag `shrink` and a variable to store the list of shrunk values. An example of a shrinker behaviour for lists is to remove an element. The following auxiliary predicate builds a list of the possible shrunk lists.

```
genL(GenA, A, Size) :- listOf(GenA, A, Size).
genL(GenA, L, shrink, Shrs) :-  shrL(L, Shrs).
shrL([], []).
shrL([A], [[]]) :- !.
shrL([A|AS], [AS|Shrs]) :-
  shrL(AS, Shrs1), maplist(cons(A), Shrs1, Shrs).

cons(X, XS, [X|XS]).
```

Most combinators do not have a default shrinking procedure. Since it is hard to decide, for example, what is a proper shrink for values generated by a random choice between generators, we default the shrinking of many combinators to an empty list of shrunk values. Instead of directly using combinators in a property quantification the user can wrap them in a generator predicate with a meaningful name, implementing the shrink behaviour for this specific type. This is exemplified by the `genL` generator predicate, which is a redefinition of `listOf` and can therefore implement a different shrinking process.

## 6 Specification Language

In this section we describe our predicate specification language. Throughout, we follow some of the principles presented by Deville [9]. There are several ways to state a predicate's specification, we do not argue that our specification process is superior to other approaches. We do believe that this approach fits naturally our needs, namely as a form to express testable predicate features.

The general specification form of a predicate `p/n` consists, at its core, of a set of uniquely identified specification clauses about *input types* or the shape of the parameters when evoking the predicate. Various aspects of the predicate for the particular input type in question can be added to a specification clause. If there is a *parameter relation* or a relation that input parameters must fulfil one can implement it as a predicate which checks if such a relation is valid for the list of input parameters given. The *modes* of each parameter can be given for the input parameters and for output answers. The language also allows stating the number of answers of a predicate, or its *range*. Last, the user may state invariant properties that should hold both before and after the predicate is executed as *pre-* and *post-conditions*. Next, we discuss the main properties that we allow in our framework.

**Types.** Types are the mandatory part of the specification. They are required to guarantee that the specification may be automatically tested. We define a type as follows:

**Definition 1.** *A* type *is a non-empty set of terms.*
*A term $t$ belongs to a type $\tau$ ($t \in \tau$) if it belongs to the set of terms that define the type.*

Types are not defined as a set of *ground* terms but rather by a set of terms. Note that types defined in this manner depict perfectly possible forms of predicate input. This approach for types already encloses, by definition, the type precondition, where the input must be compatible with the specified types.

The types mentioned in a predicate specification clause correspond to PrologCheck generators used to automatically create individual test cases. This means that the type in a specification clause is partial in the sense that it only specifies that the predicate should succeed when given elements of such types as parameters. It states nothing about parameters of other types. Other input types can be covered by other specification clauses with different generators. The behaviour of a procedure for types not covered by any of the specification clauses is considered undefined/unspecified.

We can now easily specify input types for program predicates like `app/3`. We identify the specification clauses as `{app,K}`, specification clause `K` of predicate `app`, and declare the PrologCheck types. The specifications can be tested and the predicate checked to succeed for the corresponding input types.

```
{app,1} of_type (listOf(int), value(v), variable)
{app,2} of_type (listOf(int), variable, variable)
{app,3} of_type (listOf(int), listOf(int), variable)
```

**Domain.** Correct typing of parameters is crucial but may be insufficient to express the allowed input. Sometimes the input parameters must obey a relation extending type information, based on the actual values of the parameters. The domain of a predicate is the set of parameters accepted by a predicate [9]. The domain precondition is a restriction over the set of parameters of a predicate. Suppose that `minimum(A,B,C)` is a predicate that succeeds when `C` is the minimum of `A` and `B`. The predicate has the type `(int, int, int)` and the domain is the restriction `(C==A; C==B),(C<=A, C<=B)`.

**Definition 2.** *A* domain *of a procedure p/n is a set of term n-tuples such that* $\langle t_1, ..., t_n\rangle \in (\tau_1 \times ... \times \tau_n)$
$\langle t_1, ..., t_n\rangle$ *satisfies the input parameter relation*

This definition of a domain, similarly to what happens with types, is different from the usual notion of domain. It focus on the shape of the input to a predicate and not the accepted answer set. The PrologCheck domain of a predicate is then any set of terms produced by the generator that fulfils the domain precondition. In the absence of a domain precondition relating parameters the domain is the set of terms generated. A specification clause can thus be engineered to represent a subset of a more general type. An example could be that we want to test `app` with at least one non-empty list input. This can be used, for example, to guarantee that the variable given in the third input parameter will be instantiated with a non-empty list.

```
non_empty( [[_|_],_,_] ).
non_empty( [_,[_|_],_] ).
{app,3b} of_type (listOf(int), listOf(int), variable)
   such_that m:non_empty.
```

**Directionality.** The directionality of a predicate describes its possible uses by specifying the possible forms of the parameters before and after execution. We follow Deville [9] where the main modes for a parameter are ground, variable and neither ground nor variable. Conjunction of modes is possible and all combinations are achieved by the notation for ground (*g*) and variable (*v*) as well as the negation (*n?*). This results in the main modes and their pairwise combinations: *g*, *v*, *gv*, *ng*, *nv*, *ngv*. A parameter that can be used in any form is denoted by the mode identifier *any*.

**Definition 3.** *The* modes *or* forms *a term may present are denoted by* $Modes = g, v, gv, ng, nv, ngv, any$

There are two components to a directionality: input and output. They must hold for a predicate's parameters before and after execution, respectively. This means a full directionality denotes a pre- and a post-condition to the execution of the specified predicate. In PrologCheck these properties are checked for each test case when specified before and after calling the predicate.

Input directionality acts as a sanity check for the elements of the domain, meaning that the generators must be constructed to conform to the specified input modes. Each specification clause is allowed one input directionality. If the user wishes to specify more than one input form the clause should be divided into the number of input forms and its generators adapted accordingly. This results in bigger predicate specifications with possibly duplicated code, but is a very simple way to express what happens to the parameters in finer detail.

Each input may have more than one answer and therefore more than one output form. For this reason we adopted a schema where an input directionality is paired with a list of output forms.

**Definition 4.** *A* directionality *of a specification clause of a predicate p/n is a sequence of* predicate modes*, with one input mode followed by one or more output modes.*
*A predicate mode of p, or just mode of p, is denoted as*

– $i(m_1, ..., m_n)$

– $o(M_1, ..., M_n)$

*where* $m_i, M_i \in Modes$ *and* $i, o$ *respect to input and output modes respectively.*

The specification of input and output modes is important to state predicate behaviours that may be oblivious to a library user. From using the predicate `app/3` with a list and two variables, for example, two distinct directionalities may arise. This is due to the fact that an empty list in the first input parameter does not contribute to instantiate any part of the third parameter.

```
{app, 4} of_type (listOf(int), variable, variable)
   where (i(g, v, v), o(g, v, ngv), o(g, v, v)).
```

PrologCheck does not check the specification for consistency. A parameter with modes such that *in* is ground and *out* is variable is caught during testing. Output modes that are redundant or invalid will not be exposed when part of a set of output directionalities since they are interpreted as a disjunction. Directionalities should be constructively defined and not over-specified. They should be separated according to disjoint input types and incremented as needed.

**Multiplicity.** The number of answers a predicate call has can be valuable information. Knowing a predicate has a finite search space is a termination guarantee for predicates using it. Conventionally, multiplicity information, or *range*, is given for each directionality [19]. In PrologCheck we do not require that directionality is given, in which case no tests are performed regarding parameter form and the *any* mode is assumed for all parameters. The multiplicity is tied to the domain of each specification clause where defined.

The range of answers is given with two bounds: *Min* and *Max*. These values are the lower and upper bounds to the number of answers. The lower bound should not exceed the upper bound and they both take non-negative integer values up to infinity (denoted by the atom `inf`). When no explicit multiplicity is given the default we follow is $\langle 1, inf \rangle$. When testing a specification clause, the default minimal expected behaviour is that the domain is successfully accepted by the predicate. Therefore we try to mirror this when there are other features specified but no multiplicity, expecting at least one solution. It is necessary to impose a limit when the upper bound is infinity or an excessively large number. One can state the maximum number of answers necessary to assume that the answer range is sufficiently close to the upper bound with a positive integer. We can complement the previous specification clause with a statement about the predicate behaviour regarding the number of answers. In this case we have a total function behaviour, always yielding one and only one answer.

```
{app, 4b} of_type (listOf(int), variable, variable)
   where (i(g, v, v), o(g, v, ngv), o(g, v, v))
   has_range {1,1}.
{app, 4c} of_type (variable, listOf(int), variable)
   where (i(v, g, v), o(g, g, g), o(ngv, g, ngv))
   has_range {1,inf} limit 50.
```

**Pre and Post-conditions.** Along with all the other features of a predicate we can have a connection between the relations represented by the predicate being specified and other predicates. These relations can be valid prior to or after execution. In the predicate specification language they are pre- and post-conditions and are expressed as PrologCheck properties.

A pre-condition is a property that only inspects its input. It does not change the generated values to be applied to the specified predicate. Post-conditions can use any of the specified parameters. Since they are no longer used, it does not matter if they are changed by the answer substitution. Now we can describe the property relating the lengths of `app/3`'s parameters in a post-condition of a specification clause of the respective type. We identify the parameters of `app` so that we can use them in the post-condition as `A`, `B` and `C`.

```
{app, 5} of_type (A-(listOf(int)), B-(listOf(int)), C-(variable))
  post_cond (length(A,K1), length(B,K2), length(C,K), K is K1+K2).
```

## 7 AVL Trees Case Study

We have described AVL properties and performed black-box testing of an implementation of AVL trees in a Yap [8] module, `avl.yap`, with PrologCheck[2]. Due to space restrictions we present a general description of the process and its results.

The module interface is small, with predicates to create an empty tree, insert an element and look up an element, respectively `avl_new/1`, `avl_insert/4` and `avl_lookup/3`. When performing this kind of test one does not simply test individual predicates but rather usages of the module. To do this we must be able to create sequences of interface calls and inspect intermediate results for compliance with AVL invariants. Knowledge about the shape of input/output terms can be gathered manually if it is not previously known.

[2] All the details can be found in the tool's website.

**Generator.** Creating a valid sequence of interface calls is not difficult, but requires attention to detail. First, we only want to generate valid sequences to save effort of checking validity and not suffer from sparse valid values. Using the `avl` module implies the existence of two sets of important terms: key terms, and value terms, which we represent as generators. In order to test the correct failure of wrong look-ups, a set of values for failed look-ups disjoint from the regular values is implemented.

The generator starts by creating the tree, independently of the size parameter, using `avl_new/1`. This implies that when size is 0 an empty AVL-tree is still created. Thus, we always append the tree creation to a sequence of calls to insert and look-up values. Each element of the sequence is obtained by randomly choosing between insert and look-ups.

When an insert command is added to the sequence, the value to be inserted is kept so that it can be used in later look-ups. Look-ups are divided between valid look-up and invalid look-up. Valid look-ups are only generated after the corresponding insert and invalid look-ups are based in a set of values that is never inserted. Valid look-ups can be further distinguished between looking up a key-value pair and looking up a key and retrieving its value. These elements are branded by a command identifier to recognise their correct behaviour during testing. The relative probabilities are such that we get a big variety of commands within relatively small sequences.

**Property.** The definition of the AVL property depends on several factors. It is necessary to have operations to extract information from trees, such as current node's key, key comparison, left and right sub-trees and empty tree test.

A tree may be empty, in which case it is an AVL tree of height 0. In the case of a non-empty tree we retrieve its key and sub-trees. They are used in recursive checks of the property. The recursive calls accumulate lists of keys that should be greater and less then the keys in the sub-trees. If the sub-trees are individually compliant with the property, we proceed with the last check, comparing the returned heights for balance and computing the current tree height. This is how the property is outlined in PrologCheck:

```
prop({avl, T, Gs, Ls, H}) :- if (not isNil(T)) then
 ((getKey(T, K), left(T, L),  right(T, R),
     ((forall(member(X, Ls), cmpKeys(X, K, gt))) -> error1),
     ((forall(member(X, Gs), cmpKeys(X, K, lte))) -> error2))
   and prop({avl, L, [Key|Gt], Lt, Hl})
   and prop({avl, R, Gt, [Key|Lt], Hr})
   and (( abs(Hl-Hr)>1 -> error3), H is 1+max(Hl,Hr)))
 else (H = 0).
```

We complete the property by inserting it into a loop that consumes the operations in the quantified module uses.

Table 1 summarises some relevant results of our tool applied to the AVL library. Each line corresponds to a different module version: line 1 to the original version; line 2 to a bug in the re-balancing strategy inserted by the tester; line 3 to a different bug in the re-balancing strategy inserted by someone that was not involved with the tests. The column *Tests* is the number of tests needed to achieve a particular counter-example. For the purpose of readability we will represent only the key and value input parameters of the AVL operations. Thus consider `i(N,V)` as insert an element with key `N` and value `V`, and `l(N,V)` as look up the pair `(N,V)` in the tree.

Note that the counter-example found in the original version corresponds to an unspecified behaviour in the case of two insertions with the same key. After several runs of the tool (10 for the first bug and 20 for the second) we managed to find a pattern on the counter-examples which led to the identification of the pathological behaviour caused by the bugs.

**Table 1.** `AVL` testing summary

| Version | Tests | Counter-example |
|---|---|---|
| Original | 732 | `i(1,a), i(1,b), l(1,b)` |
| Error 1 | 51 | `i(3,a), i(1,b), i(2,c)` |
| Error 2 | 213 | `i(5,a), i(2,b), i(3,c), i(4,d), i(1,e)` |

## 8 Conclusion

We present PrologCheck, an automatic tool for specification based testing of Prolog programs.

Compared to similar tools for functional languages, we deal with testing of non-deterministic programs in a logic programming language. We provide a language to write properties with convenient features, such as quantifiers, conditionals, directionality and multiplicity. PrologCheck also includes the notion of random test-data generation.

We show that specification based testing works extremely well for Prolog. The relational nature of the language allows to specify local properties quite well since all the dependencies between input parameters are explicit in predicate definitions.

Finally note that our tool uses Prolog to write properties, which, besides its use in the tool for test specification, increases the understanding of the program itself, without requiring extra learning for Prolog programmers.

**Acknowledgements.** This work is partially financed by the ERDF - European Regional Development Fund through the COMPETE Program and by National Funds through the FCT - Fundação para a Ciência e a Tecnologia (Portuguese Foundation for Science and Technology) within project ADE/PTDC/EIA-EIA/121686/2010 and by LIACC through Programa de Financiamento Pluri-anual, FCT. Cláudio Amaral is funded by FCT grant SFRH/BD/65371/2009.

## References

1. Antoy, S., Hanus, M.: Overlapping rules and logic variables in functional logic programs. In: Etalle, S., Truszczyński, M. (eds.) ICLP 2006. LNCS, vol. 4079, pp. 87–101. Springer, Heidelberg (2006)

2. Bernardy, J.-P., Jansson, P., Claessen, K.: Testing polymorphic properties. In: Gordon, A.D. (ed.) ESOP 2010. LNCS, vol. 6012, pp. 125–144. Springer, Heidelberg (2010)
3. Boberg, J.: Early fault detection with model-based testing. In: Proc. of Workshop on Erlang, pp. 9–20. ACM (2008)
4. Christiansen, J., Fischer, S.: EasyCheck — test data for free. In: Garrigue, J., Hermenegildo, M.V. (eds.) FLOPS 2008. LNCS, vol. 4989, pp. 322–336. Springer, Heidelberg (2008)
5. Claessen, K., Hughes, J.: Quickcheck: a lightweight tool for random testing of haskell programs. In: Proc. of ICFP, pp. 268–279. ACM (2000)
6. Claessen, K., Hughes, J., Pałka, M., Smallbone, N., Svensson, H.: Ranking programs using black box testing. In: Proc. of AST, pp. 103–110. ACM (2010)
7. Claessen, K., Pałka, M., Smallbone, N., Hughes, J., Svensson, H., Arts, T., Wiger, U.: Finding race conditions in erlang with quickcheck and pulse. In: Proc. of ICFP, pp. 149–160. ACM (2009)
8. Costa, V.S., Rocha, R., Damas, L.: The yap prolog system. TPLP 12(1-2), 5–34 (2012)
9. Deville, Y.: Logic programming: systematic program development. Addison-Wesley Longman Publishing Co. Inc., Boston (1990)
10. Duregård, J., Jansson, P., Wang, M.: Feat: functional enumeration of algebraic types. In: Proc. of Haskell Symposium, pp. 61–72. ACM (2012)
11. Florido, M., Damas, L.: Types as theories. In: Proc. of post-conference workshop on Proofs and Types, JICSLP (1992)
12. Frühwirth, T.W., Shapiro, E.Y., Vardi, M.Y., Yardeni, E.: Logic programs as types for logic programs. In: Proc. of LICS, pp. 300–309 (1991)
13. Hermenegildo, M.V., Bueno, F., Carro, M., López-García, P., Mera, E., Morales, J.F., Puebla, G.: An overview of ciao and its design philosophy. In: TPLP, pp. 219–252 (2012)
14. Koopman, P., Alimarine, A., Tretmans, J., Plasmeijer, R.: Gast: Generic automated software testing. In: Peña, R., Arts, T. (eds.) IFL 2002. LNCS, vol. 2670, pp. 84–100. Springer, Heidelberg (2003)
15. Mera, E., Lopez-García, P., Hermenegildo, M.: Integrating software testing and run-time checking in an assertion verification framework. In: Hill, P.M., Warren, D.S. (eds.) ICLP 2009. LNCS, vol. 5649, pp. 281–295. Springer, Heidelberg (2009)
16. Naylor, M.: A logic programming library for test-data generation (2007)
17. Papadakis, M., Sagonas, K.: A proper integration of types and function specifications with property-based testing. In: Proc. of Workshop on Erlang, pp. 39–50. ACM (2011)
18. Runciman, C., Naylor, M., Lindblad, F.: Smallcheck and lazy smallcheck: automatic exhaustive testing for small values. In: Proc. of Haskell Symposium, pp. 37–48. ACM (2008)
19. Somogyi, Z., Henderson, F.J., Conway, T.C.: Mercury, an efficient purely declarative logic programming language. Australian Computer Science Communications 17, 499–512 (1995)
20. Wielemaker, J., Schrijvers, T., Triska, M., Lager, T.: Swi-prolog. TPLP 12(1-2), 67–96 (2012)
21. Yardeni, E., Shapiro, E.: A type system for logic program. J. Log. Program. 10(2), 125–153 (1991)
22. Zobel, J.: Derivation of polymorphic types for prolog programs. In: Proc. of ICLP, pp. 817–838 (1987)