Discovering Counterexamples (and Proof Ingredients) for Knuth-like 0-1-…-k-Principles

Moritz Fürneisen and Janis Voigtländer

University of Bonn

October 9th, 2012

Knuth's 0-1-Principle [Knuth 1973]

Informally: If a comparison-swap algorithm sorts Booleans correctly, it sorts integers correctly as well.

Formally: Let (in Haskell, say):

$$\texttt{sort} :: ((\alpha,\alpha) \to (\alpha,\alpha)) \to [\alpha] \to [\alpha]$$

$$f :: (\mathrm{Int}, \mathrm{Int}) \to (\mathrm{Int}, \mathrm{Int})$$
$$f\ (x,y) = \mathit{if}\ x > y\ \mathit{then}\ (y,x)\ \mathit{else}\ (x,y)$$

$$g :: (\mathrm{Bool}, \mathrm{Bool}) \to (\mathrm{Bool}, \mathrm{Bool})$$
$$g\ (x,y) = (x\ \&\&\ y, x \mid\mid y)$$

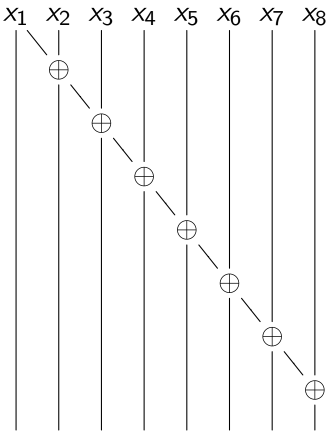
If for every xs :: [Bool], `sort` g xs gives the correct result, then for every xs :: [Int], `sort` f xs gives the correct result.

Parallel Prefix Computation

Given: inputs $x_1, \ldots, x_n$ and an associative operation $\oplus$

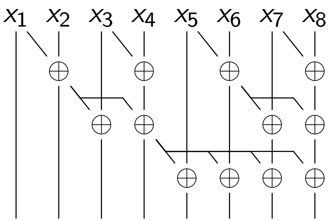
Task: compute the values $x_1, x_1 \oplus x_2, \ldots, x_1 \oplus x_2 \oplus \cdots \oplus x_n$

Solution:

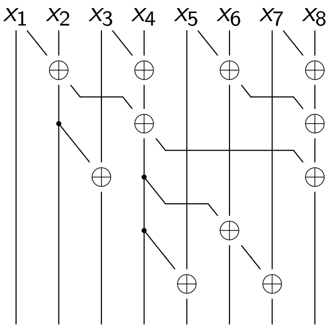


Parallel Prefix Computation

Alternative:



Or:



Or: . . .

Expressing Parallel Prefix Algorithms in Haskell

Functions of type:

$$\texttt{scanl1} :: (\alpha \to \alpha \to \alpha) \to [\alpha] \to [\alpha]$$

For example, à la [Sklansky 1960]:

$$
\begin{array}{llll}
\texttt{sklansky} :: (\alpha \to \alpha \to \alpha) \to [\alpha] \to [\alpha] & & & \\
\texttt{sklansky}\ (\oplus)\ [x] = [x] & & & \\
\texttt{sklansky}\ (\oplus)\ \mathit{xs}\ = \mathit{us} \mathbin{+\!\!+} \mathit{vs} & & & \\
\quad \textbf{where}\ t & & = & ((\texttt{length}\ \mathit{xs}) + 1)\ `\texttt{div}`\ 2 \\
\quad & (\mathit{ys}, \mathit{zs}) & = & \texttt{splitAt}\ t\ \mathit{xs} \\
\quad & \mathit{us} & = & \texttt{sklansky}\ (\oplus)\ \mathit{ys} \\
\quad & \mathit{vs} & = & [(\texttt{last}\ \mathit{us}) \oplus v \mid v \leftarrow \texttt{sklansky}\ (\oplus)\ \mathit{zs}]
\end{array}
$$

Investigating Particular Instances Only

Knuth's 0-1-Principle

If a comparison-swap algorithm sorts correctly on the Booleans, it does so on arbitrary totally ordered value sets.

A Knuth-like 0-1-Principle ?

If a parallel prefix algorithm is correct (for associative operations) on the Booleans, it is so on arbitrary value sets.

Unfortunately not !

A Knuth-like 0-1-2-Principle [V. 2008]

Given: $\texttt{scanl1} :: (\alpha \to \alpha \to \alpha) \to [\alpha] \to [\alpha]$

$\texttt{scanl1}\ (\oplus)\ (x : xs) = go\ x\ xs$

where $go\ x\ [\,] \qquad = [x]$

$go\ x\ (y : ys) = x : (go\ (x \oplus y)\ ys)$

$\texttt{candidate} :: (\alpha \to \alpha \to \alpha) \to [\alpha] \to [\alpha]$

data Three = Zero | One | Two

Theorem: If for every xs :: [Three] and associative $(\oplus)$ :: Three $\to$ Three $\to$ Three,

$$\texttt{candidate}\ (\oplus)\ xs \quad = \quad \texttt{scanl1}\ (\oplus)\ xs\,,$$

then the same holds for every type τ, $xs :: [\tau]$, and associative $(\oplus) :: \tau \to \tau \to \tau$.

Why 0-1-2? And How?

- To get going, proof uses parametricity [Reynolds 1983], deriving a "free theorem" [Wadler 1989].
- Proof crucially involves two specific associative functions:

$\oplus_1$	Zero	One	Two
Zero	Zero	One	Two
One	One	Two	Two
Two	Two	Two	Two

and

$\oplus_2$	Zero	One	Two
Zero	Zero	One	Two
One	One	One	Two
Two	Two	One	Two

  and their behavior on lists $[(\text{Zero},)^* \text{ One } (,\text{Zero})^* \ (,\text{Two})^*]$ and $[(\text{Zero},)^* \text{ One}, \text{Two } (,\text{Zero})^*]$, respectively.

- Formalisation available in Isabelle/HOL [Böhme 2007].

But:

- Does that really explain the why and how?
- What to do to get similar results for other algorithm classes?

Plan of the Talk

- Start telling the story of how "0-1-2", $\oplus_1$, $\oplus_2$, ... were found (back in 2007, never recorded, but interesting I think).

- Challenge you to suggest other approaches to discover the required counterexamples and proof ingredients?

- Invite complaints about where the presented (deliberately naive, exploratory) approach is too ad-hoc, or unacceptably pulls a rabbit out of a hat.

- Provoke investigation/proposals of other algorithm classes on which one could try to play the same or similar trick(s)?

Investigating Particular Instances Only

Knuth's 0-1-Principle

If a comparison-swap algorithm sorts correctly on the Booleans, it does so on arbitrary totally ordered value sets.

A Knuth-like 0-1-Principle ?

If a parallel prefix algorithm is correct (for associative operations) on the Booleans, it is so on arbitrary value sets.

Unfortunately not !

Investigating Particular Instances Only

Knuth's 0-1-Principle

If a comparison-swap algorithm sorts correctly on the Booleans, it does so on arbitrary totally ordered value sets.

A Knuth-like 0-1-Principle ?

If a parallel prefix algorithm is correct (for associative operations) on the Booleans, it is so on arbitrary value sets.

Unfortunately not !

Let's:

- try to find out why not,

Investigating Particular Instances Only

Knuth's 0-1-Principle

If a comparison-swap algorithm sorts correctly on the Booleans, it does so on arbitrary totally ordered value sets.

A Knuth-like 0-1-Principle ?

If a parallel prefix algorithm is correct (for associative operations) on the Booleans, it is so on arbitrary value sets.

Unfortunately not !

Let's:

- try to find out why not,
- and why 0-1-2 makes more sense to attempt proving,

Investigating Particular Instances Only

Knuth's 0-1-Principle

If a comparison-swap algorithm sorts correctly on the Booleans, it does so on arbitrary totally ordered value sets.

A Knuth-like 0-1-Principle ?

If a parallel prefix algorithm is correct (for associative operations) on the Booleans, it is so on arbitrary value sets.

Unfortunately not !

Let's:

- try to find out why not,
- and why 0-1-2 makes more sense to attempt proving,
- and let's try to do all that without pulling too many rabbits.

What a Knuth-like 0-1-Principle Would Be

Given: $\texttt{scanl1} :: (\alpha \to \alpha \to \alpha) \to [\alpha] \to [\alpha]$

$\texttt{scanl1}\ (\oplus)\ (x : xs) = go\ x\ xs$

$\quad$ **where** $go\ x\ [\,] \qquad\quad = [x]$

$\qquad\qquad go\ x\ (y : ys) = x : (go\ (x \oplus y)\ ys)$

$\texttt{candidate} :: (\alpha \to \alpha \to \alpha) \to [\alpha] \to [\alpha]$

Theorem?: If for every associative $(\oplus) :: \mathrm{Bool} \to \mathrm{Bool} \to \mathrm{Bool}$ and $xs :: [\mathrm{Bool}]$,

$$\texttt{candidate}\ (\oplus)\ xs \quad = \quad \texttt{scanl1}\ (\oplus)\ xs\,,$$

then the same holds for every type τ, associative $(\oplus) :: \tau \to \tau \to \tau$, and $xs :: [\tau]$.

What a Knuth-like 0-1-Principle Would Be

Given: $\texttt{scanl1} :: (\alpha \to \alpha \to \alpha) \to [\alpha] \to [\alpha]$

$$\begin{aligned} &\texttt{scanl1}\ (\oplus)\ (x : xs) = go\ x\ xs \\ &\quad \textbf{where}\ go\ x\ [\,] \qquad\ \ = [x] \\ &\qquad\qquad\ go\ x\ (y : ys) = x : (go\ (x \oplus y)\ ys) \end{aligned}$$

$\texttt{candidate} :: (\alpha \to \alpha \to \alpha) \to [\alpha] \to [\alpha]$

Theorem?: If for every associative $(\oplus) :: \text{Bool} \to \text{Bool} \to \text{Bool}$ and $xs :: [\text{Bool}]$,

$$\texttt{candidate}\ (\oplus)\ xs \quad = \quad \texttt{scanl1}\ (\oplus)\ xs\,,$$

then the same holds for every type τ, associative $(\oplus) :: \tau \to \tau \to \tau$, and $xs :: [\tau]$.

Let's try to find a counterexample by property-based testing.

Getting Started

Somewhat naive expression of our intent:

$$
\begin{array}{l}
\texttt{quickCheck}\ \$\ \lambda(\texttt{candidate} :: (\alpha \to \alpha \to \alpha) \to [\alpha] \to [\alpha]) \to \\
\quad (\texttt{forAll associative}\ \$\ \lambda((\oplus) :: \mathsf{Bool} \to \mathsf{Bool} \to \mathsf{Bool}) \to \\
\qquad \lambda \mathit{xs} :: [\mathsf{Bool}] \to \texttt{candidate}\ (\oplus)\ \mathit{xs} == \texttt{scanl1}\ (\oplus)\ \mathit{xs}) \\
\quad ==> \\
\quad \forall \tau.\ (\texttt{forAll associative}\ \$\ \lambda((\oplus) :: \tau \to \tau \to \tau) \to \\
\qquad\qquad \lambda \mathit{xs} :: [\tau] \to \texttt{candidate}\ (\oplus)\ \mathit{xs} == \texttt{scanl1}\ (\oplus)\ \mathit{xs})
\end{array}
$$

Getting Started

Somewhat naive expression of our intent:

$$\begin{array}{l}
\texttt{quickCheck}\ \$\ \lambda(\texttt{candidate} :: (\alpha \to \alpha \to \alpha) \to [\alpha] \to [\alpha]) \to \\
\quad (\texttt{forAll associative}\ \$\ \lambda((\oplus) :: \mathsf{Bool} \to \mathsf{Bool} \to \mathsf{Bool}) \to \\
\qquad \lambda \mathit{xs} :: [\mathsf{Bool}] \to \texttt{candidate}\ (\oplus)\ \mathit{xs} == \texttt{scanl1}\ (\oplus)\ \mathit{xs}) \\
\quad ==> \\
\quad \forall \tau.\ (\texttt{forAll associative}\ \$\ \lambda((\oplus) :: \tau \to \tau \to \tau) \to \\
\qquad\qquad \lambda \mathit{xs} :: [\tau] \to \texttt{candidate}\ (\oplus)\ \mathit{xs} == \texttt{scanl1}\ (\oplus)\ \mathit{xs})
\end{array}$$

Hmm, this is problematic in so many ways!

Getting Started

Somewhat naive expression of our intent:

$$
\begin{array}{l}
\texttt{quickCheck}\ \$\ \lambda(\texttt{candidate} :: (\alpha \to \alpha \to \alpha) \to [\alpha] \to [\alpha]) \to \\
\quad (\texttt{forAll associative}\ \$\ \lambda((\oplus) :: \mathsf{Bool} \to \mathsf{Bool} \to \mathsf{Bool}) \to \\
\qquad \lambda \mathit{xs} :: [\mathsf{Bool}] \to \texttt{candidate}\ (\oplus)\ \mathit{xs} == \texttt{scanl1}\ (\oplus)\ \mathit{xs}) \\
\quad ==> \\
\quad \forall \tau.\ (\texttt{forAll associative}\ \$\ \lambda((\oplus) :: \tau \to \tau \to \tau) \to \\
\qquad\qquad \lambda \mathit{xs} :: [\tau] \to \texttt{candidate}\ (\oplus)\ \mathit{xs} == \texttt{scanl1}\ (\oplus)\ \mathit{xs})
\end{array}
$$

Hmm, this is problematic in so many ways!

- generating (polymorphic) functions?

Getting Started

Somewhat naive expression of our intent:

$$
\begin{array}{l}
\texttt{quickCheck}\ \$\ \lambda(\texttt{candidate} :: (\alpha \to \alpha \to \alpha) \to [\alpha] \to [\alpha]) \to \\
\quad (\texttt{forAll associative}\ \$\ \lambda((\oplus) :: \mathsf{Bool} \to \mathsf{Bool} \to \mathsf{Bool}) \to \\
\qquad \lambda \mathit{xs} :: [\mathsf{Bool}] \to \texttt{candidate}\ (\oplus)\ \mathit{xs} == \texttt{scanl1}\ (\oplus)\ \mathit{xs}) \\
\quad ==> \\
\quad \forall \tau.\ (\texttt{forAll associative}\ \$\ \lambda((\oplus) :: \tau \to \tau \to \tau) \to \\
\qquad\qquad \lambda \mathit{xs} :: [\tau] \to \texttt{candidate}\ (\oplus)\ \mathit{xs} == \texttt{scanl1}\ (\oplus)\ \mathit{xs})
\end{array}
$$

Hmm, this is problematic in so many ways!

- generating (polymorphic) functions?
- randomness vs. exhaustiveness?

Getting Started

Somewhat naive expression of our intent:

$$
\begin{array}{l}
\texttt{quickCheck}\ \$\ \lambda(\texttt{candidate} :: (\alpha \to \alpha \to \alpha) \to [\alpha] \to [\alpha]) \to \\
\quad (\texttt{forAll associative}\ \$\ \lambda((\oplus) :: \mathsf{Bool} \to \mathsf{Bool} \to \mathsf{Bool}) \to \\
\qquad \lambda \mathit{xs} :: [\mathsf{Bool}] \to \texttt{candidate}\ (\oplus)\ \mathit{xs} == \texttt{scanl1}\ (\oplus)\ \mathit{xs}) \\
\quad ==> \\
\quad \forall \tau.\ (\texttt{forAll associative}\ \$\ \lambda((\oplus) :: \tau \to \tau \to \tau) \to \\
\qquad\qquad \lambda \mathit{xs} :: [\tau] \to \texttt{candidate}\ (\oplus)\ \mathit{xs} == \texttt{scanl1}\ (\oplus)\ \mathit{xs})
\end{array}
$$

Hmm, this is problematic in so many ways!

- generating (polymorphic) functions?
- randomness vs. exhaustiveness?
- a complex property as precondition of "==>"?

Getting Started

Somewhat naive expression of our intent:

$$
\begin{array}{l}
\texttt{quickCheck}\ \$\ \lambda(\texttt{candidate} :: (\alpha \to \alpha \to \alpha) \to [\alpha] \to [\alpha]) \to \\
\quad (\texttt{forAll associative}\ \$\ \lambda((\oplus) :: \mathsf{Bool} \to \mathsf{Bool} \to \mathsf{Bool}) \to \\
\qquad \lambda \mathit{xs} :: [\mathsf{Bool}] \to \texttt{candidate}\ (\oplus)\ \mathit{xs} == \texttt{scanl1}\ (\oplus)\ \mathit{xs}) \\
\quad ==> \\
\quad \forall \tau.\ (\texttt{forAll associative}\ \$\ \lambda((\oplus) :: \tau \to \tau \to \tau) \to \\
\qquad\qquad \lambda \mathit{xs} :: [\tau] \to \texttt{candidate}\ (\oplus)\ \mathit{xs} == \texttt{scanl1}\ (\oplus)\ \mathit{xs})
\end{array}
$$

Hmm, this is problematic in so many ways!

- generating (polymorphic) functions?
- randomness vs. exhaustiveness?
- a complex property as precondition of "==>"?
- generating/picking the type τ?

Transforming Polymorphism Away

Let's try to get a grip on `candidate` $:: (\alpha \to \alpha \to \alpha) \to [\alpha] \to [\alpha]$.

Transforming Polymorphism Away

Let's try to get a grip on `candidate` $:: (\alpha \to \alpha \to \alpha) \to [\alpha] \to [\alpha]$.

A question: What can such a function do, given an operation $\oplus$ and input list $[x_0, \ldots, x_{n-1}]$?

Transforming Polymorphism Away

Let's try to get a grip on `candidate` $:: (\alpha \to \alpha \to \alpha) \to [\alpha] \to [\alpha]$.

A question: What can such a function do, given an operation $\oplus$ and input list $[x_0, \ldots, x_{n-1}]$?

The answer: Create an output list consisting of "expressions" built from $\oplus$ and $x_0, \ldots, x_{n-1}$.
Independently of the α-type !

Transforming Polymorphism Away

Let's try to get a grip on `candidate` $:: (\alpha \to \alpha \to \alpha) \to [\alpha] \to [\alpha]$.

A question: What can such a function do, given an operation $\oplus$ and input list $[x_0, \ldots, x_{n-1}]$?

The answer: Create an output list consisting of "expressions" built from $\oplus$ and $x_0, \ldots, x_{n-1}$.
Independently of the α-type !
But dependent on the input list length n !

Transforming Polymorphism Away

Let's try to get a grip on `candidate` $:: (\alpha \to \alpha \to \alpha) \to [\alpha] \to [\alpha]$.

A question: What can such a function do, given an operation $\oplus$ and input list $[x_0, \ldots, x_{n-1}]$?

The answer: Create an output list consisting of "expressions" built from $\oplus$ and $x_0, \ldots, x_{n-1}$.
Independently of the α-type !
But dependent on the input list length n !

So, to any `candidate` as above corresponds a function of type $\text{Nat} \to [\text{Expr}]$, where

$$\textbf{data}\ \text{Expr} = \text{Var Nat} \mid \text{Op Expr Expr}$$

Transforming Polymorphism Away

Let's try to get a grip on `candidate` $:: (\alpha \to \alpha \to \alpha) \to [\alpha] \to [\alpha]$.

A question: What can such a function do, given an operation $\oplus$ and input list $[x_0, \ldots, x_{n-1}]$?

The answer: Create an output list consisting of "expressions" built from $\oplus$ and $x_0, \ldots, x_{n-1}$.
Independently of the α-type !
But dependent on the input list length n !

So, to any `candidate` as above corresponds a function of type Nat $\to$ [Expr], where

data Expr = Var Nat | Op Expr Expr

We can prove this (almost) isomorphism using parametricity/"free theorems" [Reynolds 1983, Wadler 1989], and can actually program appropriate conversions.

Where Are We Now?

To test:

$$
\begin{array}{l}
\texttt{quickCheck}\ \$\ \lambda(\texttt{representation} :: \mathsf{Nat} \to [\mathsf{Expr}]) \to \\
\quad (\texttt{forAll associative}\ \$\ \lambda((\oplus) :: \mathsf{Bool} \to \mathsf{Bool} \to \mathsf{Bool}) \to \\
\qquad \lambda \mathit{xs} :: [\mathsf{Bool}] \to \ldots) \\
\quad ==> \\
\quad \forall \tau.\ (\texttt{forAll associative}\ \$\ \lambda((\oplus) :: \tau \to \tau \to \tau) \to \\
\qquad\qquad \lambda \mathit{xs} :: [\tau] \to \ldots)
\end{array}
$$

Where Are We Now?

To test:

$$
\begin{array}{l}
\texttt{quickCheck}\ \$\ \lambda(\texttt{representation} :: \mathsf{Nat} \to [\mathsf{Expr}]) \to \\
\quad (\texttt{forAll associative}\ \$\ \lambda((\oplus) :: \mathsf{Bool} \to \mathsf{Bool} \to \mathsf{Bool}) \to \\
\qquad \lambda xs :: [\mathsf{Bool}] \to \ldots) \\
\quad ==> \\
\quad \forall \tau.\ (\texttt{forAll associative}\ \$\ \lambda((\oplus) :: \tau \to \tau \to \tau) \to \\
\qquad\qquad \lambda xs :: [\tau] \to \ldots)
\end{array}
$$

The standard (QuickCheck) approach here, for dealing with $\mathsf{Nat} \to [\mathsf{Expr}]$, would be:

- generate essentially partial functions

Where Are We Now?

To test:

$$
\begin{array}{l}
\texttt{quickCheck}\ \$\ \lambda(\texttt{representation} :: \mathsf{Nat} \to [\mathsf{Expr}]) \to \\
\quad (\texttt{forAll associative}\ \$\ \lambda((\oplus) :: \mathsf{Bool} \to \mathsf{Bool} \to \mathsf{Bool}) \to \\
\qquad \lambda \mathit{xs} :: [\mathsf{Bool}] \to \ldots) \\
\quad ==> \\
\quad \forall \tau.\ (\texttt{forAll associative}\ \$\ \lambda((\oplus) :: \tau \to \tau \to \tau) \to \\
\qquad\qquad \lambda \mathit{xs} :: [\tau] \to \ldots)
\end{array}
$$

The standard (QuickCheck) approach here, for dealing with $\mathsf{Nat} \to [\mathsf{Expr}]$, would be:

- generate essentially partial functions, that
- take on a "useful" / "used" result only for finitely many inputs

Where Are We Now?

To test:

$$\begin{array}{l}
\texttt{quickCheck}\ \$\ \lambda(\texttt{representation} :: \mathsf{Nat} \to [\mathsf{Expr}]) \to \\
\quad (\texttt{forAll associative}\ \$\ \lambda((\oplus) :: \mathsf{Bool} \to \mathsf{Bool} \to \mathsf{Bool}) \to \\
\qquad \lambda xs :: [\mathsf{Bool}] \to \ldots) \\
\quad ==> \\
\quad \forall \tau.\ (\texttt{forAll associative}\ \$\ \lambda((\oplus) :: \tau \to \tau \to \tau) \to \\
\qquad\qquad \lambda xs :: [\tau] \to \ldots)
\end{array}$$

The standard (QuickCheck) approach here, for dealing with $\mathsf{Nat} \to [\mathsf{Expr}]$, would be:

- generate essentially partial functions, that
- take on a “useful” / “used” result only for finitely many inputs,
- while choosing some default for all others.

Where Are We Now?

To test:

$$
\begin{array}{l}
\texttt{quickCheck}\ \$\ \lambda(\texttt{representation} :: \mathsf{Nat} \to [\mathsf{Expr}]) \to \\
\quad (\texttt{forAll associative}\ \$\ \lambda((\oplus) :: \mathsf{Bool} \to \mathsf{Bool} \to \mathsf{Bool}) \to \\
\qquad \lambda xs :: [\mathsf{Bool}] \to \ldots) \\
\quad ==> \\
\quad \forall \tau.\ (\texttt{forAll associative}\ \$\ \lambda((\oplus) :: \tau \to \tau \to \tau) \to \\
\qquad\qquad \lambda xs :: [\tau] \to \ldots)
\end{array}
$$

The standard (QuickCheck) approach here, for dealing with Nat → [Expr], would be:

- generate essentially partial functions, that
- take on a “useful” / “used” result only for finitely many inputs,
- while choosing some default for all others.

That idea is basically fine here, but we actually need some more control because of our need to check the complex precondition.

Making Partiality Explicit (and "Fixing the Default")

We go from $\mathsf{Nat} \to [\mathsf{Expr}]$ to $\mathsf{Nat} \to \mathsf{Maybe}\,[\mathsf{Expr}]$,
with the interpretation that

$$\texttt{representation}\ n = \mathsf{Nothing}$$

means to take on the `scanl1`-behavior for lists of length n.

Making Partiality Explicit (and "Fixing the Default")

We go from Nat → [Expr] to Nat → Maybe [Expr],
with the interpretation that

$$\texttt{representation}\ n = \text{Nothing}$$

means to take on the `scanl1`-behavior for lists of length n.

For example,

$$\begin{array}{llll}
\texttt{representation} :: \text{Nat} \to \text{Maybe}\ [\text{Expr}] & & & \\
\texttt{representation}\ n & \mid n == 1 & = & \text{Just}\ [\text{Op}\ (\text{Var}\ 0)\ (\text{Var}\ 0)] \\
 & \mid n == 3 & = & \text{Just}\ [\text{Var}\ 0 \\
 & & & \quad ,\text{Op}\ (\text{Var}\ 1)\ (\text{Var}\ 2)] \\
 & \mid \mathit{otherwise} & = & \text{Nothing}
\end{array}$$

corresponds to

$$\begin{array}{lll}
\texttt{candidate} :: (\alpha \to \alpha \to \alpha) \to [\alpha] \to [\alpha] & & \\
\texttt{candidate}\ (\oplus)\ [x] & = & [x \oplus x] \\
\texttt{candidate}\ (\oplus)\ [x, y, z] & = & [x, y \oplus z] \\
\texttt{candidate}\ (\oplus)\ \mathit{xs} & = & \texttt{scanl1}\ (\oplus)\ \mathit{xs}
\end{array}$$

Another Problem

Even `representation` :: Nat → Maybe [Expr] is not very "testable" for our purposes, since we can only find out which outputs are Nothing by actually applying the function.

Another Problem

Even `representation` :: Nat → Maybe [Expr] is not very "testable" for our purposes, since we can only find out which outputs are Nothing by actually applying the function.

What we need "to shortcut" the complex precondition check is explicit access to the definedness domain.

Another Problem

Even `representation` :: Nat → Maybe [Expr] is not very "testable" for our purposes, since we can only find out which outputs are Nothing by actually applying the function.

What we need "to shortcut" the complex precondition check is explicit access to the definedness domain.

So, we store only the non-Nothing positions, as key-value-pairs:

type Sparse = [(Nat, [Expr])]

Another Problem

Even `representation` :: Nat → Maybe [Expr] is not very "testable" for our purposes, since we can only find out which outputs are Nothing by actually applying the function.

What we need "to shortcut" the complex precondition check is explicit access to the definedness domain.

So, we store only the non-Nothing positions, as key-value-pairs:

```
type Sparse = [(Nat, [Expr])]
```

Previous example then corresponds to (among others):

```
sparse :: Sparse
sparse = [(1, [Op (Var 0) (Var 0)])
         , (3, [Var 0, Op (Var 1) (Var 2)])]
```

Another Problem

Even `representation` :: Nat → Maybe [Expr] is not very "testable" for our purposes, since we can only find out which outputs are Nothing by actually applying the function.

What we need "to shortcut" the complex precondition check is explicit access to the definedness domain.

So, we store only the non-Nothing positions, as key-value-pairs:

```
type Sparse = [(Nat, [Expr])]
```

Previous example then corresponds to (among others):

```
sparse :: Sparse
sparse = [(1, [Op (Var 0) (Var 0)])
         ,(3, [Var 0, Op (Var 1) (Var 2)])]
```

Actually a technical challenge now: programmatic conversion between [(Nat, [Expr])] and Nat → Maybe [Expr] ?

Excurse: from $\mathsf{Nat} \to \mathsf{Maybe}\,[\mathsf{Expr}]$ to $[(\mathsf{Nat}, [\mathsf{Expr}])]$

Problem:

- A naive “sparsification”, iterating through all natural numbers and keeping those for which the result is non-Nothing, does not work well.

Excurse: from Nat $\rightarrow$ Maybe [Expr] to [(Nat, [Expr])]

Problem:

- A naive "sparsification", iterating through all natural numbers and keeping those for which the result is non-Nothing, does not work well.
- For example, `sparsify` (`const` Nothing) would simply give $\perp$ :: Sparse, from which we could not learn anything.

Excurse: from Nat $\rightarrow$ Maybe [Expr] to [(Nat, [Expr])]

Problem:

- A naive “sparsification”, iterating through all natural numbers and keeping those for which the result is non-Nothing, does not work well.
- For example, `sparsify` (`const` Nothing) would simply give $\perp$ :: Sparse, from which we could not learn anything.

Solution:

- work with lazy natural numbers: **data** Nat $=$ *Z* | *S* Nat

Excurse: from Nat $\rightarrow$ Maybe [Expr] to [(Nat, [Expr])]

Problem:

- A naive “sparsification”, iterating through all natural numbers and keeping those for which the result is non-Nothing, does not work well.
- For example, `sparsify` (`const` Nothing) would simply give $\bot$ :: Sparse, from which we could not learn anything.

Solution:

- work with lazy natural numbers: **data** Nat $= Z \mid S$ Nat
- use an appropriately lazy function for lookup in [(Nat, [Expr])]

Excurse: from Nat $\rightarrow$ Maybe [Expr] to [(Nat, [Expr])]

Problem:

- A naive "sparsification", iterating through all natural numbers and keeping those for which the result is non-Nothing, does not work well.
- For example, `sparsify` (`const` Nothing) would simply give $\perp$ :: Sparse, from which we could not learn anything.

Solution:

- work with lazy natural numbers: **data** Nat = *Z* | *S* Nat
- use an appropriately lazy function for lookup in [(Nat, [Expr])]
- convert with care

Excurse: from $\text{Nat} \rightarrow \text{Maybe}\,[\text{Expr}]$ to $[(\text{Nat}, [\text{Expr}])]$

Problem:

- A naive “sparsification”, iterating through all natural numbers and keeping those for which the result is non-Nothing, does not work well.
- For example, `sparsify` (`const` Nothing) would simply give $\perp$ :: Sparse, from which we could not learn anything.

Solution:

- work with lazy natural numbers: **data** Nat = $Z \mid S$ Nat
- use an appropriately lazy function for lookup in $[(\text{Nat}, [\text{Expr}])]$
- convert with care
- (for convenience, actually store only the “gap lengths”)

Excurse: from Nat $\rightarrow$ Maybe [Expr] to [(Nat, [Expr])]

Solution:

- work with lazy natural numbers: **data** Nat = *Z* | *S* Nat
- use an appropriately lazy function for lookup in [(Nat, [Expr])]
- convert with care
- (for convenience, actually store only the "gap lengths")

Example:

```
representation :: Nat → Maybe [Expr]
representation n | n == 1    = Just [Op (Var 0) (Var 0)]
                 | n == 3    = Just [Var 0, ...]
                 | otherwise = Nothing
```

Excurse: from Nat → Maybe [Expr] to [(Nat, [Expr])]

Solution:

- work with lazy natural numbers: **data** Nat = *Z* | *S* Nat
- use an appropriately lazy function for lookup in [(Nat, [Expr])]
- convert with care
- (for convenience, actually store only the "gap lengths")

Example:

```
representation :: Nat → Maybe [Expr]
representation n | n == 1    = Just [Op (Var 0) (Var 0)]
                 | n == 3    = Just [Var 0, ...]
                 | otherwise = Nothing
```

now turns into:

```
sparse :: Sparse
sparse = [(S Z, [Op (Var 0) (Var 0)])
         ,(S Z, [Var 0, ...])
         ,(S (S (S ...)), ⊥)] ++ ⊥
```

Wrapping up the Transformation

We now have:

$$
\begin{array}{l}
\texttt{quickCheck}\ \$\ \lambda(\texttt{sparse} :: \mathsf{Sparse}) \to \\
\quad (\texttt{forAll associative}\ \$\ \lambda((\oplus) :: \mathsf{Bool} \to \mathsf{Bool} \to \mathsf{Bool}) \to \\
\qquad \lambda \mathit{xs} :: [\mathsf{Bool}] \to \ldots) \\
\quad {=}{=}{>} \\
\quad \forall \tau.\ (\texttt{forAll associative}\ \$\ \lambda((\oplus) :: \tau \to \tau \to \tau) \to \\
\qquad\qquad \lambda \mathit{xs} :: [\tau] \to \ldots)
\end{array}
$$

Wrapping up the Transformation

We now have:

$$\begin{array}{l}
\texttt{quickCheck}\ \$\ \lambda(\texttt{sparse} :: \mathsf{Sparse}) \rightarrow \\
\quad (\texttt{forAll associative}\ \$\ \lambda((\oplus) :: \mathsf{Bool} \rightarrow \mathsf{Bool} \rightarrow \mathsf{Bool}) \rightarrow \\
\qquad \lambda \mathit{xs} :: [\mathsf{Bool}] \rightarrow \ldots) \\
\quad {=}{=}{>} \\
\quad \forall \tau.\ (\texttt{forAll associative}\ \$\ \lambda((\oplus) :: \tau \rightarrow \tau \rightarrow \tau) \rightarrow \\
\qquad\qquad \lambda \mathit{xs} :: [\tau] \rightarrow \ldots)
\end{array}$$

Now we can:

- implement the complex precondition check
  (by handcoding or using SmallCheck machinery)

Wrapping up the Transformation

We now have:

$$
\begin{array}{l}
\texttt{quickCheck} \ \$ \ \lambda(\texttt{sparse} :: \mathsf{Sparse}) \rightarrow \\
\quad (\texttt{forAll associative} \ \$ \ \lambda((\oplus) :: \mathsf{Bool} \rightarrow \mathsf{Bool} \rightarrow \mathsf{Bool}) \rightarrow \\
\qquad \lambda \mathit{xs} :: [\mathsf{Bool}] \rightarrow \ldots) \\
\quad ==> \\
\quad \forall \tau. \ (\texttt{forAll associative} \ \$ \ \lambda((\oplus) :: \tau \rightarrow \tau \rightarrow \tau) \rightarrow \\
\qquad\qquad \lambda \mathit{xs} :: [\tau] \rightarrow \ldots)
\end{array}
$$

Now we can:

- implement the complex precondition check
  (by handcoding or using SmallCheck machinery)
- generalize to other finite types

Wrapping up the Transformation

We now have:

$$
\begin{array}{l}
\texttt{quickCheck}\ \$\ \lambda(\texttt{sparse} :: \mathsf{Sparse}) \rightarrow \\
\quad (\texttt{forAll associative}\ \$\ \lambda((\oplus) :: \mathsf{Bool} \rightarrow \mathsf{Bool} \rightarrow \mathsf{Bool}) \rightarrow \\
\qquad \lambda \mathit{xs} :: [\mathsf{Bool}] \rightarrow \ldots) \\
\quad {=}{=}{>} \\
\quad \forall \tau.\ (\texttt{forAll associative}\ \$\ \lambda((\oplus) :: \tau \rightarrow \tau \rightarrow \tau) \rightarrow \\
\qquad\qquad \lambda \mathit{xs} :: [\tau] \rightarrow \ldots)
\end{array}
$$

Now we can:

- implement the complex precondition check
  (by handcoding or using SmallCheck machinery)
- generalize to other finite types
- experiment with different choices for τ

Wrapping up the Transformation

We now have:

$$
\begin{array}{l}
\texttt{quickCheck}\ \$\ \lambda(\texttt{sparse} :: \mathsf{Sparse}) \rightarrow \\
\quad (\texttt{forAll associative}\ \$\ \lambda((\oplus) :: \mathsf{Bool} \rightarrow \mathsf{Bool} \rightarrow \mathsf{Bool}) \rightarrow \\
\qquad \lambda \mathit{xs} :: [\mathsf{Bool}] \rightarrow \ldots) \\
\quad ==> \\
\quad \forall \tau.\ (\texttt{forAll associative}\ \$\ \lambda((\oplus) :: \tau \rightarrow \tau \rightarrow \tau) \rightarrow \\
\qquad\qquad \lambda \mathit{xs} :: [\tau] \rightarrow \ldots)
\end{array}
$$

Now we can:

- implement the complex precondition check
  (by handcoding or using SmallCheck machinery)
- generalize to other finite types
- experiment with different choices for τ
- experiment with different generators for Sparse-candidates
- experiment with QuickCheck vs. SmallCheck

Some Counterexamples Thus Found

$$\begin{aligned}
&\texttt{candidate} :: (\alpha \to \alpha \to \alpha) \to [\alpha] \to [\alpha] \\
&\texttt{candidate}\ (\oplus)\ [x_0] &&= [x_0] \\
&\texttt{candidate}\ (\oplus)\ [x_0, x_1] &&= [x_0, (((x_0 \oplus x_1) \oplus x_0) \oplus x_0) \oplus (x_1 \oplus x_1)] \\
&\texttt{candidate}\ (\oplus)\ xs &&= \texttt{scanl1}\ (\oplus)\ xs
\end{aligned}$$

Some Counterexamples Thus Found

$$\begin{aligned}
&\texttt{candidate} :: (\alpha \to \alpha \to \alpha) \to [\alpha] \to [\alpha]\\
&\texttt{candidate}\ (\oplus)\ [x_0] &&= [x_0]\\
&\texttt{candidate}\ (\oplus)\ [x_0, x_1] &&= [x_0, (((x_0 \oplus x_1) \oplus x_0) \oplus x_0) \oplus (x_1 \oplus x_1)]\\
&\texttt{candidate}\ (\oplus)\ xs &&= \texttt{scanl1}\ (\oplus)\ xs
\end{aligned}$$

$$\begin{aligned}
&\texttt{candidate} :: (\alpha \to \alpha \to \alpha) \to [\alpha] \to [\alpha]\\
&\texttt{candidate}\ (\oplus)\ [x_0] &&= [x_0]\\
&\texttt{candidate}\ (\oplus)\ [x_0, x_1] &&= [x_0, ((x_0 \oplus x_0) \oplus x_0) \oplus x_1]\\
&\texttt{candidate}\ (\oplus)\ xs &&= \texttt{scanl1}\ (\oplus)\ xs
\end{aligned}$$

Some Counterexamples Thus Found

$$\begin{aligned}
&\texttt{candidate} :: (\alpha \to \alpha \to \alpha) \to [\alpha] \to [\alpha] \\
&\texttt{candidate}\ (\oplus)\ [x_0] &&= [x_0] \\
&\texttt{candidate}\ (\oplus)\ [x_0, x_1] &&= [x_0, (((x_0 \oplus x_1) \oplus x_0) \oplus x_0) \oplus (x_1 \oplus x_1)] \\
&\texttt{candidate}\ (\oplus)\ xs &&= \texttt{scanl1}\ (\oplus)\ xs
\end{aligned}$$

$$\begin{aligned}
&\texttt{candidate} :: (\alpha \to \alpha \to \alpha) \to [\alpha] \to [\alpha] \\
&\texttt{candidate}\ (\oplus)\ [x_0] &&= [x_0] \\
&\texttt{candidate}\ (\oplus)\ [x_0, x_1] &&= [x_0, ((x_0 \oplus x_0) \oplus x_0) \oplus x_1] \\
&\texttt{candidate}\ (\oplus)\ xs &&= \texttt{scanl1}\ (\oplus)\ xs
\end{aligned}$$

$$\begin{aligned}
&\texttt{candidate} :: (\alpha \to \alpha \to \alpha) \to [\alpha] \to [\alpha] \\
&\texttt{candidate}\ (\oplus)\ [x_0, x_1, x_2] &&= [x_0, (x_0 \oplus x_0) \oplus (x_0 \oplus x_1) \\
& &&\quad , x_0 \oplus (x_1 \oplus x_2)] \\
&\texttt{candidate}\ (\oplus)\ xs &&= \texttt{scanl1}\ (\oplus)\ xs
\end{aligned}$$

Recall: Plan of the Talk

- Start telling the story of how "0-1-2", $\oplus_1$, $\oplus_2$, ... were found (back in 2007, never recorded, but interesting I think).
- Challenge you to suggest other approaches to discover the required counterexamples and proof ingredients?
- Invite complaints about where the presented (deliberately naive, exploratory) approach is too ad-hoc, or unacceptably pulls a rabbit out of a hat.
- Provoke investigation/proposals of other algorithm classes on which one could try to play the same or similar trick(s)?

References I

G.E. Blelloch.
Prefix sums and their applications.
In J.H. Reif, editor, *Synthesis of Parallel Algorithms*, pages 35–60. Morgan Kaufmann, 1993.

S. Böhme.
Much Ado about Two. Formal proof development.
In *The Archive of Formal Proofs*.
`http://afp.sf.net/entries/MuchAdoAboutTwo.shtml`, 2007.

N.A. Day, J. Launchbury, and J.R. Lewis.
Logical abstractions in Haskell.
In *Haskell Workshop, Proceedings*, 1999.

References II

D.E. Knuth.
The Art of Computer Programming, volume 3: Sorting and Searching.
Addison-Wesley, 1973.

J.C. Reynolds.
Types, abstraction and parametric polymorphism.
In *Information Processing, Proceedings*, pages 513–523.
Elsevier Science Publishers B.V., 1983.

M. Sheeran.
Searching for prefix networks to fit in a context using a lazy functional programming language.
Hardware Design and Functional Languages, 2007.

References III

J. Sklansky.
Conditional-sum addition logic.
IRE Transactions on Electronic Computers, EC-9(6):226–231, 1960.

J. Voigtländer.
Much ado about two: A pearl on parallel prefix computation.
In *Principles of Programming Languages, Proceedings*, pages 29–35. ACM Press, 2008.

P. Wadler.
Theorems for free!
In *Functional Programming Languages and Computer Architecture, Proceedings*, pages 347–359. ACM Press, 1989.