A PropEr Integration of Types and Function Specifications with Property-Based Testing

Manolis Papadakis [1] Konstantinos Sagonas [1,2]

[1] School of Electrical and Computer Engineering, National Technical University of Athens, Greece
[2] Department of Information Technology, Uppsala University, Sweden
manopapad@softlab.ntua.gr kostis@cs.ntua.gr

Abstract

We present a tight integration of the language of types and function specifications of Erlang with property-based testing. To achieve this integration we have developed from scratch PropEr, an open-source QuickCheck-inspired property-based testing tool. We present technical details of this integration, most notably how the conversion of recursive types into appropriate generators takes place and how function specifications can be turned automatically into simple properties in order to exercise the code of these functions. Finally, we present experiences and advice for the proper use of PropEr.

Categories and Subject Descriptors D.2.5 [*Software Engineering*]: Testing and Debugging—Testing tools (e.g., data generators, coverage testing); D.3.3 [*Programming Languages*]: Language Constructs and Features—Data types and structures

General Terms Algorithms, Languages

Keywords property-based testing, type declarations, function signatures, test generators, Erlang

1. Introduction

For some years now, Erlang comes with a language of type declarations and function specifications, so called specs. This language has so far been used for expressing users' intentions, detecting discrepancies between these intentions and the actual implementation of functions, and for generating documentation for Erlang programs. By now it is quite common to see Erlang modules or even complete applications that are full of specs. Concurrently with this activity, property-based testing has become quite popular among Erlang developers. Sadly, however, the two technologies have been disconnected till now. Type-aware Erlang programmers can not use type information to get some simple tests "for free" and QuickCheck-conscious users have to manually write (often complicated) generators for data structures manipulated by their programs.

Considering this situation, we decided to do something about it: we embarked on a project that would integrate these technologies. Since the main tool for property-based testing in Erlang is proprietary and closed-source, we had to create our own tool. It is called PropEr and is freely available as open-source [7].

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

Erlang'11, September 23, 2011, Tokyo, Japan.
Copyright © 2011 ACM 978-1-4503-0859-5/11/09…$10.00

This paper presents issues that have to be addressed in order to achieve a tight and proper integration of types and specs with property-based testing in Erlang. As we will see, PropEr is capable of automatically using types, both module-local and remote, as generators and turning function specifications into simple properties that can test the agreement between the programmers' intentions and the implementation of these functions. Achieving all this is not trivial and the paper describes in detail the techniques that make this integration possible.

The next section describes the Erlang type language in detail. It is followed by Section 3 that reviews property-based testing and a section that describes the PropEr tool and its built-in language of generators (Section 4). Section 5, which is the main section of this paper, describes the integration of Erlang's type language with the language of PropEr generators, focusing on the translation scheme used to automatically create generators from recursive types. Section 6 presents some experiences from using PropEr in Erlang code bases and the paper ends with some concluding remarks.

2. The Erlang Type Language

The type language used in Erlang [5] is quite similar to the type notation used by the EDoc documentation tool [4] and has been designed with expressive power in mind and in order to best capture the style of programming that programmers of dynamic languages often follow. For example, the language allows for singleton types, for taking arbitrary unions of existing types and for constructing a new type without having to wrap terms in tagged tuples (i.e. in constructors). Type declarations and function specifications are not used to guarantee type safety, because the Erlang language already provides this guarantee through runtime checks. Instead, they have been primarily designed to allow detection of definite type errors by the Dialyzer static analysis tool [6]. Nowadays, types and function specifications are quite widespread and it is quite common to find code bases where most exported functions have type signatures.

2.1 Built-in Types

Types are used to describe sets of Erlang terms, both finite and infinite. The Erlang type language provides a few predefined types, most of which are refinements over the runtime representation-based classification of terms. These types are shown in Table 1 grouped by term category.

The type language also defines a number of other built-in types, such as `boolean()` for `'true' | 'false'`, `binary()` for `<<_:_*8>>`, and `string()` for `[0..16#10FFFF]`, i.e. a list of Unicode characters. Improper lists, i.e. lists where the tail of a cons cell is a non-list value, are supported by the type language, but are of little practical use, therefore we will not consider them for the rest of this paper (although PropEr handles them properly).

Integers mathematical integers
  I only a specific integer I, e.g. 42 (singleton type)
  L..H integers between L and H
  integer() all integers
Floats floating point numbers
  float() all floats
Atoms named constants, e.g. 'hello', 'World'
  A only a specific atom (singleton type)
  atom() the set of all atoms
Bitstrings untyped series of bits, e.g. <<8:4,1:4>> (<<129:8>>)
  <<>> only the empty bitstring (singleton type)
  <<_:_*U>> bitstrings whose size is a multiple of U, $U \in \mathbb{N}$
  <<_:B*U>> bitstrings whose size is $B \times U$
Pids handles for communicating with other Erlang processes
  pid() all process identifiers
Ports handles for communicating with external programs
  port() all ports
References identifiers that are unique within a runtime environment
  reference() all references
Funs callable function objects, e.g. fun(X) -> X + 42 end
  fun((...) -> R) functions that return values of type R
  fun(($T_1, \ldots, T_N$) -> R) functions that accept N arguments of types $T_1, \ldots, T_N$ and return values of type R
Tuples compound terms with a fixed number of elements, e.g. {0,'foo',3.14,'bar',<<42>>}, {'answer',42}
  tuple() all tuples
  $\{T_1, \ldots, T_N\}$ tuples of N elements, of types $T_1, \ldots, T_N$
Lists compound terms with a variable number of elements, not necessarily of the same type, e.g. [42,'answer']
  [] only the empty list (singleton type)
  [T] lists with elements of type T
  [T,...] non-empty lists with elements of type T
Special types that do not correspond to a specific term category
  any() all Erlang terms (top type)
  none() no terms (bottom type)
  T_1 | T_2 | ... | T_N all the terms represented by at least one of $T_1, T_2, \ldots, T_N$ (union type)

Table 1. Built-in types of Erlang

2.2 User-defined Types

Users can define their own, custom types using type declarations. Listing 1 shows some examples. The basic syntax of a custom type is similar to that of a function call: tname($TV_1, \ldots, TV_N$) where the TV_i's are types or type variables. The definition of such a type must be provided in the right hand side (RHS) of its type declaration and can be any valid type expression.

Inside their defining module, user-defined types may appear in any type expression, even in the RHS of their own declaration. Types defined in other modules, also known as *remote types*, can only be used if they have been exported from their defining module using an export_type declaration. Such types can then be referenced by using their module qualified type name (e.g., the type b:bar() in Listing 1).

Custom types can be used as aliases for other type expressions. Additionally, custom types can be parametric, allowing for the reuse of a structure specification with various different element types. Finally, recursive types can be expressed using type declarations where the name of the type appears in the RHS of the declaration, or even through mutually recursive declarations.

Erlang also allows users to define *record* data types, which behave like tuples with named fields. Records may optionally be typed, by providing a type declaration for one or more of their fields. In Erlang, however, records are not really a separate data type, since they are converted to tuples at compile time. Records

```
-module(a).

%% time() is local: we can refer to it using just its name
%% bar() is remote: it must be exported from its defining
%% module and the module name must be used when referring to it
-type foo() :: {time(),b:bar()}.

%% simple type:
-type time() :: {0..23,0..59,0..59}.

%% recursive type:
-type tree() :: 'leaf' | {'node',integer(),tree(),tree()}.

%% mutually recursive types:
-type do() :: 'do' | {'do',re()}.
-type re() :: 're' | {'re',mi()}.
-type mi() :: 'mi' | {'mi',do()}.

%%===========================================================

-module(b).
-export_type([bar/0]).

-type bar() :: integer() | #rec{}.

%% record declaration:
-record(rec, {a, b = 12, c :: atom(), d = 3.1 :: float()}).

%% parametric type:
-type kvlist(K,V) :: [{K,V}].

%% parametric, recursive type:
-type tree(T) :: 'leaf' | {'node',T,tree(T),tree(T)}.
```

Listing 1. Examples of type declarations

may be referenced inside any type expression in their defining module using the notation: #rec{}. Such references may optionally refine the types of some fields. For example, #rec{a :: atom()} further refines the types of the record declaration in Listing 1.

2.3 Function Specifications

A function specification (or contract) is a way for the programmer to explicitly state the intended uses of a function. Contracts are declared using spec attributes and follow the syntax:

-spec fname($AT_1, \ldots, AT_N$) -> RT.

where $AT_1, \ldots, AT_N$ and RT can be any type expression allowed in the context of the current module.

Erlang functions are often designed to operate on different data types in an overloaded fashion. In order to accurately capture this behaviour, the type language allows for *overloaded contracts*. These are specified as a sequence of simple contracts (referred to as *contract clauses*) separated by semicolons, as below:

```
-spec pnt(pos_integer(), pos_integer()) -> pos_integer();
         (pos_integer(), neg_integer()) -> neg_integer();
         (neg_integer(), pos_integer()) -> neg_integer();
         (neg_integer(), neg_integer()) -> pos_integer().
```

Another feature of the contract language is support for *parametric polymorphism*. Type variables can be used in contracts to specify relations among the input arguments and with the return value, as in this specification of the lists:map/2 function:

```
-spec map(fun((T) -> S), [T]) -> [S].
```

By default, type variables are universally quantified, i.e. they can be instantiated to any Erlang type (even singleton types). Users can constrain the types that a variable is allowed to represent by adding

a guard-like subtype constraints to the contract, thus achieving *bounded quantification* as in the following example:

```
-spec af(X,Y) -> {X,Y} when X::atom(), Y::float().
```

Some functions in Erlang are not meant to return. For example, they may define servers which are supposed to run infinitely or may simply be functions that only throw exceptions. Such functions should be specified as returning the special `none()` type or its alias `no_return()`.

3. Property-Based Testing

The process of testing a piece of code can be conceptually divided into three steps:

1. Acquire a valid input.
2. Run the code on that input and capture any output.
3. Decide if the input-output pair conforms to the code's intended behaviour.

The second step of this process can nowadays be fully automated (e.g., by using a unit testing framework like EUnit [2]). Various approaches have been suggested for solving the third step, most of which involve defining some sort of specification for the code under test. In many cases, programmers are required to provide a full, formal specification of their code, whose size is often comparable to that of the original program. The first step, how to efficiently generate valid inputs representative of a program's domain, has also been studied [8]. However, most conventional testing tools leave it upon programmers to come up with test inputs for their code.

Property-based testing (PBT) is a novel approach to software testing, where the tester only needs to specify the generic structure of valid inputs to the program under test, along with a number of properties (regarding the program's behaviour and the input-output relation) which are expected to hold for every valid input. A PBT tool, when supplied with this information, will automatically produce progressively more complex random valid inputs, then apply those inputs to the program while monitoring its execution, to test that it behaves as expected. By following this methodology, a tester's manual tasks are reduced to correctly specifying the program's input format and formulating a set of properties that accurately describe its intended behaviour.

PBT tools operate on *properties*, which are essentially partial specifications of the program under test, meaning that they are more compact and easy to write and understand than full specifications. Users can make full use of the host language when writing properties, and thus can accurately describe a wide variety of input-output relations. They may also write their own test data generators, should they require greater control over the input generation process. Compared to testing code with manually-written test cases, testing with properties is a faster and less mundane process. The resulting properties are also much more concise than a long series of test cases, but, if used properly, can accomplish a more thorough testing of the program, by subjecting it to a much greater variety of inputs than any human tester would be willing to write. Moreover, properties can serve as a checkable partial specification of a program, one that is considerably more general than any set of instantiated test cases, and thus one that is much better at documenting the behaviour of the program.

Because test inputs are generated randomly, the part of a failing test case that is actually responsible for the failure can easily become lost inside a lot of irrelevant data. A PBT tool often aids the programmer in extracting that part by simplifying the counterexample, through an automated process called *shrinking*. In most cases, shrinking works in the same way as a human tester, by consecutively removing parts of the failing input until no more can be removed without making the test pass. This "minimal" test case will serve as a good starting point for the debugging process. The shrinking process can often be fine-tuned through user-defined shrinking strategies.

The first implementation of a PBT tool was QuickCheck for Haskell [3], a combinator library designed for testing pure functions. Since this pioneering work, QuickCheck-like tools have been developed for a variety of other languages. Erlang versions include Quviq QuickCheck [1], a proprietary closed-source tool marketed as a commercial product, Triq [9], which was written by Kresten Krab Thorup and released as open-source software, and our own implementation, PropEr.

3.1 Testing Process

A test run of a single property typically follows this workflow:

1. Randomly generate valid instances for each universally quantified variable in the property, using the generator that the user has provided for each such variable.
2. Call the property code with this input.
3. If the property evaluated to `'true'` without throwing an exception, repeat from 1. Else:
4. While it is possible to shrink the test case to one that is simpler and fails the property in the same way, shrink it.
5. Report the failing test case and the shrinked input to the user.

In PBT tools, this process can typically be configured in various ways through options. For example, users can control the maximum number of tests to run, the size of produced inputs, the maximum number of shrinking attempts, etc. Failing test cases, i.e. the values of the universally quantified variables that caused the property to fail, can be saved and re-applied to the same property, allowing users to easily check if they have successfully fixed the problem.

To better illustrate this process, we provide an example: Suppose that we want to test our implementation of a `delete` function for lists (Listing 2), which should take a term `X` and a list `L` and return a copy of `L` with all occurrences of `X` removed. We translate this specification into the following property: "For any pair of values `X` and `L`, where `X` is an integer and `L` a list of integers, `delete(X,L)` must not contain any element matching `X`".

```
-module(mylists).
-export([delete/2]).
-include_lib("proper/include/proper.hrl").

delete(X, L) ->
    delete(X, L, []).

delete(_, [], Acc) ->
    lists:reverse(Acc);
delete(X, [X|Rest], Acc) ->
    lists:reverse(Acc) ++ Rest;
delete(X, [Y|Rest], Acc) ->
    delete(X, Rest, [Y|Acc]).

prop_delete_removes_every_x() ->
    ?FORALL({X,L}, {integer(),list(integer())},
            not lists:member(X, delete(X,L))).
```

Listing 2. Example of source code with PropEr tests

Listing 3 shows how this property is checked using PropEr. We can see that PropEr managed to find an error in our implementation. Judging from the format of the result, it seems that lists containing two (or more) instances of the element which is to be deleted are not handled correctly. Indeed, upon careful inspection it becomes apparent that our code only deletes the first occurrence of this element from the input list. The code and the property are not in agreement and one of them should better be changed.

```
2> proper:quickcheck(mylists:prop_delete_removes_every_x()).
.........................................!
Failed: After 42 test(s).
{-4,[6,-22,8,-4,46,1,-13,-4,56,-22,-10,50,
     -149,6,-7,-3,-20,17,22,1,-32,28,-37,0]}
Shrinking ........(8 time(s))
{-4,[-4,-4]}
false
```

Listing 3. Testing a property on the Erlang shell using PropEr

3.2 Property Declaration

In PropEr, properties are written using Erlang expressions, with the help of a few predefined macros. Expressions used as properties are expected to evaluate to `'true'` on success, and either evaluate to `'false'` or throw an exception on failure. More complex properties can be composed by wrapping such an expression with a property wrapper. A commonly used property wrapper is `?FORALL`:

`?FORALL(Xs, XsGen, Prop)`: The `Xs` argument should only contain variables, optionally contained in arbitrarily nested lists and/or tuples. The `XsGen` argument must then contain a PropEr generator (Section 4) that will produce a value of the same structure as that of `Xs`. All the variables inside `Xs` can (and should) be present as free variables inside the wrapped property `Prop`. When a `?FORALL` wrapper is encountered, a random instance of `XsGen` is produced and each variable in `Xs` is replaced inside `Prop` by its corresponding value.

PropEr comes with a variety of helper wrappers, which can be used for such purposes as declaring preconditions for the produced instances, setting a timeout for the code under test, combining many properties into one, collecting statistics on the randomly produced instances, attaching debug output to property code, working with code that spawns processes and configuring the testing process [7].

4. PropEr and its Internals

In this section, we present some internals of PropEr in detail. PropEr's interface is mostly compatible with other PBT tools for Erlang, but also closer to Erlang's type language.

The input domain of functions is specified through the use of a set of custom generators. Each PropEr generator specifies how the following three operations are performed:

Instance generation Given a generator, run it to randomly produce a valid instance. At the start of every testing run, PropEr produces only small instances of generators in `?FORALL`s, but gradually increases their complexity as tests pass. To accomplish this, PropEr uses a global `size` parameter, which starts at one and increases by one with each successful test. Its purpose is to control the maximum size of produced instances: the actual size of a produced instance is chosen randomly, but can never exceed the value of the `size` parameter at the moment of generation. The actual size of a term is measured differently for each data type, e.g. the actual size of a list could be defined as the sum of sizes of all its elements + 1, while the actual size of a tree could be defined as the number of its internal nodes.

Instance checking Given a term and a generator, decide if the term is a valid instance of the generator. This operation is used when shrinking instances of generators in a `?FORALL` that contains nested `?FORALL`s. In this case, the generator declarations in the nested `?FORALL`s may depend on the instances generated for the containing `?FORALL`, therefore we need to ensure that the old instances for the nested generators are still valid.

Shrinking Given a term and its generator, produce (perhaps lazily) all[1] the simpler instances of the generator which can be derived from this particular term.

4.1 Built-in Generators

Let's see a brief overview of the basic generators of PropEr:

integer(L,H) generates integers between L and H; either bound can be set to `'inf'`, in which case it represents $-\infty$ or $+\infty$ respectively.

float(L,H) generates floats between L and H; either bound can be set to `'inf'`, in which case it represents $-\infty$ or $+\infty$ respectively.

atom() generates all atoms.

bitstring(B,U) generates bitstrings of size $B \times U$; B can be set to `'any'` to indicate a bistring of arbitrary size.

function(N,G) generates pure functions of arity N that return values which are instances of G.

$\{G_1, \ldots, G_N\}$ generates tuples of N elements which are instances of $G_1, \ldots, G_N$.

loose_tuple(G) generates tuples of any size, whose elements are all instances of G.

list(G) generates lists with elements that are instances of G.

vector(Len,G) generates lists of length Len, whose elements are all instances of G.

$[G_1, \ldots, G_N]$ generates lists of exactly N elements, instances of $G_1, \ldots, G_N$.

any() generates all Erlang terms.

weighted_union($[\{w_1,G_1\},\ldots,\{w_N,G_N\}]$) generates the terms which are instances of at least one of $G_1, \ldots, G_N$; the weights $w_1, \ldots, w_N$ are used when deciding which one of $G_1, \ldots, G_N$ to use: choices with greater weights are more likely to be chosen.

(literal term) always generates that specific integer, float, atom or bitstring literal; counts as a singleton type that represents only that particular term.

The instance checking algorithm for built-in generators follows directly from these definitions. One exception is function(N,G): it is normally impossible to verify that every value returned by a function is an instance of a certain generator, simply because the domain of most non-trivial functions is infinite. For this reason, PropEr tags every function it produces with its range generator, and uses this information when instance checking.

Table 2 demonstrates how PropEr will attempt to shrink an instance of each built-in generator.

In addition to the information presented above, we note the following:

- We have omitted pids, ports and references from the PropEr generators, because these data types normally serve as identifiers to application-specific resources, therefore there is no point in producing random ones for testing.
- The language of PropEr's generators is compatible with Erlang's type language, with the exception of shorthand `[`T`]` for `list(`T`)`, which has a different meaning in PropEr. This is because we wanted to make the generator language more

[1] For efficiency, we relax this requirement a little when shrinking instances of variable-length terms, such as lists. In particular, we never try to shorten such an instance by removing two or more non-contiguous parts at once.

Term generated from	To get a simpler instance...
integer(L,H)	take an integer closer to 0
float(L,H)	take a float closer to 0.0
atom()	drop some characters
bitstring(B,U)	drop some bits, or convert some 1's to 0's
function(N,G)	take a function that returns simpler values
$\{G_1,\ldots,G_N\}$	simplify some elements
loose_tuple(G)	drop or simplify some elements
list(G)	drop or simplify some elements
vector(Len,G)	simplify some elements
$[G_1,\ldots,G_N]$	simplify some elements
any()	(depends on the instance)
weighted_union(...)	take an instance of some generator closer to the head of the list, or simplify the selected instance according to its generator
(literal term)	(doesn't shrink)

Table 2. Built-in PropEr generators' shrinking strategies

expressive than the type language in this case. Specifically, we allow users to specify lists of length N, using the syntax $[G_1, \ldots, G_N]$. Therefore, generator $[G]$ will always produce lists with exactly one element (an instance of G), while the type `[`T`]` represents lists of arbitrary length. Specifying lists of a given length is not possible in the type language.

- PropEr also defines a few generator aliases like integer() or boolean(), that correspond to standard Erlang types, and a few compatibility generators for interoperability with other PBT tools.
- Because instances of a weighted_union shrink towards the first choice, users should write the simplest case first in weighted unions. A weighted_union($[\{k,G_1\},\ldots,\{k,G_N\}]$) generator whose every choice has the same weight k will be denoted simply as union($[G_1, \ldots, G_N]$).

4.2 User-defined Generators

Users can define their own, custom generators as functions. Like the Erlang type language, the PropEr generator language allows users to define aliases for commonly used expressions, e.g.

```
day() -> union(['mo','tu','we','th','fr','sa','su']).
```

parametric generators, e.g.

```
kvlist(K,V) -> list({K,V}).
```

and, as we will soon see, recursive generators. Programmers can also define customized generators, using macros like the following:

`?LET(Parts, PartsGen, In)` This macro allows users to define dependent generators. The `Parts` argument should only contain variables, optionally contained in arbitrarily nested lists and/or tuples. The `PartsGen` argument must then contain a PropEr generator that will produce a value of the same structure as that of `Parts`. All the variables inside `Parts` can (and should) be present as free variables inside the wrapped expression `In`.

In order to produce an instance, PropEr generates a random instance of `PartsGen`, matches it with `Parts` and replaces all variables inside `In` with their corresponding values. `In` is allowed to evaluate to a generator, in which case an instance of the inner generator is generated recursively; this allows for nested `?LET`s.

To be able to instance check and shrink instances of `?LET`s, we need to save the generated instance of `PartsGen` along with the final value, since that information may be partially lost when applied to `In`. Consider, for example, instances produced by `?LET({X, Y}, {integer(17,21),integer(23,54)}, X+Y)`. If all we knew about an instance of this was its final value, e.g. `42`, we would have no way of knowing what values of `X` and `Y` were used to produce it. In fact, in the general case we would be unable to determine if that value could have ever been produced by some instance of `PartsGen`. For this reason, instances of `?LET`s are saved in an intermediate form: `{'$used',PartsValue,Value}`.

To shrink an instance of a `?LET`, we first shrink each of the `Parts` and re-apply them to `In`. When this is finished, we move on to shrinking the `Value`. If `In` evaluates to a generator, we need to check that the old `Value` is a valid instance of the new generator of `In`. This is done, as with `?FORALL`s, through instance checking.

`?SUCHTHAT(X, Gen, Condition)` This macro produces a specialization of `Gen`, which includes only those terms that satisfy the constraint `Condition`, i.e. those terms for which the function `fun(X) -> Condition end` returns `'true'`. Currently, PropEr performs no constraint satisfaction analysis; instead, it produces instances of `Gen` randomly until it finds a valid one. Any shrunk instance of a specialized generator must also satisfy the constraint, else it is rejected. A typical example of the use of `?SUCHTHAT` is the `non_empty` wrapper:

```
non_empty(T) -> ?SUCHTHAT(L, T, L =/= []).
```

By wrapping a list generator with this wrapper, we ensure that the empty list will never be produced.

4.3 Recursive Generators

Without the integration of Erlang type declarations and PropEr generators that we will present in Section 5, if users need to work with recursive data types, they have to manually guide their generation process. It is the user's responsibility to:

- provide a base case;
- ensure that generation always terminates;
- place all recursive cases in a weighted_union;
- add a fallback to the base case as the first choice;
- calculate the weights for the recursive choices, to ensure a satisfactory average size of produced instances;
- add laziness to ensure efficient generation;
- define a shrinking strategy.

In addition, to ensure that the generation of a recursive data type always terminates, users must handle the value of `size` manually: they have to write a recursive generator function that accepts (at least) a `Size` parameter, which should be distributed among all recursive calls of each recursion path. The base case should be provided in a separate clause for `Size = 0`. PropEr needs to be told how to apply the value of `size` to this generator, through the use of a `?SIZED` macro. To accomplish all the above, testers need to use the following macros and operators:

`?SIZED(Size, Gen)` Constructs a new generator from a *sized* generator `Gen` (i.e., a generator that handles `size` directly). To produce instances of this generator, PropEr will apply the current value of `size` to the function `fun(Size) -> Gen end`.

`resize(NewSize, Gen)` This declaration instructs PropEr to use `NewSize` instead of the `size` value to produce instances of `Gen`.

`?LETSHRINK(Parts, PartsGen, In)` This construct is equivalent to a simple `?LET`, but `Parts` can only be a list of variables. When shrinking an instance of this generator, the parts that were combined to produce it are first tried in place of the failing instance, before proceeding with normal `?LET` shrinking strategies. This can result in much more effective shrinking, therefore users are advised to always use `?LETSHRINK` when combining recursively generated values.

```
tree() ->
  ?SIZED(Size, tree(Size)).

tree(0) ->
  'leaf';
tree(Size) ->
  weighted_union([
    {1, ?LAZY(tree(0))},
    {5, ?LAZY(?LETSHRINK([L,R],
                         [tree(Size div 2),tree(Size div 2)],
                         {node,integer(),L,R}))}]).
```

Listing 4. Example of a PropEr recursive generator

`?LAZY(Gen)` This construct delays the execution of `Gen` until the generated value is actually needed. Users should wrap each recursive choice with a `?LAZY` macro, or else PropEr will prematurely generate an instance for each alternative, when only one will eventually be used. Because this process will be repeated recursively for each choice, the total generation time can be exponential on the size of the produced instance. Conversely, by making proper use of `?LAZY` we can achieve linear generation time.

For an example of what the user would need to write to obtain a recursive generator see Listing 4, where we provide a generator for the `tree()` type declaration of Listing 1.

5. Integrating PropEr with the Type Language

PropEr was designed from the ground up to be properly integrated with Erlang's type language, a feature currently unique among PBT tools for Erlang. Before delving into the technical aspects of this integration, we briefly examine the motivation behind the creation of a PBT tool that can work with type information directly:

- While the language of generators is richer that the language of types, it is often the case that a user-written generator is almost completely equivalent with the type declaration for the same data structure. In these cases, a type-aware PBT tool can save considerable programmer effort.
- Using types as generators helps to reduce code redundancy: the type declaration becomes the single point of specification for datatypes, so programmers do not have an extra set of places in their code (the generators) to update every time a type changes.
- The burden of writing generators for recursive data structures is lifted, because the tool can extract all the required information from the corresponding type declaration.
- Adding type declarations to Erlang programs becomes more valuable and users do not need to learn a new type specification language to start testing with properties[2].
- Function signatures, being essentially a form of lightweight specification, can be leveraged to provide simple properties for free. Programmers now have one more reason to make their `specs` more descriptive.

As hinted above, there are two kinds of type information that we can utilize for PropEr testing:

- type declarations, which can be used to derive generators for the data structures they represent, and
- function specifications, which can be converted into simple properties

[2] In pursuit of the same goal, we have also strived to keep PropEr's API as similar as possible to the Erlang type language.

Erlang Type	PropEr Generator
$L..H$	integer(L,H)
`integer()`	integer('inf','inf')
`non_neg_integer()`	integer(0,'inf')
`pos_integer()`	integer(1,'inf')
`neg_integer()`	integer('inf',-1)
`float()`	float('inf','inf')
`<<_:_*`U`>>`	bitstring('any',U)
`<<_:`B`*`U`>>`	bitstring(B,U)
`pid()`	—
`port()`	—
`reference()`	—
`fun((...) -> `R`)`	?LET(N, integer(0,255), function(N, R'))
`fun((`$T_1,\ldots,T_N$`) -> `R`)`	function(N,R')
`tuple()`	loose_tuple(any())
$\{T_1,\ldots,T_N\}$	$\{G'_1,\ldots,G'_N\}$
`[`T`]`	list(G')
`[`T`,...]`	non_empty(list(G'))
`none()`	—
T_1 | T_2 | ... | T_N	union([$G'_1, G'_2, \ldots, G'_N$])

Table 3. Conversion table from built-in Erlang types to PropEr generators. Types that have the same representation (e.g. `atom()`) in both languages have been omitted. G' stands for the PropEr equivalent of the Erlang type T; similarly for R' and R.

Let us see the first component: a type-to-generator converter.

5.1 Converting Non-recursive Types

Converting built-in Erlang types to PropEr generators is a straightforward process: we just follow the conversion formula outlined in Table 3. Note that the types `pid()`, `port()` and `reference()` do not have any equivalent generators in PropEr, for reasons explained in Section 4.1.

Non-recursive user-defined types are also relatively easy to handle: we simply recurse into their type structure. If the type is parametric, we also need to apply the actual parameters to it. For module-local types, we can simply replace every instance of a variable in the definition with its corresponding value, on a syntactic level. In the case of remote types, we also need to mark the value of each parameter with its originating module, because parameters are considered to be in the scope of the referring module, and may therefore contain references to types which are local to that module.

5.2 Converting Recursive Types

PropEr can also handle self-recursive and mutually recursive types. However, the method used to convert such types is much more complicated than the one for non-recursive types.

To make the presentation easier to follow, we are going to ignore some constructs of the Erlang type language that do not add any significant complexity to the procedure. Specifically, we will be ignoring funs (which can be treated similarly to lists), records (which can be treated like user-defined types) and parametric user-defined types (which can be treated exactly like non-parametric types, after instantiating them with their actual parameters). We will only use types that are all declared in the same module, but in the actual implementation we also allow recursion paths to span multiple modules. Among the remaining built-in Erlang types, only $\{T_1,\ldots,T_N\}$, `[`T`]`, `[`T`,...]` and T_1 | ... | T_N require special handling; the rest can be converted using the simple procedure of Section 5.1. We will be referring to the latter collectively as *simple types*.

```
-type nat()  :: non_neg_integer()
              | {'+', nat(), nat()}
              | {'if', cond(), nat(), nat()}
              | {'from_bits', bits()}.
-type cond() :: {'=', nat(), nat()}
              | {'=', real(), real()}.
-type real() :: {'from_nat', nat()}
              | {'+', real(), real()}.
-type bits() :: {'from_nat', nat()}
              | {'concat', [bits() | nat()]}.
```

Listing 5. Type declarations as given by the user

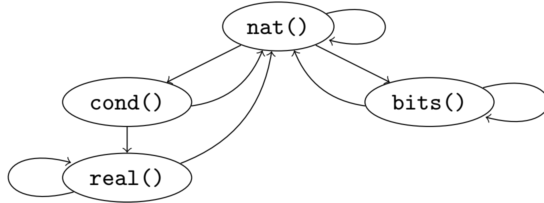

Figure 1. Initial dependency graph

Given a set of mutually recursive type declarations, we can sketch the dependencies among the types using a directed graph. In such a graph, nodes represent types and edges represent dependencies between types: there exists an edge from node `a()` to node `b()` if and only if the definition of type `a()` contains a reference to type `b()`. This graph may contain self-loops, which represent direct self-recursion. We can elect a particular type (normally, the one we wish to translate) as the root, and view the graph as a tree. Back-edges on that tree will then represent indirect recursion paths.

To help illustrate each step of the translation process, we will be using the type `nat()`, declared in Listing 5, as a running example (this type was adapted from a specification for valid syntax expressions in some programming language). Figure 1 depicts the dependency graph for the types declared in the listing.

5.2.1 Detect Recursion

The first issue we need to solve is detecting when the type we are processing is recursive. To achieve this, we augment the type translation algorithm of Section 5.1 with a *type stack*, where we record the names of user-defined types as we recurse into them. If at some point we come across a reference to a type that is present in the stack, we know we are dealing with a recursive type.

In our example, it becomes immediately obvious that `nat()` is a recursive type: a self-reference is encountered in the second clause of the top union.

5.2.2 Inline Type Definitions

Before we start the actual translation process, we run the recursive type declaration through a pre-processing step designed to minimize the number of mutually recursive types that need to be handled, and thus reduce the length of recursive generator call chains.

During this phase, we repeatedly replace type references inside the recursive type's structure with their definitions, being careful not to inline references to the type we are currently working on, or to other self-recursive types (if we did, the process would never terminate). To decide if a type is safe to inline, we test whether doing so and continuing with the process would result in a new reference to the type appearing in the resulting sub-expression. The inlining process is then applied recursively to all the remaining types of the recursive type's declaration. When processing such an *internal* type, we need to keep track of all the types we have recursed into so far, to avoid inlining them in the declaration.

```
 nat() :: non_neg_integer()
        | {'+', nat(), nat()}
        | {'if',
           {'=', nat(), nat()} | {'=', real(), real()},
           nat(), nat()}
        | {'from_bits', bits()}.
real() :: {'from_nat', nat()} | {'+', real(), real()}.
bits() :: {'from_nat', nat()} | {'concat', [nat() | bits()]}.
```

Listing 6. Types after inlining `cond()`

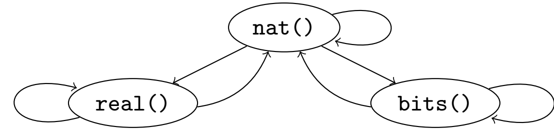

Figure 2. Dependency graph after inlining

If we re-sketch the type dependency graph at this point, we will notice that any node which did not have a self-loop or a back-edge pointing to it has been removed. Every one of the remaining types will be handled separately in the following steps, and each will produce at least one recursive generator.

Listing 6 and Figure 2 show the state of our running example after the end of the inlining phase. As these types are now internal to PropEr, we henceforth omit the `-type` attribute from the listings.

5.2.3 Push Unions to the Top-level

At this point, we rewrite each type declaration in a special form that will allow us to translate it efficiently.

In the following, we will use the term *r-union* to refer to a union expression that contains at least one reference to a user-defined type. We also define the following kinds of type expressions:

kind A contains no references to user-defined types

kind B contains at least one reference to a user-defined type, but no r-unions

kind C contains exactly one r-union, on the top-level

Union expressions are used heavily in recursive type declarations, as a way to separate the base case(s) from the recursive case(s). Therefore, most recursive type declarations will contain r-unions. However, if an r-union is present in any point of a recursive type declaration other than the top-level, we can no longer be certain about the number of sub-instances that are going to be used during generation, and thus it becomes much harder to set up an efficient shrinking strategy and control the size of produced instances. The following type is a good example of this:

```
t() :: {'node', t() | 'null', t() | 'null'}.
```

For this reason, we devote a step to transforming every type declaration to an equivalent one that has at most one r-union, necessarily on the top-level, i.e. one that is of kind B or C. Type expressions that satisfy this condition are said to be in *normal form*. The aforementioned transformation essentially involves pushing all the r-unions of a type declaration to its top-level.

We present our normalization algorithm in Table 4. In this table, the sub-expressions of each input are named according to their kind; sub-expressions of type A are named A_i etc. The entries for tuples, lists and unions should be read as follows: Whenever we encounter such a type expression, we first normalize all its sub-

Input	Result	
	Expression	Kind
T_{simple}	T_{simple}	A
`my_type()`	`my_type()`	B
$\{A_1, A_2, A_3\}$	$\{A_1, A_2, A_3\}$	A
$\{A_1, B_2, A_3\}$	$\{A_1, B_2, A_3\}$	B
$\{A_1, (C_{2a} \vert C_{2b} \vert C_{2c}), B_3, (C_{4a} \vert C_{4b})\}$	$\{A_1, C_{2a}, B_3, C_{4a}\}$ $\vert \{A_1, C_{2a}, B_3, C_{4b}\}$ $\vert \{A_1, C_{2b}, B_3, C_{4a}\}$ $\vert \ldots$	C
$[A_1]$	$[A_1]$	A
$[B_1]$	$[B_1]$	B
$[C_{1a} \mid C_{1b}]$	`[helper()]`	B
$A_1 \mid A_2 \mid A_3$	$A_1 \mid A_2 \mid A_3$	A
$A_1 \mid B_2 \mid A_3$	$A_1 \mid B_2 \mid A_3$	C
$A_1 \mid (C_{2a} \vert C_{2b}) \mid B_3$	$A_1 \mid C_{2a} \mid C_{2b} \mid B_3$	C

Table 4. How to push r-unions to the top-level of recursive types

expressions, then decide based on the table how we should combine them. As can be seen from the last column of the table, every type expression produced by the algorithm will be of kind A, B or C, so the following three cases are sufficient to cover every possible combination of sub-expressions (after they have been normalized):

- All sub-expressions are of kind A.
- At least one sub-expression is of kind B, but none is of kind C.
- At least one sub-expression is of kind C.

We provide a representative example of how we handle each of these cases, in the context of lists, tuples and unions (non-empty lists can be handled exactly like normal lists). The third rule for lists introduces a new helper type, which is defined as follows:

$$\texttt{helper() ::}\ C_{1a} \mid C_{1b}.$$

and is already in normal form.

It can easily be concluded that pushing all the r-unions of a type to its top-level does not alter its meaning. However, it does alter the union structure that the programmer had specified, which we intend to use as a guide when assigning probabilities to the recursive cases of the final generator. For example, take the last line of Table 4. According to the initial form of the type, choices A_1, C_{2a}, C_{2b} and B_3 should be assigned probabilities 1/3, 1/6, 1/6 and 1/3 respectively. After normalizing the whole union, however, the same method would assign an equal probability of $1/4$ to all four choices. To counter this effect, we annotate the r-unions on the top-level of kind C expressions with weights, which we update in accordance with the type's structure changes. These weights will be used later, when constructing the corresponding recursive generators.

Listing 7 and Figure 3 show what our abstract expressions example look like after the end of the normalization phase.

5.2.4 Find Base Cases

On this step, we detect the base case(s) of each recursive type declaration. At the start of the process, each type declaration is processed separately. Suppose we are trying to extract the base case of some type `a()`, whose definition is a type expression of kind C (if it is of kind B, we simply treat it as we would a union with one choice). We take each of the top-level union's clauses and run it through the function described in Table 5 (unions found inside the type expression require no further processing, because every r-union has already been pushed to the top-level, therefore all internal unions contain no custom type references).

```
nat()  :: non_neg_integer()₂
        | {'+', nat(), nat()}₂
        | {'if', {'=',nat(),nat()}, nat(), nat()}₁
        | {'if', {'=',real(),real()}, nat(), nat()}₁
        | {'from_bits', bits()}₂.
real() :: {'from_nat', nat()}₁ | {'+', real(), real()}₁.
bits() :: {'from_nat', nat()}₁ | {'concat', [bits_helper()]}₁.
bits_helper() :: nat()₁ | bits()₁.
```

Listing 7. Types after union pushing, with weight annotations

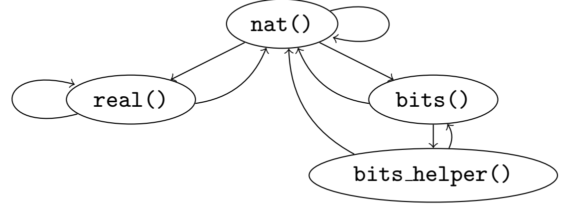

Figure 3. Dependency graph after union pushing

Type	bc(Type)
T_{simple}	T_{simple}
`a()`	(ABORT)
`b()`	`b()`
$\{T_1,\ldots,T_N\}$	$\{\text{bc}(T_1),\ldots,\text{bc}(T_N)\}$
$[T]$	[]
$[T,\ldots]$	$[\text{bc}(T),\ldots]$
T_{union}	T_{union}

Table 5. How to extract a base case (function bc) from some clause of the top-level union of type `a()`

If the function fails on all the clauses, then we are certain that `a()` does not have a base case, so the whole translation process terminates with an error. Otherwise, we collect all the clauses for which the bc function returned a result and categorize them based on whether the corresponding result contains a reference to some other type or not. The former are characterized as *conditional* base case clauses, while the latter as *clear*. If `a()` has at least one clear base case clause, we keep only the clear clauses and mark `a()` as an *independent* type. Otherwise, we mark `a()` as a *reliant* type. The intuition behind these names is that types in the second category do not have a base case themselves; instead they fall back to some other type.

At this point, we consider our set of recursive type declarations as a whole. Our goal is to ensure that every type has a well-defined base case, even if it can only be reached indirectly. If every type has been marked as independent, then they all have at least one clear base case, and the above condition is trivially satisfied. Otherwise, we have to verify that for every reliant type in the set, every possible series of clause choices is guaranteed to terminate. Consider, for example, the following types:

```
-type foo() :: bar().
-type bar() :: foo() | baz().
-type baz() :: 'ok' | {'boo', baz()}.
```

Of these types, `baz()` is independent (it has a clear base case, `'ok'`), `foo()` is reliant (it has a single, conditional base case clause, `bar()`) and `bar()` is also reliant (it has two conditional base case clauses, `foo()` and `baz()`). If we keep both conditional clauses of `bar()`, then `foo()` and `bar()` will not have well-defined

base cases: we can keep selecting the clause `foo()` when processing `bar()` and the clause `bar()` when processing `foo()`, thus creating an infinite loop. The problem, however, can easily be solved, by removing the first of `bar()`'s base case clauses.

In conclusion, we need to find a subset of each reliant type's base case clauses for which no reference cycles are created. (Of course, we must leave at least one clause per reliant type.) This problem can be restated as a graph problem, and solved as such.

The types from our running example have the following base cases:

```
nat() : non_neg_integer()
real() : {'from_nat', nat()}
bits() : {'concat', []}
bits_helper() : nat() and bits()
```

Thus, types `nat()` and `bits()` are independent, while `real()` and `bits_helper()` are reliant. In this particular example, we can keep all the base case clauses of the two reliant types.

5.2.5 Prepare the Recursive Calls

On this step, we convert type references to generator calls.

Each top-level clause of each type in our set of recursive type declarations is processed separately. Given the type expression of some clause, we first identify and enumerate all the type references it contains. At this point, we do not treat self-references differently from other references, and we consider a list that contains multiple references (e.g., this one: `[{a(),b()}]`), as a single reference. Let k be the total number of references contained in the clause.

Each reference is then converted into a call to the corresponding sized generator, with the size parameter equal to a $1/k$-fraction of a parameter S (this parameter will be bound at a later time). A special case arises when $k = 1$ and the single reference is a back-edge or self-loop. In that case, we have to call the sized generator for size equal to $S - 1$, to ensure the termination of the generation process. Additionally, references contained in lists have to be translated somewhat differently, e.g. the sub-expression `[a()]` would be converted to resize(S div k, list(a())). Notice that we do not use the sized generator in this case.

As an example of this procedure, consider the type declarations of Listing 7. The second `'if'` clause of `nat()` contains four type references, all of which would be converted to generator calls at size S div 4. The `'concat'` clause of `bits()` contains a single list of reference-containing elements, which would be converted to resize(S, list(bits_helper())). The `nat()` clause of `bits_helper()` contains a single reference, which also happens to be a back-edge, so it will be converted into a call to the `nat/1` generator at size $S - 1$.

5.2.6 Determine Shrinking Behaviour

It is often the case with recursive types that recursively produced instances are simply combined using tuples (the type declarations in Listing 7 are a good example of this). In these cases, instead of generating the sub-instances as we build the tuple, we can first generate them all and then combine them into a tuple. The two idioms will produce essentially the same end result, but the second one allows the use of `?LETSHRINK`, which can significantly improve the effectiveness of the shrinking process. The same technique can be applied to recursive clauses that contain lists of self-references: we simply pre-produce a list of sub-instances.

5.2.7 Compose a Generator

In the final step of the translation process, we compose all the pieces we have created so far into a recursive generator. For each type in our set of recursive type declarations, we do the following:

1. Set up a sized generator with two clauses, one for `size` = 0 and one for `size` = $S > 0$.
2. Fill in the 0-size clause with a union of all base case clauses assigned to the type, after replacing any type references inside those clauses with calls to the corresponding sized generators at `size` = 0.
3. Fill in the S-size clause with a weighted_union containing all of the type's clauses, and set the weights according to the top-level r-union's weights.
4. Add a base case fallback clause (a self-call with `size` = 0) to the head of the S-size weighted_union, and recalculate the weights of all clauses to achieve the desired fallback probability.
5. Wrap each weighted_union with a `?LAZY` wrapper.
6. Convert the non-recursive parts of the type declarations using the method described in Section 5.1.
7. Add a no-size generator for the type, created by wrapping the sized generator with a `?SIZED` macro.

Listing 8 shows the final output of our abstract expressions example. We assume that the user has specified a $1/5$ base case fallback probability.

5.3 Integration with PropEr Notation

The type-to-generator converter is not enough. We also need to provide a way for testers to write properties using Erlang types. To make this process as intuitive as possible, we decided to allow the use of Erlang types anywhere a PropEr generator would be expected, i.e. inside both property declarations and user-defined generators. In particular, if PropEr detects that some part of a generator expression does not correspond to a valid generator, but would constitute a legal Erlang type, it will assume that the expression represents an Erlang type and use the method illustrated in the previous sections to convert it. See Listing 9 for an example of the use of Erlang types in properties.

The use of Erlang types in modules containing properties is subject to the following rules:

- Any local user-defined Erlang type may be used, as long as it is not shadowed by a local or imported function (or an auto-imported Built-In Function) of the same name and arity.
- Any remote Erlang type may be used, as long as it is exported from its defining module and not shadowed by some function of the same name and arity, exported from the same module.
- Types and generators can be combined in arbitrarily nested lists and/or tuples.
- All other constructs of the Erlang type language, such as the union operator `'|'`, are not allowed, because they are rejected by the Erlang parser.
- The parameters of an Erlang type can only be other types.
- The parameters of PropEr generators and the arguments of PropEr constructors like `?LET` can include both Erlang types and PropEr generators.
- If an expression can be interpreted both as a PropEr generator and as an Erlang type, the former takes precedence. This may cause some confusion when list syntax is used (cf. Section 4.1).

To implement this behaviour, we need to write a component that locates all the "call" structures[3] in a module and decide for each one

[3] A call is a syntactic entity of the form `name(...)` (a local call) or `mod:name(...)` (a remote call), which can either be a function call or a reference to a type.

```
nat() ->
    ?SIZED(Size, nat(Size)).

nat(0) ->
    non_neg_integer();
nat(S) ->
    weighted_union([
        {2, ?LAZY(nat(0))},
        {2, ?LAZY(non_neg_integer())},
        {2, ?LAZY(?LETSHRINK([X,Y], vector(2,nat(S div 2)),
                {'+', X, Y}))},
        {1, ?LAZY(?LETSHRINK([X,Y,Z,W], vector(4,nat(S div 4)),
                {'if', {'=', X, Y}, Z, W}))},
        {1, ?LAZY(?LETSHRINK([X,Y], vector(2,nat(S div 4)),
                {'if', {'=', real(S div 4),real(S div 4)},
                       X, Y}))},
        {2, ?LAZY({'from_bits',from_bits(S)})}]).

real() ->
    ?SIZED(Size, real(Size)).

real(0) ->
    {'from_nat',nat(0)};
real(S) ->
    weighted_union([
        {2, ?LAZY(real(0))},
        {3, ?LAZY({'from_nat',nat(S-1)})},
        {3, ?LAZY(?LETSHRINK([X,Y], vector(2,real(S div 2)),
                {'+', X, Y}))}]).

bits() ->
    ?SIZED(Size, bits(Size)).

bits(0) ->
    {'concat',[]};
bits(S) ->
    weighted_union([
        {2, ?LAZY(bits(0))},
        {3, ?LAZY({'from_nat',nat(S-1)})},
        {3, ?LAZY({'concat',resize(S,list(bits_helper()))})}]).

bits_helper() ->
    ?SIZED(Size, nat(Size)).

bits_helper(0) ->
    union([nat(0), bits(0)]);
bits_helper(S) ->
    weighted_union([{2, ?LAZY(bits_helper(0))},
                    {3, ?LAZY(nat(S-1))},
                    {3, ?LAZY(bits(S-1))}]).
```

Listing 8. The final generator

of them whether it refers to a function (i.e., a PropEr generator) or an Erlang type. Calls to PropEr generators are left unchanged, while references to Erlang types are replaced by calls to the type conversion subsystem.

For local calls, this decision can be made safely at compile time. Because we have access to the code of the module, we can easily check which one of the following cases is true for each local call:

- There exists a function in scope of that name and arity, therefore the call represents a PropEr generator.
- There exists a locally-defined Erlang type of that name and arity which it is not shadowed by any similar function in scope, therefore the call represents an Erlang type.
- Neither a function nor a type with that name and arity exists in the current scope, therefore the call is an error.

```
%% Inside myprops.erl:

prop_new_array_handles_any_opt_combination() ->
    ?FORALL(Opts, array:array_opts(),
            array:is_array(array:new(Opts))).

%% Inside array.erl:

-type array_opt() :: 'fixed' | non_neg_integer()
                   | {'default', term()} | {'fixed', boolean()}
                   | {'size', non_neg_integer()}.
-type array_opts() :: array_opt() | [array_opt()].
```

Listing 9. Example of the use of Erlang types in properties: Verifying that `array:new/0` can handle any combination of options.

The above method cannot be applied to remote types, because Erlang modules are not linked until runtime, therefore it is possible that the referenced remote modules will not be present during compilation. Instead, we have to delay the processing of remote calls until runtime. When such a call is encountered during testing, PropEr first tries to find an exported function with the correct signature. Only if this fails will PropEr search the `-export_type` declarations of the remote module for a type that matches the original remote call.

A little extra work is required when handling remote calls with one or more arguments. If at least one of them cannot possibly be interpreted as an Erlang type, then the remote call must surely be a function call (we do not allow PropEr generators as parameters to Erlang types). If, instead, some argument cannot be interpreted as a PropEr generator, but only as an Erlang type, then we process it as such. The problem arises when an argument can be interpreted in both ways. In this case we have to keep two representations of the argument, one as a normal value and one as a type expression, because we do not know which one will eventually be used.

We implemented these operations as part of a parse transform which is applied automatically to any module that includes PropEr's header file.

5.4 Automatic Function Specification Testing

Having described the type translation scheme, we are ready to turn our attention to the other component of the Erlang type language, function signatures. Function signatures (or specs) are essentially a form of lightweight specification. A `spec` of the form:

$$\texttt{-spec foo}(A_1, \ldots, A_N)\ \texttt{->}\ R.$$

can be interpreted as: "function `foo/`N, if called with arguments of types $A_1, \ldots, A_N$ and returns some value, then that value will be of type R". It should be easy, then, to convert such descriptions into testable properties, and thus provide a method for testing functions automatically. Our implementation of such a system can test an exported function against its spec by calling it with increasingly complex valid inputs (as specified in the spec's domain) and checking that no unexpected value (according to the spec's range) is returned. We will describe these steps in detail (each clause of a multi-clause `spec` is treated like a separate contract).

5.4.1 Generate Valid Inputs

To produce valid inputs for the function under test, we simply extract the domain type from its `spec`, convert it to PropEr's generator format and pass it to the random instance generator. Some extra work is required for `specs` that contain variables, as the type translator can only work with fully instantiated types. On each testing run, every type variable present inside the `spec` is instantiated to

increasingly broader types, taking care to remain inside the limits enforced by any subtype constraints.

5.4.2 Check the Return Value

If the function under test returns normally for a random input, we need to check that the returned value is in accordance with the function's `spec`. Essentially, we need to write a component for instance checking terms against Erlang types. The algorithm used by such a component can easily be derived from the description of the type language given in Section 2, with only a few cases requiring special attention:

- Record expressions are expanded into tuple types. Each field's type, if not overridden in the reference, is copied from the record's original definition.
- References to user-defined types (either local or remote) are expanded using the corresponding type definitions. If a user-defined type is parametric, we first replace every variable in its definition with the value of the corresponding actual parameter. In the case of parametric remote types, we also need to annotate the parameters with their originating module, for reasons explained in Section 5.1.
- Recursing into a type structure generally requires that we consume some bit of the input term. For example, in order to recurse into a tuple we have to consume the tuple structure encasing the elements. Therefore, since Erlang terms are of finite size, the instance checking process is guaranteed to terminate.
- Unions and references to user-defined types are an exception to the previous rule. The incorrect use of these types in recursive type declarations has the potential to create an infinite loop in our instance checker. The problem arises when a recursive type contains at least one recursion path of just unions and type references, like in these (non-sensical) type declarations:

```
-type a()  :: atom() | a().
-type b()  :: float() | c().
-type c()  :: integer() | b().
-type d(T) :: T | d({'bar', T}).
```

  By keeping a recursion stack, we are able to detect such declarations and stop the process early.
- It is impossible, in the general case, to fully test the domain of funs, simply because in most cases it is infinite. Therefore, we can do little to verify that returned funs abide by their specification beyond checking their arity.

Abnormal function returns also need to be classified. However, the Erlang type language in its current form lacks a standard way of specifying exceptional function behaviour. Thus, in our effort to avoid false positives, we are currently forced to accept any thrown exception as a normal return, as long as it does not signify a system error (we also accept `badarg` errors, which are commonly used to signify an illegal input).

6. Practical Experiences

In this section, we briefly comment on the effectiveness of PropEr as a type-aware PBT tool and an automatic function tester. We base our analysis both on our own experience and on feedback we have received from PropEr's users.

6.1 Sources of Feedback

In an effort to assess the usefulness of PropEr's type language integration component, we have tried PropEr on various open-source code bases that contain sufficient type information. The most significant of these code bases was Erlang/OTP's standard library. The bugs that PropEr's spec tester uncovered on this code were mostly errors in specs:

- The specs for functions `filename:join/1`, `gb_sets:intersection/1` and `ordsets:intersection/1` incorrectly stated that they could also accept the empty list.
- The specs for `lists:merge/1` and `lists:umerge/1` read `([T]) -> [T]` instead of the correct `([[T]]) -> [T]`.
- The spec for `orddict:filter/2` stated that the predicate used to filter the dictionary could return `any()`, whereas only `boolean()` is actually acceptable.

Although at the time of this writing PropEr has not been properly released yet, various open-source Erlang projects (e.g. riak, hibari and mochiweb), and even big companies, are already adopting it as their PBT framework of choice. Support for PropEr was recently added to the popular Erlang project management tool rebar. On our part, we are actively working towards expanding our user base, and have remained in close communication with most of our users, who have supplied us with a good amount of feedback, in the form of bug reports, patches and suggestions.

6.2 Observations

From the feedback we received on PropEr's integration with the type language, we were able to make the following observations:

- The Erlang type language cannot always accurately express the characteristics of data structures. For example, users cannot specify that a certain binary should represent valid UTF-8 character sequences, or that some list should always have exactly k elements, or an even number of elements. In such cases, the user has to manually write appropriate generators, something which is of course possible in PropEr.
- Function `specs` cannot express dependencies that might exist between different arguments of a function. Good examples of this are the `lists:nth/2` function whose first argument is a positive integer less or equal to the length of the list in its second argument, or the `lists:zip/2` function whose documentation states that it only accepts two lists of the same length. However, the closest a `spec` can get to specifying the actual behaviour of this function is: `zip([T],[S]) -> {T,S}`, which also allows for lists of different lengths. Even if a function accepts all combinations of arguments, it is likely that independently generated arguments will mostly exercise a subset of all possible use cases of the function. For example, suppose we independently generate two arguments for the function `lists:delete/2`: a term `X` and a list `L`. It is very unlikely that `X` will happen to be a member of `L`, therefore a big part of the function's behaviour will not be tested adequately by values generated in that manner.
- Despite the previous points, Erlang's type language may well be adequate for accurately describing the input domains of functions. As a testament to this, the developers of mochiweb were able to make full use of PropEr as a spec tester in a module for numeric calculations, where specs were total.
- When using PropEr as a spec tester, users must be careful not to underspecify their function's domains, because doing so may cause PropEr to run the function with illegal inputs. A common case of overapproximating a function's domain are specs for functions which can only process non-empty lists of some type T, stating that the function accepts arguments of type `[`T`]` instead of the correct `[`T`,...]`. The use of the `any()` type in function specifications is also often an overapproximation.

- Function signatures are a rather simple form of specification: they cannot be expected to discover subtle errors, nor can they be used to test (among others) inter-functional properties. Therefore, at their current form, specs cannot replace user-written properties.

After taking these and other observations into account, some parts of our first iteration of PropEr had to change. The most important one of them was that in order to help users overcome the limitations of Erlang's type language, we had to make our type integration component more customizable. Users can now mark function specs as untestable, or provide their own generators for datatypes that cannot be fully specified using types.

6.3 Lessons Learned and Some Advice

Through our interaction with PropEr's users, we have gained a greater understanding of how programmers go about using PBT tools, and which aspects of PBT are the most difficult for programmers to understand.

- Switching from testing with unit tests to testing with properties is often a challenge for programmers, because PBT requires a completely different mindset. New users often find it hard to come up with interesting properties to test, and sometimes have trouble expressing them properly. A significant part of their time is spent debugging the properties rather than the code.
- Writing generators, especially recursive ones, often requires considerable time and effort. New users find it hard to make efficient use of the generator primitives, and to express their generators in a declarative manner. In this respect, PropEr improves significantly over other existing PBT tools for Erlang.
- Users are generally not familiar with the intricacies of Erlang's type language, and have a hard time tightening their specs, to allow for the efficient use of spec testing.
- The list syntax discrepancy between the Erlang type language and PropEr's generator language has been a source of confusion in some cases.

Based on what we have learned, we have these pieces of advice to give to (new) users of PropEr; some apply to PBT in general.

- Users should start with the functional core of their application, which is usually much easier to test than the front-end, and is often amendable to spec testing.
- Testing the full functionality of your program using a single property is not a good idea. Instead, one should test many, smaller, partial properties with various kinds of inputs. Such properties are surprisingly effective at uncovering bugs, and are easier to get right than full specifications.
- Properties should be written for readability. Each property should be viewed as a way of documenting a specific aspect of the code. Property and variable names should be descriptive and the input generators should be as declarative as possible.
- Generators for recursive datatypes are hard to get right and PropEr's automatically produced generators often are enough.
- When trying to tightly spec your code, make use of Dialyzer's `-Wunderspecs` option, which warns about specs that are strictly more allowing than the success typing inferred for that function.
- Slight alterations to the syntax of type declarations can make them better suited to their role as term generators:
  - Putting the simplest cases first in unions achieves better shrinking performance.
  - Sometimes, adding redundant clauses to unions improves the distribution of produced instances. For example, consider the following type declaration for compiler attributes: `'type' | 'spec' | atom()`. Although this declaration is semantically equivalent to just `atom()`, it will result in a specialized generator which produces the more "interesting" compiler attributes `'type'` and `'spec'` 66.6% of the time.
- When a type declaration (or a simple generator) is almost sufficient to fully specify the structure of some data type (in the sense that only a small fraction of the produced values are not valid instances of the datatype), a simple use of a `?SUCHTHAT` macro to reject the invalid values is preferable to writing a more complex generator.
- Type variables in function specs that express relations between arguments and results allow PropEr to test those functions more effectively.

7. Concluding Remarks

We have presented PropEr, a property-based testing tool that is tightly integrated with the Erlang type language. We have described the approach we took in providing such an integration, which consists of a system for the conversion of types to term generators, an extension to the language of properties and generators that allows the use of types in place of generators, and a component that can test functions automatically based solely on their signatures. We focused particularly on the conversion algorithm for recursive types, which was the main technical challenge of this work.

Based on the feedback we have received from PropEr's users, as well as the results of our own experimentation with the tool, it is clear that our approach currently suffers from expressivity limitations of the Erlang type language. We plan to explore possible solutions to these issues, which will probably involve extensions to the Erlang type language, as we continue our work on PropEr.

References

[1] T. Arts, J. Hughes, J. Johansson, and U. Wiger. Testing Telecoms Software with Quviq QuickCheck. In *Proceedings of the 2006 ACM SIGPLAN Workshop on Erlang*, pages 2–10. ACM, 2006.

[2] R. Carlsson and M. Rémond. EUnit: A Lightweight Unit Testing Framework for Erlang. In *Proceedings of the 2006 ACM SIGPLAN Workshop on Erlang*, pages 1–1, New York, NY, USA, 2006. ACM.

[3] K. Claessen and J. Hughes. QuickCheck: a Lightweight Tool for Random Testing of Haskell Programs. In *Proceedings of the 5th ACM SIGPLAN International Conference on Functional Programming*, pages 268–279. ACM, 2000.

[4] EDoc. User's Guide, 2011. `http://www.erlang.org/doc/apps/edoc/users_guide.html`.

[5] M. Jimenez, T. Lindahl, and K. Sagonas. A Language for Specifying Type Contracts in Erlang and its Interaction with Success Typings. In *Proceedings of the 2007 SIGPLAN Workshop on Erlang*, pages 11–17. ACM, 2007.

[6] T. Lindahl and K. Sagonas. Practical Type Inference Based on Success Typings. In *Proceedings of the 8th ACM SIGPLAN International Conference on Principles and Practice of Declarative Programming*, pages 167–178. ACM, 2006.

[7] PropEr. Property-based testing for Erlang, 2011. `http://proper.softlab.ntua.gr/`.

[8] J. Rushby. Automated Test Generation and Verified Software. In B. Meyer and J. Woodcock, editors, *Verified Software: Theories, Tools, Experiments*, volume 4171 of *Lecture Notes in Computer Science*, pages 161–172. Springer, Berlin / Heidelberg, 2008.

[9] K. K. Thorup. Triq: Trifork QuickCheck, 2011. `http://krestenkrab.github.com/triq/`.