Εθνικό Μετσόβιο Πολυτεχνείο
Σχολή Ηλεκτρολόγων Μηχανικών και Μηχανικών Υπολογιστών
Τομέας Τεχνολογίας Πληροφορικής και Υπολογιστών

Αυτόματος Τυχαίος Έλεγχος Ιδιοτήτων Συναρτήσεων από τις Προδιαγραφές τους

Διπλωματική Εργασία

του

Εμμανουήλ Παπαδάκη

Επιβλέπων: Κωστής Σαγώνας
Αν. Καθηγητής Ε.Μ.Π.

Εργαστήριο Τεχνολογίας Λογισμικού
Αθήνα, Οκτώβριος 2010

Εθνικό Μετσόβιο Πολυτεχνείο
Σχολή Ηλεκτρολόγων Μηχανικών και Μηχανικών Υπολογιστών
Τομέας Τεχνολογίας Πληροφορικής και Υπολογιστών
Εργαστήριο Τεχνολογίας Λογισμικού

Αυτόματος Τυχαίος Έλεγχος Ιδιοτήτων Συναρτήσεων από τις Προδιαγραφές τους

Διπλωματική Εργασία

του

Εμμανουήλ Παπαδάκη

Επιβλέπων: Κωστής Σαγώνας
Αν. Καθηγητής Ε.Μ.Π.

Εγκρίθηκε από την τριμελή εξεταστική επιτροπή την 22η Οκτωβρίου, 2010.

.....	.....	.....
Κωστής Σαγώνας	Νικόλαος Παπασπύρου	Κώστας Κοντογιάννης
Αν. Καθηγητής Ε.Μ.Π.	Επικ. Καθηγητής Ε.Μ.Π.	Αν. Καθηγητής Ε.Μ.Π.

Αθήνα, Οκτώβριος 2010

.....
Εμμανουήλ Παπαδάκης
Διπλωματούχος Ηλεκτρολόγος Μηχανικός και Μηχανικός Υπολογιστών Ε.Μ.Π.

Περίληψη

Τα τελευταία χρόνια, όλο και περισσότεροι προγραμματιστές της γλώσσας Erlang χρησιμοποιούν εργαλεία ελέγχου βάσει ιδιοτήτων για τον έλεγχο των προγραμμάτων τους. Σήμερα, τέτοια εργαλεία έχουν ελάχιστη σύνδεση με το σύστημα τύπων της γλώσσας. Σε αυτή τη Διπλωματική Εργασία, διερευνούμε ορισμένους τρόπους για την ενσωμάτωση στοιχείων του συστήματος τύπων της Erlang σε ένα τέτοιο εργαλείο. Συγκεκριμένα, περιγράφουμε πώς ένα τέτοιο εργαλείο θα μπορούσε να χρησιμοποιήσει τις δηλώσεις τύπων διάφορων τύπων δεδομένων προκειμένου να παράγει αυτόματα τις αντίστοιχες γεννήτριες, πώς θα πρέπει να αντιμετωπίσει αφηρημένους τύπους δεδομένων με κρυφή εσωτερική αναπαράσταση, και επίσης πώς θα μπορούσε να χρησιμοποιήσει την πληροφορία που περιέχεται στην υπογραφή μίας συνάρτησης προκειμένου να την ελέγξει αυτόματα. Έχουμε αναπτύξει ένα πρωτότυπο ενός τέτοιου συστήματος, το οποίο ονομάσαμε PropEr. Από δοκιμές του PropEr ως εργαλείο αυτόματου ελέγχου συναρτήσεων γίνεται φανερό ότι, ενώ ένα τέτοιο σύστημα είναι οπωσδήποτε σε θέση να εντοπίσει σφάλματα λογισμικού, η προσέγγισή μας παρουσιάζει ορισμένα εγγενή προβλήματα, τα οποία αφήνουμε ως μελλοντική εργασία.

Λέξεις Κλειδιά

έλεγχος λογισμικού βάσει ιδιοτήτων, αυτόματος έλεγχος λογισμικού, τυχαίος έλεγχος λογισμικού, προδιαγραφές συναρτήσεων, παραγωγή ελέγχων λογισμικού βάσει τεκμηρίωσης

Abstract

Property-based testing tools have recently become quite popular among Erlang developers. Such tools currently have little connection with the type system of the language. In this thesis, we explore some possible ways of integrating elements of the language's type system into a property-based testing tool. Specifically, we describe how such a tool could utilize type declarations to produce the corresponding term generators automatically, how it should handle abstract data types with hidden representations, and also how it could use the type information contained in the signature of a function to test it automatically. We have developed a prototype for such a system, called PropEr. Tests from using PropEr as an automatic function tester show that, while such a system is definitely capable of finding errors in programs, our approach suffers from some key limitations, which we leave as future work.

Keywords

property-based testing, automated testing, random testing, function specifications, deriving tests from documentation

Ευχαριστίες

Θα ήθελα κατ' αρχάς να ευχαριστήσω τον επιβλέποντα καθηγητή μου Κωστή Σαγώνα για την καθοδήγησή του κατά την εκπόνηση της παρούσας εργασίας. Η εργατικότητα και ο ενθουσιασμός του αποτέλεσαν και αποτελούν έμπνευση για μένα.

Ευχαριστώ τους καθηγητές μου Νίκο Παπασπύρου, Στάθη Ζάχο, Άρη Παγουρτζή και Κώστα Κοντογιάννη, που με μύησαν στα μυστικά της Πληροφορικής.

Ευχαριστώ όλα τα παιδιά του Εργαστηρίου Λογισμικού, που με βοήθησαν, ο καθένας με τον τρόπο του, σε κάθε βήμα της διαδικασίας.

Τέλος, ευχαριστώ την οικογένειά μου για την κατανόηση και την υποστήριξή τους όλα αυτά τα χρόνια.

Παπαδάκης Εμμανουήλ

Contents

List of Tables

List of Listings

Chapter 1

Introduction

For a long time, users of the Erlang programming language had no practical way of including type information in their code. This recently changed, with the addition of a standard type annotation language, which programmers can use to write type signatures for their functions. Programmers are encouraged to do so, due primarily to the usefulness of function signatures as program documentation, but also because static analysis tools like Dialyzer can utilize them to improve their error detection rate. The use of this type language has been steadily increasing ever since its introduction.

Recent years have also seen a rise in the popularity of property-based testing tools within the Erlang community. Programmers can use such tools to test their programs by providing properties which are expected to hold on every execution. The tool itself, then, tries to falsify these properties through a series of random tests. Each testable property must be accompanied by a specification of the terms for which it should hold (this specification provides a guide for the random generation of tests). To write such a specification, programmers need to use a custom type annotation language, which usually overlaps with Erlang's type language in many respects.

As part of this thesis, we have explored ways of integrating Erlang's type annotation language with a property-based testing tool. By providing users with one such tool that can work with Erlang type specifications directly, we effectively make it easier for programmers to start using these tools, while adding more value to the presence of type information in user code. We have also examined ways of using type signatures to test functions automatically, a feature which would add even more value to function specifications, further encouraging programmers to invest time in writing them. Such a system could also be employed as a complementary tool to static analyzers in the process of verifying the accuracy of user-provided specifications.

In order to test our ideas, we required a property-based testing tool to use as a base. To ensure maximum flexibility, we decided to implement such a tool from scratch instead of working with an existing one; we named that tool PropEr (PROPerty-based testing tool for ERlang). PropEr was then extended with the addition of a compatibility layer, which allowed the use of Erlang type expressions in properties. This was followed by a component that automatically produces generators for abstract data types based on their API, and, finally, a component that tests functions automatically based on their specs.

PropEr's aptitude as an automatic spec testing tool was measured by applying it to various modules of the Erlang standard library, and also on itself. We observed that, while this

component is clearly capable of detecting faults, it is inherently limited in its applicability and requires further extensions before it can be used in the general case.

Outline of the Thesis

The rest of this thesis is organized as follows: Chapter 2 gives an introduction of the Erlang language and describes its type system in detail. Chapter 3 describes property-based testing and introduces the PropEr tool. This is followed by the main chapter of the thesis, Chapter 4, where we present the extensions we have made to PropEr in order to make it compatible with Erlang's type system, and introduce a system for the automatic testing of function specs. In Chapter 5 we report on our experiences from using PropEr's automatic spec tester on some code bases. Finally, Chapter 6 gives an overview of related work and Chapter 7 concludes.

Chapter 2

The Erlang Language and Type System

2.1 The Erlang Language

Developed within Ericsson's research labs in the 1980's, Erlang is a programming language designed for writing embedded control software for telecommunications systems, where high availability is a major requirement. As such, Erlang was built from the ground up to facilitate the development of scalable and robust non-stop, soft real-time systems [6, 5]. Features like fault-tolerance, lightweight concurrency, efficient distribution and communication, even on-the-fly code reloading, most often an afterthought to language designers, were considered of vital importance. Aiming to provide all these, the makers of Erlang made two major design decisions. First, Erlang was chosen to be a strict, dynamically-typed functional language, sacrificing the better runtime efficiency of imperative languages for the compactness and clarity of side-effects-free code. Second, Erlang differs from most other concurrent programming languages in that it implements the actor concurrency model: Erlang processes can only communicate with one another through asynchronous message-passing, making it much easier to program and debug concurrent applications.

As mentioned above, Erlang is a dynamically typed language, meaning that no kind of type annotation is required in the source code and the compiler performs very little type-checking during the compilation process. Instead, all values are tagged with type information at runtime, allowing the Erlang runtime system to detect and prevent the improper use of a function or language operator, thus providing type safety. This system is a good fit for Erlang's typical uses, i.e. telecom applications, where servers (implemented as processes) are often expected to gracefully handle all incoming messages, even invalid ones, and therefore benefit from untyped routines and communication channels. Moreover, not having to conform to a static type system allows for greater expressiveness and flexibility (polymorphism can be taken to the extreme), and may help to speed up development.

This freedom, however, comes with a price. No type checking means that programmers get very little feedback from the compiler. Instead, they have to actually test their program to uncover even the most trivial of errors (e.g. typos), errors that any static type system would have easily detected. In addition, programmers that aren't forced to spell out their intention to the compiler can easily end up writing convoluted code that's hard for

other people to understand. Even if a programmer wished to clearly define a function's behaviour, until recently she couldn't easily document it, since there was no standard type annotation system.

2.2 Adding a Type System

2.2.1 Past Efforts

Over the years, a number of solutions to the shortcomings of Erlang's dynamic type system have been proposed. Marlow and Wadler developed a type system based on subtyping [68], which never caught on with Erlang developers, because it essentially tried to impose on them a programming style more suited to statically-typed programming languages. More recently, Nyström introduced a soft type system [76], which only covered a subset of the language and was hard to use in practice. The EDoc documentation tool [1] is also worth mentioning, since it allows programmers to specify the interface to their functions through signatures written in a custom type notation language. These signatures, however, are never checked against the code (they are really just comments) and are thus susceptible to code rot.

2.2.2 The Dialyzer Tool

In 2004, Lindahl and Sagonas presented a tool called Dialyzer (DIscrepancy AnaLYZer for ERlang programs) [62], that used static analysis methods to detect various kinds of software defects in Erlang code. Dialyzer was fast, easy to use, imposed no restrictions on function use, required no user guidance at all and proved to be quite effective in practice. Most importantly, Dialyzer was sound for defect detection: it only reported provable type errors, i.e. code points that would definitely raise an exception due to a type clash. While it made no guarantees on its error detection rate, its lack of false positives made it very appealing to programmers. From that point on, Dialyzer has seen increasing use in the Erlang community, eventually becoming part of the official Erlang distribution, Erlang/OTP, in 2007.

Inferring Success Typings

Traditional type systems, in their effort to prove type safety, often have to restrict the use of functions. Therefore, such type systems will often reject perfectly reasonable programs. The function signatures that they infer only allow those inputs for which the function is guaranteed to evaluate without type errors. At its core, Dialyzer, too, functions like a type inferencer, by deducing the implicit type information which exists in Erlang programs [64]. Dialyzer, however, doesn't aim to prove type safety (this is already guaranteed by the implementation of the Erlang language), but locate definite type errors. Its purpose is essentially to capture the biggest set of terms for which it can be proven that type clashes will occur. The type signatures that Dialyzer infers, called "success typings", are the complement of that set of terms. Success typings are an over-approximation to the set of terms for which a function can evaluate: the domain of the signature includes all possible values that the function could accept as parameters, and its range includes all

possible return values for this domain. Success typings are guaranteed to capture all intended uses of a function, along, perhaps, with some erroneous ones. Thus, any use of a function that is incompatible with its success typing will definitely fail. In effect, success typings approach the type inference problem from a direction opposite to that of type systems for statically typed languages.

2.2.3 Creating a Type System

Dialyzer's creators soon realized that success typings could also be used as program documentation, since they will never fail to capture some possible use of a function [63]. Their next tool, TypEr, was a fully automatic type annotator which utilized Dialyzer's inference engine to reconstruct a significant portion of the type information implicit in an Erlang program. That way, TypEr provided users with automatic documentation that would evolve together with the program and would not rot.

It was obvious, however, that inferred success typings, being over-approximations, could often be greatly refined through user intervention. Thus, a specification language for Erlang was designed [50], one that would allow user-provided function contracts, both as a means of documenting a function's intended use and as extra information for Dialyzer to work with. This addition largely solved Erlang's documentation problem, because type signatures expressed in this type system were verifiable, and therefore much less susceptible to code rot. Nowadays, most of Erlang/OTP's libraries, as well as many open source applications written in Erlang, come with type contracts for functions, especially those which are part of the public API.

2.3 The Erlang Type System

The Erlang type system was designed to be compatible with the language's dynamic semantics. For that reason, it could not be modeled after the traditional unification-based Hindley-Milner type systems, but had to be based on unrestricted subtyping. Its makers also placed more importance on intuitiveness, readability and simplicity than on expressive power. The Erlang type system was also influenced from EDoc's type notation language.

In this section, we describe the Erlang type system in detail.

2.3.1 Built-in Types

In Erlang, a piece of data of any kind is called a term. Based on its runtime representation, a term will belong to exactly one[1] of the categories outlined in Table 2.1. Types can be used to describe sets of Erlang terms, both bounded and infinite. A few predefined types are provided (see Table 2.2), most of which are refinements over the representation-based classification of terms. In addition to the information presented in Tables 2.1 and 2.2, we note the following:

[1]Note, however, that binaries are also bitstrings and, correspondingly, bitstrings of a size divisible by 8 are also binaries.

Category	Description	Examples
Integer	A mathematical integer	−31, 0, 17, 42
Float	A floating point number	−0.123, 3.14
Atom	A named constant	hello, 'World'
Binary	An untyped series of bytes	«255,0,98», «42»
Bitstring	An untyped series of bits	«99,3:2», «1:1,0:1», «4»
Pid	A handle for talking to an Erlang process	–
Port	A handle for talking to an external program	–
Reference	A term unique within a runtime environment	–
Fun	A callable function object	fun(X) → X + 1 end, fun lists:reverse/1
Tuple	A compound term with a fixed number of elements	{0,alabama,3.14}, {answer,42}
List	A compound term with a variable number of elements (not necessarily of the same type)	[1,2,3], [42,answer], [for,whom,the,bell,tolls]

Table 2.1: Categories of Erlang terms

- Erlang treats characters as simple integers instead of defining a separate character data type.

- Strings are represented as lists of characters.

- There is no boolean data type in Erlang. Instead, the atoms 'true' and 'false' are used to denote boolean values.

- Erlang also provides a "record" data type, which behaves like a tuple with named fields. Records, however, are not really a separate data type, since they are converted to tuples at compile time. A record declaration (see Listing 2.1 for an example) consists of the record's name (an atom) and zero or more field entries. Each field entry specifies the field's name (also an atom) and may contain an initialization expression and/or a type declaration. If both are present, the initialization value must be an instance of the field type. Uninitialized fields of newly created records will contain the value 'undefined'. Records may be referenced inside any type expression in their defining module using the notation: "#rec{}". Such references may optionally (re)define the types of one or more fields, like so: "#rec{a :: atom(), c :: binary()}".

- The list types mentioned above all refer to "proper" lists, i.e. lists that can be represented either as the empty list or as the cons of a head element and a (proper) list tail. Erlang, however, allows the tail of a cons cell to be a non-list value. Lists constructed this way (called "improper" lists) are of little practical use, therefore we will ignore them in the rest of the thesis and consider only proper lists.

- To conserve space in Table 2.2, we have omitted some predefined types which are simply aliases for other types.

Term Group	Related Types	Represented Terms
Integers	*<Int>*	only a specific integer, *<Int>* (singleton type)
	<Lo>..<Hi>	integers between *<Lo>* and *<Hi>*
	integer()	all integers
	non_neg_integer()	non-negative integers
	pos_integer()	positive integers
	neg_integer()	negative integers
Floats	float()	all floats
Atoms	*<Atom>*	only a specific atom, *<Atom>* (singleton type)
	atom()	all atoms
Binaries	binary()	all binaries
	«»	only the empty binary (singleton type)
	«_:*<Base>*»	binaries of length *<Base>* (in bytes)
Bitstrings	bitstring()	all bitstrings
	«»	only the empty bitstring (singleton type)
	«_:_*<Unit>*»	bitstrings of length $k\times$*<Unit>* (in bits)
	«_:*<B>*, _:_*<U>*»	bitstrings of length *<B>*$\times$*<U>* (in bits)
Pids	pid()	all pids
Ports	port()	all ports
References	reference()	all references
Funs	fun()	all functions
	fun((...) $\rightarrow$ *Type*)	functions of any arity returning *Type*
	fun(() $\rightarrow$ *Type*)	zero-arity functions returning *Type*
	fun((T_1,...,T_N) $\rightarrow$ R)	N-arity functions accepting arguments of types T_1,...,T_N and returning R
Tuples	tuple()	all tuples
	{}	only the zero-size tuple (singleton type)
	{$Type_1$,...,$Type_N$}	tuples of N elements, of types $Type_1$,...,$Type_N$
Lists	[]	only the empty list (singleton type)
	[*Type*]	lists with elements of type *Type*
	[*Type*,...]	non-empty lists with elements of type *Type*
—	any()	all Erlang terms
	none()	no terms (special type)
	T_1 | T_2 | ... | T_N	the union of all terms represented by T_1, T_2, ..., or T_N

Table 2.2: Built-in Erlang types

```
1  -record(rec, {a,
2                % no initialization or type declaration
3                % equivalent to a = 'undefined' :: any()
4                b = 12,
5                % initialization only
6                % equivalent to b = 12 :: any()
7                c :: integer(),
8                % type declaration only
9                % equivalent to c = 'undefined' :: 'undefined' | integer()
10               d = 3.1 :: float()}).
11               % both initialization and type declaration
```

Listing 2.1: Example of an Erlang record declaration

2.3.2 User-Defined Types

Using only the predefined types presented above, users can accurately describe most of the term patterns commonly encountered in Erlang applications. It was decided, however, to also support user-defined types, since they offer several benefits:

- Type expression reuse: a custom type can serve as an alias for a type expression that appears often
- Support for parametric types: the same structure specification can be used with many different element types
- Support for recursive types

Users can define their own, custom types using "-type" declarations (see Listings 2.2 and 2.3). The basic syntax of a custom type is an atom (referred to as the type's name) followed by closed parentheses. The definition of such a type must be provided in the 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 module (using an "-export_type" declaration). Such types can then be referenced by using their fully qualified type name (i.e. the type name prepended with '*module_name*:').

Custom types can be parametric, i.e. they can accept one or more type parameters as arguments. The declaration of a parametric type is syntactically similar to that of a function, with type parameters denoted using type variables. Type variables are syntactically equivalent to Erlang variables (i.e. they start with an uppercase letter or underscore and may appear in the RHS of the type declaration, anywhere a type would be expected. Note that a user-defined type cannot override a predefined type of the same name and arity.

2.3.3 Function Specifications

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

$$\textbf{-spec}\ Function\,(ArgType_1, \ldots, ArgType_N) \rightarrow RetType.$$

```
1  -module(a).
2
3  -type foo() :: [integer() | atom()].
4  -type bar() :: {foo(),b:boo()}.
5  % foo() is local, it needn't be exported, no module identifier is used
6  % boo() is remote, it must be exported, the module name must be used
7
8  % recursive type
9  -type baz() :: 'done' | {'one_more',baz()}.
10
11 % mutually recursive types
12 -type do() :: 'do' | {'do',re()}.
13 -type re() :: 're' | {'re',mi()}.
14 -type mi() :: 'mi' | {'mi',do()}.
```

Listing 2.2: Examples of type declarations, module 'a'

```
1  -module(b).
2  -export_type([boo/0]).
3
4  -record(rec, {f :: integer(), g :: atom()}).
5  % exported type
6  -type boo() :: integer() | #rec{}.
7
8  % parametric, recursive type
9  -type tree(T) :: 'leaf' | {'single',T,tree(T)} | {'node',T,tree(T),tree(T)}.
```

Listing 2.3: Examples of type declarations, module 'b'

where $ArgType_1$, ..., $ArgType_N$ and $RetType$ are type expressions, which may contain predefined types, record expressions and references to user-defined types, local or remote. Users can, for documentation purposes, also give names to one or more of the arguments:

$$\textbf{-spec}\ Function\,(ArgName_1 :: ArgType_1, \ldots, ArgName_N :: ArgType_N) \rightarrow RetType.$$

Erlang functions are often designed to operate on different data types in an overloaded fashion. In order to accurately capture this behaviour, contracts are allowed to be overloaded as well. Overloaded contracts are specified as a sequence of simple contracts (referred to as "contract clauses") separated by semicolons:

$$\begin{aligned}\textbf{-spec}\ Function\,(ArgType_{a1}, \ldots, ArgType_{aN}) &\rightarrow RetType_a\\ ;\quad (ArgType_{b1}, \ldots, ArgType_{bN}) &\rightarrow RetType_b.\end{aligned}$$

Currently, the domain types of contract clauses are not allowed to overlap, as in this example:

$$\begin{aligned}\textbf{-spec}\ \mathrm{wrong}\,(\mathrm{pos\_integer}()) &\rightarrow \mathrm{pos\_integer}()\\ ;\quad (\mathrm{integer}()) &\rightarrow \mathrm{integer}().\end{aligned}$$

Another feature of the contract language is support for parametric polymorphism. Type variables can be used in contracts to specify relations among the input and output arguments of a function, as in this specification of the 'filter' function for lists:

$$\textbf{-spec}\ \mathrm{filter}\,(\mathrm{fun}\,((T) \rightarrow \mathrm{boolean}())\,, [T]) \rightarrow [T]\,.$$

By default, type variables are universally quantified, i.e. they count as the type 'any()'. Users can constrain the types that a variable is allowed to represent by adding a guard-like subtype constraint to the contract, thus achieving bounded quantification:

$$\textbf{-spec}\ \mathrm{foo}\,(X, Y) \rightarrow \{X, Y\}\ \textbf{when}\ X :: \mathrm{atom}(),\ Y :: \mathrm{float}().$$

These type variable constraints can also be combined with contract overloading. The scope of a subtype constraint is the contract clause after which it appears:

$$\begin{aligned}\textbf{-spec}\ \mathrm{tag\_number}\,(T) &\rightarrow \{\text{'int'}, T\}\ \textbf{when}\ T :: \mathrm{integer}()\\ ;\quad (T) &\rightarrow \{\text{'float'}, T\}\ \textbf{when}\ T :: \mathrm{float}().\end{aligned}$$

Some functions in Erlang are not meant to return (e.g. they define servers or they are used to throw exceptions). Such functions should be specified as returning the special 'no_return()' type[2].

[2] Actually, 'no_return()' is an alias for the type 'none()'.

2.3.4 Opaque Types

The Erlang type system provides a special kind of type declaration for opaque data types [67], i.e. data types whose representation is not supposed to be visible in any way outside of their defining module. Only functions in the defining module of an opaque data type are allowed to depend on its term structure. In contrast, external code should never:

- pattern match on the structure of an opaque data type
- compare an opaque term for equality or inequality against another term, opaque or not (Even two opaque terms of the same kind cannot be compared for equality. A special 'equals' function should have been provided for such an operation by the opaque type's defining module.)
- inspect the type of an opaque term using type test BIFs[3] (e.g. 'is_atom/1', 'is_list/1', 'is_tuple/1')
- apply an opaque term to a BIF which works only for some subset of all terms (e.g. 'atom_to_list/1', 'length/1', 'tuple_size/1', 'element/2')

Dialyzer will detect such erroneous uses of an opaque type and report them to the user.

Opaque types are declared using "-opaque" declarations, which are syntactically identical to normal "-type" declarations. Opaque types don't make much sense as module-local, since module-local types are, by definition, inaccessible to other modules. Therefore, opaque types should always be exported.

Opaque types are mainly useful for representing purely functional abstract data types (ADTs) with hidden internal representations. Programmers should regard instances of an ADT as black boxes and handle them only through API functions exported from the ADT's defining module. This way, they can be certain that the ADT's invariants will always hold and that their code is safe from future changes to the ADT's representation. In Erlang, however, the internal representation of a purely functional ADT is simply a term, which could easily be manipulated in ways that break this convention. Declaring an ADT as opaque helps detect such illegal uses and enforces the idea of an ADT. For this reason, most of the ADTs in Erlang's standard library are nowadays declared as opaque types.

Listing 2.4 illustrates how an ADT implementation module might look like. The 'stack' module presented in this listing contains a list-based implementation of a simple stack data type. Apart from the usual operations of pushing and popping, this particular implementation supports $O(1)$ calculation of a stack's size.

[3] BIF = Built-In Function, a function that is considered part of the Erlang language. BIFs are usually implemented in C.

```
-module(stack).
-export([is_empty/1, size/1, new/0, push/2, pop/1, safe_pop/1]).
-export_type([stack/1]).

-opaque stack(T) :: {non_neg_integer(),[T]}.
% The internal representation of a stack consists of an element count and the
% actual elements of the stack in a list.

-spec is_empty(stack(_T)) -> boolean().
is_empty({0, []}) ->
    true;
is_empty({_N, [_Top|_Rest]}) ->
    false.

-spec size(stack(_T)) -> non_neg_integer().
size({N, _Elems}) ->
    N.

-spec new() -> stack(_T).
new() ->
    {0, []}.

-spec push(T, stack(T)) -> stack(T).
push(X, {N,Elems}) ->
    {N+1, [X|Elems]}.

-spec pop(stack(T)) -> {T,stack(T)}.
pop({0, []}) ->
    throw(stack_empty);
pop({N, [Top|Rest]}) when N > 0 ->
    {Top, {N-1,Rest}}.

-spec safe_pop(stack(T)) -> {'ok',T,stack(T)} | 'error'.
safe_pop({0, []}) ->
    error;
safe_pop({N, [Top|Rest]}) when N > 0 ->
    {ok, Top, {N-1,Rest}}.
```

Listing 2.4: ADT example: simple implementation of a stack

Chapter 3

Property-Based Testing and PropEr

3.1 Software Testing

3.1.1 Unit Testing

Testing is the most direct way for programmers to get feedback on the quality of their code. Manual testing of programs, however, is a time-consuming and tedious process, and often amounts to a considerable portion of software production costs. Because of this, there is an ongoing effort within the software engineering field to automate as much of the testing process as possible. One of the methodologies that lends itself well to automation is Unit Testing [11], whereby each unit of a software system is considered a black box and tested separately from the others. In functional languages such as Erlang, the smallest unit of any application is a single function.

The process of testing a single function can be conceptually divided into three steps:

1. Acquire a valid input for the function.
2. Run the function for that input and capture its output.
3. Decide if the input-output pair conforms to the function's intended behaviour (also known as the "oracle problem" [54, 95]).

The second step of this workflow can nowadays be fully automated using unit testing frameworks like xUnit. Most conventional testing tools, however, leave it upon the tester to both provide the inputs and decide if the corresponding outputs are acceptable. xUnit frameworks, for example, require the user to provide a series of test inputs, along with their corresponding expected outputs. This approach essentially forces the tester to manually calculate the output for each test input, a process that tends to become impractical as input sizes grow, meaning that most such test cases will end up rather small. Moreover, no matter how big a test suite they write, users can never be certain that they have covered all aspects of a program's behaviour.

3.1.2 Property-Based Testing

Property-based testing 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 property-based testing 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 ensure that it behaves according to its specification. In fact, property-based testing tools allow full use of the host language for writing assertions, meaning that users can accurately describe a wide variety of logical properties, even ones that involve more than one functions. Users may also write their own test data generators, should they require greater control over the input generation process.

By following this methodology, a tester's manual tasks are reduced to correctly specifying the test function's input format and formulating a set of properties that accurately describe the program's intended behaviour. While not trivial, these tasks only require a fraction of the time it takes to write enough test cases to achieve the same goal. The resulting properties are also much more concise than a long series of input-output pairs, but, if used properly, can accomplish a much more thorough testing of the program. Moreover, properties can serve as a checkable partial specification of a program, one that is much more general than any set of instantiated test cases, and thus much better at documenting the behaviour of the program.

To illustrate the merits of property-based testing as opposed to traditional testing methodologies, consider the following example: We wish to test our implementation of a 'delete' function for lists (Listing 3.1), which should take a term 'X' and a list 'L' and return a list identical to 'L', but with all occurences of 'X' removed. We will test our code using EUnit (Erlang's main implementation of an xUnit framework [2]) and PropEr (our implementation of a property-based testing tool). EUnit will test the 'delete' function using the 9 test cases (input-expected output pairs) we have written. PropEr will test the function by checking whether 'X' is really absent from the returned list in every one of a serires of random inputs.

Listing 3.2 contains the results of testing. We can see that PropEr managed to find an error in our implementation, one that manifests for a class of inputs not exercised by our test cases, namely lists containing two (or more) instances of 'X'. Indeed, upon careful inspection it becomes apparent that our code only deletes the first occurence of 'X' from the input list.

3.1.3 The Need for Shrinking

Because test inputs are randomly generated, the part of a failing test case that is actually responsible for the failure can easily become lost inside a lot of irrelevant data. In our last example, only 2 of the 49 elements of the initial failure-inducing list actually contributed to the fault. Most programmers are already familiar with this problem, since they often have to devote a lot of time to manually extract the useful part from a long failure report, before moving on to pinpointing the fault. The property-based testing tool can aid the programmer in this regard by automatically simplifying a failing test case before presenting it to the user, through a process called "shrinking". In most cases, shrinking works in the

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

%%--------------------------------------------------------------------------------

delete_test_() ->
    [?_assert(delete(X,L1) =:= L2) || {X,L1,L2} <- test_cases()].

test_cases() ->
    [{1,[],[]},
     {1,[1],[]},
     {2,[1],[1]},
     {2,[2,1],[1]},
     {1,[1,2,3],[2,3]},
     {2,[1,2,3],[1,3]},
     {3,[1,2,3],[1,2]},
     {4,[1,2,3],[1,2,3]},
     {100,lists:seq(1,200),lists:seq(1,99) ++ lists:seq(101,200)}].

%%--------------------------------------------------------------------------------

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

Listing 3.1: Property-based testing vs. xUnit: code under test

```
1> eunit:test(mylists:delete_test_()).
  All 9 tests passed.
ok
2> proper:check(mylists:prop_delete()).
.................................................!
Failed, after 54 test(s).
{38,[-139,-29,-105,16,0,-11,26,-15,7,-23,-2,-82,-3,-24,-162,45,5,-36,-11,-22,-6,
3,401,38,101,-120,38,129,-11,-49,-30,17,41,26,-56,40,66,92,-4,23,39,-32,-3,27,
28,-34,-67,-195,-18]}
Shrinking ..............(14 time(s))
{0,[0,0]}
false
```

Listing 3.2: Property-based testing vs. xUnit: testing results

same way as a human tester, by consecutively removing parts of the failing input until no more can be removed without making the test succeed. In our last example, the shrinking process eventually returns a list of 2 elements, which are both necessary for triggering the error. This "minimal" test case, whose every part is essential to the failure, will serve as a good starting point for the debugging process. The shrinking process can often be fine-tuned through user-defined shrinking strategies.

3.1.4 Implementations

The first implementation of a property-based testing tool was QuickCheck [26], a combinator library written in Haskell, initially designed for testing pure functions. QuickCheck-like tools have since been developed for a variety of other languages (e.g. Clean [56]). Erlang versions include Quviq QuickCheck [8], a proprietary application written by the original authors of QuickCheck and marketed as a commercial product, and Triq [90], which was written by Kresten Krab Thorup and released as open-source software.

3.2 Our Implementation: PropEr

We have implemented a property-based testing tool for the Erlang language, called PropEr (PROPerty-based testing tool for ERlang). In its current form, PropEr is mainly useful for testing pure functions. We have strived to keep PropEr's interface compatible with other such tools for Erlang, but also bring it closer to Erlang's type system.

3.2.1 PropEr Workflow

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

1. Parse the property.
2. Randomly generate valid instances for each universally quantified variable in the property, based on its type.
3. If for this input the property code evaluates to 'true' without throwing an exception, repeat from 2. Else:
4. Report the failing test case to the user.
5. Try to shrink the test case to one that is simpler but fails the property in the same way.
6. If you cannot shrink the test case any further, report it to the user and exit. Else, repeat from 5.

The user can specify both a maximum number of tests to run (default is 100 tests) and a maximum number of shrinks (default is 500 shrinks). Failing test cases can be saved and re-applied to the same property, allowing the user to easily check if he has successfully fixed the problem.

3.2.2 Writing Properties

Properties are written using Erlang expressions, with the help of a few predefined macros[1]. The simplest properties that PropEr can test consist of a single boolean expression, which is expected to evaluate to 'true'. An expression that evaluates to 'false' or throws an exception during evaluation is considered a failing property. Thus, the test 'true' always succeeds, while the tests 'false' and 'throw(some_exc)' always fail. More complex (and useful) properties can be written by wrapping a boolean expression with one or more of the following wrappers:

?FORALL(*Xs*, *Xs_type*, *Prop*)
: The *Xs* field can contain either a single variable, a tuple of variables or a list of variables. The *Xs_type* field must then contain a single PropEr type (for more on the PropEr type system, see the next section), a tuple of types of the same size as the tuple of variables or a list of types of the same length as the list of variables, respectively. Tuples and lists can be arbitrarily nested, as long as *Xs* and *Xs_type* are compatible. 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 *Xs_type* is produced and each variable in *Xs* is replaced inside *Prop* by its corresponding instance.

?IMPLIES(*Precondition*, *Prop*)
: This wrapper only makes sense when in the scope of at least one ?FORALL. The *Precondition* field must contain a boolean expression, normally one that includes some of the universally quantified variables in scope. If the precondition evaluates to 'false' for the variable instances produced in the enclosing ?FORALL wrappers, the test input is considered invalid and thus is rejected, ending the testing round early.

?TIMEOUT(*Time_limit*, *Prop*)
: This wrapper adds an extra failure condition: *Prop* will be considered failing if it takes more than *Time_limit* milliseconds to return. The purpose of this wrapper is to test code that may hang if something goes wrong.

We also define a few helper wrappers for collecting statistics on the produced instances. It is a good idea to observe the distribution of generated test data, at least while developing the properties, to ensure that a sufficient number of non-trivial tests are run.

At this point we give a few examples of properties (Listing 3.3) and the output we get from testing them (Listing 3.4). PropEr uses intermediate output to report on the testing progress: each '.' signifies a successful test or shrink, each 'x' a rejected test and each '!' a failing test.

3.2.3 The PropEr Type System

The input domain of functions is specified through the use of a custom type system, modeled closely after the type system of the language itself. The basic operations that involve types in PropEr are:

[1] Macros in Erlang always begin with a '?'.

```
prop_correct() ->
    ?FORALL({X,Y}, {integer(),integer()}, X + Y =:= Y + X).

prop_incorrect() ->
    ?FORALL(X, integer(), X*X > X).

prop_nested() ->
    ?FORALL(X, pos_integer(),
            ?FORALL(Y, range(0,X-1), X*X > Y*Y)).

prop_good_precondition() ->
    ?FORALL({X,Y}, {integer(),integer()},
            ?IMPLIES(Y =/= 0, (X div Y) * Y + X rem Y =:= X)).

prop_bad_precondition() ->
    ?FORALL(X, neg_integer(), ?IMPLIES(X >= 0, X*X >= X)).
```

Listing 3.3: Examples of properties

```
1> proper:check(props:prop_correct()).
................................................................................
....................
OK, passed 100 test(s).
2> proper:check(props:prop_incorrect()).
...!
Failed, after 4 test(s).
1
Shrinking .(1 time(s))
0
3> proper:check(props:prop_nested()).
................................................................................
....................
OK, passed 100 test(s).
4> proper:check(props:prop_good_precondition()).
x.............................................................x.................
........x..............
OK, passed 100 test(s).
5> proper:check(props:prop_bad_precondition()).
xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
xxxxxxxxxxxxxxxxxxxx
Error: no valid test could be generated.
```

Listing 3.4: Output from example properties

Instance generation
Given a type, produce a valid instance of that type. At the start of every test run, PropEr produces only small instances of types in ?FORALLs, but gradually increases their complexity as more tests pass. To accomplish this, PropEr uses a global 'size' non-negative integer parameter, which influences the generation of every type. 'Size' starts at 1 and increases by 1 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. A more accurate definition is this: the maximum instance of 'size' S can never be smaller than the maximum instance of 'size' $S - 1$. The actual size of an instance is measured differently for each type. For example, the actual size of a list is the sum of sizes of all its elements + 1, that of a union is the size of the selected element, while that of a tree may be the number of its internal nodes.

Instance checking
Given a term and a type, decide if the term is a valid instance of the type. This operation comes in useful when shrinking instances of types in a ?FORALL that contains other ?FORALLs, since the nested types' structures may depend on the instance generated for the top-level ?FORALL (e.g. see the prop_nested property in Listing 3.3). In this case, we must ensure that the old instances of the nested types are also valid instances for the new nested types.

Shrinking
Given a term and its type, produce all[2] simpler instances of the type (this can be done lazily). In our case, "simpler" usually means "smaller". The top layer of PropEr will only accept a shrunk instance if it fails the property in the same way as the original instance.

Table 3.1 gives an overview of the basic type constructs of the PropEr type system, along with a description of all valid instances for each type. Table 3.2 explains what constitutes a "simpler" instance of each type, i.e. how PropEr will attempt to shrink an instance of this type. In addition to the information presented in these tables, we note the following on the basic PropEr types:

- Function types can also be specified using a list of argument types in place of the arity parameter. Currently, however, these are only used to calculate the arity of the function.
- We have omitted pids, ports and references from the PropEr type system, because these data types normally serve as identifiers to application-specific resources, therefore there is no point in producing random ones for testing.
- List syntax in the PropEr type system differs from list syntax in the Erlang type system, see Table 3.3.
- Because instances of a union or wunion shrink to the first type choice, users should write the simplest case first in the choices list.

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

Term Group	Related Types	Represented Terms
Integers	range(<*Lo*>,<*Hi*>)	integers between <*Lo*> and <*Hi*>
	integer()	all integers
	non_neg_integer()	non-negative integers
	pos_integer()	positive integers
	neg_integer()	negative integers
Floats	float()	all floats
	float(<*Lo*>,<*Hi*>)	floats between <*Lo*> and <*Hi*>
	non_neg_float()	non-negative floats
Atoms	atom()	all atoms (except some special ones used internally by PropEr)
Binaries	binary()	all binaries
	binary(<*Len*>)	binaries of length <*Len*> (in bytes)
Bitstrings	bitstring()	all bitstrings
	bitstring(<*Len*>)	bitstrings of length <*Len*> (in bits)
Pids	—	—
Ports	—	—
References	—	—
Funs	function(<*N*>,*Type*)	<*N*>-arity pure functions returning *Type*
Tuples	{T_1,T_2,...,T_N}	tuples of N elements, of types T_1,T_2,...,T_N
	loose_tuple(*Type*)	any-size tuples whose elements are all of type *Type*
Lists	list(*Type*)	lists with elements of type *Type*
	vector(<*Len*>,*Type*)	lists of <*Len*> elements, all of type *Type*
	[T_1,T_2,...,T_N]	lists of N elements, of types T_1,T_2,...,T_N
—	union([T_1,T_2,...,T_N])	instances of T_1, T_2, ..., or T_N, each of the types T_1,T_2,...,T_N is equally likely to be picked for generation
	wunion([{w_1,T_1},...])	equivalent to union([T_1,...,T_N]), but the type to generate is picked based on the weights w_1,...,w_N
	any()	all Erlang terms (for efficiency reasons, we never produce functions)
	<*Term*>	only the term <*Term*> (singleton type)

Table 3.1: Basic PropEr types

Term Group	Related Types	Instances shrink to
Integers	range(<*Lo*>,<*Hi*>)	integers closer to abs_min(<*Lo*>,<*Hi*>)
	integer()	integers closer to 0
	non_neg_integer()	integers closer to 0
	pos_integer()	integers closer to 1
	neg_integer()	integers closer to −1
Floats	float()	floats closer to 0.0
	float(<*Lo*>,<*Hi*>)	floats closer to abs_min(<*Lo*>,<*Hi*>)
	non_neg_float()	floats closer to 0.0
Atoms	atom()	atoms of smaller length (avoiding the special atoms PropEr uses internally)
Binaries	binary()	shorter binaries with byte values closer to 0
	binary(<*Len*>)	same-length binaries with byte values closer to 0
Bitstrings	bitstring)	shorter bitstrings with bit values closer to 0
	bitstring(<*Len*>)	same-length bitstrings with bit values closer to 0
Pids	—	—
Ports	—	—
References	—	—
Funs	function(<*N*>,*Type*)	(doesn't shrink)
Tuples	{T_1,T_2,...,T_N}	same-size tuples with simpler elements
	loose_tuple(*Type*)	smaller tuples with simpler elements
Lists	list(*Type*)	shorter lists with simpler elements
	vector(<*Len*>,*Type*)	same-length lists with simpler elements
	[T_1,T_2,...,T_N]	same-length lists with simpler elements
—	union([T_1,T_2,...,T_N])	instances of types closer to the head of the list, simpler instances of the selected type
	wunion([{w_1,T_1},...])	instances of types closer to the head of the list, simpler instances of the selected type
	any()	(depends on the structure of the instance)
	<*Term*>	(doesn't shrink)

Table 3.2: Basic PropEr types' shrinking strategies

- Apart from these basic types, we also define a few aliases and QuickCheck compatibility types.

Type Expression	Type System	
	Erlang	PropEr
[integer()]	any-length list of integers	lists of 1 integer
[integer(),integer()]	(illegal)	lists of 2 integers

Table 3.3: List syntax differences

Programmers can also use the following macros to customize basic types:

?LET(*Parts*, *Parts_ type*, *In*)
In order to produce an instance of this type, PropEr generates a random instance of *Parts_ type*, matches it with *Parts* and replaces all variables inside *In* with their corresponding values from the generated instance. The *In* part is allowed to evaluate to a type, in which case an instance of the inner type is generated recursively – this allows for nested ?LETs. To be able to instance-check and shrink instances of ?LETs, we need to save the generated instance of *Parts_ type*, since that information may be partially lost when applied to *In* (consider, for example, this type: ?LET({X,Y}, {float(),float()}, X+Y)). To accomplish this, instances of ?LETs are saved in an intermediate form: {'$used',*Parts*,*Value*}. To shrink an instance of a ?LET, we first shrink each of the *Parts* and re-apply them to *In*, then when this is done we move on to shrinking the *Value*. When *In* evaluates to a type, we need to check that the old *Value* is a valid instance of *In* after the parts have been shrunk – we do this through instance-checking.

?SUCHTHAT(*X*, *X_ Type*, *Condition*)
This produces a specialization of *Type*, which only includes those terms that satisfy the constraint *Condition*, i.e. those terms for which the function 'fun(X) $\rightarrow$ *Condition* end' returns 'true'. Currently, PropEr performs no constraint satisfaction analysis – instead, it produces instances of *Type* randomly until it finds a valid one. This means constraints shouldn't be very strict, i.e. most instances of *Type* should satisfy the condition, else it will take a lot of tries to find a valid one. This would result in slower testing or even an error, if a constraint tries limit is reached. Users should also make sure that even small instances can satisfy the constraint, since PropEr will only try small instances at the start of testing. Any shrunk instance of a specialized type must also satisfy the constraint, else it is rejected.

?SHRINK(*Type*, *Alternatives*)
This adds one or more alternative types to *Type*. The *Alternatives* field should be a list of types to be used when shrinking an instance of this type: Instances of each of the *Alternatives* will first be tried in place of the original instance, with the types in the head of the list taking precedence. Therefore, types in *Alternatives* should be simpler than *Type*, with the simplest cases declared first.

3.2.4 Recursive Generators

If users need to work with recursive data types, they have to manually guide their generation process; it's the user's responsibility to provide the base case and to ensure that generation always terminates. To accomplish this, a user must handle the 'size' parameter manually: he has 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 $Size = 0$ and all other clauses should contain a fallback to that clause. Finally, users have to tell PropEr how to apply the value of 'size' to this generator, by using a ?SIZED macro.

The following macros and functions are mainly useful for writing recursive type declarations:

?SIZED(*Size*, *Generator*)
: Constructs a new type from a sized generator. To produce instances of this type, PropEr will apply the current value of 'size' to the function 'fun(*Size*) → *Generator* end'.

resize(*New_size*, *Type*)
: This declaration instructs PropEr to use *New_size* instead of the value of the 'size' parameter to produce instances of *Type*.

?LETSHRINK(*Parts*, *Parts_type*, *In*)
: This construct is equivalent to a simple ?LET, but *Parts* must necessarily be a list of variables. Also, when shrinking an instance of this type, 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 make the shrinking process much more effective, therefore users should always use ?LETSHRINKs when combining recursively generated instances.

?LAZY(*Type*)
: This construct delays the generation of *Type*. Users should wrap each recursive choice in the non-zero-size clause of a recursive generator with a ?LAZY macro, so as to achieve linear generation time.

For an example of a recursive type declaration see Listing 3.5, where we provide a generator for the 'tree(T)' type declared in Listing 2.3.

3.2.5 Symbolic Instances

As explained in Section 2.3.4, instances of ADTs should always be constructed through successive calls to public API functions exported from their defining module. When writing a (recursive) generator for an ADT, however, simply limiting oneself to API calls is not sufficient. An instance produced by such a generator will essentially be a term describing the internal representation of the ADT instance, which the user can't reason about. It is much more practical to represent a randomly produced instance of an ADT as the series of API calls that will be used to construct it [7]. Working with such, so-called, "symbolic" instances has several benefits:

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

tree(0,_T) ->
    leaf;
tree(Size,T) ->
    wunion([
        {1, ?LAZY(tree(0,T))},
        {5, ?LAZY(?LETSHRINK([SubTree],
                             [tree(Size - 1,T)],
                             {single,T,SubTree}))},
        {5, ?LAZY(?LETSHRINK([L,R],
                             [tree(Size div 2,T),tree(Size div 2,T)],
                             {node,T,L,R}))}
    ]).
```

Listing 3.5: Example of a PropEr recursive type declaration

- Failing test cases are easier to read and understand.
- Failing test cases are easier to shrink.
- It is especially useful when testing the data type itself: Certain implementation errors may depend on some particular selection and ordering of API calls, thus it is important to cover the entire ADT construction API.

PropEr supports the symbolic representation of datatypes, using the following syntax: {call, *Module*, *Function*, *Arguments*}. A term like that represents a call to the API function *Module*:*Function* with arguments *Arguments*. Each of the arguments may be a symbolic call itself or contain other symbolic calls in lists or tuples of arbitrary depth. The 'eval/1' function can be used to evaluate a symbolic instance, i.e. calculate the concrete term it represents. The user normally needs to evaluate symbolic instances manually inside the property-testing code, unless she uses '$call' tuples, which are evaluated automatically before being applied to a property.

Listing 3.6 presents a generator for the stack data type (see Listing 2.4), along with a sample property that uses it.

Note that the above generator may produce symbolic instances like this one: {'$call', erlang, element, [2, {'$call', stack, pop, [{'$call', stack, new, []}]}]}, which will raise an exception when evaluated. Such symbolic instances are ill-defined, that is, they don't correspond to a real value. We can apply a 'well_defined' attribute to an ADT generator to constrain it to only well-defined symbolic instances. This works by checking each symbolic instance returned by the generator to see if it can be evaluated without throwing an exception. If it cannot, the generator is called again, until it produces a valid symbolic value. This strategy obviously assumes that the majority of generated symbolic values are well-defined.

```
stack(T) ->
    ?SIZED(Size, stack(Size,T)).

stack(0,_T) ->
    {'$call',stack,new,[]};
stack(Size,T) ->
    wunion([
        {1, ?LAZY(stack(0,T))},
        {5, ?LAZY(?LETSHRINK([S],
                             [stack(Size - 1,T)],
                             {'$call',stack,push,[T,S]}))},
        {2, ?LAZY(?LETSHRINK([S],
                             [stack(Size - 1,T)],
                             {'$call',erlang,element,
                              [2,{'$call',stack,pop,[S]}]}))},
        {2, ?LAZY(?LETSHRINK([S],
                             [stack(Size - 1,T)],
                             {'$call',erlang,element,
                              [3,{'$call',stack,safe_pop,[S]}]}))}
    ]).

prop_push_pop() ->
    ?FORALL({X,S}, {integer(),stack(integer())},
            begin
                {Y,_} = pop(push(X,S)),
                X =:= Y
            end).
```

Listing 3.6: Stack generator and sample property

Chapter 4

Utilizing Types in Testing

4.1 Motivation

As illustrated in the previous chapter, property-based testing tools can help to significantly speed up the testing process. Utilizing such tools efficiently, however, presents a new set of challenges for the programmer.

One of the first things a programmer will realize when working with such tools is that finding the right properties to test is often a non-trivial matter. Consider, for example, the situation where one wants to test his solution to an optimization problem. The straight-forward course of action would be to write a property that tests whether, for every one of a series of random inputs, the program under test produces the optimal solution. Checking that a solution is optimal, however, may involve comparing it to every other possible solution, a process that quickly becomes impractical as input size grows, in many cases (e.g. NP-Complete optimization problems, such as the TSP) exhibiting exponential complexity. Even for less extreme cases, it is ofter hard to translate one's abstract understanding of a program's expected behaviour into concrete properties. Solutions to this problem are inherently program-specific, therefore we can do little to automate the process.

A secondary inconvenience for new users of a property-based testing tool such as PropEr is the need to become familiar with a new type declaration language for writing term generators. To make this process as painless as possible, PropEr's type system has been modeled closely after Erlang's type system. Still, programmers have to write generators separately from the functions under test, meaning that changes to a function's interface will often need to be replicated in one or more generators. This cannot be avoided in cases where the programmer needs full control over the random generation process, e.g. if she wishes to fine-tune the size frequencies of produced terms. Often, however, programmers will end up writing generators that are almost equivalent to the types in a function's spec. Such use cases would benefit from a testing tool that can work with type information directly, therefore eliminating the need for the presence of redundant generators. A tool like that would be especially useful when working with recursive datatypes, since generators for such types require manual specification of both the size distribution and shrinking behaviour of produced instances.

4.2 Converting Types to Generators

The first component we need to implement is a type-to-generator converter.

4.2.1 Simple Types

Converting built-in Erlang types to PropEr types is a straight-forward process: we just follow the conversion formula outlined in Table 4.1[1]. Note that some Erlang types do not have an equivalent in the PropEr type system, most notably types for pids, ports and references, i.e. datatypes which PropEr cannot generate, for reasons explained in Section 3.2.3. PropEr also lacks types for describing bitstrings with a unit size greater than 1 and functions of unspecified arity; these types, however, are not commonly encountered in function specifications.

User-defined types and records are also relatively easy to handle, so long as they are not recursive. Converting a user-defined type or record is simply a matter of recursing into its type structure, except for two special cases, which require some extra bookkeeping: First, converting a record expression with field type updates involves an extra step of applying those updates to the record's original definition. Second, we cannot recurse into the body of a parametric type before applying the actual parameters to it. In this case, however, working on a purely syntactic level is not enough. We cannot simply substitute each variable in the type definition with its corresponding value, since the arguments of a type reference are considered to be in the same scope as the reference, not the definition. Consider, e.g., modules 'a' and 'b' from Listings 4.1 and 4.2: In the process of converting the type reference 'a:foo()' we would have to convert the type expression 'b:bar(boo())'. Simply replacing 'X' with'boo()' in the RHS of bar's declaration would produce the type expression '{baz(),boo()}', which is illegal in the context of either module. Instead, we pre-process each of the actual parameters, match the resulting PropEr types to variable names, then use this mapping while recursing to lazily substitute variables as we find them.

```
1 -module(a).
2 -export_type([foo/0]).
3
4 -type foo() :: b:bar(boo()).
5 -type boo() :: atom().
```

Listing 4.1: Parameter substitution example, module 'a'

4.2.2 Recursive Types

PropEr can also handle self-recursive and mutually recursive types. We require, however, that such types are written in a way that makes their base case clear. In particular, PropEr currently only accepts recursive types whose top level is either a maybe empty list (then the base case is the empty list) or a union that contains at least one choice which doesn't

[1] In this table, *Type′* stands for the PropEr equivalent of the Erlang type *Type*. Also, 'non_empty(*Type*)' is shorthand for '?SUCHTHAT(X, *Type*, X =/= [])'.

Term Group	Erlang Types	PropEr Types
Integers	*<Int>*	*<Int>*
	<Lo>..<Hi>	range(*<Lo>*,*<Hi>*)
	integer()	integer()
	non_neg_integer()	non_neg_integer()
	pos_integer()	pos_integer()
	neg_integer()	neg_integer()
Floats	float()	float()
Atoms	*<Atom>*	*<Atom>*
	atom()	atom()
Binaries	binary()	binary()
	«»	«»
	«_:*<Base>*»	binary(*<Base>*)
Bitstrings	bitstring()	bitstring()
	«»	«»
	«_:_*1»	bitstring()
	«_:_*<Unit>*»	—
	«_:*<Base>*, _:_*1»	bitstring(*<Base>*)
	«_:*<Base>*, _:_*<Unit>*»	—
Pids	pid()	—
Ports	port()	—
References	reference()	—
Funs	fun()	—
	fun((...) → *Type*)	—
	fun(() → *Type*)	function(0,$Type'$)
	fun((T_1,...,T_N) → *Type*)	function(*<N>*,$Type'$)
Tuples	tuple()	loose_tuple(any())
	{}	{}
	{$Type_1$,...,$Type_N$}	{$Type'_1$,...,$Type'_N$}
Lists	[]	[]
	[*Type*]	list($Type'$)
	[*Type*,...]	non_empty(list($Type'$))
—	any()	any()
	none()	—
	$Type_1$ | ... | $Type_N$	union([$Type'_1$,...,$Type'_N$])

Table 4.1: Built-in Erlang types to PropEr types

```
1 -module(b).
2 -export_type([bar/1]).
3
4 -type bar(X) :: {baz(),X}.
5 -type baz() :: float().
```

Listing 4.2: Parameter substitution example, module 'b'

directly reference the type (then the base case is formed as a union of all such choices). In this section we give a high-level description of PropEr's recursive type handling subsystem.

Detecting Recursion

The first issue we need to solve is detecting when the type we are processing is recursive. We achieve this by maintaining a stack during our recursion, where we record each type we recurse into. If at some point we come across a reference to a type that's present in the stack, we know we are dealing with a recursive type. A recursion path may span multiple modules, therefore each type reference stored in the stack must record the type's module. If a type is parametric, we also need to store the parameter values of each reference, so that we can differentiate between them. Records may be recursive as well, so we store them on the stack too (we use a flag to distinguish between user-defined types and records in the stack). Instances of the same record that contain different field type updates essentially represent different types, therefore we need to distinguish them. To achieve this, we have to store the field type updates of record references in the stack. Type parameters and record field type updates are stored in their processed forms, as PropEr types.

Handling Recursion

When we detect a recursive call, we notify the parent type by returning a rec_fun, i.e. a function that takes a list of recursive generators and arranges them according to the type structure. A rec_fun also accepts a second, integer valued argument, which controls the size parameters passed to the generators. A rec_fun that corresponds to a simple recursive call should accept a single generator (the one for the recursive type) and call it directly with the size parameter passed to it. The code for a derived type (list, tuple, fun or union) will collect all rec_funs returned to it by inner types and place them as arguments to its type's generator. The result will in turn be propagated upwards in a rec_fun, which should also somehow distribute its size parameter to all the combined generators. The conversion code also needs a way to keep track of the types of each of a rec_fun's expected generators. Consequently, each rec_fun is accompanied by a list of type references (in the same format as the elements of the recursion stack), which correspond to the generators in the rec_fun's first argument; we call this the list of rec_args. When a rec_fun that expects at least one generator for a specific type eventually reaches that type's conversion code, we essentially instruct it to use itself recursively, through the use of the Y combinator.

Top Level Behaviour

As mentioned above, not every construct is allowed at the top level of a recursive type; we use the recursion stack to enforce this. The recursion-handling code for every type construct starts with a top level check: it compares each of the rec_args returned to it with the top of the recursion stack. For this to work correctly, we need to record to the stack every entry to a compound type, by pushing special values that don't correspond to user-defined types. The code for most types will return an error if it detects that it's on top level, except for unions and lists, which will enter base case preparation mode instead. In this mode, the code for lists will set up the empty list as the base case and lazify its

element type. The code for unions will partition its choices into three categories: non-recursive, non-self-recursive (doesn't contain references to the type itself, but to types that are mutually recursive with it) and self-recursive. The choices of the first two categories will form the union of the 0-size base case, while the choices in the second and third categories are combined with a base case fallback in a weighted union, to form the recursive clause of the generator. Every recursive choice in this clause is then lazified, to speed up recursive generation. Then, the code for the type itself only needs to apply the Y combinator and wrap the produced recursive generator in a ?SIZED macro.

Specifying Shrinking Behaviour

A rec_fun normally expects to receive a list of generators, but if a generator will surely be called exactly once, we could just pre-produce that single sub-instance and pass it to the rec_fun in place of the generator. Thus, if the union-handling code at the top level of a type knows that the rec_fun for a particular recursive choice will call some of the type's generators passed to it exactly once, it can instruct the code for that choice to pre-produce the sub-instances of the type, then pass them to the rec_fun via a ?LETHSRINK, thus greatly enhancing the shrinking behaviour of produced instances. The only constructs that accommodate such use are simple recursive calls and tuples. Lists of references to a recursive type (optionally wrapped in tuples with all other elements being non-recursive) can also be constructed from a pre-produced list of instances. To keep track of the instance-accepting arguments of rec_funs, each rec_arg also carries an 'accepts instance' flag, which takes the values 'true', 'false' and 'list'. If a construct (e.g. a fun) cannot work with instances, it resets all flags in the rec_args before returning.

Handling Size

A generator for a recursive data type should always be constructed in such a way that generation is guaranteed to terminate and the size of produced terms grows at a linear rate. The generators that PropEr derives from type specifications achieve this by distributing the value of size among recursive calls, i.e. they call the rec_funs returned by internal types with a fraction of their size argument. Specifically, tuples distribute their size evenly among the recursive elements, lists randomly distribute it among their elements, funs pass it unchanged to their return value generator and unions pass it unchanged to each of the recursive choice generators. Simply distributing the size, however, is not enough if some recursion paths only call the recursive generator once. To counter this, we additionally subtract 1 from the size on each recursive call (unless the size is already at 0). Finally, we increment the size from inside the ?SIZED macro, so as to balance out the first subtraction. When constructing the recursive clause of a generator, we assign a weight of 1 to the base case fallback, and a weight of size / num_rec_choices to all recursive choices (or 1, if the above expression evaluates to 0).

Examples

To illustrate this process, we provide a step-by-step analysis of the conversion of the expression 'tree(atom())' (the tree/1 type was first defined in Listing 2.3). See Figure 4.1 for a graphic representation of the conversion process.

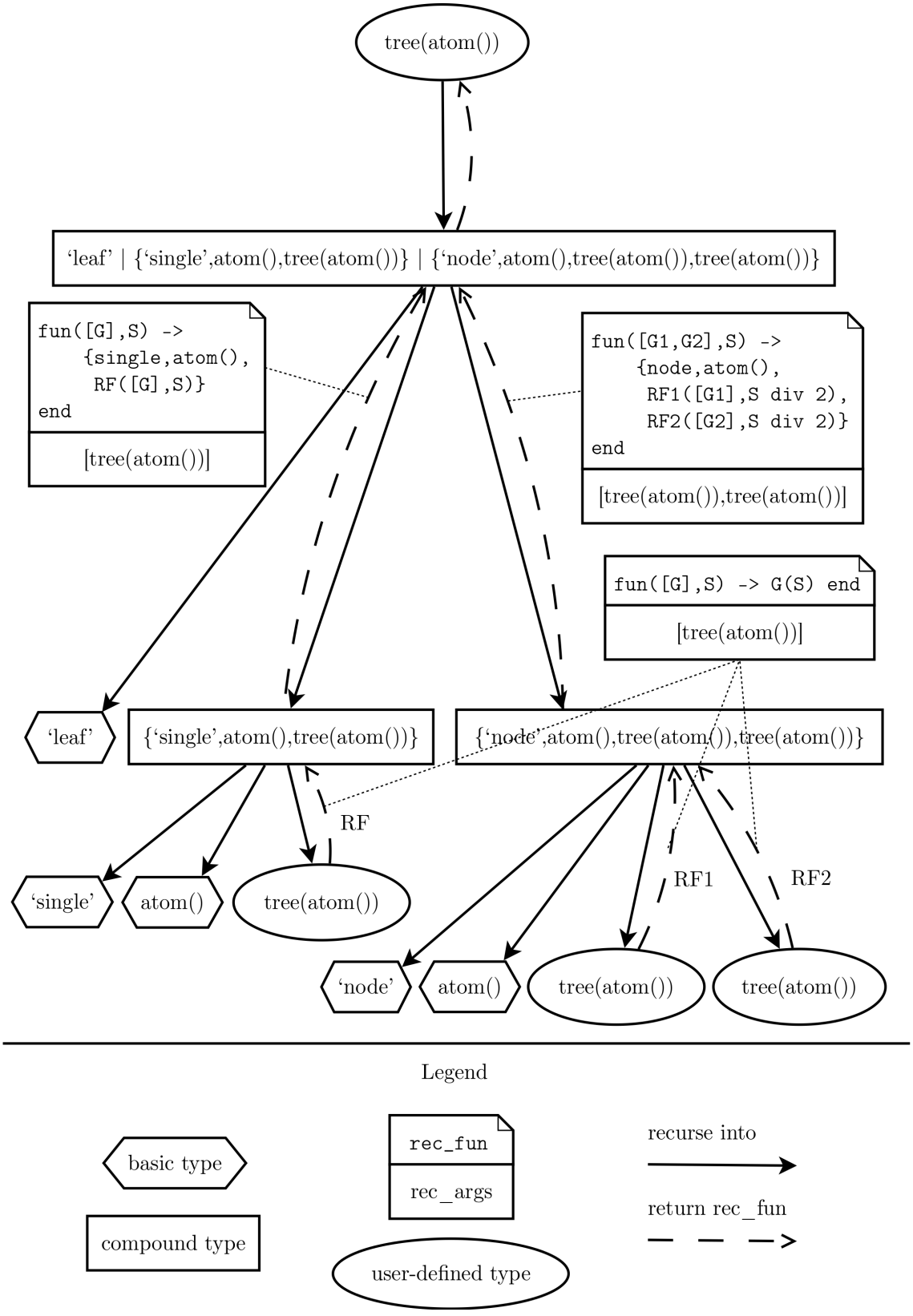

Figure 4.1: Conversion process for 'tree(atom())'

The type conversion system is asked to translate the type:

$$\text{tree(atom())}$$

First, we convert 'atom()' to its corresponding PropEr type, create a binding from the type variable 'T' to that type, push 'tree(atom())' to the recursion stack and recurse into the type structure of 'tree/1':

$$\text{`leaf'} \mid \{\text{`single'}, T, \text{tree}(T)\} \mid \{\text{`node'}, T, \text{tree}(T), \text{tree}(T)\}$$

The union-handling code recurses into the first choice (after pushing 'union' to the recursion stack):

$$\text{`leaf'}$$

We convert the singleton type 'leaf' to to its equivalent PropEr type and return it to the union handler. The union-handling code then recurses into the second choice (after pushing 'union' to the recursion stack):

$$\{\text{`single'}, T, \text{tree}(T)\}$$

The tuple-handling code pushes 'tuple' to the recursion stack and recurses into the first element:

$$\text{`single'}$$

We convert the singleton type 'single' to its equivalent PropEr type and return it to the tuple handler. The tuple-handling code then recurses into the second element (after pushing 'tuple' to the recursion stack):

$$T$$

The type variable 'T' is replaced by its value, the PropEr equivalent of 'atom()', and returned to the tuple handler. The tuple-handling code then recurses into the third element (again, after pushing 'tuple' to the recursion stack):

$$\text{tree}(T)$$

We first process the single parameter of the type reference, which evaluates to the PropEr equivalent of 'atom()'. We then search the recursion stack for an instance of 'tree(atom())': indeed, we find one three levels above, therefore we know that this one is a recursive call. We notify the tuple handler directly above by returning the following:

$$\begin{aligned} rec\_fun &= \textbf{fun}\ ([G], S) \rightarrow G(S)\ \textbf{end} \\ rec\_args &= [\text{tree(atom())}] \end{aligned}$$

Having received at least one rec_fun from its elements, the tuple-handling code first verifies that it's not at the top level of a recursive type, then combines all its elements into a new rec_fun-rec_args pair:

$$\begin{aligned} rec\_fun &= \textbf{fun}\ ([G], S) \rightarrow \{\text{single}, \text{atom}(), RF([G], S)\}\ \textbf{end} \\ rec\_args &= [\text{tree(atom())}] \end{aligned}$$

where RF is the rec_fun returned by the tuple's third element. This pair is then returned to the union handler. The union-handling code then recurses into the third choice (after pushing 'union' to the recursion stack):

$$\{\text{`node'}, T, \text{tree}(T), \text{tree}(T)\}$$

Following a similar process as above, we find that the first and second elements are easily converted to PropEr types, while the third and fourth both return the same rec_fun-rec_args pair:

$$rec\_fun = \textbf{fun}\ ([G], S) \to G(S)\ \textbf{end}$$
$$rec\_args = [\text{tree(atom())}]$$

Having received at least one rec_fun from its elements, the tuple-handling code first verifies that it's not at the top level of a recursive type, then combines all its elements into a new rec_fun-rec_args pair:

$$\textbf{fun}\ ([G1, G2], S) \to \{\text{node}, \text{atom}(), RF1([G1], S \text{ div } 2), RF2([G2], S \text{ div } 2)\}\ \textbf{end}$$
$$[\text{tree(atom())}, \text{tree(atom())}]$$

where $RF1$ and $RF2$ represent the rec_funs returned by the tuple's third and fourth elements (notice that this rec_fun distributes its size among the two recursive calls). This pair is then returned to the union handler. Having received at least one rec_fun from its choices, the union-handling code first checks if it's at the top level of a recursive type. Indeed, we are directly below a reference to 'tree(atom())', therefore we enter base case preparation mode: Of the three choices in the union, the first is not recursive, while the other two are self-recursive. Thus, the base case for the type 'tree(atom())' is formed as the union of a single choice: 'leaf'. As for the other choices, all the generators passed to them will be called exactly once, therefore we set up the generation code to use the recursive generators early, in order to pre-produce all the corresponding sub-instances and pass those to the rec_funs instead (see Listing 4.3 for the resulting recursive generator). This recursive generator is then passed up to 'tree(atom())', which identifies itself among the rec_args, therefore uses the Y combinator to instruct the returned rec_fun to use itself recursively, then wraps the resulting generator in a ?SIZED macro to produce the final PropEr type.

At this point, we provide the automatically generated recursive generators for some example types (Listings 4.3, 4.4, 4.5 and 4.6). Note that such generators are never output to the user; in fact, PropEr doesn't generate any code when converting types. The examples are presented in source form to make them more accessible to the reader.

Limitations

PropEr currently requires that the base case of a recursive type is made explicit by the user. The reason for this is that PropEr flattens non-top level unions (combines all the choices into one rec_fun). We made this design choice for reasons of performance: the alternative would be to work with sets of rec_funs, taking the set product of all elements every time we need to combine recursive generators into a compound type.

It is common practice, however, to write recursive types with the base case union at top level, therefore this limitation will rarely be an issue. Even if an (otherwise valid) recursive type declaration can't be parsed, it is often trivial to rewrite it in a way that fixes the problem: all the user has to do is move the base case union to the top level, as in this example:

$$\textbf{-type}\ \text{a()} :: \{\text{'a'}, \text{'none'} \mid \text{a()}\}.$$
$$\Rightarrow \textbf{-type}\ \text{a()} :: \{\text{'a'}, \text{'none'}\} \mid \{\text{'a'}, \text{a()}\}.$$

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

%--------------------------------------------------------------------------------

% This generator corresponds to tree(atom()).
tree_atom() ->
    ?SIZED(Size, tree_atom(Size + 1)).
% We add 1 to (partially) balance out the -1 below.

tree_atom(Size) ->
    tree_atom_(erlang:max(0, Size - 1)).
% We subtract 1 so that paths like 'single' will terminate.

tree_atom_(0) ->
    union([
        leaf
    ]);
tree_atom_(Size) ->
    W = erlang:max(1, Size div 2), % Size / num_rec_choices
    wunion([
        {1, ?LAZY(tree_atom(0))}, % fallback to base case
        {W, ?LAZY(?LETSHRINK([A],
                             [tree_atom(Size)],
                             {single,atom(),A}))},
        {W, ?LAZY(?LETSHRINK([A,B],
                             [tree_atom(Size div 2),tree_atom(Size div 2)],
                             {node,atom(),A,B}))}
    ]).
```

Listing 4.3: Example: instance-accepting recursion path, compare this generator with the manually written generator of Listing 3.5

```
-type delayer() :: 'done' | fun(() -> delayer()).

%--------------------------------------------------------------------------------

delayer() ->
    ?SIZED(Size, delayer(Size + 1)).

delayer(Size) ->
    delayer_(erlang:max(0, Size - 1)).

delayer_(0) ->
    union([
        done
    ]);
delayer_(Size) ->
    W = erlang:max(1, Size),
    wunion([
        {1, ?LAZY(delayer(0))},
        {W, ?LAZY(function(0, delayer(Size)))}
    ]).
```

Listing 4.4: Example: non-instance-accepting recursion path

```
-type gen_tree(T) :: 'leaf' | {'node',T,[gen_tree(T),...]}.
% general tree: each node has 1 or more branches

%-------------------------------------------------------------------------------

% This generator corresponds to gen_tree(atom()).
gen_tree_atom() ->
    ?SIZED(Size, gen_tree_atom(Size + 1)).

gen_tree_atom(Size) ->
    gen_tree_atom_(erlang:max(0, Size - 1)).

gen_tree_atom_(0) ->
    union([
        leaf
    ]);
tree_atom_(Size) ->
    W = erlang:max(1, Size),
    wunion([
        {1, ?LAZY(gen_tree_atom(0))},
        {W, ?LAZY(?LETSHRINK(A,
                             non_empty(list(gen_tree_atom())),
                             {node,atom(),A}))},
    ]).
```

Listing 4.5: Example: list-instance-accepting recursion path

Another shortcoming of this translation strategy is that the distribution of generated instances cannot be fine-tuned. Users, however, can often improve the distribution simply by modifying their type declarations; redundant type declarations[2] are especially useful in this regard. For example, the type 'a' | 'b' | atom(), while semantically equivalent to atom(), will result in a specialized generator, which returns the atoms 'a' or 'b' 66.6% of the time. Users can also improve the shrinking behaviour of instances simply by writing the simplest cases first in unions.

4.3 Integration with PropEr Notation

Having implemented an Erlang-to-PropEr type converter, we need to provide a way for testers to write properties using Erlang types. We decided not to introduce a new wrapper, but to extend the ?FORALL wrapper to also accept Erlang type references. If some part of the type specification in a ?FORALL doesn't correspond to a PropEr type, but would constitute a legal Erlang type, we assume it represents that Erlang type and use the method illustrated in the previous section to convert it. We allow the following uses of Erlang types in a ?FORALL:

- Any local user-defined Erlang type may be used, as long as it's not shadowed by a local or imported function (or an auto-imported BIF) of the same name and arity.

[2]Note that redundant type declarations won't confuse Dialyzer.

```
-type expr() :: non_neg_integer() | {expr_binop(),expr(),expr()}
              | {'if',bcond(),expr(),expr()}.
-type bcond() :: 'true' | 'false' | {bcond_unop(),bcond()}
               | {bcond_binop(),bcond(),bcond()}
               | {expr_comp(),expr(),expr()}.
-type expr_binop() :: '+' | '-' | '*' | '/'.
-type bcond_unop() :: 'not'.
-type bcond_binop() :: 'and' | 'or'.
-type expr_comp() :: '=' | '<' | '>'.

%%--------------------------------------------------------------------------------

% The user has requested a generator for expr().

% ...

expr_(0) ->
    union([
        non_neg_integer()
    ]);
expr_(Size) ->
    W = erlang:max(1, Size div 2),
    wunion([
        {1, ?LAZY(expr(0))},
        {W, ?LAZY(?LETSHRINK([A,B],
                             [expr(Size div 2),expr(Size div 2)],
                             {expr_binop(),A,B}))},
        {W, ?LAZY(?LETSHRINK([A,B],
                             [expr(Size div 2),expr(Size div 2)],
                             {'if',bcond(),A,B}))}
    ]).

bcond_(0) ->
    union([
        true,
        false,
        {expr_comp(),expr(0),expr(0)}
    ]);
bcond_(Size) ->
    W = erlang:max(1, Size div 3),
    wunion([
        {1, ?LAZY(bcond(0))},
        {W, ?LAZY(?LETSHRINK([A],
                             [bcond(Size)],
                             {bcond_unop(),A}))},
        {W, ?LAZY(?LETSHRINK([A,B],
                             [bcond(Size div 2),bcond(Size div 2)],
                             {bcond_binop(),A,B}))},
        {W, ?LAZY({expr_comp(),expr(Size div 2),expr(Size div 2)})}
    ]).

% ...
```

Listing 4.6: Example: mutual recursion

- Any remote Erlang type may be used, as long as it's exported from its defining module and not shadowed by an exported function from that module, of the same name and arity.
- Types (of any kind) can be combined in tuples and lists (the constructs '[...]', '{...}' and '++' are all allowed).
- No other construct of Erlang's type system is allowed. This includes union syntax, binary type syntax, function type syntax and record syntax.
- The parameters of an Erlang type can only be other Erlang types.
- The parameters of a PropEr type can include both Erlang and PropEr types.
- In general, if an expression can be interpreted both as a PropEr and as an Erlang type, the former takes precedence. This may cause some confusion when list syntax is used (see Table 3.3).

The type system detection component that implements this behaviour essentially has a single responsibility: to find all the ?FORALLs in a module and decide for each call[3] in the type field of every ?FORALL whether it refers to a function (i.e., a PropEr type) or an Erlang type. Calls to PropEr types will be left unchanged, while references to Erlang types will be replaced by calls to the type conversion subsystem. While type conversion can be delayed until testing time, the procedure we just described must take place at compile time, or the testing module may fail to compile. To illustrate this, consider module 'a' from Listing 4.7. The compiler will refuse to compile this module, since it considers 'foo()' to be a call to a non-existent local function.

```
1 -module(a).
2 -export([prop_foo/0]).
3
4 -type foo() :: atom().
5
6 prop_foo() ->
7     ?FORALL(X, foo(), is_atom(X)).
```

Listing 4.7: Example of a module that won't compile without a parse transform

We implemented this step as a parse transform, i.e. a code transformation that works at an abstract code level and is injected into the compilation process just before the source program is checked for errors. Our parse transform works by recursing into any syntactic construct in search of ?FORALLs. When we find one, we recurse into its type field (but only into call arguments and the '[...]', '{...}' and '++' constructs) in search of calls. If we find a local call, we check whether a function of that name and arity is visible in the module's scope (i.e., one such function is declared in the module, or the user has imported one such function, or there is an automatically-imported BIF with that name and arity). If not, we assume it's a reference to a local type and replace it with a call to the type conversion code. This call will contain a reference to the module this ?FORALL was found in, plus the whole type expression (including the arguments) printed to a string (we can't use the abstract code directly, because it's in expression format; we need to re-parse it as a type). If we find a remote call, we proceed in a more strict manner: we extract the remote module's exported functions and types and only consider the call as a remote

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

type if the remote module does export such a type, but doesn't export a function of the same name and arity. The process of consulting remote modules during compilation goes against standard Erlang practice, but any other solution would be much more complicated and could potentially incur great runtime penalties.

4.4 Handling Opaque Datatypes

PropEr's type conversion subsystem, as described in the previous sections, makes no distinction between normal and opaque types. Opaque types, however, are most often used to describe ADTs with structure invariants. Producing an instance of such a data type using only the information in its type declaration (essentially, its internal representation) will most probably not be a valid ADT instance. Consider our simple ADT example of a list-based stack, provided in Listing 2.4: It is very unlikely that a randomly produced instance of '{N :: non_neg_integer(),L :: [T]}' will satisfy the stack's single invariant, that N is always equal to the length of list L. It is, therefore, necessary that opaque types receive special treatment.

The best way to work with ADTs, as explained in Section 3.2.5, is to use symbolic instances. The symbolic generator for an ADT is, at its core, a listing of all possible ways to produce an instance of the ADT using exported API functions. This information, however, should already be available in the API functions' specs. We have extended PropEr to consider exported opaque types as ADT types, i.e. to ignore their internal representation and instead produce symbolic instances. PropEr's symbolic generators make use of all exported functions from an ADT's defining module that return at least one instance of the ADT (according to their specs). In this section we give a high-level description of our method for creating a symbolic generator for an opaque type.

4.4.1 Identifying Useful API Functions

The first step is to identify which functions from the opaque type's defining module can be used to produce instances of the ADT. We do this by scanning the specs of all exported functions from that module, looking for a way to extract an ADT instance from the return type. When working with parametric ADTs, we only want generic instances, therefore we only look for opaque type references with universally quantified variables as parameters (not bound by any subtype constraint). Currently, we only search the first clause of a multi-clause spec, but this can easily be extended to try all clauses. Our search works on a syntactic level: we only recurse into the type constructs '[...]', '{...}' and '...|...' and ignore references to custom types and record expressions; those are fairly uncommon in the return type of API functions anyway. Table 4.2 gives a description of our search and extraction strategies.

An API function may return an ADT instance inside a tuple; therefore, we need a method for extracting the useful field from a tuple. A straight-forward solution would be to use the 'element/2' BIF, which takes a tuple and a position and extracts the element at the specified position in the tuple. We have chosen to extract the value through pattern matching instead, for reasons that will become apparent in our analysis of unions. For this purpose, we have defined a minimal pattern language: We only intend to pattern match against tuples, therefore every pattern is a tuple. Each field of a tuple pattern can

Type construct	Is it usable?	How to extract the ADT from an instance of this type
ADT reference	Yes	(no action required)
Tuple	Only if one of its fields is usable	Extract the usable field through matching, then recurse into it.
List	Only if its element type is usable	Take the head of the list and recurse into it.
Union	Only if one of the choices is usable and can be distinguished from all the others	Use the extraction method for the usable choice (it's distinguishable, therefore there should be no problem).
Anything else	No	(we can't)

Table 4.2: API functions' return type search strategy

take one of a few possible values, listed in Table 4.3. A pattern that we can match against an M-size tuple to extract its Nth element would be an M-size tuple with all elements equal to '0', except for the Nth element, which would be a '1'. Atoms can be used in patterns to test for the presence of an expected tag; this feature is of little use if a function always returns a specific tuple, but is quite useful for distinguishing between tagged results in a union. Our matching procedure will throw an exception if it cannot match a term with the provided pattern (e.g. if the term is not a tuple or it's a tuple of the wrong size or it's a tuple of the correct size but has the wrong tags).

Pattern Field	Matching behaviour
0	Matches any term and throws it away (equivalent to the underscore variable in Erlang patterns).
1	Matches any term and saves it as the return value (there must be exactly one such field in each pattern).
<*Atom*>	Matches only the atom <*Atom*>.

Table 4.3: Description of custom match specifications

Unions may also be present in the return type of an API function, in most cases between a 'success' value and a 'failure' value, both of which are either an atom or a tagged tuple (a common return format for API functions is: {'ok',...} | 'error'). In the case of unions it's not enough to just detect the choice that contains an ADT reference, we must also make sure that its structure is unique among all the choices of the union. If that is not the case, i.e. some other choice has a similar format with the selected choice, we cannot be certain if a term returned by the function indeed contains an ADT or not. See Table 4.4 for a description of the distinguishability criteria we apply. We only check the top-level type constructors of a union's choices to determine whether they are distinguishable, but this is sufficient for most uses.

To illustrate this process, consider the stack ADT: Of all the functions in its defining module, only 'new/0', 'push/2', 'pop/1' and 'safe_pop/1' contain a reference to the opaque type in the range of their specs. Functions 'new/0' and 'push/2' return a clean ADT instance, therefore require no extra processing. 'pop/1' always returns an untagged two-

element tuple with an ADT instance as its second element. Thus, we would need to match its return value against the pattern '{0,1}'. 'safe_pop/1' will either return an atom ('error') or a three-element tuple tagged 'ok', with an ADT instance as its third element. Of these possible return values, only the second contains an ADT instance, which we can extract by matching against the pattern '{'ok',0,1}'. Since the other return value is not a tuple, it can never match against any tuple pattern, therefore we can safely use the pattern '{'ok',0,1}' to match against any value returned by this function.

Selected choice	Is it distinguishable?
ADT reference	No (we cannot count on the structure of an opaque term, thus we can't distinguish it from other terms)
Tuple	Only if all other choices will definitely not match this choice's pattern (i.e., their instances are never tuples of the same size and with the same tags as the selected choice)
List	Only if all other choices will definitely raise an exception if passed to erlang:hd/1 (e.g., 'tuple()' will surely fail, as will '[]', but we can't be sure about 'term()')
Union	Only if the selected choice in the nested union is distinguishable from all other choices of both the nested and the outer union.

Table 4.4: Choice distinguishability criteria in unions

4.4.2 Constructing a Symbolic Generator

After we have collected all the API functions that we can use, along with a way to extract an ADT instance from each function's return value, we need to combine them into a symbolic generator. We observe at this point that the process by which one chooses which functions will constitute the base case of such a generator is not very different from the way we select the base case for a recursive type: We pick those functions that don't require another ADT instance to operate on, i.e. those functions that don't have a reference to the ADT's opaque type in their domain. This is similar to how a recursive type's base case is formed by combining all the choices that don't reference the type. Inspired by this observation, we attempt to find a way to reuse PropEr's recursive type handler in the creation of symbolic generators.

It is actually almost trivial to accurately express the format of a symbolic call to a 0-arity function using the Erlang type system. This type, for example:

$$\textbf{-type}\ \text{erlang\_now\_symb}() :: \{\text{'\$call'}, \text{erlang}, \text{now}, []\}$$

describes a symbolic call to the function erlang:now/0. This technique, however, cannot be extended to functions of arity 1+, because the Erlang type language doesn't have support for fixed-length lists. To remedy this, we have created our own fixed_list type, which PropEr recognizes as a built-in (users don't have access to this type; it only makes sense in the context of PropEr's type converter). The type conversion system handles this type exactly like a tuple, but wraps its elements in a 'fixed_list/1' PropEr type instead of a 'tuple/1'. Using this type, we can accurately describe any symbolic call.

Therefore, we replace each ADT's opaque declaration with a new (simple '-type') declaration which lists all the symbolic calls that can produce it as choices in a top level union[4]. The argument types for each function's symbolic call are simply copied from its spec and wrapped in a fixed_list type. If a function doesn't return an ADT instance directly, we have to wrap the base call with symbolic calls to all the functions needed to extract the actual instance. Listing 4.8 shows what the symbolic type generated for the stack ADT would look like in source form (in this listing, the delimiters '<' and '>' are used in place of our fixed_list type).

```
1  -type stack(T) :: {'$call',stack,new,<>}
2                  | {'$call',stack,push,<T,stack(T)>}
3                  | {'$call',proper_typeserver,match,
4                     <{0,1},{'$call',stack,pop,<stack(T)>}>}
5                  | {'$call',proper_typeserver,match,
6                     <{ok,0,1},{'$call',stack,safe_pop,<stack(T)>}>}.
7
8  %------------------------------------------------------------------------------
9
10 % Examples of valid instances for stack(atom()):
11 {'$call',stack,new,[]}
12 {'$call',stack,push,[foo,{'$call',stack,push,[bar,{'$call',stack,new,[]}]}]}
13 {'$call',proper_typeserver,match,
14  [{ok,0,1},{'$call',stack,safe_pop,
15             [{'$call',stack,push,[baz,{'$call',stack,new,[]}]}]}]}
```

Listing 4.8: Inferred type specification for symbolic stack instances, with examples

This process is a bit more complicated when the ADT in question is parametric. First of all, we can only work with generic instances of the ADT, therefore every reference to the opaque type should have universally quantified variables as parameters[5]. Separate instances of the same ADT are allowed to have different variables in the same argument position, but this cannot introduce an implicit binding on the ADT's parameters, as in this spec:

$$\textbf{-spec}\ \text{weird}(\text{dict}(T, S), \text{dict}(S, T)) \to \text{boolean}().$$

Similarly, an ADT shouldn't contain duplicate parameters. Any spec that violates these constraints is automatically rejected. Additionally, we have to rewrite the specs' domains so that they make sense in the RHS of the ADT's symbolic type declaration: We update each spec's variables in such a way that all ADT instances in all specs have the same parameters as the LHS of the original opaque declaration. We also apply all subtype constraints to each spec (by replacing each bound variable with its supertype) and replace any remaining (unbound) variables with 'any()'.

At this point, we are done; the task of turning the symbolic type declaration into a recursive generator is left to the type conversion system. The resulting generator will be wrapped with a 'well_defined/1' attribute (see Section 3.2.5), therefore every produced symbolic instance is guaranteed to evaluate successfully. Both illegal, exception-throwing uses of API functions (e.g. calling 'stack:pop/1' on an empty stack, which throws a 'stack_empty'

[4]If we could find no suitable API functions for some opaque type, we have no choice but to treat it like an ordinary type.

[5]If this requirement would result in singleton variables, as in the spec for stack:new/0, users should use variable names that begin with an underscore.

exception) and unexpected 'failure' returns (e.g. calling 'stack:safe_pop/1' on an empty stack, which returns 'error', while we were expecting '{ok,_,S}') are avoided. The produced symbolic calls will be in '$call' format, so they will be evaluated automatically.

4.5 Automatic Spec Testing

Having implemented a fairly robust type translation system, we are ready to turn our attention to the other component of the Erlang type system, function signatures. Since function specs are essentially a form of lightweight specification ("this function, if called with arguments of types $A_1, \ldots, A_N$, will return a value of type *Ret*"), it should be easy to convert them into testable properties. We have implemented a prototype for a system which builds on this idea to enable the automatic testing of functions, based solely on information extracted from their signatures. At its current form, our implementation tests 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.

4.5.1 Generating 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 type format and pass it to the random instance generator. Specs that contain variables, however, require an extra pre-processing step: every bound variable is replaced by its supertype and all remaining (unbound) variables are converted to 'any()'. Currently, we only test the first clause of multi-clause specs, but this can easily be extended to testing each clause separately.

4.5.2 Checking the Return Value

If the function under test returns normally for a random input, we should check that the returned value is in accordance with the function's spec. Essentially, we need to write a component for instance checking against Erlang types. The straight-forward way to achieve this is through synchronized recursion in the term's and the type's structures. Most of the type tests we perform while recursing are self-explanatory, 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 (local or remote) types are expanded using the corresponding type definitions. For opaque types, we obviously use their original declaration and not the symbolic one that PropEr created. Testing API functions, however, would be much more effective if, instead of just checking a returned ADT instance against its internal representation format, we also tested its invariants, utilizing a user-provided invariants checking function.

- If a user-defined type is parametric, we first replace every variable in its definition with the value of the corresponding actual parameter before recursing into its structure. In the case of parametric remote types, we have to annotate the parameters with their originating module before copying them into the definition, because they may not make sense in the context of the remote type's defining module.

- Recursing into any type structure requires that we consume some bit of the input term (e.g., to recurse into a list we have to consume the list structure containing the elements, to recurse into a tuple we have to consume the tuple structure encasing the field values), therefore the process is guaranteed to terminate (since Erlang terms are of finite size). The exception to this is unions and references to user-defined types (record expressions are expanded into tuples, therefore pose no risk). This, coupled with the type system's support for recursive types, has the potential to create an infinite loop in our instance-checker. The problem arises when some spec's range contains a reference to a (directly or indirectly) recursive type that contains at least one recursion path of just unions and type references, like in these examples:

  -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 types and stop the process early. We don't consider this to be a serious limitation of our tool, since such types are rare in practice and can easily be translated into an acceptable form.

- We cannot safely check either the argument types or the return type of funs. 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. Due to the lack of a standard way to specify exceptional function behaviour in the current form of the Erlang type system, we decided to be conservative and accept any thrown exception (plus 'badarg' errors, which are commonly used to signify an illegal input) as a normal return.

Chapter 5

Practical Evaluation

In this chapter, we summarize the results of our experiments on using PropEr as an automatic spec tester, and comment on them.

5.1 Context

PropEr's source code is well-documented, contains specs for all functions, and is essentially stateless, therefore it was trivial to adapt it so that it could test itself. Specifically, we have used *proper:check_spec/1* to test the main PropEr function, *proper:check/1*. Additionally, we have used PropEr to test the specs of various modules from Erlang's standard library. Not all modules could be tested: modules that implement servers, contain impure code or handle stateful resources (e.g. processes, files, databases) are out of the scope of PropEr's automatic spec tester at its current form. Also, we couldn't test modules that manipulate Erlang code in AST form, because the AST format is currently unspecified type-wise. Apart from these applications, most of the modules from the public Erlang code base either didn't contain specs, or were unfit for automatic spec testing (most often, their operation depended on some kind of state that couldn't be described using only specs).

5.2 Self-Testing

5.2.1 Results Summary

Through self-testing, we were able to uncover these errors in our implementation:

- ?TIMEOUT wrappers with failing internal properties would cause PropEr to crash. To understand why this happened, we need to know how ?TIMEOUT is implemented: When a ?TIMEOUT(*Time*,*Inner_Prop*) wrapper is reached, PropEr spawns a child process, which is then tasked with carrying out the testing of *Inner_Prop*. This child is expected to return the result (via a message) within *Time* milliseconds, or else a timeout error occurs. In our initial implementation, the spawned process would try to read some values stored in the process dictionary[1] of its parent, but

[1]Each Erlang process has a private key-value store, called the process dictionary, which can be accessed in an impure way.

only if the internal test failed (such a property was never exercised in our unit tests). To fix this error, we made the spawning process copy the contents of its process dictionary to every spawned child.

- Some unusual wrapper combinations (namely, ?FORALLs with branching internal properties, where each branch contained a different number of stats-collecting calls), that both the specs and the program documentation allowed, weren't handled properly.
- Most union- and wunion-related functions were underspecified: their specs allowed for an empty list of choices.

5.2.2 Conclusions

From our overall experience of using PropEr for self-testing purposes, we have arrived to the following conclusions:

- Erlang's type language is often adequate for accurately describing a function's input domain. However, users must be careful not to underspecify when writing specs that they wish to test automatically. Overapproximating a function signature is fine for the purposes of finding type errors, but may cause PropEr to test the function with inputs it's not supposed to handle.
- Function specifications 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.
- Users may need to alter the syntax of their type declarations to make them better suited to the role of term generator specifications, e.g. by adding redundant choices or writing the simplest case first in unions (see also Section 4.2.2).
- PropEr's special handling of opaque types is very practical for spec testing, but there should also be a way for users to exclude one or more API functions from symbolic generators (e.g. because these functions are expensive to run).
- A more powerful system for the outputting and shrinking of generated functions could be very useful for certain use cases.

5.3 Standard Library Testing

We used PropEr to test some modules from Erlang's standard library, as it was on the latest major release of the Erlang platform (R14B).

5.3.1 Results Summary

Most of the modules we tested contained functions whose input format could not be accurately represented using just the Erlang type language. Their specs would overapproximate their real domain, causing PropEr to generate illegal input values, which

most functions didn't handle gracefully. As an example, consider the *lists:zip/2* function, whose documentation states that it only accepts two lists of the same length. The closest a spec can get to specifying the actual behaviour of this function is this:

$$\textbf{-spec}\ \mathrm{zip}([A],[B]) \rightarrow [\{A,B\}].$$

The last spec also allows lists of different lenghts, which cause *lists:zip/2* to crash. Also, consider the functions inside the *calendar* module, which accept dates in the format:

$$\{Day :: 1..31,\ Month :: 1..12,\ Year :: \mathrm{non\_neg\_integer}()\}$$

This specification allows for some illegal date declarations, like {31,4,2000} (April only has 30 days), which will cause any date-handling function from the *calendar* module to crash.

The real errors we found were mostly errors in specs (we weren't really expecting to find errors in the code, since the standard library is one of the most well-tested applications in Erlang/OTP):

filename:join/1, gb_sets:intersection/1, ordsets:intersection/1:
These functions' specs incorrectly stated that they could also accept the empty list.

gb_sets:next/1, gb_trees:next/1:
These functions' specs incorrectly stated that they would accept any term, while they can really only work with an 'iterator()' data type, which actually has an undocumented representation, and therefore should have been declared as an opaque type.

lists:merge/1, lists:umerge/1:
These functions' specs read $([T]) \rightarrow [T]$ instead of the correct $([[T]]) \rightarrow [T]$.

orddict:filter/2:
This function's spec stated that the predicate used to filter the dictionary could return 'any()', whereas only 'boolean()' is actually acceptable.

5.3.2 Conclusions

After using PropEr to automatically test specs from modules of the standard library, we have the following observations to make:

- As mentioned above, the Erlang type annotation language is not always enough to accurately specify a function's input domain.
- Many kinds of functions could be tested much more efficiently if their specs contained exceptional behaviour information. For example, in the documentation page for the *queue* module it is noted that many of the module's functions throw an 'error:empty' exception if called with an empty queue. PropEr, however, currently has no way of knowing this, therefore considers the empty queue as a failure-inducing input for all of these functions.

- Deeply nested symbolic instances (of size 40 and over) tend to significantly slow down the testing process, because they consume a lot of memory and take long to evaluate.
- The ‘any()’ predefined type, while common is specs, is not well-suited to testing: It is very expensive (both in terms of processing time and in terms of consumed memory) to produce and to shrink random structured terms (producing such instances in large quantities often resulted in out-of-memory errors). Using a smaller subset of values (e.g. integers) is almost always just as effective. Therefore, we should either streamline the handling of ‘any()’, or stop using it altogether while spec testing.

Chapter 6

Related Work

In this chapter, we review a number of techniques related to automated, specification-based testing. See also the recent paper by Hierons et al. [45] for a more comprehensive survey.

6.1 Oracle Generation

A testing oracle, i.e. a mechanism for determining whether the program has passed or failed a test, can often be derived from software specifications in a fully automatic manner. In this section, we give a brief overview of the major oracle generation techniques found in the literature. See the paper by Baresi and Young [10] for a more comprehensive survey.

Contract-Based Specifications Specification languages belonging to this category include JML, Object-Z, Eiffel and others. These languages follow a "Design by Contract"™ approach: functions are specified using preconditions, postconditions, invariants and assertions. See the paper by Richardson et al. [86] for an overview of the theory behind automated testing techniques for such languages.

The most common form of oracle derived from contract-based specifications is the "passive" oracle, i.e. one that simply checks the behaviour of a program without reproducing it [20, 84, 70, 83, 52, 25]. Assertion, invariant and postcondition checks serve as a (partial) oracle, which can then reused by various testing tools (e.g. the JML-based testing tools JML-JUnit [23], Jartege [80], Korat [15], JET [24], JML-TT [14] and JMLAutoTest [97] all use the same passive oracle).

Another, less common, type of oracle that can be derived from contract-based specifications is the "active oracle", i.e. one that mimics the behaviour of the program under test. Active oracles are essentially alternative implementations: to test a program against an active oracle, a testing tool would execute the two implementations in parallel and verify that their behaviour is always identical. An active oracle may be produced either by generating executable code from the specification, or, as in the case of jmle [58], by executing the specification itself symbolically, with the help of a constraint solver.

Model-Based Specifications Reactive systems (i.e., systems that continuously interact with their environment through events) are typically specified using abstract models. State transition systems (FSMs, EFSMs, labeled transition systems, flow charts, Markov chains etc.) representing the possible configurations of the system under test are most often used as models [12, 19, 9, 34]. Executable paths through such a state-based specification can then be translated into test cases [92, 77, 88, 32]. It is also feasible to animate the specification (translate the specification into an implementation) [65, 74, 89, 75, 55, 69]. The resulting program can then be used as an active oracle, as a prototype implementation, or as a way to test the specification itself. See the papers by El-Far and Whittaker [35], Boberg [13] and Utting et al. [93] for overviews of model-based testing.

Operational Profile Extraction Some testing tools utilize reverse engineering techniques, such as symbolic execution [59] or runtime invariant detection [81], to extract an operational profile of the program directly from the source code. This profile functions like an (approximate) specification of the program and can be used in the same manner as a user-provided one.

Algebraic Specifications Algebraic specification languages describe software by making formal statements, called axioms, about relationships among instances of data types and the functions that operate on them. A testing tool can use such specifications to derive a test oracle [4, 38] (active oracles are relatively easy to produce), a test case generator [4] (inputs are essentially nested symbolic calls), or both [49]. See the thesis of P. Machado [66] for a more in-depth presentation of algebraic specification testing.

6.2 Test Data Generation

Another part of the testing process that can easily be automated is the creation of test inputs. In this section, we briefly introduce various families of test generation techniques proposed in the literature. See the papers by Edvardsson [33] and Rushby [87] for more comprehensive reviews.

Random Testing All of the testing tools mentioned in Chapter 3 (including PropEr), as well as many others (e.g. JCrasher [28], Jartege [80]) generate test inputs in a completely random manner. While this approach has been proven effective in practice, it suffers from a few inherent shortcomings: it can be almost impossible to randomly produce instances that satisfy a strict precondition, and random inputs are not guaranteed to achieve sufficient code coverage of the program under test. To counter these issues, random testing tools use a variety of approaches, such as allowing users to assign weights to constructors, specify a size distribution for the produced inputs, or even write their own, custom data generators.

Incremental Generation A popular approach among automated testing tools for object-oriented languages (e.g. RANDOOP [82], JET [24] and AutoTest [25]) is the incremental generation of objects. The tool maintains an object pool, which is initially seeded with a few small values (either provided by the user or produced randomly). To construct a new object, the tool will randomly choose a method from the class under test and call it with

arguments randomly selected from the object pool (only objects that satisfy the method's precondition can be used). Provided that the method returns normally and the resulting object doesn't violate any postcondition, the new object is inserted into the pool. In order to suppress redundancy among the successfully built call sequences, many of these tools employ some king of equivalence checking: newly constructed objects that are found to be isomorphic to some older input are discarded.

Exhaustive Testing Most programs are designed to accept an infinite number of possible inputs, therefore complete coverage of their input space is impossible. However, exhaustive testing can be achieved as long as we confine ourselves to inputs up to a certain size. This method is based on the "regularity hypothesis": we expect that, if the implementation works correctly for all tests up to size k, it will also work for all tests of size greater than k.

The Korat tool [15] follows this approach when generating instances of structured objects: the size of an input is considered to be the number of its internal nodes. The Symstra tool [96] also uses bounded exhaustive generation, but produces instances through a series of API calls, therefore considers the size of an instance to be the number of calls made to construct it. Both of these tools avoid the generation of isomorphic (equivalent) objects, so as to reduce the number of produced test inputs. The SITE tool [49] produces all combinations of ADT API calls up to a certain nesting depth.

To further reduce the number of produced tests, testing tools can utilize a pairwise argument generation technique [29]: Testers supply a number of values for each input argument and the tool, instead of trying all $Vals_1 \cdot Vals_2 \cdot \ldots \cdot Vals_n$ combinations, produces just enough input vectors to ensure that, for all $\binom{n}{2}$ pairs of arguments, all $Vals_i \cdot Vals_j$ combinations of values appear in at least one test.

Domain Testing Domain testing [48, 60] works by partitioning the input space (domain) of a program into classes of equivalence (sub-domains): all the inputs in a sub-domain (should) cause the program to execute in a similar way. These partitions are usually derived solely from the specification of the program, by parsing the input type declarations and considering them together with the preconditions (after transforming them to a suitable form, such as the disjunctive normal form). Domain information can also be extracted from the source code, e.g. through symbolic execution. Each partition should be covered by at least one test, with input values on and around the boundaries taking precedence over random values.

Domain testing works on the basis of two hypotheses: The first hypothesis states that the program behaves uniformly in each sub-domain. Therefore, if it works correctly (or fails) for some input in the sub-domain, then it will work correctly (or fail) for all inputs in that sub-domain. The second hypothesis states that faults tend to cluster around the boundaries of domain partitions.

Special Values Testing This strategy involves the testing of programs on input values that have special properties, because such values, experience indicates, are likely to trigger faults. This approach has little theoretical basis and requires manual tester intervention, therefore it is rarely considered by researchers on automated testing techniques.

Symbolic Execution and Constraint Solving The idea of using symbolic execution to derive test cases dates back from the 1970's [16, 27, 85], but this research area is still active [42, 72]. Testing tools that follow this approach do not execute the program at all during test case generation. Instead, they work on the source code itself, by selecting a code path and following it while recording the constraints that input variables will have to satisfy for a program execution to actually follow that path. Solutions to these constraints, which can be achieved through the use of Boolean solvers (e.g. SAT solvers) or by numerical analysis (e.g. Gaussian elimination), represent the test data.

The main limitations of such techniques stem from their use of symbolic execution: They cannot easily process programs that contain recursion, dynamic data structures, array indices which depend on input data, pointer variables and some loop structures. Also, the problem of solving arbitrary constraint systems is known to be intractable. Combining symbolic with actual execution [40] can help to ameliorate this situation.

Constraint solving techniques can also be applied on specification preconditions [21], in cases where they contain complex predicate expressions.

Model Checker-Assisted Generation Model checking normally involves processing the model of a software system exhaustively, to verify that some property holds for any implementation of that model. However, model checkers, due to their ability to produce counterexamples for (most) falsified properties, have been used extensively to support testing [94, 39, 44, 46, 18, 17, 36, 51]. Most uses of model checkers for testing purposes follow this pattern:

1. A model of the program under test is provided by the user or extracted from a suitable specification.
2. A set of test purposes (e.g. a set of code paths to cover) is formulated (manually or automatically) as a set of temporal properties in an appropriate language (e.g. LTL).
3. The properties are negated.
4. The model checker is asked to prove the validity of each negated property.
5. The model checker finds a counterexample for each negated property (assuming that the corresponding test purpose is satisfiable, e.g. the desired path it traversable).
6. Each (abstract) counterexample is translated to a (concrete) test input.

Search-Based Generation Combinatorial search-based optimization algorithms, such as local search [57], simulated annealing [91] and genetic algorithms [22], may be employed by the input generator. The fitness function that such algorithms attempt to minimize (or maximize) should be defined in such a manner that it favors inputs that possess a desirable property:

- they exercise a particular (unexplored) program path
- they lie on the boundaries of an input partition
- they satisfy a precondition

- they fail a postcondition
- they cause the program to exhibit incorrect or dangerous behaviour (e.g. crash or access memory illegally)
- they cause the program to execute for a long period of time
- ...

See the paper by McMinn [71] for a recent survey on the use of such search-based techniques for automated testing purposes.

Fuzz Testing Fuzz testing (or fuzzing) [73] is a black-box testing technique that involves feeding unexpected, random, or randomly mutated well-formed inputs to a program, in an attempt to trigger crashes. Because it can essentially only verify that a system can handle unexpected input gracefully, fuzz testing is mostly useful for uncovering security bugs.

The simplest fuzzers treat the input simply as a series of bits. This approach makes it hard to produce test cases that resemble valid data enough to reach the inner parts of the program. The most successful fuzzers have detailed understanding of the format or protocol being tested [53]. In order to produce an input, they walk through the input format specification and add a small number of anomalies to the data contents, structures, messages or sequences, thus producing inputs that are "almost valid".

White-box techniques, such as source-level tracing [37] and symbolic execution [41], can be combined with input mutation to more effectively guide the fuzzing process.

6.3 Test Set Adequacy Evaluation

Many techniques have been developed for estimating the completeness of a test suite. Since most testing procedures do not test programs exhaustively, there is always a possibility that an error has been overlooked, therefore any test set, regardless of how "complete" it is estimated to be, can never verify the correctness of a program. Having a measure of adequacy, however, allows us to set a target for the testing effort, and thus know when it it relatively safe to stop. Testing tools can also use this information to guide the input generator.

At its current form, PropEr doesn't include support for any test set adequacy metric.

Source Code Coverage Criteria A common criterion used to evaluate test sets is that of code coverage: a set of test cases should exercise as great a portion of the program as possible. Various concrete metrics have been proposed as a measure of code coverage (the paper by Zhu et al. [98] gives a comprehensive review):

- Control flow based:

  Statement Coverage
  Every statement of the program has to be executed at least once.

  Branch Coverage
  Every possible outcome of all decisions must be exercised at least once.

Path Coverage
Every possible path through the code has to be executed.

...

- Data flow based:

  All Definitions Coverage
  For each value-binding statement of each variable declared in the program, at least one definition-use path[1] starting at that definition must be executed.

  All Uses Coverage
  For each value-binding statement of each variable declared in the program, all the definition-use paths starting at that definition must be executed.

  ...

Specification Coverage Criteria Black-box testing tools may also include a way of measuring the completeness of a test set derived solely from the specification (rather than the source code) of the program [79, 78, 61, 47]. Some of the most common specification coverage criteria, targeted at model-based specifications, include state coverage, transition coverage and path coverage.

Runtime invariant detection techniques can also be used to reason about test suite adequacy [43]: We consider an additional test case as redundant when adding it to the test suite doesn't affect the set of invariants inferred from the execution of all tests.

Mutation Analysis Through mutation analysis [30], test suites are evaluated on their ability to distinguish the original program from slightly modified versions. Each such version, called a "mutant", is derived from the original by deliberately inserting one fault somewhere in the code. A fault injected by the mutator might be the change of some constant value (e.g. 0 to 1), the swapping of an operator with a different one (e.g. $<$ to $>$) etc. If at least one test case from a test suite causes the mutant to exhibit different (observable) behaviour than the original, then the test suite is said to "kill" the mutant. A test suite is assigned a mutation score according to the percentage of mutants it managed to kill. That score is an approximation of the relative adequacy of the test data set (live mutants point out inadequacies in the test suite).

Mutation analysis works on the basis of two hypotheses: The "competent programmer" hypothesis states that programmers write programs that are almost perfect, therefore program faults are usually syntactically small. Mutants, then, simulate the likely effects of real faults, thus a test set that is efficient at killing mutants will also be good at uncovering real faults. The "coupling" hypothesis states that, should there be any big and dramatic effects that arise from bugs in the software, then these will be closely coupled to small and simple bugs. Based on these hypotheses, proponents of mutation analysis claim that, if the program contains a fault, it is likely that there is a mutant which can only be killed by a test case that also reveals the fault.

Black-box testing tools can also employ mutation analysis [3]. In the case of such tools, mutations are applied to the specification instead of the program. A model-based specification, for example, can be mutated by adding, moving or deleting states and transitions,

[1] A definition-use path (du-path) for a variable x inside a program P is a series of statements (extracted from some execution of P) that starts at some binding of a value to x and ends at the first use of that value.

by altering preconditions etc. A model checker can then be asked to verify that the mutated model is equivalent to the original one. If the two models are distinguishable, the model checker will produce a counterexample that proves it. That counterexample can then be translated into a concrete test case.

Mutation analysis can also be used to guide the generation of new test cases. The Godzilla tool [31] utilizes symbolic execution to derive constraints on the input values that can distinguish a specific mutant from the original. A constraint solver is then used to derive a concrete test case that will kill the mutant.

Chapter 7

Conclusion

7.1 Concluding Remarks

We have presented PropEr, a property-based testing tool that can work with type information directly, thus liberating users from having to write redundant term generators. We have also described our extensions to that tool, which allow it to handle opaque data types properly, and test functions automatically based solely on their specifications. We have tested the effectiveness of our implementation by spec testing PropEr's own code, as well as various modules from the standard Erlang library. Based on the results of this testing, we conclude that this direction shows promise, but our approach, due in part to expressivity problems in the language of specs, suffers from a few inherent limitations, that currently make our tool inapplicable to certain categories of functions. As part of our future work in this area, we intend to explore possible solutions to some of these issues.

7.2 Future Work

The first extension we need to make concerns PropEr's handling of exceptional function returns. Users need to have some standard way of declaring what exceptions a function is expected to throw, and perhaps under what circumstances. Apart from their immediate usability in the context of automatic spec testing, such declarations will also be valuable as program documentation, and might be useful to static analysis tools like Dialyzer.

To make PropEr usable for a wider variety of functions, we need to introduce a way for users to more accurately specify the input domain of a function. We are currently considering the addition of precondition declarations to function domains, along with an efficient method of producing satisfying instances. Related to this is the addition of support for postcondition declarations to range types, which PropEr could check against every returned value.

Aside from these more pressing concerns, we should also take steps to improve the memory requirements and shrinking behaviour of generated terms, especially symbolic instances and instances of the 'any()' type. As a temporary measure, we could try introducing an upper limit to the value of the 'size' parameter. It may also be worth developing a more powerful system for the outputting and shrinking of generated functions.

We would also like to explore the possibility of extending our analysis to more diverse directions. At this point, we are considering extending our methodology to the testing of stateful or impure code. It would also be interesting to see if we can somehow make use of invariant detection tools to augment our fault detection strategy.

Bibliography

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

[2] EUnit: a Lightweight Unit Testing Framework for Erlang. `http://erlang.org/doc/apps/eunit/users_guide.html`.

[3] P. E. Ammann, P. E. Black, and W. Majurski. Using Model Checking to Generate Tests from Specifications. In *Proceedings of the 1998 International Conference on Formal Engineering Methods*, page 46. IEEE Computer Society, 1998.

[4] S. Antoy and D. Hamlety. . *IEEE Transactions on Software Engineering*, 26(1):55–69, January 2000.

[5] J. Armstrong. *Programming Erlang: Software for a Concurrent World*. The Pragmatic Bookshelf, 2007.

[6] J. Armstrong, R. Virding, C. Wikström, and M. Williams. *Concurrent Programming in Erlang*. Prentice Hall Europe, second edition, 1996.

[7] T. Arts, L. M. Castro, and J. Hughes. Testing Erlang Data Types with Quviq Quickcheck. In *Proceedings of the 7th ACM SIGPLAN Workshop on Erlang*, pages 1–8. ACM, 2008.

[8] 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.

[9] 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.

[10] L. Baresi and M. Young. Test Oracles. *Department of Computer and Information Science, University of Oregon, CIS-TR-01-02*, August 2001.

[11] K. Beck. *Kent Beck's Guide to Better Smalltalk*, chapter 30: "Simple Smalltalk Testing". Cambridge University Press, December 1998.

[12] J. Blom and B. Jonsson. Automated Test Generation for Industrial Erlang Applications. In *Proceedings of the 2003 ACM SIGPLAN Workshop on Erlang*, pages 8–14. ACM, 2003.

[13] J. Boberg. Early Fault Detection with Model-Based Testing. In *Proceedings of the 7th ACM SIGPLAN Workshop on Erlang*, pages 9–20. ACM, 2008.

[14] F. Bouquet, F. Dadeau, and B. Legeard. Automated Boundary Test Generation from JML Specifications. In J. Misra, T. Nipkow, and E. Sekerinski, editors, *FM 2006: Formal Methods*, volume 4085 of *Lecture Notes in Computer Science*, pages 428–443. Springer, Berlin / Heidelberg, 2006.

[15] C. Boyapati, S. Khurshid, and D. Marinov. Korat: Automated Testing Based on Java Predicates. *SIGSOFT Software Engineering Notes*, 27(4):123–133, July 2002.

[16] R. S. Boyer, B. Elspas, and K. N. Levitt. SELECT–A Formal System for Testing and Debugging Programs by Symbolic Execution. In *Proceedings of the International Conference on Reliable Software*, pages 234–245. ACM, 1975.

[17] J. R. Callahan, S. M. Easterbrook, and T. L. Montgomery. Generating Test Oracles via Model Checking. Technical Report NASA-IVV-98-015, NASA/WVU Software Research Lab, Fairmont, WV, 1998.

[18] J. R. Callahan, F. Schneider, and S. M. Easterbrook. Automated Software Testing Using Model-Checking. In *Proceedings of the 1996 SPIN Workshop*, volume 353, 1996.

[19] D. Carrington, I. MacColl, J. McDonald, L. Murray, and P. Strooper. From Object-Z Specifications to ClassBench Test Suites. *Software Testing, Verification and Reliability*, 10(2):111–137, 2000.

[20] Y. Cheon. *A Runtime Assertion Checker for the Java Modeling Language*. PhD thesis, Department of Computer Science, Iowa State University, April 2003.

[21] Y. Cheon, A. Cortes, and G. T. Leavens. Integrating Random Testing with Constraints for Improved Efficiency and Diversity. In *Proceedings of the 20th International Conference on Software Engineering and Knowledge Engineering*, July 2008.

[22] Y. Cheon, M. Y. Kim, and A. Perumandla. A Complete Automation of Unit Testing for Java Programs. In *Proceedings of the 2005 International Conference on Software Engineering Research and Practice (SERP)*, pages 290–295. CSREA Press, June 2005.

[23] Y. Cheon and G. T. Leavens. A Simple and Practical Approach to Unit Testing: The JML and JUnit Way. In B. Magnusson, editor, *ECOOP 2002 – Object-Oriented Programming*, volume 2374 of *Lecture Notes in Computer Science*, pages 1789–1901. Springer, Berlin / Heidelberg, 2006.

[24] Y. Cheon and C. E. Rubio-Medrano. Random Test Data Generation for Java Classes Annotated with JML Specifications. In *Proceedings of the 2007 International Conference on Software Engineering Research and Practice*, pages 385–392, June 2007.

[25] I. Ciupa and A. Leitner. Automatic Testing Based on Design by Contract. In *Proceedings of the 6th Annual International Conference on Object-Oriented and Internet-based Technologies, Concepts, and Applications for a Networked World*, pages 545–557, 2005.

[26] 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.

[27] L. A. Clarke. A System to Generate Test Data and Symbolically Execute Programs. *IEEE Transactions on Software Engineering*, 2:215–222, 1976.

[28] C. Csallner and Y. Smaragdakis. JCrasher: An Automatic Robustness Tester for Java. *Software: Practice and Experience*, 34(11):1025–1050, 2004.

[29] S. R. Dalal, A. Jain, N. Karunanithi, J. M. Leaton, C. M. Lott, G. C. Patton, and B. M. Horowitz. Model-Based Testing in Practice. In *Proceedings of the 1999 International Conference on Software Engineering (ICSE)*, page 285. ACM, May 1999.

[30] R. A. DeMillo, R. J. Lipton, and F. G. Sayward. Hints on Test Data Selection: Help for the Practicing Programmer. *Computer*, 11(4):34–41, April 1978.

[31] R. A. DeMillo and J. Offutt. Constraint-Based Automatic Test Data Generation. *IEEE Transactions on Software Engineering*, 17(9):900–910, September 1991.

[32] J. Dick and A. Faivre. Automating the Generation and Sequencing of Test Cases from Model-Based Specifications. In J. Woodcock and P. Larsen, editors, *FME '93: Industrial-Strength Formal Methods*, volume 670 of *Lecture Notes in Computer Science*, pages 268–284. Springer, Berlin / Heidelberg, 1993.

[33] J. Edvardsson. A Survey on Automatic Test Data Generation. In *Proceedings of the 2nd Conference on Computer Science and Engineering*, pages 21–28. ECSEL, October 1999.

[34] S. H. Edwards. Black-Box Testing Using Flowgraphs: An Experimental Assessment of Effectiveness and Automation Potential. *Software Testing, Verification and Reliability*, 10(4):249–262, 2000.

[35] I. K. El-Far and J. A. Whittaker. Model-based Software Testing. *Encyclopedia on Software Engineering*, 2001.

[36] A. Engels, L. Feijs, and S. Mauw. Test Generation for Intelligent Networks Using Model Checking. In E. Brinksma, editor, *Tools and Algorithms for the Construction and Analysis of Systems*, volume 1217 of *Lecture Notes in Computer Science*, pages 384–398. Springer, Berlin / Heidelberg, 1997.

[37] V. Ganesh, T. Leek, and M. Rinard. Taint-based Directed Whitebox Fuzzing. In *Proceedings of the 31st IEEE International Conference on Software Engineering (ICSE)*, pages 474–484. IEEE Computer Society, May 2009.

[38] J. Gannon, P. McMullin, and R. Hamlet. Data Abstraction, Implementation, Specification, and Testing. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 3(3):211–223, July 1981.

[39] A. Gargantini and C. Heitmeyer. Using Model Checking to Generate Tests from Requirements Specifications. In O. Nierstrasz and M. Lemoine, editors, *Software Engineering – ESEC/FSE '99*, volume 1687 of *Lecture Notes in Computer Science*, pages 146–162. Springer, Berlin / Heidelberg, 1999.

[40] P. Godefroid, N. Klarlund, and K. Sen. DART: Directed Automated Random Testing. In *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 213–223. ACM, 2005.

[41] P. Godefroid, M. Y. Levin, and D. Molnar. Automated Whitebox Fuzz Testing.

[42] A. Gotlieb, B. Botella, and M. Ruehe. Automatic Test Data Generation using Constraint Solving Techniques. In *Proceedings of the 1998 ACM SIGSOFT international Symposium on Software Testing and Analysis (ISSTA)*, pages 53–62. ACM, 1998.

[43] M. Harder, J. Mellen, and M. D. Ernst. Improving Test Suites via Operational Abstraction. In *Proceedings of the 25th International Conference on Software Engineering (ICSE)*, pages 60–71. IEEE Computer Society, 2003.

[44] M. P. Heimdahl, S. Rayadurgam, W. Visser, G. Devaraj, and J. Gao. Auto-generating Test Sequences Using Model Checkers: A Case Study. In A. Petrenko and A. Ulrich, editors, *Formal Approaches to Software Testing*, volume 2931 of *Lecture Notes in Computer Science*, page 1100. Springer, Berlin / Heidelberg, 2004.

[45] R. M. Hierons, K. Bogdanov, J. P. Bowen, R. Cleaveland, J. Derrick, J. Dick, M. Gheorghe, M. Harman, K. Kapoor, P. Krause, G. Lüttgen, A. J. H. Simons, S. Vilkomir, M. R. Woodward, and H. Zedan. Using Formal Specifications to Support Testing. *ACM Computing Surveys*, 41(2):9:1–9:76, February 2009.

[46] H. Hong, I. Lee, O. Sokolsky, and S. Cha. Automatic Test Generation from Statecharts Using Model Checking. In *Proceedings of the 1st Workshop on Formal Approaches to Testing of Software*, pages 15–30, 2001.

[47] H. Hong, I. Lee, O. Sokolsky, and H. Ural. A Temporal Logic Based Theory of Test Coverage and Generation. In J.-P. Katoen and P. Stevens, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, volume 2280 of *Lecture Notes in Computer Science*, pages 151–161. Springer, Berlin / Heidelberg, 2002.

[48] H.-M. Hörcher and J. Peleska. Using Formal Specifications to Support Software Testing. *Software Quality Journal*, 4(4):309–327, 1995.

[49] P. Jalote. Specification and Testing of Abstract Data Types. *Computer Languages*, 17(1):75–82, 1992.

[50] 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.

[51] T. Jéron and P. Morel. Test Generation Derived from Model-Checking. In N. Halbwachs and D. Peled, editors, *Computer Aided Verification*, volume 1633 of *Lecture Notes in Computer Science*, pages 108–122. Springer, Berlin / Heidelberg, 1999.

[52] J.-M. Jézéquel, D. Deveaux, and Y. L. Traon. Reliable Objects: Lightweight Testing for OO Languages. *IEEE Software*, 18(4):76–83, July 2001.

[53] R. Kaksonen. A Functional Method for Assessing Protocol Implementation Security. *VVT Publications*, October 2001.

[54] C. Kaner and J. Bach. Black Box Software Testing: The Oracle Problem. `http://www.testingeducation.org/BBST/BBSTIntro1.html`, 2005. Lecture Notes, Center for Software Testing Education & Research, Florida Institute of Technology.

[55] R. A. Kemmerer. Testing Formal Specifications to Detect Design Errors. *IEEE Transactions on Software Engineering*, 11(1):32–43, January 1985.

[56] P. Koopman, A. Alimarine, J. Tretmans, and R. Plasmeijer. Gast: Generic Automated Software Testing. In R. Peña and T. Arts, editors, *Implementation of Functional Languages*, volume 2670 of *Lecture Notes in Computer Science*, pages 991–991. Springer, Berlin / Heidelberg, 2003.

[57] B. Korel. Automated Software Test Data Generation. *IEEE Transactions on Software Engineering*, 16(8):870–879, 1990.

[58] B. Krause and T. Wahls. jmle: A Tool for Executing JML Specifications Via Constraint Programming. In L. Brim, B. Haverkort, M. Leucker, and J. van de Pol, editors, *Formal Methods: Applications and Technology*, volume 4346 of *Lecture Notes in Computer Science*, pages 293–296. Springer, Berlin / Heidelberg, 2007.

[59] D. Kung, J. Gao, P. Hsia, Y. Toyoshima, C. Chen, Y.-S. Kim, and Y.-K. Song. Developing an Object-Oriented Software Testing and Maintenance Environment. *Communications of the ACM*, 38(10):75–87, October 1995.

[60] B. Legeard, F. Peureux, and M. Utting. Automated Boundary Testing from Z and B. In L.-H. Eriksson and P. Lindsay, editors, *FME 2002: Formal Methods–Getting IT Right*, volume 2391 of *Lecture Notes in Computer Science*, pages 221–236. Springer, Berlin / Heidelberg, 2002.

[61] B. Legeard, F. Peureux, and M. Utting. Controlling Test Case Explosion in Test Generation from B Formal Models. *Software Testing, Verification and Reliability*, 14(2):81–103, June 2004.

[62] T. Lindahl and K. Sagonas. Detecting Software Defects in Telecom Applications Through Lightweight Static Analysis: A War Story. In W.-N. Chin, editor, *Proceedings of the Second Asian Symposium on Programming Languages and Systems*, volume 3302 of *Lecture Notes in Computer Science*, pages 91–106, Berlin / Heidelberg, 2004. Springer.

[63] T. Lindahl and K. Sagonas. TypEr: a Type Annotator of Erlang Code. In *Proceedings of the 2005 ACM SIGPLAN Workshop on Erlang*, pages 17–25. ACM, 2005.

[64] 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.

[65] S. Liu and H. Wang. An Automated Approach to Specification Animation for Validation. *The Journal of Systems and Software*, 80(8):1271–1285, August 2007.

[66] P. D. L. Machado. *Testing from Structured Algebraic Specifications: The Oracle Problem*. PhD thesis, University of Edinburgh, 2000.

[67] M.-V. Manoukian. Detection of Opaque Violations in Erlang Using Static Analysis. Diploma thesis, Department of Electrical and Computer Engineering, National Technical University of Athens, 2009.

[68] S. Marlow and P. Wadler. A practical subtyping system for Erlang. In *Proceedings of the 2nd ACM SIGPLAN International Conference on Functional Programming*, pages 136–149. ACM, 1997.

[69] J. McDonald, L. Murray, and P. Strooper. Translating Object-Z Specifications to Object-Oriented Test Oracles. In *Proceedings of the 4th Asia-Pacific Software Engineering and International Computer Science Conference (APSEC)*, page 414. IEEE Computer Society, 1997.

[70] J. McDonald and P. Strooper. Translating Object-Z Specifications to Passive Test Oracles. In *Proceedings of the 2nd IEEE International Conference on Formal Engineering Methods (ICFEM)*, page 165. IEEE Computer Society, 1998.

[71] P. McMinn. Search-based Software Test Data Generation: A Survey. *Software Testing, Verification and Reliability*, 14(2):105–156, 2004.

[72] C. Meudec. ATGen: Automatic Test Data Generation Using Constraint Logic Programming and Symbolic Execution. *Software Testing, Verification and Reliability*, 11(2):81–96, June 2001.

[73] B. P. Miller, L. Fredriksen, and B. So. An Empirical Study of the Reliability of UNIX Utilities. *Communications of the ACM*, 33(12):32–44, 1990.

[74] T. Miller and P. Strooper. Model-Based Specification Animation Using Testgraphs. In C. George and H. Miao, editors, *Formal Methods and Software Engineering*, volume 2495 of *Lecture Notes in Computer Science*, pages 192–203. Springer, Berlin / Heidelberg, 2002.

[75] T. Miller and P. Strooper. Supporting the Software Testing Process through Specification Animation. In *Proceedings of the 1st International Conference on Software Engineering and Formal Methods (SEFM)*, page 14. IEEE Computer Society, 2003.

[76] S.-O. Nyström. A soft-typing system for Erlang. In *Proceedings of the 2003 ACM SIGPLAN Workshop on Erlang*, pages 56–71. ACM, 2003.

[77] J. Offutt and A. Abdurazik. Generating Tests from UML Specifications. In R. France and B. Rumpe, editors, *UML '99 – The Unified Modeling Language*, volume 1723 of *Lecture Notes in Computer Science*, page 76. Springer, Berlin / Heidelberg, 1999.

[78] J. Offutt, S. Liu, A. Abdurazik, and P. Ammann. Generating Test Data from State-Based Specifications. *Software Testing, Verification and Reliability*, 13(1):25–53, March 2003.

[79] J. Offutt, Y. Xiong, and S. Liu. Criteria for Generating Specification-based Tests. In *Proceedings of the 5th International Conference on Engineering of Complex Computer Systems (ICECCS)*, pages 119–129. IEEE Computer Society, October 1999.

[80] C. Oriat. Jartege: a Tool for Random Generation of Unit Tests for Java Classes. In R. Reussner, J. Mayer, J. A. Stafford, S. Overhage, S. Becker, and P. J. Schroeder, editors, *Quality of Software Architectures and Software Quality*, volume 3712 of *Lecture Notes in Computer Science*, pages 242–256. Springer, Berlin / Heidelberg, 2005.

[81] C. Pacheco and M. D. Ernst. Eclat: Automatic Generation and Classification of Test Inputs. In A. P. Black, editor, *ECOOP 2005 – Object-Oriented Programming*, volume 3586 of *Lecture Notes in Computer Science*, pages 504–527. Springer, Berlin / Heidelberg, 2005.

[82] C. Pacheco, S. K. Lahiri, M. D. Ernst, and T. Ball. Feedback-directed Random Test Generation. In *29th International Conference on Software Engineering (ICSE)*, pages 75–84. IEEE Computer Society, 2007.

[83] A. Pasquini and B. Aichernig. Automated Black-Box Testing with Abstract VDM Oracles. In K. Kanoun, editor, *Computer Safety, Reliability and Security*, volume 1698 of *Lecture Notes in Computer Science*, page 688. Springer, Berlin / Heidelberg, 1999.

[84] D. K. Peters and D. L. Parnas. Using Test Oracles Generated from Program Documentation. *IEEE Transactions on Software Engineering*, 24(3):161–173, March 1998.

[85] C. V. Ramamoorthy, S.-B. F. Ho, and W. Chen. On the Automated Generation of Program Test Data. *IEEE Transactions on Software Engineering*, 2(4):293–300, December 1976.

[86] D. J. Richardson, O. O'Malley, and C. Tittle. Approaches to Specification-Based Testing. In *Proceedings of the 3rd ACM SIGSOFT Symposium on Software Testing, Analysis, and Verification*, pages 86–96. ACM, 1989.

[87] 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.

[88] K. Sabnani and A. Dahbura. A Protocol Test Generation Procedure. *Computer Networks and ISDN Systems*, 15(4):285–297, 1988.

[89] M. Satpathy, M. Leuschel, and M. Butler. ProTest: An Automatic Test Environment for B Specifications. In *Proceedings of the 2004 Workshop on Model Based Testing (MBT)*, volume 111 of *Electronic Notes in Theoretical Computer Science*, pages 113–136. Elsevier, January 2005.

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

[91] N. Tracey, J. Clark, and K. Mander. Automated Program Flaw Finding using Simulated Annealing. *ACM SIGSOFT Software Engineering Notes*, 23(2):73–81, 1998.

[92] H. Ural. Formal Methods for Test Sequence Generation. *Computer Communications*, 15(5):311–325, 1992.

[93] M. Utting, A. Pretschner, and B. Legeard. A Taxonomy of Model-Based Testing. Working paper, April 2006.

[94] W. Visser, C. S. Păsăreanu, and S. Khurshid. Test Input Generation with Java PathFinder. In *Proceedings of the 2004 ACM SIGSOFT international Symposium on Software Testing and Analysis*, pages 97–107. ACM, 2004.

[95] E. J. Weyuker. On Testing Non-Testable Programs. *The Computer Journal*, 25(4):465–470, 1982.

[96] T. Xie, D. Marinov, W. Schulte, and D. Notkin. Symstra: A Framework for Generating Object-Oriented Unit Tests Using Symbolic Execution. In N. Halbwachs and

L. D. Zuck, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, volume 3440 of *Lecture Notes in Computer Science*, pages 365–381. Springer, Berlin / Heidelberg, 2005.

[97] G. Xu and Z. Yang. JMLAutoTest: A Novel Automated Testing Framework Based on JML and JUnit. In A. Petrenko and A. Ulrich, editors, *Formal Approaches to Software Testing*, volume 2931 of *Lecture Notes in Computer Science*, pages 1103–1104. Springer, Berlin / Heidelberg, 2004.

[98] H. Zhu, P. A. V. Hall, and J. H. R. May. Software Unit Test Coverage and Adequacy. *ACM Computing Surveys*, 29(4):366–427, December 1997.