

1 Introduction

This is a literate Haskell file that contains several examples of testing *Prelude* functions for minimal strictness. This file is intended to give an introduction to Sloth and it is used as unit test for the implementation. You can generate a pdf from this file using `lhs2tex`¹ by invoking `lhs2TeX -o Examples.tex Examples.lhs --poly & pdflatex Examples.tex`.

2 Booleans

First we check whether the Boolean conjunction (\wedge) is minimally strict. To use Sloth we have to import the module *Test.Sloth*.

```
import Test.Sloth
```

The function *strictCheck* takes a function and an integer as arguments. The integer specifies the maximal size of the test cases that are considered. The size of a test case (a value of a certain type) is the number of constructors it contains.

```
conjunctionExample = strictCheck ( $\wedge$ ) 10
```

```
Main> conjunctionExample  
Finished 5 tests.
```

As *strictCheck* does not state any counter-examples (\wedge) is probably minimally strict. In this case the absence of a counter-example even proves that (\wedge) is minimally strict as the domain of (\wedge) has only finitely many elements and all elements have a size smaller than 10.

In contrast to Sloth, a tool called *StrictCheck* [1, 2] that has been the inspiration for Sloth identifies (\wedge) as unnecessarily strict. Sloth uses the same approach as *StrictCheck* to check whether a function is unnecessarily strict with respect to a specific argument. But, in contrast to *StrictCheck*, Sloth restrains the number of test cases to propose sequential functions only. In contrast, *StrictCheck* proposes the parallel *and* as it is less strict than (\wedge).

Next we check the Boolean instance of the function (\leq).

```
leExample = strictCheck (( $\leq$ ) :: Bool  $\rightarrow$  Bool  $\rightarrow$  Bool) 10
```

```
Main> leExample  
2: \False _ -> True  
Finished 7 tests.
```

Sloth states that this function is unnecessarily strict. It presents a test case and the proposed result of a minimally strict implementation of (\leq). The underscore represents \perp (the smallest element of the corresponding cpo) which denotes an error or a non-terminating expression. The test case `\False _ -> True` denotes that a minimally strict implementation of (\leq) yields *True* if it is applied to *False*

¹<http://hackage.haskell.org/package/lhs2tex>

and \perp . If we replace the sub-term that is highlighted red by \perp we get the current result of the function. That is, we have $(\leq) \text{ False } \perp \equiv \perp$.

Besides *strictCheck*, Sloth provides the function *check* that additionally takes a configuration as first argument. For example, we have $\text{strictCheck} = \text{check defaultConfig}$ where *defaultConfig* is a configuration provided by Sloth. A configuration is a record that, among others, contains a Boolean field called *detailed* that specifies whether Sloth is supposed to present detailed informations about test cases. For example, the following application yields detailed information about the counter-example for (\leq) .

```
detailedLeExample =
  check (defaultConfig { detailed = True })
  ((\leq) :: Bool -> Bool -> Bool) 10
```

```
Main> detailedLeExample
2: Argument(s): False _
Current Result: _
Proposed Result: True
Finished 7 tests.
```

The function $(\leq) :: \text{Bool} \rightarrow \text{Bool} \rightarrow \text{Bool}$ is unnecessarily strict because it is defined by means of $\text{compare} :: \text{Bool} \rightarrow \text{Bool} \rightarrow \text{Ordering}$, which checks whether its arguments are equal. Therefore, (\leq) evaluates both arguments even if its first argument is *False*. There are two possible implementations that are less strict than the current implementation of (\leq) . One implementation satisfies $(\leq) \text{ False } \perp \equiv \text{True}$ and the other satisfies $(\leq) \perp \text{ True} \equiv \text{True}$. Note that we cannot satisfy both without employing non-sequential features like concurrency. If there are several ways of defining a less strict implementation Sloth selects the one that uses a pattern matching order that is most similar to the pattern matching order of the original implementation. For example, $(\leq) :: \text{Bool} \rightarrow \text{Bool} \rightarrow \text{Ordering}$ first performs pattern matching on its first argument and, therefore, Sloth proposes a function which satisfies $(\leq) \text{ False } \perp \equiv \text{True}$.

Next we check the function $\text{and} :: [\text{Bool}] \rightarrow \text{Bool}$. Instead of using the function *strictCheck* we use *verboseCheck* which additionally presents successful test cases.

```
andExample = verboseCheck and 3
```

```
Main> andExample
1: \_ -> _
2: \[] -> True
3: \(_:_) -> _
4: \(False:_) -> False
5: \(True:_) -> ?
6: \(True:[]) -> True
7: \(True:_:_) -> ?
Finished 7 tests.
```

Sloth presents seven test cases and the results of *and* for the corresponding arguments. As the application $\text{and} (\text{False} : \perp)$ yields a total result *and* cannot

be too strict for this test case. By monotonicity *and* furthermore cannot be too strict for more defined arguments. Therefore, Sloth does not consider test cases that are more defined than $False : \perp$.

The question marks in test cases five and seven state that Sloth cannot decide whether the function is minimally strict for these test cases or not. To check whether a function yields the correct result for a test case Sloth considers the behaviour of the function for more defined arguments. In the standard configuration Sloth yields a question mark if it considers less than two more defined arguments.

The record field called *minInfSize* specifies the number of more defined arguments Sloth has to consider. The following checks the function *and* and demands Sloth to use 20 more defined arguments.

```
andExample20 = check (verboseConfig { minInfSize = 20 }) and 3
```

```
Main> andExample20
1: \_ -> _
2: \[] -> True
3: \(_:_)-> _
4: \(False:_)-> False
5: \(True:_)-> ?
6: \(True:[])-> True
7: \(True:_:_)-> ?
Finished 7 tests.
```

Surprisingly Sloth yields the same results as before. The parameter *minInfSize* only affects test cases which Sloth is uncertain about. That is, in the test cases one to four Sloth definitely knows that the behaviour of *and* is minimally strict without considering 20 more defined test cases.

If we set the parameter *minInfSize* to one the output of Sloth changes.

```
andExample1 = check (verboseConfig { minInfSize = 1 }) and 3
```

```
Main> andExample0
1: \_ -> _
2: \[] -> True
3: \(_:_)-> _
4: \(False:_)-> False
5: \(True:_)-> True
6: \(True:[])-> True
7: \(True:_:_)-> ?
Finished 7 tests.
```

First of all Sloth still yields a question mark for test case seven. In this case there are no values more defined than the test case and Sloth always prints a question mark in this case. In contrast, test case five now is a potential counter-example. Sloth distinguishes **potential** counter-examples and **definite** counter-examples. If a counter-example is potential it might be no counter-example if we consider test cases of larger sizes. For example, if we increase the size of the previous test from three to four Sloth does not identify $True : \perp$ as counter-example. We will return to the difference between potential and definite counter-examples later.

3 Polymorphism

To check a polymorphic function we use the monomorphic instance where all type variables are instantiated with the opaque data type A that is provided by Sloth. Sloth uses symbolic values as test cases of type A . For example, if we check a function whose argument type is $[A]$ Sloth generates test cases \perp , $[]$, $a : \perp$, $a : []$, $a : b : \perp$, \dots where a and b are representing arbitrary values of equal type. To test a function that takes an argument of type $[A]$ Sloth in fact replaces elements of type A by distinct integers, that is, it generates test cases \perp , $[]$, $0 : \perp$, $0 : []$, $0 : 1 : \perp$. These integers are displayed as variables as they actually represent arbitrary values (for more details see [4]).

The following application checks whether the polymorphic identity is minimally strict.

```
idExample = verboseCheck (id ::  $A \rightarrow A$ ) 10
```

```
Main> idExample  
1: \a -> a  
Finished 1 tests.
```

We employ properties of polymorphic functions that are guaranteed by free theorems to increase the efficiency of testing polymorphic functions. For example, id is only checked for the test case a , that is, for 0. Note that even a test of the simplest monomorphic instance $id :: () \rightarrow ()$ generates two test cases, namely, \perp and $()$.

In fact, in contrast to $id :: \alpha \rightarrow \alpha$, the function $id :: () \rightarrow ()$ is unnecessarily strict. That is, we may not check the monophonic unit instance to check whether the polymorphic function is minimally strict.

```
unitIdExample = verboseCheck (id ::  $() \rightarrow ()$ ) 10
```

```
Main> unitIdExample  
1: \_ -> ()  
2: \() -> ()  
Finished 2 tests.
```

The function $id :: () \rightarrow ()$ yields $()$ for all total arguments. That is, the function does not have to check its argument. In other words, the function $const ()$ is less strict than $id :: () \rightarrow ()$.

If we check the strict evaluation primitive seq ² Sloth states that it is minimally strict.

```
seqExample = verboseCheck (seq ::  $A \rightarrow A \rightarrow A$ ) 10
```

```
Main> seqExample  
1: \a b -> b  
Finished 1 tests.
```

²The function $seq :: \alpha \rightarrow \beta \rightarrow \beta$ satisfies the laws $seq \perp y \equiv \perp$ and $seq x y \equiv y$ if $x \not\equiv \perp$.

This is quite surprising as *seq* is by definition unnecessarily strict. For example, the function $\lambda_ y \rightarrow y$ is less strict than *seq*. Sloth does not observe that *seq* is unnecessarily strict because it only considers checks the application *seq a b*, that is, *seq* 0 1. If we instantiate the type variables of the type of *seq* by *Bool* the result looks more promising.

```
boolSeqExample = strictCheck (seq :: Bool → Bool → Bool) 10
```

```
Main> boolSeqExample 10
2: \_ False -> False
3: \_ True -> True
Finished 7 tests.
```

The results of Sloth for *A* instances might be wrong if the polymorphic function uses *seq*. The behaviour of Sloth for the *A* type is based on free theorems that fail in the presence of *seq*. It is possible to handle polymorphic functions that use *seq* correctly but we have to check significantly more test cases in this case. Therefore, we expect that polymorphic functions do not use *seq* to keep the number of test cases small (see [4] for more details).

4 Lists

In this chapter we consider functions on lists.

```
appendExample = strictCheck ((+) :: [A] → [A] → [A]) 10
```

```
Main> appendExample
Finished 91 tests.
```

The function $(+)$ is minimally strict. Furthermore note that the number of test cases is quite small if we consider that we check $(+)$ for all pairs of lists up to size ten. This is due to the fact that $(+)$ is polymorphic and cannot be too strict in the polymorphic component.

As examples for minimally strict function we check *tail* and *zip*.

```
tailExample = strictCheck (tail :: [A] → [A]) 100
```

```
Main> tailExample
Finished 201 tests.
```

```
zipExample = strictCheck (zip :: [A] → [A] → [(A, A)]) 100
```

```
Main> zipExample
Finished 199 tests.
```

In contrast to *zip* the function *unzip* is not minimally strict.

```
unzipExample = strictCheck (unzip :: [(A, A)] → ([A], [A])) 3
```

```

Main> unzipExample
1: \_ -> (_,_)
3: \(_:_)-> (_:_,_:_)
Finished 8 tests.

```

The first counter-example states that *unzip* yields a tuple in all cases and, therefore, should yield a tuple even if its argument is \perp . To understand the other counter-example we consider the implementation of *unzip*.

$$\begin{aligned} \text{unzip} &:: [(\alpha, \beta)] \rightarrow ([\alpha], [\beta]) \\ \text{unzip} &= \text{foldr } (\lambda(a, b) \sim (as, bs) \rightarrow (a : as, b : bs)) ([], []) \end{aligned}$$

The second counter-example states that $\text{unzip } (\perp : \perp) = \perp$ while we can have $\text{unzip } (\perp : \perp) = (\perp : \perp, \perp : \perp)$. That is, if we apply *unzip* to a list with at least one element the result is always a tuple where both lists have at least one element. In other words, the functional argument of *foldr* performs pattern matching on its for argument. That is, *unzip* is head-strict although it does not have to be.

If we replace the strict tuple matching of the functional argument of *foldr* by a lazy matching we get a less strict implementation of *unzip*. To satisfy the first counter-example we use a lazy pattern matching by means of a **where** clause and explicitly construct the tuple.

$$\begin{aligned} \text{unzip}' &:: [(\alpha, \beta)] \rightarrow ([\alpha], [\beta]) \\ \text{unzip}' \text{ } xs &= (ys, zs) \\ &\textbf{where} \\ & \quad (ys, zs) = \text{foldr } (\lambda \sim (a, b) \sim (as, bs) \rightarrow (a : as, b : bs)) ([], []) \text{ } xs \end{aligned}$$

If we check this less strict implementation, Sloth does not state any counter-examples.

$$\text{unzip}'\text{Example} = \text{strictCheck } (\text{unzip}' :: [(A, A)] \rightarrow ([A], [A])) 100$$

```

Main> unzip'Example
Finished 599 tests.

```

If we check whether *reverse* is minimally strict, Sloth states a couple of potential counter-examples.

$$\text{reverseExample} = \text{strictCheck } (\text{reverse} :: [A] \rightarrow [A]) 5$$

```

Main> reverseExample
3: \(\a:_)-> _:_
5: \(\a:b:_)-> _:_:
7: \(\a:b:c:_)-> _:_::_
Finished 11 tests.

```

We can confirm that a potential counter-example is indeed a counter-example by checking *reverse* for inputs of larger sizes. If we check *reverse* for polymorphic lists up to size ten all potential counter-examples for size five stay potential counter-examples. As these potential counter-examples may still be no counter-examples, we can verify a potential counter-example by hand. A potential

counter-example is definitely a counter-example if all more defined total inputs lead to results that are at least as defined as the recommended result. For example, consider the first counter-example. For all total inputs that are more defined than $a : \perp$, that is, for all lists with at least one element, the function *reverse* yields a list with at least one element. A list with at least one element is at least as defined as the recommended result $\perp : \perp$. Therefore, the first counter-example is definitely a counter-example.

A less strict implementation of *reverse* is only advantageous in very uncommon use cases. For example, the evaluation of *null (reverse xs)* completely evaluates the spine of *xs* while a minimally strict implementation of *reverse* would only evaluate *xs* to head normal form in this case.

As two more interesting examples we consider the functions *intersperse* and *inits* from *Data.List*³.

```
intersperseExample = strictCheck (intersperse :: A -> [A] -> [A]) 4
```

```
Main> intersperseExample
3: \a (b:_) -> a:_
5: \a (b:c:_) -> b:a:c:_
Finished 7 tests.
```

This nicely demonstrates the power of the symbolic testing of polymorphic functions. For example, the first counter-example does not only state that *intersperse* is too strict if it is applied to a non-terminated list but also that the first element of the list is supposed to be the element of the second argument. The unnecessarily strict implementation of *intersperse* can cause a serious space leak as it is shown in [3].

The implementation of *intersperse* is unnecessarily strict because the function checks whether the list has exactly one element.

```
intersperse :: alpha -> [alpha] -> [alpha]
intersperse _ [] = []
intersperse _ [x] = [x]
intersperse sep (x : xs) = x : sep : intersperse sep xs
```

As the first element of the result list is *x* no matter whether the list has one element or more we can “delay” the pattern matching as follows.

```
intersperse' :: alpha -> [alpha] -> [alpha]
intersperse' _ [] = []
intersperse' sep (x : xs) = x : go xs
  where
    go [] = []
    go (y : ys) = sep : y : go ys
```

The implementation of *inits* is in a similar manner unnecessarily strict.

```
initsExample = strictCheck (inits :: [A] -> [[A]]) 3
```

³We consider the definitions of *intersperse* and *inits* from **base-4.3.1.0** as these functions have been improved in **base-4.4.0.0**

```

Main> initsExample
1: \_ -> []:_
3: \(a:_) -> []:(a:[]):_
5: \(a:b:_) -> []:(a:[]):(a:b:[]):_
Finished 7 tests.

```

```

inits :: [α] → [[α]]
inits [] = [[]]
inits (x : xs) = [[]] ++ map (x:) (inits xs)

```

To define a less strict implementation we delay the pattern matching that checks whether the argument is the empty list or not.

```

inits' :: [α] → [[α]]
inits' xs =
  [] : case xs of
        [] → []
        x : xs → map (x:) (inits' xs)

```

The function `tails :: [α] → [[α]]` shows a similar behaviour as `inits`, that is, it is unnecessarily strict.

5 Integers

Using Sloth we can check functions that use integers. For example, the following application checks whether the multiplication of two `Ints` is minimally strict.

```

multExample = strictCheck ((* :: Int → Int → Int) 5)

```

```

Main> multExample
2: \0 _ -> 0
Finished 21 tests.

```

As the counter-example shows the multiplication is too strict. If the first argument is zero we do not have to evaluate the second argument. But obviously the primitive multiplication evaluates both arguments.

When we check the monomorphic integer instance of `id` we can observe the correspondence between the size parameter and the integers considered as test cases.

```

intIdExample = verboseCheck (id :: Int → Int) 4

```

```

Main> intIdExample
1: \_ -> _
2: \0 -> 0
3: \-1 -> -1
4: \1 -> 1
5: \-2 -> -2
6: \2 -> 2
7: \-3 -> -3
8: \3 -> 3
Finished 8 tests.

```

The number of test cases is linear in the size, more precisely we check the integers from $-size-1$ to $size-1$ if we check $id :: Int \rightarrow Int$ up to size $size$.

If we check $drop$ for test cases up to size ten Sloth presents a couple of potential counter-examples.

```
dropExample = strictCheck (drop :: Int → [A] → [A]) 10
```

```
Main> dropExample
42: \4 _ -> []
61: \4 (a:_) -> []
62: \5 _ -> []
83: \4 (a:b:_) -> []
85: \5 (a:_) -> []
86: \6 _ -> []
Finished 146 tests.
```

If we check $drop$ for test cases up to size eleven some of the counter-examples disappear while additional potential counter-examples appear. All these potential counter-examples are no counter-examples. For example, let us consider the first one. Sloth suspects that $drop$ always yields the empty list if the first argument is four. The lists generated by Sloth do not exceed length four. Therefore, the result of all considered test cases is indeed the empty list. This example nicely demonstrates that not all potential counter-examples are indeed counter-examples.

As examples for minimally strict functions we check $(!!)$ and $take$.

```
indexingExample = strictCheck ((!!) :: [A] → Int → A) 100
```

```
Main> indexingExample
Finished 5098 tests.
```

```
takeExample = strictCheck (take :: Int → [A] → [A]) 100
```

```
Main> takeExample
Finished 5000 tests.
```

6 Character

Sloth handles characters in a similar way as integers. That is, it does not enumerate all characters for test cases of size one. The following example demonstrates the enumeration of characters by employing the $Char$ instance of id .

```
charIdExample = verboseCheck (id :: Char → Char) 4
```

```
1: \_ -> _
2: \'??' -> '??'
3: \'?>' -> '>?'
4: \'?@' -> '@?'
```

```

5: \'=\' -> '= '
6: \<' -> '<'
7: \'A' -> 'A'
8: \'B' -> 'B'
9: \';' -> ';'
10: \':' -> ':'
11: \'9' -> '9'
12: \'8' -> '8'
13: \'C' -> 'C'
14: \'D' -> 'D'
15: \'E' -> 'E'
16: \'F' -> 'F'

```

In contrast to the enumeration of integers the enumeration of characters is exponential in the size. That is, Sloth checks $id :: Char \rightarrow Char$ for 2^{size} test cases if we check it for inputs up to size $size$.

To demonstrate one of the limitations of Sloth we check whether the function `words` is unnecessarily strict. It is very difficult to check `words` because it yields a singleton list as result unless the argument contains a newline character. As there are many characters it is not very likely that a test case contains this character. Therefore, Sloth very likely assumes that `words` always yields lists with only a single element as result. For example, if we check `words` for strings up to size ten Sloth reports many potential counter-examples.

Because there are more test cases that are more defined, potential counter-examples that are reported first are more likely to be counter-examples than counter-examples reported later. That is, we would like to check `words` for strings up to a large size but only consider the first few counter-examples. For this purpose Sloth provides a function called `interactCheck`, which interactively asks the user whether to present more counter-examples.

```
wordsExample = interactCheck words 15
```

```

Main> wordsExample
175: Argument(s): \('?:'??:'>':_)
Current Result: ('?:'??:'>':_):_
Proposed Result: ('?:'??:'>':_):
More? [y(es)/n(o)/a(ll)]n

```

If we check `words` for strings up to size 15 test case 175 is a potential counter-example. By default `interactCheck` uses the detailed presentation of test cases.

At last we want to present one unnecessarily strict function that, at first sight, might be surprising. If we check the Boolean instance of the `show` function, Sloth states that it is unnecessarily strict.

```
showExample = strictCheck (show :: Bool -> String) 5
```

```

Main> showExample
1: \_ -> _:~::~:~::~:_
Finished 3 tests.

```

As both possible results, the string `"True"` as well as the string `"False"`, have at least four characters `show :: Bool → String` always yields a list with at least four elements. Therefore, Sloth proposes to yield a list structure with at least four elements without evaluating the argument.

References

- [1] O. Chitil. Promoting non-strict programming. In *IFL'06 Draft Proceedings*, 2006.
- [2] O. Chitil. StrictCheck: a tool for testing whether a function is unnecessarily strict. Technical Report 2-11, University of Kent, School of Computing, June 2011.
- [3] J. Christiansen. Sloth - A Tool for Checking Minimal Strictness. In *PADL'11 Proceedings*. Springer, 2011.
- [4] J. Christiansen and D. Seidel. Minimally strict polymorphic functions. In *PPDP'11 Proceedings*. ACM, 2011.