Errors for the Common Man

Hiding the unintelligable in Haskell

Matthew Sackman and Susan Eisenbach

Imperial College, London,
{ms, sue}@doc.ic.ac.uk

Abstract. If a library designer takes full advantage of Haskell's rich type system and type-level programming capabilities, then the resulting library will frequently inflict huge and unhelpful error messages on the library user. These error messages are typically in terms of the library and do not refer to the call-site of the library by the library user, nor provide any guidance to the user as to how to fix the error.
The increasing appetite for programmable type-level computation makes this a critical issue, as the advantages and capabilities of type-level computation are nullified if useful error messages cannot be returned to the user.
We present a novel technique that neatly side-steps the default error messages and allows the library programmer to control the generation of error messages that are statically returned to the user. Thus with this technique, there is no longer any drawback to using the full power of Haskell's type system.

1 Introduction

Haskell's rich type system allows type-level computation: using the type system as a programming environment. This creates many possibilities, for example being able to enforce rich and complex invariants statically, thus removing the need for either an additional preprocessor or run-time checks.

In Haskell, type-level computation typically requires the type checker to find an instance of a type class which satisfies certain constraints. If no such type class can be found then an error message is generated. It indicates the type class instance that was required, and the location within the code that requested such a type class instance; so far, so good.

When a type class instance is found, it itself can require instances of other type classes, thus a tree of constraints is constructed. The problem that arises is that a constraint that cannot be satisfied may have been requested by a type class instance that is a long way from the function the library user called from their code, and consequently the error message is unrelated to the user's code. Thus the error message is unhelpful and confusing to the user.

For example, consider type-level booleans, and a type class expressing the *not* relation:

$$\textbf{data}\ \mathit{True} = \mathit{TT}$$
$$\textbf{data}\ \mathit{False} = \mathit{FF}$$

$$\textbf{class}\ \mathit{Not}\ x\ y$$
$$\textbf{instance}\ \mathit{Not}\ \mathit{True}\ \mathit{False}$$
$$\textbf{instance}\ \mathit{Not}\ \mathit{False}\ \mathit{True}$$

Given this type class, we could write a function that requires two arguments to be in a *not* relationship:

$$f :: (\mathit{Not}\ x\ y) \Rightarrow x \rightarrow y \rightarrow ...$$

If the library user then calls $f\ \ TT\ \ TT$ then they will get a `No instance for (Not True True)` error. This may make sense to the user and they may realise how to fix this problem. But this constraint could be added by some other type class instance buried deep within a library:

$$\textbf{instance}\ (\mathit{Not}\ x\ y, ...) \Rightarrow \mathit{Foo}\ w\ x\ y\ z\ \textbf{where}\ ...$$

$$\textbf{instance}\ (\mathit{Foo}\ b\ c\ d\ \mathit{False}, ...) \Rightarrow \mathit{Bar}\ a\ b\ c\ d\ \textbf{where}\ ...$$

$$f :: (\mathit{Bar}\ a\ b\ c\ d) \Rightarrow a \rightarrow b \rightarrow c \rightarrow d \rightarrow ...$$

Thus we have an instance of the type class *Foo* which, if it is selected by the type checker, adds the requirement that two of its parameters, x and y are in a *not* relation; and an instance of the type class *Bar* which, if selected adds the requirements that three of its arguments and the constant type *False* form an instance of *Foo*. If the user, at the public API level of the library supplied arguments to the function f that eventually resulted in x and y within the *Foo* instance *not* being in a *not* relation then the same `No instance for...` error message will be generated. But the user almost certainly won't be able to understand what it is they did to cause this error to be generated, and may not even be aware of the *Not* type class. Furthermore, the line number and file indicated in the error message will indicate the location of this particular *Foo* instance, which will be of no value to the user. When functional dependencies or associated data types are involved, the error messages can exponentially lose meaning and the user is only able to conclude that *something* is wrong.

- We present a technique that allows the library writer to control the generation of error messages and present the library user with as useful as possible error reports. We do not require any modifications to compilers or any prepossessing steps. Only library code must be written incorporating this technique.
- The technique is highly flexible and allows the library writer to cull the tree of constraints as early or late as they wish in order to ensure that the error messages are accurate and contain as much helpful information as possible.
- We demonstrate our technique with a running example that builds up a library of type-level boolean operators and list functions. These are, in our experience, very common requirements for type-level computation, and as such form a core of functionality for which meaningful error messages must be provided.

2 Boolean Mayhem

If we are to wrestle generation of error messages away from the type checker, then we must stop the type checker rejecting programs for which we want to provide more helpful error messages. This means the type checker must accept the program, and then we indicate to the library user, through some other means, that their program is bad and provide an error message. We do not intend to accept all programs: there are many errors that the user can make for which GHC provides perfectly good error messages. We only wish to accept bad programs (which would otherwise be rejected as a result of complex invariants being expressed through type-level programming) for which GHC does not provide useful error messages.

The idea is to manually track errors through type parameters and then give the user a means of accessing the contents of errors, should they occur. This will be achieved through duplication of a library's public API such that every exported function has a *debug* variant which exists to expose any error messages at runtime.

We start by introducing some new types which we are going to use to track errors within the type system. We have two error types and a type class that allows us to create and modify the error message within either type. Of course, in the case of *NoError*, there is no error message, so modification does nothing.

```
data NoError = NoError       deriving (Show)
data Error   = Error String deriving (Show)

class UpdateErrorMessage err where
  updateErrorMessage :: (String → String) → err → err
  blankError :: err

instance UpdateErrorMessage NoError where
  updateErrorMessage _ err = err
  blankError = NoError

instance UpdateErrorMessage Error where
  updateErrorMessage f (Error str) = Error (f str)
  blankError = Error ""
```

There are several ways of using these error types and we, as a library developer, shall explore some of these. The simplest is to add a single additional type parameter where necessary. For our *Not* type class:

```
class Not err x y | x y → err
instance Not NoError True False
instance Not NoError False True
instance Not Error True True
instance Not Error False False
```

Thus we have provided all possible instances within our Boolean types and so we will no longer get the `No instance for...` message (assuming Boolean types are used as the type parameters) from constraints requesting instances of the *Not* type class. We have also introduced a functional dependency [1] which allows

the type checker to determine the error type from the other type parameters.[1] This is highly desirable as without the functional dependency, we would not be able to compute whether an error has occurred or not.

At this point, no error message has been created: all we know is whether or not the Boolean type parameters presented to the *Not* type class have resulted in an error. However, this is the *only* information we have at this stage and so creating an error message here would be no different from the `No instance for...` error messages that we are trying to avoid. No, the place to create the error messages is in the type class instances that add constraints involving our *Not* type class.

In our *Foo* type class instance, we can now extract whether an error has occurred and return this back up the tree of the type class constraints. In order to provide a useful error message, we need a function which can supply the text into the error type, and the functional dependency also is repeated to indicate that from all the non-error type parameters of the type class, the error type parameter can be derived.

```
class Foo err w x y z | w x y z → err where
  foo :: w → x → y → (z, err)

instance (Not err x y, UpdateErrorMessage err) ⇒
         Foo err w x y Int where
  foo w x y = (0, err)
    where
      err = updateErrorMessage (const ("The instance Foo required "
                ++ "the type parameters x and y to be in a not "
                ++ "relationship, but they were not")) blankError
```

By using the *UpdateErrorMessage* type class, we can create and modify the error message generically and not care about the case where no error has occurred. In this example, the error message isn't a spectacular improvement over the default `No instance for...` error message, but the point is that the error message is now in the programmer's control, and for a type class which had some real purpose (rather than our fictional *Foo* type class), it is possible to provide more helpful messages.

This error type parameter then needs to propagate up through all *parent* type class instances until it emerges at the public API of our library. At this point, as a library developer, we must now provide two functions, rather than one. The first will enforce that the program can only be accepted if no type-checking errors occur; and should an error occur, we will be back to GHC's default error messages:

```
f :: (Foo NoError w x y z) ⇒ ...
```

The other version of the function does not restrict the error type parameter at all, and simply returns it. This allows the error to be inspected by the library user:

```
f_debug :: (Foo err w x y z) ⇒ ... → err
```

[1] This can equally be achieved through an associated data type [2–4].

The expected development pattern is that as the user makes use of the library API they use the *debug* API and iteratively develop their application, switching to the non-*debug* API as each section of their application is finished. As soon as they hit an error which they don't understand (e.g. an error which is the result of type-level programming and thus causes GHC to produce an unhelpful error message), they can switch back to the *debug* API and inspect the library author's manually constructed error messages.

This particular pattern of threading type errors upwards works well when errors only occur in the leaves of the constraint tree. However, frequently errors can occur throughout the constraint tree and this particular pattern does not deal well with this case. For example, a single type class instance can add several constraints, each of which can error:

instance $(A\ err\_a\ w\ y, B\ err\_b\ y\ z, C\ err\_c\ w\ x) \Rightarrow$
$Foo\ err\_f\ w\ x\ y\ z$ **where** ...

The problem here is how to construct err_f: which of the sub-errors do we use, or if we wish to combine them, how should that be done? Furthermore, if a type class's instances are recursive, how do we terminate recursion early as soon as an error is encountered? Our solution is to allow the error type parameter to flow both up and down the tree of type class constraints.

3 The errors of Lizst

As lists are widely used values in Haskell, it should be no surprise that they are also very useful at the type-level [5]. From the humble list, we can build all manner of type-level data structures, from maps and sets to entire type-level domain specific languages. One of the most common operations to perform on a list (at least, in our experience of using type-level lists) is to check whether an element is a member of a list or not. The type class instances for this recurse on the structure of the list and stop when they find a matching list element. If they fail to find such a list element, then we are left with the standard error message with parameters that look like we're trying to find an element in an empty list (which indeed, the recursion has resulted in). This is often a baffling error message: particularly when several lists are in use and all you know is that you fell off the end of one of them without finding what you wanted; so providing useful error messages in this case is particularly helpful. The basic implementation of a type level list and of an *Elem* type class are shown below:[2]

data $Nil = Nil$ **deriving** $(Show)$
data $Cons\ val\ nxt = Cons\ val\ nxt$ **deriving** $(Show)$

[2] The *Elem* type class can only work with both the *Overlapping instances* and *Undecidable instances* flags enabled for GHC. Without the *Overlapping instances* flag, any element and list that matched the second instance would also match the third and would be rejected by the compiler, and without the *Undecidable instances* flag, the functional dependency is rejected as the instances fail the *Coverage Condition*: the *res* type (which is either *True*, *False*, or *res* in the recursive case) do not appear in the left hand side of the functional dependency.

class *Elem lst val res* | *lst val* → *res*

instance *Elem Nil val False*

instance *Elem* (*Cons val nxt*) *val True*

instance (*Elem nxt val res*) ⇒ *Elem* (*Cons val′ nxt*) *val res*

Although the type class will never fail if the result is left unspecified, the typical use case is to assert that an element either *is*, or *is not* a member of a particular list. For example:

instance (*Elem lst Int True*) ⇒ *Foo lst x y Int* **where** ...

asserts that within the list *lst* we need to find an *Int* type. When this fails to be the case, the default error message is of the form `No instance for Elem Nil val True`; providing a more helpful error message higher up in the tree of constraints is clearly very valuable.

From the list and the element, there are four possible outcomes: the cartesian product of the *actual* result of the search (i.e. the element *may*, or *may not* be found in the list), and *desired* outcome of the search (i.e. the element *may* or *may not* have been wanted to be found as indicated by the type class parameter *res*). We need to be able to express that *if the result of the search is not what was desired, then raise an error*. So we introduce a type-level *If* type class, and then handle the error case explicitly. The *If c x y z* type class should be read as $z =$ **if** c **then** x **else** y.

```
class If c x y z | c x y → z
instance If True x y x
instance If False x y y
instance (Elem lst Int isElem, If isElem NoError Error err,
          Show lst, UpdateErrorMessage err) ⇒
         Foo err lst x y Int where
  foo lst x y = (0, err)
    where
      err = updateErrorMessage (const ("I was expecting to find"
              ++ " an Int in the list " ++ (show lst)
              ++ " but didn't.")) blankError
```

Within the *foo* function body, we can create a blank error as before, and provide a more helpful message: now we can inspect what the list was *before* we started recursing on it in our search for the *Int*. This alone makes a big difference to debugging and understanding why a program is failing to type check. Again, we would use the same machinery as before at the public API level with a *"normal"* function which statically permits no errors and a *"debug"* function which statically permits an error to occur and allows the library user to inspect it dynamically. As before, all we have done is identified where within a library a confusing and unhelpful error message could occur should the library user make a mistake, and ensured that such a mistake will not (when using the *"debug"* API) cause GHC to statically reject the program. Instead the program is statically accepted and permits the user to dynamically access a more helpful error message.

Another common operation to perform on a list is to *map* one list to another. This too is desirable for type-level lists. The difficulty is then what to do if the function transforming each element of the list generates an error. We could abstract to a general *Map* type class (e.g. see [5]), but for clarity, we instead inline the transformation function type class:

```
class C a b | a → b
class MapWithC lstIn lstOut | lstIn → lstOut
instance MapWithC Nil Nil
instance (MapWithC nxt nxt', C val val') ⇒
  MapWithC (Cons val nxt) (Cons val' nxt')
```

The type class C should provide an *err* type parameter which would indicate whether it encountered an error. We could simply use that type parameter in the instance head of the recursive instance of *MapWithC*, but really we want the error as a *value* from C as it should contain useful information: the transformation function should be able to provide more accurate information about what went wrong. Then, having encountered an error, we want to stop recursion as soon as possible to ensure that we get the first error message out and that it doesn't get lost under a subsequent error. In order to do this, we have added *errIn* and *errOut* type parameters to the *MapWithC* type class and demanded that both type classes have real member functions:

```
class C err a b | a → b err where
  c :: a → (b, err)
class MapWithC errIn errOut lstIn lstOut | errIn lstIn → errOut lstOut
  where
    mapWithC :: lstIn → errIn → (lstOut, errOut)
instance MapWithC NoError NoError Nil Nil where
  mapWithC lst err = (lst, err)
instance MapWithC Error Error lst lst where
  mapWithC lst err = (lst, err)
instance (MapWithC errIn errOut nxt nxt', C errIn val val') ⇒
  MapWithC NoError errOut (Cons val nxt) (Cons val' nxt') where
    mapWithC (Cons val nxt) _ = ((Cons val' nxt'), errOut)
      where
        (val', errIn) = c val
        (nxt', errOut) = mapWithC nxt errIn
```

Thus recursion is permitted only when no error has been encountered so far (the third instance), and as soon as an error is encountered, the error is returned and no more recursion is performed (the second instance). Finally, the absence of an error is only confirmed once we have successfully reached the end of the list (the first instance). Using this *MapWithC* type class is no different to before, only now we also need to supply the initial *NoError* parameter:

```
instance (MapWithC NoError err lst lst') ⇒ Foo err lst x y lst' where
  foo lst x y = mapWithC lst NoError
```

This pattern of passing the error through an incoming and outgoing type parameter is also useful when more than one constraint can generate errors; it allows us to express which errors should take precedence, which in turn allows only the most useful errors to be emitted. For example:

```
instance (MapWithC NoError err_a w x, MapWithC err_a err_b x y) ⇒
  Foo err_b w x y z where ...
```

However, there are also cases where we might like to combine errors that come from disjoint constraints. In general, it is a challenge to get the balance right between overloading the library user with error information and providing sufficient information such that the user isn't needlessly round-tripping between the interpreter and their editor. As such there are certainly cases where presenting two or more independent error messages is desirable. For example:

```
instance (MapWithC NoError err_a w x, MapWithC NoError err_b y z) ⇒
  Foo err w x y z where ...
```

Here, *err_a* and *err_b* are completely independent (as they are generated by disjoint type parameters) so it would be reasonable to present these error messages together should both *MapWithC* constraints produce an error. Thus we introduce a new type class that allows us to achieve this.

```
class CombineErrorMessages errA errB errC | errA errB → errC where
  combineErrorMessages :: errA → errB → errC

instance CombineErrorMessages NoError NoError NoError where
  combineErrorMessages _ _ = NoError

instance CombineErrorMessages NoError Error Error where
  combineErrorMessages _ err = err

instance CombineErrorMessages Error NoError Error where
  combineErrorMessages err _ = err

instance CombineErrorMessages Error Error Error where
  combineErrorMessages (Error errA) (Error errB)
    = Error (errA ++ ('\n' : errB))
```

And finally we can now combine the independent error messages from our *Foo* instance:

```
instance (MapWithC NoError err_a w x, MapWithC NoError err_b y z,
          CombineErrorMessages err_a err_b err) ⇒
  Foo err w x y z where ...
```

These error messages can be further improved by using the *updateErrorMessage* function to annotate each sub-error message with information about which type parameters led to which errors.

4 Monadic Fail

Haskellers love monads and so we have extended this technique so that it will work with monads too. For this to work, we need to redefine the standard Haskell

Monad type class into a type indexed monad so that it can carry our type parameters indicating the error state:[3]

```
class Monad m where
  (>>)  :: m x y a → m y z b → m x z b
  (>>=) :: m x y a → (a → m y z b) → m x z b
  return :: a → m x x a
```

Thus the monad has gained two extra type parameters, the first can be thought of as the *incoming* error state, and the second can be thought of as the *outgoing* error state. We can create an instance of this *Monad* type class:

```
newtype ErrState x y a = ErrState{ runErrState :: x → (a, y) }

instance Monad ErrState where
  f >> g   = ErrState (λx → let (_, y) = runErrState f x
                            in runErrState g y)
  f >>= g  = ErrState (λx → let (a, y) = runErrState f x
                            in runErrState (g a) y)
  return a = ErrState (λx → (a, x))
```

This should look and feel very much like the normal Haskell *State* monad instance, but with one change that allows the type of the state to vary. We now require that for functions built out of these monadic combinators, the first error that occurs should be the error reported and that we have a means to report a error. This latter part is really an extended *return* function. So we introduce a new type class, *ErrorReturn* which through a functional dependency states that from the error being reported and the previous error we can determine the consequent error type (this is the same functional dependency as in the *CombineErrorMessages* type class). It has just one function, *returnErr*, which takes a result (as normal for *return*) and an error (the *selfErr* parameter) and lifts these into our monad.

```
class (Monad m) ⇒
  ErrorReturn m selfErr errIn errOut res | selfErr errIn → errOut where
    returnErr :: res → selfErr → m errIn errOut res

instance ErrorReturn ErrState Error NoError Error res where
    returnErr res errOut = ErrState (λ_ → (res, errOut))

instance ErrorReturn ErrState Error Error Error res where
    returnErr res _      = ErrState (λerrIn → (res, errIn))

instance ErrorReturn ErrState NoError errOut errOut res where
    returnErr res _      = ErrState (λerrIn → (res, errIn))
```

Thus we have supplied the only instances that are required which ensure that the error supplied to *returnErr* (the *selfErr* parameter) is only used when going from a *NoError* state to an *Error* state. In all other cases, the existing error

[3] In this presentation we redefine the standard Haskell *Monad* type class as shown, which is legal Haskell 98 but isn't accepted by GHC prior to version 6.10. For versions of GHC prior to 6.10, we have to rename the monad functions and abandon **do**-syntax. Using **do**-syntax makes the presentation more familiar and simpler.

state is used. The library developer now only needs to switch from using *return* to using *returnErr* when they wish to present an error. For example:

```
foo :: ∀ selfErr x y errIn errOut .
      (Not selfErr x y, ErrorReturn ErrState selfErr errIn errOut Int,
       UpdateErrorMessage selfErr) ⇒
      x → y → ErrState errIn errOut Int
foo x y = returnErr 5 (updateErrorMessage
              (const "a useful error message") (blankError :: selfErr))
```

Thus the function *foo* will supply an error message if and only if the x and y parameters to the *Not* type class result in an error. The function *foo* can be combined with others inside **do**-blocks as normal. Finally, we provide a means to extract the error, should one occur from such a function. Variations on this would be the bulk of the *"debug"* API for a library.

```
extractError :: ErrState NoError err res → err
extractError f = snd (runErrState f NoError)
```

We simply run the monadic function, supplying it with an initial *NoError* state and discarding all results but for the error returned to us.

5 Related Work

This work was born out of the difficulties of using a large and complex library which made extensive use of type-level programming within Haskell, and frequently resulted in error messages of literally thousands of lines. Such error messages made the library completely unusable for anyone but the original authors and a solution was repeatedly demanded. To our knowledge, no one has proposed such a solution before or considered how to escape from GHC's default error messages when desirable.

Frequently, the limit of useful default error messages is reached when using GADTs [6] which regularly feature in domain specific languages as a means to enforce some level of type soundness [7]. There are other projects such as the HaskellDB project [8] which make use of large and complex type signatures for which we believe our technique of providing custom error messages would be beneficial and appropriate.

Sadly, much research on dependently typed languages [9–11] has failed so far to consider issues such as error messages; indeed for many of these languages, popular requests on their wikis and websites are for better error messages. It seems that there is a significant research potential in investigating how to enable programmers to create error messages when type checking fails in dependently typed languages. As type systems become ever more powerful and expressive, so rises the importance of useful error messages. As type systems effectively become programming languages, it is utterly unreasonable to expect users of a library to read and understand both the runtime code and the compile-time code of the library in order to try and decipher an error message.

6 Conclusions and Future Work

Type-level programming offers the programmer increased levels of expression and permits easy extension of the type system. Whilst the current interest in dependently typed languages and extensions of lesser type systems such as Haskell's has stimulated discussion and development of type-level programming, there is still a barrier to adoption. One of these barriers we solve with this work: namely that an error that occurs as a result of type-level programming must have a useful error message attached to it. Without this, the advantages of type-level programming are rendered void as development against rich libraries becomes almost a matter of guess-work in order to decipher unhelpful error messages and debug faulty programs.

The technique presented here requires that libraries be written with our technique in mind, and must adopt changes to their APIs to make debugging and inspection of manually created error messages possible. Furthermore, our technique requires some additional type class instances to be written by library authors and demands extra type parameters in order to track error states. However, in our experience, this does not amount to a significant overhead for the library and we have presented means (for example in extending monads) to minimise these requirements.

One of the most obvious deficiencies with our technique is that no line numbers or positional information is presented in the error messages. This is because that information isn't available when creating the error message. It would be an interesting and useful extension to modify GHC in order to be able to capture such information in error messages. With the *"debug"* API, users now not only have to have programs that type check but then must also remember to run the *"debug"* function variant in order to inspect the error. This is an unfortunate additional step which the user must remember to do. We would like to investigate whether using techniques such as Template Haskell [12] would allow these checks to be merged with the general type-checking process and thus eliminate this extra step from the user. On the other hand, the current additional step is present with test-driven development using tools such as QuickCheck [13] where after successful type checking, one is expected to run the suite of tests in order to verify invariants of the program which are not captured in the type system.

References

1. Duck, G.J., Peyton Jones, S.L., Stuckey, P.J., Sulzmann, M.: Sound and Decidable Type Inference for Functional Dependencies. In: ESOP. (2004) 49–63
2. Chakravarty, M.M.T., Keller, G., Peyton Jones, S., Marlow, S.: Associated types with class. SIGPLAN Not. **40**(1) (2005) 1–13
3. Chakravarty, M.M.T., Keller, G., Peyton Jones, S.: Associated type synonyms. SIGPLAN Not. **40**(9) (2005) 241–253
4. Tom Schrijvers, Simon Peyton Jones, M.S., Chakravarty, M.: Towards open type functions for haskell. In: Proceedings of the symposium on Implementation and Application of Functional Languages. Lecture Notes In Computer Science, Springer-Verlag (2007)

5. Kiselyov, O., Lämmel, R., Schupke, K.: Strongly typed heterogeneous collections. In: Haskell '04: Proceedings of the ACM SIGPLAN workshop on Haskell, ACM Press (2004) 96–107
6. Peyton Jones, S.L., Vytiniotis, D., Weirich, S., Washburn, G.: Simple unification-based type inference for GADTs. In: ICFP. (2006) 50–61
7. Rhiger, M.: A foundation for embedded languages. ACM Trans. Program. Lang. Syst. **25**(3) (2003) 291–315
8. Leijen, D., Meijer, E.: Domain specific embedded compilers. SIGPLAN Not. **35**(1) (2000) 109–122
9. Norell, U.: Towards a practical programming language based on dependent type theory. PhD thesis, Chalmers University of Technology and Gteborg University (2007)
10. Sheard, T.: Languages of the future. SIGPLAN Not. **39**(12) (2004) 119–132
11. McBride, C.: The Epigram Prototype: a nod and two winks. http://www.e-pig.org/downloads/epigram-system.pdf (April 2005)
12. Sheard, T., Peyton Jones, S.: Template metaprogramming for Haskell. In Chakravarty, M.M.T., ed.: ACM SIGPLAN Haskell Workshop 02, ACM Press (October 2002) 1–16
13. Claessen, K., Hughes, J.: QuickCheck: a lightweight tool for random testing of Haskell programs. In: ICFP '00: Proceedings of the fifth ACM SIGPLAN international conference on Functional programming, ACM Press, New York, NY, USA (2000) 268–279