Why can’t I get a stack trace?

Simon Marlow
Motivation

Simon Marlow - Aug 9, 2011 - Public
After bashing my head against this problem on and off for several years, I think I finally understand how to track call stacks properly in a lazy functional language. If this pans out, we'll get backtraces in GHCi and more accurate profiling.

- Andrew Sackville-West, Benedict Eastaugh, Carl Howells, David Waern, Don Stewart and 26 more

23 comments

Gabriel Dos Reis - An upcoming ICFP paper?
Aug 9, 2011

Debasish Ghosh - Please give here a shout in case you decide to document it in a paper or a blog post.
Aug 9, 2011 +7

Manuel Chakravarty - That would be awesome!
Aug 9, 2011 +1

David Leuschner - Great news! We're already looking forward to testing the new profiler! :-)
Aug 9, 2011

Thomas Schilling - So, that would only work in GHCi? Will it have a performance impact?
Aug 9, 2011
A stack trace (or lexical call context) contains a lot of information, often enough to diagnose a bug.

In an imperative language, where every function call pushes a stack frame, the execution stack contains enough information to reconstruct the lexical call context.

The same isn’t true in Haskell, for various reasons...
1. Tail Call Optimisation

- TCO means that important information about the call chain is not retained on the stack
- But TCO is essential, we can’t just turn it off

```haskell
main = do
    [x] <- getArgs
    print (f (read x))

f :: Int -> Int
f x = g (x-1)

g :: Int -> Int
g x = 100 `div` x
```

Execution stack:
- main
- g
2. Lazy evaluation

- Lazy evaluation results in an execution stack that looks nothing like the lexical call stack.
- When a computation is suspended (a thunk) we should capture the call stack and store it with the thunk.

```haskell
main = do
  [x] <- fmap (fmap read) getArgs
  print (head (f x))

f x = map g [x .. x+10]

g :: Int -> Int

# g x = 100 `div` x
```

Execution stack:
- main
- print
- g
3. Transformation and optimisation
   – we do not want the transformations done by GHC’s optimiser to lose information or mangle the call stack.
   – we’ve already established that strictness analysis should not distort the stack.
   – But even inlining a function will lose information if we aren’t careful.
4. Even if we fix 1—3, high-level abstractions like monads result in strange stacks
   — examples coming...

• We need a framework for thinking about the issues.
A construct for pushing on the stack

- “push label L on the stack while evaluating E”
- this is a construct of the source language and the intermediate language (Core)
- Compiler can add these automatically, or the user can add them
- Think {-# SCC .. #-} in GHC
- We get to choose how detailed we want to be:
  - exported functions only
  - top-level functions only
  - all functions (good for profiling)
  - call sites (good for debugging)
  - all sub-expressions (fine-grained debugging or profiling)
Define stacks:

```
type Stack = [Label]
push :: Label -> Stack -> Stack
call :: Stack -> Stack -> Stack
```
Define stacks:

```haskell
type Stack = [Label]
push :: Label -> Stack -> Stack
call :: Stack -> Stack -> Stack
```

stack at the call site
• Define stacks:

```haskell
  type Stack = [Label]
push :: Label -> Stack -> Stack
call :: Stack -> Stack -> Stack
```

- stack at the call site
- stack of the function
• Define stacks:

```haskell
type Stack = [Label]
push :: Label -> Stack -> Stack
call :: Stack -> Stack -> Stack
```

stack at the call site

stack of the function

stack for the call
eval :: \text{Stack} \to \text{Expr} \to \text{E (Stack,Expr)}

eval \text{stk} (\text{EInt} \, i) = \text{return} \((\text{stk}, \text{EInt} \, i)\)

eval \text{stk} (\text{ELam} \, x \, e) = \text{return} \((\text{stk}, \text{ELam} \, x \, e)\)

\begin{align*}
\text{eval \text{stk} (\text{EPush} \, l \, e)} &= \text{eval} \ ((\text{push} \, l \, \text{stk}) \, e) \\
\text{eval \text{stk} (\text{ELet} \, (x, e1) \, e2)} &= \text{do} \\
&\quad \text{insertHeap} \, x \, (\text{stk}, e1) \\
&\quad \text{eval \, stk} \, e2 \\
\text{eval \text{stk} (\text{EApp} \, f \, x)} &= \text{do} \\
&\quad (\text{lam\_stk}, \text{ELam} \, y \, e) \leftarrow \text{eval \, stk} \, f \\
&\quad \text{eval \, lam\_stk} \, (\text{subst} \, y \, x \, e)
\end{align*}
Executable semantics

\[
\text{eval} :: \text{Stack} \to \text{Expr} \to E (\text{Stack}, \text{Expr})
\]

\[
\text{eval stk (EInt } i\text{)} = \text{return (stk, EInt } i\text{)}
\]
\[
\text{eval stk (ELam } x \text{ e)} = \text{return (stk, ELam } x \text{ e)}
\]

\[
\text{eval stk (EPush } l \text{ e)} = \text{eval (push } l \text{ stk) e}
\]

\[
\text{eval stk (ELet (x,e1) e2)} = \text{do}
\begin{align*}
\text{insertHeap } x \text{ (stk,e1)} \\
\text{eval stk e2}
\end{align*}
\]

\[
\text{eval stk (EApp } f \text{ x)} = \text{do}
\begin{align*}
\text{lam_stk, ELam y e)} & \leftarrow \text{eval stk } f \\
\text{eval lam_stk (subst y x e)}
\end{align*}
\]
Evaluable semantics

\[
eval :: \text{Stack} \to \text{Expr} \to \text{E} (\text{Stack}, \text{Expr})
\]

\[
eval \text{stk} \ (\text{EInt} \ i) = \text{return} \ (\text{stk}, \text{EInt} \ i)
\]

\[
eval \text{stk} \ (\text{ELam} \ x \ e) = \text{return} \ (\text{stk}, \text{ELam} \ x \ e)
\]

\[
eval \text{stk} \ (\text{EPush} \ l \ e) = \text{eval} \ (\text{push} \ l \ \text{stk})
\]

\[
eval \text{stk} \ (\text{ELet} \ (x,e1) \ e2) = \text{do}
\]
\[
\quad \text{insertHeap} \ x \ (\text{stk}, e1)
\]
\[
\quad \text{eval} \ \text{stk} \ e2
\]

\[
eval \text{stk} \ (\text{EApp} \ f \ x) = \text{do}
\]
\[
\quad (\text{lam} \ _\text{stk}, \text{ELam} \ y \ e) \leftarrow \text{eval} \ \text{stk} \ f
\]
\[
\quad \text{eval} \ \text{lam} \ _\text{stk} \ (\text{subst} \ y \ x \ e)
\]

E is a State monad containing the Heap: a mapping from \text{Var} to (\text{Stack}, \text{Expr})
Executable semantics

\[
eval :: \text{Stack} \to \text{Expr} \to E (\text{Stack}, \text{Expr})
\]

\[
eval \text{stk} (\text{EInt } i) = \text{return } (\text{stk}, \text{EInt } i)
\]

\[
eval \text{stk} (\text{ELam } x \ e) = \text{return } (\text{stk}, \text{ELam } x \ e)
\]

\[
eval \text{stk} (\text{EPush } l \ e) = \text{eval } (\text{push } l \ \text{stk}) \ e
\]

\[
eval \text{stk} (\text{ELet } (x,e1) e2) = \text{do}
\begin{align*}
    \text{insertHeap } x \ (\text{stk}, e1) \\
    \text{eval stk } e2
\end{align*}
\]

\[
eval \text{stk} (\text{EApp } f \ x) = \text{do}
\begin{align*}
    (\text{lam_stk}, \text{ELam } y \ e) & \leftarrow \text{eval stk } f \\
    \text{eval } \text{lam_stk} & \ (\text{subst } y \ x \ e)
\end{align*}
\]

Values are straightforward
Evaluable semantics

\[\text{eval} :: \text{Stack} \rightarrow \text{Expr} \rightarrow E (\text{Stack}, \text{Expr})\]

\[\text{eval} \ \text{stk} \ (\text{EInt} \ i) \ = \ \text{return} \ (\text{stk}, \ \text{EInt} \ i)\]

\[\text{eval} \ \text{stk} \ (\text{ELam} \ x \ e) \ = \ \text{return} \ (\text{stk}, \ \text{ELam} \ x \ e)\]

\[\text{eval} \ \text{stk} \ (\text{EPush} \ l \ e) = \ \text{eval} \ (\text{push} \ l \ \text{stk}) \ e\]

\[\text{eval} \ \text{stk} \ (\text{ELet} \ (x,e1) \ e2) = \ \text{do}\]
\[\text{insertHeap} \ x \ (\text{stk}, e1)\]
\[\text{eval} \ \text{stk} \ e2\]

\[\text{eval} \ \text{stk} \ (\text{EApp} \ f \ x) = \ \text{do}\]
\[\ (\text{lam}\_\text{stk}, \ \text{ELam} \ y \ e) \leftarrow \ \text{eval} \ \text{stk} \ f\]
\[\text{eval} \ \text{lam}\_\text{stk} \ (\text{subst} \ y \ x \ e)\]
Executable semantics

eval :: Stack -> Expr -> E (Stack,Expr)

eval stk (EInt i)    = return (stk, EInt i)
eval stk (ELam x e)  = return (stk, ELam x e)
eval stk (EPush l e) = eval (push l stk) e

eval stk (ELet (x,e1) e2) = do
    insertHeap x (stk,e1)
    eval stk e2

eval stk (EApp f x) = do
    (lam_stk, ELam y e) <- eval
    eval lam_stk (subst y x e)
Evaluated semantics

\[
\text{eval :: Stack} \rightarrow \text{Expr} \rightarrow E (\text{Stack,Expr})
\]

\[
\text{eval stk (EInt i)} = \text{return} (\text{stk, EInt i})
\]
\[
\text{eval stk (ELam x e)} = \text{return} (\text{stk, ELam x e})
\]
\[
\text{eval stk (EPush l e)} = \text{eval} (\text{push l stk}) e
\]
\[
\text{eval stk (ELet (x,e1) e2)} = \text{do}
  \text{insertHeap} x \text{ (stk,e1)}
  \text{eval stk e2}
\]
\[
\text{eval stk (EApp f x)} = \text{do}
  (\text{lam_stk}, \text{ELam y e}) \leftarrow \text{eval stk f}
  \text{eval lam_stk (subst y x e)}
\]

Application continues with the stack returned by evaluating the lambda.
Executable semantics (variables)

```haskell
eval stk (EVar x) = do 
  r <- lookupHeap x 
  case r of 
    (stk', EInt i) -> return (stk', EInt i) 
    (stk', ELam y e) -> return (call stk stk', ELam y e) 
    (stk', e) -> do 
      deleteHeap x 
      (stkv, v) <- eval stk' e 
      insertHeap x (stkv, v) 
      eval stk (EVar x) 
```

Here's where we are "calling" a function.
Given this semantics, define push & call

- The problem now is to find suitable definitions of **push** and **call** that
  - Behave like a call stack
  - Have nice properties:
    - transformation-friendly
    - predictable/robust
    - implementable
Lazy evaluation is dealt with

- Lazy evaluation is dealt with by
  - capturing the current stack when we suspend a computation as a thunk in the heap
  - temporarily restoring the stack when the thunk is evaluated
- Nothing controversial at all – we just need a mechanism for capturing and restoring the stack.

```haskell
eval stk (ELet (x,e1) e2) = do
  insertHeap x (stk,e1)
  eval stk e2

eval stk (EVar x) = do
  r <- lookupHeap x
  case r of
    ...
      (stk',e) -> do
        deleteHeap x
        (stkv, v) <- eval stk' e
        insertHeap x (stkv,v)
        eval stk (EVar x)
```
Tail calls are dealt with

- The semantics says nothing about tail calls – `push` always pushes on the stack.
- Even if the underlying execution model is doing TCO, the call stack simulation must not.
Examples

• The heap is initialised with the top-level bindings (give each the stack <CAF>)
• When we get to (f y), current stack is <main>
• f is already evaluated
• call <main> <CAF> = <main>
• eval <main> (push f y+y)
• eval <main,f> (y+y)
• at the +, the current stack is <main,f>

\[
\begin{align*}
  f &= \lambda x. \text{push "f" } x+x \\
  \text{main} &= \lambda x. \text{push "main" } \\
  \text{let } y = 1 \text{ in } f y
\end{align*}
\]

Let’s assume, for now, call Sapp Slam = Sapp
Use the call-site stack?

- Previous example suggests this might be a good choice?
- After all, *this gives exactly the call stack you would get in a strict language*
But we have to be careful

• If instead of this:

```latex
f = \lambda x. \text{push "f" } x+x
main = \lambda x. \text{push "main" } \\
\text{let } y = 1 \text{ in } f \ y
```

• We wrote this:

```latex
f = \text{push "f" } (\lambda x. x+x) \\
main = \lambda x. \text{push "main" } \text{let } y = 1 \text{ in } f \ y
```

• Now it doesn’t work so well: the “f” label is lost.

• In this semantics, the scope of push does not extend into lambdas
Just label all the lambdas?

- Idea: make the compiler label all the lambdas automatically
- e.g. the compiler inserts a push inside any lambda:

```
f = push "f" (λx . push "f1" x+x)
main = λx. push "main"
  let y = 1 in f y
```

- Now we get a useful stack again: <main,f1>
Some properties

- Adding an extra binding doesn’t change the stack

\[
f = \text{push } \text{“f”} \, (\lambda x \ . \ \text{push } \text{“f1”} \, x+x) \\
g = \text{push } \text{“g”} \, f \\
\text{main} = \lambda x . \ \text{push } \text{“main”} \\
\text{let } y = 1 \text{ in } g \, y
\]

- In this semantics ‘push L x == x’
- Arguably useful: the stack is robust with respect to this transformation (by the compiler or user)
But...

- eta-expansion changes the stack

\[
f = \text{push } \text{"f" } (\lambda x . \text{push } \text{"f1" } x + x)
\]
\[
g = \lambda x . \text{push } \text{"g" } f \ x
\]
\[
\text{main } = \lambda x . \text{push } \text{"main" }
\]
\[
\text{let } y = 1 \text{ in } g \ y
\]

- Now the stack at the + will be <main,g,f>
Concrete example

• When we tried this for real, we found that in functions like

\[ h = f \cdot g \]

• \( h \) does not appear on the stack, although in

\[ h \ x = (f \cdot g) \ x \]

• now it does. This is surprising and undesirable.
Worse...

• Let’s make a state monad:

```haskell
newtype M s a = M { unM :: s -> (s,a) }

instance Monad (M s) where
  (M m) >>= k = M $
    λ s -> case m s of
      (s',a) -> unM (k a) s'

return a = M $
  λ s -> (s,a)

errorM :: String -> M s a
errorM s = M $ λ _ -> error s

runM :: M s a -> s -> a
runM (M m) s = case m s of (_,a) -> a
```

Suppose we want the stack when error is called, for debugging.
Using a monad

- Simple example:

```haskell
main = print (runM (bar ["a","b"]) "state")

bar :: [String] -> M s [String]
bar xs = mapM foo xs

foo :: String -> M s String
foo x = errorM x
```
Using a monad

• Simple example:

```haskell
main = print (runM (bar ["a","b"])) "state")

bar :: [String] -> M s [String]
bar xs = mapM foo xs

foo :: String -> M s String
foo x = errorM x
```

• We are looking for a stack like <main,runM,bar,mapM,foo,errorM>
Using a monad

• Simple example:

```haskell
main = print (runM (bar ["a","b"]) "state")

bar :: [String] -> M s [String]
bar xs = mapM foo xs

foo :: String -> M s String
foo x = errorM x
```

• We are looking for a stack like
  <main,runM,bar,mapM,foo,errorM>

• Stack we get: <runM>
Why?

• Take a typical monadic function:

\[
f = \text{do } p; q
\]

• Desuraging gives

\[
f = p >> q
\]

• Adding push:

\[
f = \text{push “f” (} p >> q \text{)}
\]

• Expanding out (\(\gg\)):

\[
f = \text{push “f” (} \lambda s \rightarrow \text{case } p s \text{ of } (a, s') \rightarrow b s' \text{)}
\]

• recall that \(\text{push } L (\lambda x . e) = \lambda x . e\)
The IO monad

- In GHC the IO monad is defined like the state monad given earlier.
- We found that with this stack semantics, we get no useful stacks for IO monad code at all.
- When profiling, all the costs were attributed to main.
We recovered the non-lazy non-TCO call stack, which is the stack you would get in a strict functional language.

But it isn’t good enough.

– at least when used with monads or other high-level functional abstractions
Can we find a better semantics?

- call $S_{app} S_{lam} = ?$
- non-starter: call $S_{app} S_{lam} = S_{lam}$
  - ignores the calling context
  - gives a purely lexical stack, not a call stack
  - (possibly useful for flat profiling though)
- Clearly we want to take into account both $S_{app}$ and $S_{lam}$ somehow.
The definitions I want to use

- Behaves nicely with inlining:
  - “common prefix” is intended to capture the call stack up to the point where the function was defined
- useful for profiling/debugging: the top-of-stack label is always correct, we just truncate the stack on recursion.

```haskell
call Sapp Slam = Sapp ++ Slam'
  where (Spre, Sapp', Slam') = commonPrefix Sapp Slam

push l s | l `elem` s = dropWhile (/= l) s
  | otherwise = l : s
```
Status

• GHC 7.4.1 has a new implementation of profiling using **push**
• +RTS –xc prints the call stack when an exception is raised
• Programmatic access to the call stack:
Status

• GHC 7.4.1 has a new implementation of profiling using push
• +RTS –xc prints the call stack when an exception is raised
• Programmatic access to the call stack:

```haskell
traceStack :: String -> a -> a
errorWithStackTrace :: String -> a
```
Demo
Programmatic access to stack trace

• The GHC.Stack module provides runtime access to the stack trace
• On top of which is built this:

  • e.g. now when GHC panics it emits a stack trace (if it was compiled with profiling)
Programmatic access to stack trace

- The GHC.Stack module provides runtime access to the stack trace
- On top of which is built this:

```haskell
-- | like 'trace', but additionally prints a call stack if one is available.
traceStack :: String -> a -> a
```

- e.g. now when GHC panics it emits a stack trace (if it was compiled with profiling)
Properties

- This semantics has some nice properties.

\[
\begin{align*}
\text{push } L \ x & \Rightarrow x \\
\text{push } L \ (\lambda x . \ e) & \Rightarrow \lambda x . \ e \\
\text{push } L \ (C \ x_1 .. x_n) & \Rightarrow C \ x_1 .. x_n \\
\text{let } x = \lambda y . \ e \ \text{in} \ \text{push } L \ e' & \Rightarrow \text{push } L \ (\text{let } x = \lambda y . \ e \ \text{in} \ e') \\
\text{push } L \ (\text{let } x = e \ \text{in} \ e') & \Rightarrow \text{let } x = \text{push } L \ e \ \text{in} \ \text{push } L \ e
\end{align*}
\]
Properties

- This semantics has some nice properties:

  \[
  \begin{align*}
  \text{push } L \ x & \Rightarrow x \\
  \text{push } L \ (\lambda x . e) & \Rightarrow \lambda x . e \\
  \text{push } L \ (C \ x_1 \ldots x_n) & \Rightarrow C \ x_1 \ldots x_n \\
  \text{let } x = \lambda y . e \text{ in } \text{push } L \ e' & \Rightarrow \text{push } L \ (\text{let } x = \lambda y . e \text{ in } e') \\
  \text{push } L \ (\text{let } x = e \text{ in } e') & \Rightarrow \text{let } x = \text{push } L \ e \text{ in } \text{push } L \ e
  \end{align*}
  \]

since the stack attached to a lambda is irrelevant (except for heap profiling)
Properties

- This semantics has some nice properties.

\[
\begin{align*}
push_L \ x & \Rightarrow x \\
push_L \ (\lambda x. \ e) & \Rightarrow \lambda x. \ e \\
push_L \ (C \ x_1 .. x_n) & \Rightarrow C \ x_1 .. x_n \\
let \ x = \lambda y. \ e \ \text{in} \ push_L \ e' & \Rightarrow push_L \ (let \ x = \lambda y. \ e \ \text{in} \ e') \\
push_L \ (let \ x = e \ \text{in} \ e') & \Rightarrow let \ x = push_L \ e \ \text{in} \ push_L \ e
\end{align*}
\]

O(1) change to cost attribution, no change to profile shape
### Properties

- This semantics has some nice properties.

<table>
<thead>
<tr>
<th>Expression</th>
<th>Simplified</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>push L x</code></td>
<td><code>x</code></td>
</tr>
<tr>
<td><code>push L (\lambda x . e)</code></td>
<td><code>\lambda x . e</code></td>
</tr>
<tr>
<td><code>push L (C x1 .. xn)</code></td>
<td><code>C x1 .. xn</code></td>
</tr>
</tbody>
</table>

```
let x = \lambda y . e in push L e'
=> push L (let x = \lambda y . e in e')
```

```
push L (let x = e in e')
=> let x = push L e in push L e
```

Note if `e` is a value, the `push L` will disappear.
Inlining

• We expect to be able to substitute a function’s definition for its name without affecting the stack. e.g.

\[
\begin{align*}
f &= \lambda x . \text{push} \ "f1" \ x+x \\
\text{main} &= \lambda x . \text{push} \ "main" \\
&\quad \text{let } y = 1 \text{ in } f \ y
\end{align*}
\]

• should be the same as

\[
\begin{align*}
\text{main} &= \lambda x . \text{push} \ "main" \\
&\quad \text{let } y = 1 \text{ in} \\
&\quad (\lambda x . \text{push} \ "f1" \ x+x) \ y
\end{align*}
\]

• and indeed it is in this semantics.
  – (inlining functions is crucial for optimisation in GHC)
Think about what properties we want

• Push inside lambda:

\[
\text{push } \bot (\lambda x. e) = \lambda x. \text{push } \bot e
\]

– (recall that the previous semantics allowed dropping the push here)

– This will give us a push that scopes over the inside of lambdas, not just outside.

• which will in turn give us that stacks are robust to eta-expansion/contraction
What does it take to make this true?

• Consider

\[
\text{let } f = \text{push } \text{"f" } \lambda x . e \\
\text{in } \ldots f \ldots
\]

\[
\text{let } f = \lambda x . \text{push } \text{"f" } e \\
\text{in } \ldots f \ldots
\]

• If we work through the details, we find that we need

\[
\text{call } S (\text{push } L S_f) == \text{push } L (\text{call } S S_f)
\]

• Not difficult: e.g.

\[
\text{type Stack } = [\text{Label}] \\
\text{push } = (:) \\
\text{call } = \text{foldr push}
\]

like flip (++), but useful to define it this way
Recursion?

• We do want finite stacks
  – the mutator is using tail recursion
• Simplest approach: push is a no-op if the label is already on the stack somewhere:

\[
\text{push } l \ s \ | \ l \ `\text{elem}` \ s \ = \ s \\
| \ \text{otherwise} \quad = \ l : s
\]

• still satisfies the push-inside-lambda property
• but: not so good for profiling or debugging
  – the label on top of the stack is not necessarily where the program counter is
Inlining of functions

- (remember, allowing inlining is crucial)

- Consider

```
let g = λx.e in
let f = push "f" g in
f y
```

```
let f = push "f" λx.e in
f y
```

- Work through the details, and we need that

```
call (push L S) S == push L S
```

- interesting: calling a function whose stack is a prefix of the current stack should not change the stack.
Break out the proof tools
Break out the proof tools

• QuickCheck.
Break out the proof tools

• QuickCheck.

prop_append2 = forallShrink stacks shrinkstack $ \ s \rightarrow 
forall Main.labels $ \ x \rightarrow 
call (s `push` x) s == s `push` x

*** Failed! Falsifiable (after 8 tests and 2 shrinks):
(E :> "e") :> "b"
"e"
Break out the proof tools

• QuickCheck.

```
prop_append2 = forAllShrink stacks shrinkstack $ s ->
  forAll Main.labels $ x ->
    call (s `push` x) s == s `push` x

*** Failed! Falsifiable (after 8 tests and 2 shrinks):
(E :> "e") :> "b"
"e"
```

• but this corresponds to something very strange:
Break out the proof tools

- QuickCheck.

```haskell
prop_append2 = forallShrink stacks shrinkstack $ \s ->
  forall Main.labels $ \x ->
    call (s `push` x) s == s `push` x
```

*** Failed! Falsifiable (after 8 tests and 2 shrinks):
(E :> "e") :> "b"
"e"

- but this corresponds to something very strange:

```haskell
push "f"
...
let g = \x.e in
let f = push "f" g in
f y
```
A more restricted property

- This is a limited form of the real property we need for inlining
- The push-inside-lambda property behaves similarly: we need to restrict the use of duplicate labels to make it go through.
A more restricted property

```haskell
prop_stack2a = forAllShrink stacks shrinkstack $ \s ->
  forAll Main.labels $ \x ->
    x `elemStack` s ||
    call (s `push` x) s == s `push` x
```

*Main> quickCheck prop_stack2a
+++ OK, passed 100 tests.

• This is a limited form of the real property we need for inlining
• The push-inside-lambda property behaves similarly: we need to restrict the use of duplicate labels to make it go through.