Verifying a Simple Compiler Using Property-based Random Testing

Pedro Vasconcelos

Technical Report Series: DCC-2013-06

Abstract

This paper reports on the use of the Haskell *QuickCheck* library for testing the correctness of a simple functional compiler and abstract machine. We use QuickCheck to express the correctness of the abstract machine against a denotational semantics, to generate well-formed test programs and to automatically shrink counterexamples obtained when a test fails.

1 Introduction

This paper reports on the use the Haskell *QuickCheck* testing library [2] for verifying the implementation correctness of a simple functional language. The implementation consists of a syntax-directed compiler and an interpreter for (a variant of) the well-known SECD abstract machine [5]. The denotational semantics, compiler and abstract machine are written in Haskell and correctness is expressed as a QuickCheck property relating the execution of the compiled code against the denotational semantics. We use QuickCheck to define generators for random well-formed programs and a "shrinking" heuristic based on *partial evaluation* to simplify failed test cases. Furthermore, we show that this heuristic allows us to obtain short and insightful counterexamples from randomly-generated test cases.

The motivation for this work was pedagogical: the author is lecturing a course on programming language implementation; part of the student's work in the course consists of modifying such a toy implementation (e.g. adding language features or optimizations). The ability to automatically test a large number of cases and obtain short counter-examples should help students find errors that might go unnoticed under less thorough testing. Moreover, property-based testing re-enforces the fundamental connections between theory (semantics) and implementations (machines).

While random testing has been used to verify both real and toy compilers [9, 3], the use of partial evaluation to shrink counterexamples is, to the best of our knowledge, novel.

The complete source code complementing this paper is available from the author's web page: `htp://www.dcc.fc.up.pt/~pbv/compcheck`.

2 The *Fun* language

2.1 Syntax and semantics

Fun is a minimal language intended for teaching principles of implementation of functional languages. It consists of the call-by-value λ-calculus extended with primitive integers, arithmetic operations, conditionals and local definitions. The abstract syntax and semantics of *Fun* are given in Figure 1.

The formal meaning of *Fun* terms is defined by a standard denotational semantics: the denotable values are the disjoint union of primitive integers and functions and are represented by the `Value` data type. We define partial projections `toInt` and `toFun` from values to integers and functions, respectively. As is usual, the meaning of free variables is given using environments (i.e. a functions from variables to values).

The denotational semantics of Figure 1 is an evaluation function `eval` from terms and environments to values. The semantics is standard except that `eval` is a partial function: erroneous terms cause the Haskell execution to diverge (hence the absence of explicit representation for errors). We consider here only non-recursive functions (in particular, *let*-expressions are not recursive); this is done to ensure termination of test cases (cf. Section 3).

Finally, we note the use of strictness annotations (`!`) in the fields of the `Value` data type and strict application (`$!`) in the semantics of application; this is because Haskell is non-strict by default and we intend to define a strict semantics for *Fun*.[1]

2.2 Compiler and abstract machine

The implementation of *Fun* is a compiler translating *Fun* terms to SECD abstract machine code and an interpreter for the SECD machine; the complete implementation is listed in Appendix A. The translation from *Fun* terms to SECD code is given as a function

```
compile :: Term -> [Ident] -> Code
```

[1]In fact, using a non-strict semantics for our testing purpose would suffice because we will only ever generate normalizing terms (cf. Section 3).

```
module Fun where

type Ident = Char     -- identifiers

-- abstract syntax of Fun terms
data Term = Const Int
          | Var Ident
          | Lambda Ident Term
          | App Term Term
          | Let Ident Term Term
          | If Term Term Term
          | Term :+ Term

-- denotational semantics
data Value = Prim { toInt :: !Int }
           | Fun { toFun :: !(Value -> Value) }

type Env = Ident -> Value

eval :: Term -> Env -> Value
eval (Var x) env = env x
eval (Const n) env  = Prim n
eval (Lambda x e) env
    = Fun (\v -> eval e (extend env x v))
eval (App e1 e2) env= f $! v
    where f = toFun (eval e1 env)
          v = eval e2 env
eval (If e1 e2 e3) env
    | v==0             = eval e3 env
    | otherwise        = eval e2 env
    where v = toInt (eval e1 env)
eval (Let x e1 e2) env
    = eval (App (Lambda x e2) e1) env
eval (e1 :+ e2) env = Prim (v1+v2)
  where v1 = toInt (eval e1 env)
        v2 = toInt (eval e2 env)
```

Figure 1: Abstract syntax and denotational semantics of *Fun*.

$$\begin{tikzcd}
\mathtt{Term} \arrow[r, "\mathtt{eval}"] \arrow[dd, "\mathtt{compile}"] & \mathtt{Fun.Value} \arrow[dr, "\mathtt{Fun.toInt}"] & \\
& & \mathtt{Int} \\
\mathtt{Code} \arrow[r, "\mathtt{run}"] & \mathtt{SECD.Value} \arrow[ur, "\mathtt{SECD.toInt}"'] &
\end{tikzcd}$$

Figure 2: Correctness of the implementation

The first argument is a term to be compiled; the second is a list of bound variables; the result is a list of instructions for the SECD machine. The translation is syntax-directed and follows the Henderson's textbook presentation [5].

A configuration of the SECD machine has four components (whose initials name the machine): a value *stack*, an *environment*, a *control* and *dump* stack. We represent stacks by lists and configurations by a 4-tuples in Haskell: the stack and environment are lists of *values*, i.e. primitive integers or closures; the control is a list of *instructions*; and the dump is a list of triples of stack, environment and code. The machine's instructions are represented by the `Instr` data type. The single-step transition between configurations is given by the function

```
execute :: SECD -> SECD
```

where `SECD` is a 4-tuple of the machine configuration. We also define an auxiliary function

```
run :: Code -> Value
```

which iterates the single-step transition from an initial configuration and yields the final value on top of the stack. As with the denotational semantics, run-time errors cause exceptions or non-termination in the Haskell interpreter.

3 Testing the correctness of the implementation

Informally, the implementation is correct if, for well-formed terms, the execution the compiled code yields a value equivalent to one given by the denotational semantics. To make this both precise and amendable to automatic testing, we remark that:

- We can restrict to testing *simply-typed terms*: following Milner's slogan this ensures that programs "don't go wrong", i.e. attempt to treat an integer as a function;
- In fact, we can test just *integer* terms (for which equivalence testing is trivial) rather than arbitrary values (e.g. functions)[2];
- Because *let*-bindings are non-recursive, termination of of evaluation is ensured for simply-typed terms by the standard strong normalization result for the simply-typed λ-calculus.

The correctness of the implementation is then expressed by the commutativity of the diagram in Figure 2.

3.1 A QuickCheck primer

QuickCheck is a library for testing *properties* of Haskell programs [2]. The simplest form of a property is a Boolean function; for example, the following property states that composing list reverse with itself is the identity function:

[2]This corresponds to testing *adequacy* of the denotational semantics with respect to the operational semantics rather than full abstraction [8].

```
prop_rev_rev :: [Int] -> Bool
prop_rev_rev xs = reverse (reverse xs) == xs
```

The above property is implicitly universally quantified over the argument `xs`. QuickCheck can test the property with randomly generated lists:

```
> quickCheck prop_rev_rev
+++ OK, passed 100 tests.
```

Testing aborts if a counterexample is found and the failing test case is shown. For example:

```
prop_wrong :: [Int] -> Bool
prop_wrong xs = reverse (reverse xs) == reverse xs

> quickCheck prop_wrong
*** Failed! Falsifiable (6 tests and 4 shrinks):
[0,1]
```

Note that the counterexample found is the "smallest" one that falsifies the conjecture. While it is not possible to guarantee minimality in all cases, there are some techniques to automatically shrink counterexamples; we will describe these in Section 3.4.

QuickCheck defines default test case generators for basic types (such as `Int`) and for structured ones (such lists and tuples). The library also defines a small monadic domain-specific language for writing custom generators; for details see the QuickCheck documentation[3].

3.2 Specifying correctness in QuickCheck

We can express the correctness of our implementation by the following congruence property (cf. Figure 2):

```
prop_correct :: Term -> Bool
prop_correct e
  = Fun.toInt (eval e emptyEnv) ==
    SECD.toInt (run $ compile e [])
```

To test this property we first need to define a custom generator for terms; as discussed earlier, we will generate well-typed integer terms. We start by defining simple types as Haskell data, considering only integer and functional types:

```
data Type = Tint | Tfun Type Type
```

Furthermore, we introduce a synonym for a *typing context*, i.e. an association list of identifiers to types:

```
type Context = [(Ident,Type)]
```

The term generator is given in Figure 3 as a function `genTerm` parametrized by the size, context and result type:

```
genTerm :: Int -> Context -> Type -> Gen Term
```

The generator is defined by case analysis of the type argument; for example, to generate an integer expression we (randomly) choose of the following: an integer constant, two integer terms joined by an arithmetic operator; an application, let or conditional that yields an integer; or one integer variable from the current context. Either the type or the size parameter (or both) get smaller when generating sub-expressions; this controls the size of generated term and ensures termination.

The auxiliary function `genAppTerm` generates an application, let-expression or conditional term. For example: to generate an application of type t we first generate a type t' and two terms of types $t' \to t$ and t' respectively. Because *Fun* is higher-order, t and t' may themselves be functional types.

We can specify an instance of the `Arbitrary` class for terms using `genTerm`, i.e. a generator for *closed* terms of integer type:

[3]`http://hackage.haskell.org/package/QuickCheck`

```
instance Arbitrary Term where
   arbitrary = sized (\n -> genTerm n [] Tint)
```

3.3 Testing using QuickCheck

We can test our implementation using QuickCheck by combining the correctness property in Section 3.2 and generator in Figure 3. As would be expected for a simple language and well-known abstract machine, this did not reveal any errors in our implementation. It did, however, reveal an error our first generator which could incorrectly pick a "shadowed" identifier from an outer scope; the solution in Figure 3 hides such occurrences (cf. the use of `trim` in function `pickVar`).

Using our implementation as a starting point, we experimented with artificially introducing errors in either the compiler or abstract machine in an attempt to mimic typical student's mistakes; these ranged from simple typos (e.g. compiling conditional branches in the wrong order) to more subtle errors (e.g. incorrect state restore on function return). All such errors are indeed quickly detected by random testing. However, the counterexamples obtained can be relatively large, making it difficult to identify the actual cause of the failure. Fortunately, QuickCheck allows the definition of *shrinking functions* that implement some heuristic for reducing the size of counterexamples [1]. In the following section we present the shrinking heuristic for terms and defer a discussion on the effectiveness of testing to section 3.5.

3.4 Shrinking counterexamples

Along with the test data generator, the `Arbitrary` type class allows the definition of a *shrinking function*:

```
class Arbitrary a where
    arbitrary :: Gen a
    shrink :: a -> [a]
    shrink _ = []       -- default: no shrinking
```

The `shrink` function should yield a (possibly empty) finite list of "smaller" values derived from its argument. The default shrink is the constant function yielding the empty list (i.e. no shrinking). Note that, unlike generation, shrinking does not depend on a random seed (i.e., it is a function solely of the test data).

When a property fails, `shrink` is applied to the counterexample found; if this yields a value that still falsifies the property, shrinking continues; otherwise the process terminates and a (locally) minimal counterexample has been found. To guarantee termination, the shrink function should be well-founded, i.e. there should not exist an infinite chains of shrinks; this can easily be ensured for inductive data e.g., by yielding structurally smaller components.

Care must be taken when shrinking abstract syntax terms to prevent breaking non-local properties; for example, sub-terms of a closed term may be open, or admit different a type than the enclosing term. The implementation presented in Figure 4 preserves types and closure of terms by combining structural shrinking with *partial evaluation*:

- the auxiliary function `shrinkRedex` attempts a a single reduction step, yielding a singleton list if the reduction is possible or the empty list otherwise;

- the main function `shrinkTerm` first attempts partial evaluation, followed by structural sub-terms (when applicable) and finally recursively shrinks sub-terms.

Partial evaluation preserves types and closure, while termination is ensured by the strong normalization result for simply typed terms. Structural shrinking is allowed only in cases where types and closure are also preserved (i.e. conditional branches and arguments of arithmetic operators).

The shrinking function for terms is specified in the instance of the `Arbitrary` class:

```
genTerm :: Int -> Context -> Type -> Gen Term
genTerm size ctx Tint
  | size>0 = oneof ([liftM Const arbitrary, liftM2 (:+) gt gt,
                     genAppTerm size ctx Tint] ++
                     pickVar ctx Tint)
  | otherwise= oneof (liftM Const arbitrary : pickVar ctx Tint)
    where gt = genTerm (size`div`2) ctx Tint

genTerm size ctx t@(Tfun t1 t2)
  | size>0 = oneof ([genLambda size ctx t1 t2, genAppTerm size ctx t] ++
                     pickVar ctx t)
  | otherwise = oneof (genLambda size ctx t1 t2 : pickVar ctx t)

-- generate applications, let or conditionals
genAppTerm size ctx t = oneof [genApp,genLet,genIf]
    where
     gt = genTerm (size`div`2)
     gt'= genTerm (size`div`3)
     genApp = do t' <- pickType ctx
                 liftM2 App (gt ctx (Tfun t' t)) (gt ctx t')
     genLet = do x <- genIdent
                 t'<- pickType ctx
                 liftM2 (Let x) (gt ctx t') (gt ((x,t'):ctx) t)
     genIf = liftM3 If (gt' ctx Tint) (gt' ctx t)  (gt' ctx t)

-- generate a lambda abstraction
genLambda size ctx t1 t2
    = do x <- genIdent
         liftM (Lambda x) (genTerm size ((x,t1):ctx) t2)

-- generate a variable name
genIdent :: Gen Ident
genIdent = elements ['a'..'z']

-- pick a variable from a context by type
-- hides identifiers in outer scopes
pickVar :: Context -> Type -> [Gen Term]
pickVar ctx t = optElements [Var x | (x,t')<-trim ctx, t'==t]

-- trim a context hiding outer scopes
trim :: Context -> Context
trim = nubBy ((==)`on`fst)

-- optional generator from a list of elements
optElements :: [a] -> [Gen a]
optElements [] = []
optElements xs = [elements xs]

-- pick a simple type from the context
pickType :: Context -> Gen Type
pickType ctx = elements $ nub (Tint : [t | (x,t)<-trim ctx])
```

Figure 3: Generator for simply-typed *Fun* terms.

```
instance Arbitrary Term where
   arbitrary = sized (\n -> genTerm n [] Tint)
   shrink = shrinkTerm
```

3.5 Effectiveness of random testing

We shall now illustrate the effectiveness of random testing through some examples.

Compiling conditional branches in wrong order. As a first example, let us consider an error introduced by compiling the branches `e2` and `e3` of a conditional `If e1 e2 e3` in the wrong order. QuickCheck testing finds on counterexample very quickly:

```
*** Failed! Falsifiable (after 7 tests):
If (Let 'b' (Const (-2)) (Var 'b') :+ Const 5)
   (Const (-4))
   (If (Const (-4)) (Const (-3)) (Const 6))
```

Because of pseudo-random generation, this term contains many irrelevant details; thankfully, our shrinking heuristic can simplify it considerably. Here is the counterexample obtained after enabling shrinking:

```
*** Failed! Falsifiable (7 tests and 8 shrinks):
If (Const 0) (Const 0) (Const 1)
```

We observe that the heuristic was very effective, as this is now a minimal counterexample, namely, a conditional expression with two distinct branches where all sub-expressions are fully evaluated.

Failing to restore the stack. Let us now consider an error in the abstract machine implementation, namely, failing to restore the stack on function return. Again testing detects the error (which manifests itself as a pattern-matching failure due to stack underflow in `execute`):

```
*** Failed! Exception (12 tests and 3 shrinks):
Const 0 :+ App (Lambda 't' (Var 't')) (Const 0)
```

Note that the counterexample above is indeed a local minimum; consider the following expressions obtained by shrinking:

```
Const 0
Const 0 :+ Const 0
App (Lambda 't' (Var 't')) (Const 0)
```

The first two expressions do not evaluate applications and so don't trigger the buggy return; last one does evaluate an application, but in the particular case where the saved stack is empty and thus does not lead to the stack underflow.

Failing to restore the environment. We now consider an error introduced in a simple optimization: compiling more efficient code for *let* by avoiding the closure creation and the transfer of control. We add an instruction `AA` ("add argument") to push a value from the stack to the environment:

```
execute (v:stack, env, AA:code, dump, store)
    = (stack, v:env, code, dump, store)
```

To evaluate the *let*, we evaluate the first expression, push the result value from the stack to the environment and evaluate the second one; here is a first attempt:

```
compile (Let x e1 e2) sym
    = compile e1 sym ++ [AA] ++ compile e2 (x:sym)
```

```
shrinkTerm :: Term -> [Term]
shrinkTerm e@(App e1 e2)
  = shrinkRedex e ++
    [App e1' e2 | e1'<-shrinkTerm e1] ++
    [App e1 e2' | e2'<-shrinkTerm e2]
shrinkTerm (Lambda x e)
    = [Lambda x e' | e'<-shrinkTerm e]
shrinkTerm e@(Let x e1 e2)
  = shrinkRedex e ++
    [Let x e1' e2 | e1'<-shrinkTerm e1] ++
    [Let x e1 e2' | e2'<-shrinkTerm e2]
shrinkTerm e@(If e1 e2 e3)
  = shrinkRedex e ++
    [e2, e3] ++
    [If e1' e2 e3 | e1'<-shrinkTerm e1] ++
    [If e1 e2' e3 | e2'<-shrinkTerm e2] ++
    [If e1 e2 e3' | e3'<-shrinkTerm e3]
shrinkTerm e@(e1 :+ e2)
  = shrinkRedex e ++
    [e1, e2] ++
    [e1':+ e2 | e1'<-shrinkTerm e1] ++
    [e1 :+ e2' | e2'<-shrinkTerm e2]
shrinkTerm (Const n) = [Const n' | n'<-shrink n]
shrinkTerm _ = []

-- shrink a redex by partial evaluation
shrinkRedex :: Term -> [Term]
shrinkRedex (App (Lambda x e1) e2)
    | x`notElem`fv e1 = [e1]
    | null (bv e1`intersect`fv e2) = [subst x e2 e1]
shrinkRedex (Let x e1 e2)
    | x`notElem`fv e2 = [e2]
    | null (bv e2`intersect`fv e1) = [subst x e1 e2]
shrinkRedex (If (Const 0) e1 e2) = [e2]
shrinkRedex (If (Const _) e1 e2) = [e1]
shrinkRedex (Const n1 :+ Const n2) = [Const (n1+n2)]
shrinkRedex _ = []
```

Figure 4: Shrinking function for *Fun* terms.

However, testing with QuickCheck highlights an error:

```
*** Failed! Falsifiable (56 tests and 41 shrinks):
Let 'q' (Const 0)
  (Let 's' (Const 1) (Var 'q') :+ Var 'q')
```

The error results from failing to restore the environment after the evaluation: the second reference to variable `q` will pick the value bound introduced by the binding for `s`. The solution is to add another instruction `PA` ("pop argument") to remove the last entry in the environment:

```
compile (Let x e1 e2) sym
    = compile e1 sym ++ [AA] ++
      compile e2 (x:sym) ++ [PA]

execute (stack, v:env, PA:code, dump, store)
    = (stack, env, code, dump, store)
```

The revised implementation passes all tests.

3.6 Extension to mutable references

We now consider extending our term language with mutable references in the style of ML; we start by augmenting the abstract syntax:

```
data Term = {- ... as before ... -}
          | Ref Term       -- new reference
          | Deref Term     -- de-reference
          | Term := Term   -- assignment
          | Seq Term Term  -- sequencing
          | Skip           -- null op
```

To define the semantics of references we re-write the denotational evaluator in monadic style [7] using a state monad of *stores* (mappings from locations to values). We extend denotations with *unit* and *location* values; following the call-by-value monadic translation, the co-domain of function values also becomes monadic.

```
data Value = PrimUnit
           | PrimInt { toInt :: !Int }
           | PrimLoc { toLoc :: !Loc }
           | Fun { toFun :: !(Value->EvalM Value) }

type Store = Map Loc Value
type EvalM = State Store

evalM :: Term -> Env -> EvalM Value
```

Because of space limitations we omit the full definition of `evalM` (a straightforward textbook exercise) and the modification of `genTerm`. For the later, we need to augment the syntax of types with *reference* and *unit type constructors*; the key modifications to term generation are then:

- a term of unit type is either a `Skip`, an application, sequencing, or an assignment to some reference variable in the current context;
- a term of some type t, we can do as before but also generate a sequence `Seq e1 e2` where `e1` has the unit type and `e2` has type t;
- to pick an expression of type t from a context, we can dereference a variable of type `Tref` t; this generalizes to an arbitrary number of dereferences.

```
shrinkTerm (Seq e1 e2)
  = [e2] ++
    [Seq e1' e2 | e1'<-shrinkTerm e1] ++
    [Seq e1 e2' | e2'<-shrinkTerm e2]
shrinkTerm (e1 := e2)
  = [Skip] ++
    [e1':= e2 | e1'<-shrinkTerm e1] ++
    [e1 := e2' | e2'<-shrinkTerm e2]
shrinkTerm (Ref e) = [Ref e' | e'<-shrinkTerm e]
shrinkTerm t@(Deref e)
  = shrinkRedex t ++ [Deref e' | e'<-shrinkTerm e]

shrinkRedex (Deref (Ref e)) = [e]
```

Figure 5: Extension of shrinking for mutable references.

Note that in the presence of side-effects, the evaluation order must be explicitly defined in all cases (not just for sequencing), e.g. in the term `(e1 :+ e2)` we must choose to which sub-term `e1` or `e2` to evaluate first. If the denotational semantics and implementation do not agree in evaluation order, testing the correctness property with QuickCheck finds a minimal counterexample witnessing the mismatch:

```
*** Failed! Falsifiable (37 tests and 37 shrinks):
Let 'r' (Ref (Const 0))
  (Seq (Var 'r' := Const 1) (Const 0) :+
       Deref (Var 'r'))
```

The body of the *let*-expression above is a sum `e1 :+ e2` of the two sub-expressions:

```
e1 = Seq (Var 'r' := Const 1) (Const 0)
e2 = Deref (Var 'r')
```

Because `e1` modifies the reference used in `e2`, the result *let*-expression can be either 0 or 1 depending on the order in which `:+` evaluates its arguments. As before, shrinking was essential to obtain a short counterexample; the extension to the shrink function to handle mutable references is shown in Figure 5; our heuristic needs only a few extra cases:

- shrinking `Seq e1 e2` may discard `e1` (but not `e2`, because this determines the result value);
- shrinking may discard any assignment (yielding `Skip`);
- the redex `Deref (Ref e)` can be simplified to `e`.

4 Related work

Claessen, Runciman, Chitil, Hughes and Wallace have shown how to use QuickCheck for testing the correctness of a compiler for simple imperative language [3]. Unlike our approach, term generation is not directed by types (because the language is first-order and all values are numbers). They also do not ensure termination of generated test term but instead specify correctness by relating the partial traces of an interpreter against compiled code. Also, this work does not address the issue of shrinking counterexamples.

Yang, Chen, Eide and Regehr have used random testing to find errors in production C compilers [9]. Instead of attempting a formal definition of the C-language semantics they rely on a "black-box" approach by generating random C programs and comparing the output of several compilers.

Although the generator is parametrized by a notion size, they report best results (more bugs found) with large test cases (8K-16K tokens) and rely on manual techniques to reduce the size for bug reporting.

Danvy has shown how to constructively derive the SECD machine from a denotational semantics for the lambda-calculus by employing successive refinements given as standard program transformations [4]. The approach allows re-constructing alternative machines by performing the transformations in a different order. Because the emphasis is foundational, this work considers only the pure λ-calculus omitting primitive values, conditionals and let-expressions.

5 Discussion and further work

This paper presented a case-study of using the QuickCheck library for testing correctness of a simple functional language implementation. We have also include some experimental evidence for the effectiveness of this methodology in generating insightful counterexamples. For pedagogical reasons we have developed this approach for a simple SECD-like implementation. Note, however, that term generation and shrinking are independent of the particular abstract machine and would be applicable to testing other implementations.

We have not attempted to collect evidence as to whether the use of property-based testing was considered beneficial. This was mainly because, at the time of writing, the course is still underway. Furthermore, the low number of students involved would limit the statistical relevance of this data.

A more ambitious research direction work would be to scale this methodology to testing compilers for real languages like Haskell or O'Caml. A first obstacle is the absence of completely formal semantics for most real languages. This is not however, an unsurpassable difficulty: one could employ the approach used in [9] and compare the output of two compilers or the optimized binary against the non-optimized one within a single compiler. We also conjecture that a partial semantics based on rewrite rules should suffice to implement the shrinking strategy for a reasonable language subset. Another important requirement for testing realistic languages is the need to deal with runtime exceptions (e.g. division by zero or pattern-matching failure). Moreover, this raises the issue how precisely should one compare exceptions. We believe that the imprecise exception semantics proposed in [6] provides a starting point for this.

References

[1] T. Arts, J. Hughes, J. Johansson, and U. Wiger. Testing telecoms software with Quviq QuickCheck. In *Erlang Workshop*, pages 2–10, 2006.

[2] K. Claessen and J. Hughes. QuickCheck: a lightweight tool for random testing of Haskell programs. In *ICFP*, pages 268–279, 2000.

[3] K. Claessen, C. Runciman, O. Chitil, J. Hughes, and M. Wallace. Testing and tracing lazy functional programs using quickcheck and hat. In *Advanced Functional Programming*, pages 59–99, 2002.

[4] O. Danvy. A rational deconstruction of Landin's SECD machine. In C. Grelck, F. Huch, G. Michaelson, and P. W. Trinder, editors, *IFL*, volume 3474 of *Lecture Notes in Computer Science*, pages 52–71. Springer, 2004.

[5] P. Henderson. *Functional Programming: Application and Implementation.* Prentice-Hall International, 1980.

[6] S. Peyton Jones, A. Reid, F. Henderson, T. Hoare, and S. Marlow. A semantics for imprecise exceptions. In *Proceedings of the ACM SIGPLAN 1999 conference on Programming language design and implementation*, PLDI '99, pages 25–36, 1999.

[7] P. Wadler. Monads for functional programming. In *Advanced Functional Programming*, pages 24–52, 1995.

[8] G. Winksel. *The Formal Semantics of Programming Languages: An Introduction.* Foundations of Computing. MIT Press, 1993.

[9] X. Yang, Y. Chen, E. Eide, and J. Regehr. Finding and understanding bugs in C compilers. In *PLDI*, pages 283–294, 2011.

A SECD compiler and interpreter

```
module SECD where
data Value = Prim Int | Closure Code Env

type Stack = [Value]
type Env   = [Value]
type Code  = [Instr]
type Dump  = [(Stack,Env,Code)]

data Instr = LDC Int               -- load constant
           | LD Int                -- load variable
           | ADD                   -- addition
           | SEL [Instr] [Instr]   -- select on zero
           | JOIN                  -- close branch
           | LDF [Instr]           -- load a closure
           | AP                    -- apply
           | RTN                   -- return
           | STOP                  -- finished

-- machine configuration
type SECD = (Stack, Env, Code, Dump)

-- single-step transition
execute :: SECD -> SECD
execute (stack, env, LDC n:code, dump)
    = (Prim n:stack, env, code, dump)
execute (Prim v2:Prim v1:stack, env, ADD:code, dump)
    = (Prim (v1+v2):stack, env, code, dump)
execute (stack, env, LD k:code, dump)
    = let v = env!!k
      in (v:stack, env, code, dump)
execute (stack, env, LDF code':code, dump)
    = (Closure code' env:stack, env, code, dump)
execute (arg:(Closure code' env'):stack, env, AP:code, dump)
    = ([], arg:env', code', (stack,env,code):dump)
execute (v:stack, env, RTN:code, (stack',env',code'):dump)
    = (v:stack', env', code', dump)
execute (Prim n:stack, env, SEL code1 code2:code, dump)
    | n==0      = (stack, env, code2, ([],[],code):dump)
    | otherwise = (stack, env, code1, ([],[],code):dump)
execute (stack, env, JOIN:code, (_,_,code'):dump)
    = (stack, env, code', dump)
execute (stack, env, STOP:code, dump)
    = (stack, env, [], dump)

-- many-step transition function from initial state
run :: Code -> Value
run code = v
```

```
    where states = iterate execute ([],[],code++[STOP],[])
          trace = takeWhile (not.final) states
          final (s, e, c, d) = null c
          (v:[], _, _, _) = last trace

-- translation from Fun programs to SECD code
compile :: Term -> [Ident] -> Code
compile (Const n) sym = [LDC n]
compile (Var x) sym
    = case elemIndex x sym of  -- index of x
        Nothing -> error ("unbound var: "++show x)
        Just k -> [LD k]
compile (Lambda x e) sym
    = [LDF (compile e (x:sym) ++ [RTN])]
compile (App e1 e2) sym
    = compile e1 sym ++ compile e2 sym ++ [AP]
compile (If e1 e2 e3) sym
    = compile e1 sym ++
      [SEL (compile e2 sym ++ [JOIN])
           (compile e3 sym ++ [JOIN])]
compile (Let x e1 e2) sym
    = compile (App (Lambda x e2) e1) sym
compile (e1 :+ e2) sym
    = compile e1 sym ++ compile e2 sym ++ [ADD]
```