Program Construction and Reasoning

Shin-Cheng Mu

Institute of Information Science, Academia Sinica, Taiwan

2008 Formosan Summer School on
Logic, Language, and Computation
June 30 – July 11, 2008

So I Was Asked...

- "So, you study about computers? What programs have you written?"

So I Was Asked...

- "So, you study about computers? What programs have you written?"
- I had to explain that my research is more about how to construct correct programs.

So I Was Asked...

- "So, you study about computers? What programs have you written?"
- I had to explain that my research is more about how to construct correct programs.
- *Correctness*: that a program does what it is supposed to do.

So I Was Asked...

- "So, you study about computers? What programs have you written?"
- I had to explain that my research is more about how to construct correct programs.
- *Correctness*: that a program does what it is supposed to do.
- "What do you mean? Doesn't a program always does what it is told to do?"

Maximum Segment Sum

- Given a list of numbers, find the maximum sum of a *consecutive* segment.
  - $[-1, 3, 3, -4, -1, 4, 2, -1] \;\Rightarrow\; 7$
  - $[-1, 3, 1, -4, -1, 4, 2, -1] \;\Rightarrow\; 6$
  - $[-1, 3, 1, -4, -1, 1, 2, -1] \;\Rightarrow\; 4$

Maximum Segment Sum

- Given a list of numbers, find the maximum sum of a *consecutive* segment.
  - $[-1, 3, 3, -4, -1, 4, 2, -1] \Rightarrow 7$
  - $[-1, 3, 1, -4, -1, 4, 2, -1] \Rightarrow 6$
  - $[-1, 3, 1, -4, -1, 1, 2, -1] \Rightarrow 4$
- Not trivial. However, there is a linear time algorithm.

Maximum Segment Sum

- Given a list of numbers, find the maximum sum of a *consecutive* segment.
  - $[-1, 3, 3, -4, -1, 4, 2, -1] \;\Rightarrow\; 7$
  - $[-1, 3, 1, -4, -1, 4, 2, -1] \;\Rightarrow\; 6$
  - $[-1, 3, 1, -4, -1, 1, 2, -1] \;\Rightarrow\; 4$

$$
\begin{array}{cccccccccl}
-1 & 3 & 1 & -4 & -1 & 1 & 2 & -1 & & \\
3 & 4 & 1 & 0 & 2 & 3 & 2 & 0 & 0 & (up + right) \uparrow 0 \\
4 & 4 & 3 & 3 & 3 & 3 & 2 & 0 & 0 & up \uparrow right
\end{array}
$$

A Simple Program Whose Proof is Not

- The specification: $max\,\{\,sum\,(i,j)\,|\,0 \leq i \leq j \leq N\,\}$, where $sum\,(i,j) = a[i] + a[i+1] + \ldots + a[j]$.
- The program:

```
s = 0; m = 0;
for (i=0; i<=N; i++) {
   s = max(0, a[j]+s);
   m = max(m, s);
}
```

- They do not look like each other at all!
  - Moral: programs that appear "simple" might not be really that simple!

A Simple Program Whose Proof is Not

- The specification: $max\,\{\,sum\,(i,j)\,|\,0 \le i \le j \le N\,\}$, where $sum\,(i,j) = a[i] + a[i+1] + \ldots + a[j]$.
- The program:

```
s = 0; m = 0;
for (i=0; i<=N; i++) {
    s = max(0, a[j]+s);
    m = max(m, s);
}
```

- They do not look like each other at all!
  - Moral: programs that appear "simple" might not be really that simple!
- When we are given only the specification, can we construct the program?

Verification v.s. Derivation

How do we know a program is correct with respect to a specification?

- Verification: given a program, prove that it is correct with respect to some specification.
- Derivation: start from the specification, and attempt to construct *only* correct programs!

Theoretical development of one side benefits the other.

Program Derivation

- Wikipedia: *program derivation* is the derivation of a program from its specification, by mathematical means.
- To write a formal specification (which could be non-executable), and then apply mathematically correct rules in order to obtain an executable program.
- The program thus obtained is correct by construction.

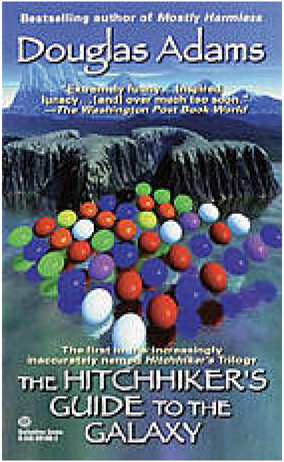
A Typical Derivation

$$
\begin{array}{ll}
& max\,\{\,sum\,(i,j)\,|\,0 \leq i \leq j \leq N\,\} \\
= & \quad \{\text{Premise 1}\} \\
& max \cdot map\,sum \cdot concat \cdot map\,inits \cdot tails \\
= & \quad \{\text{Premise 2}\} \\
& \ldots \\
= & \quad \{\ldots\} \\
& \text{The final program!}
\end{array}
$$

It's How We Get There That Matters!

$$
\begin{array}{cl}
 & \text{Meaning of Life} \\
= & \quad \{\text{Premise 1}\} \\
 & \ldots \\
= & \quad \{\text{Premise 2}\} \\
 & \ldots \\
= & \quad \{\ldots\} \\
 & 42!
\end{array}
$$

The answer may be simple. It is how we get there that matters.



Functional Programming

- In program derivation, programs are entities we manipulate. Procedural programs (e.g. C programs), however, are difficult to manipulate because they lack nice properties.
- In C, we do not even have $f(3) + f(3) = 2 \times f(3)$.
- In functional programming, programs are viewed as mathematical functions that can be reasoned algebraically, thus more suitable for program derivation.
- However, we will talk about procedural program derivation in the latter part of this course.

Functions

- For the purpose of this lecture, it suffices to *assume* that functional programs actually denote functions from sets to sets.
  - The reality is more complicated. But that is out of the scope of this course.
- Functions can be viewed as sets of pairs, each specifies an input-output mapping.
  - E.g. the function *square* is specified by $\{(1,1),(2,4),(3,9)\ldots\}$.
  - Function application is denoted by juxtaposition, e.g. $\mathit{square}\,3$.
- Given $f :: \alpha \rightarrow \beta$ and $g :: \beta \rightarrow \gamma$, their composition $g \cdot f :: \alpha \rightarrow \gamma$ is defined by $(g \cdot f)\, a = g\,(f\, a)$.

Recursively Defined Functions

- Functions (or total functions) can be recursively defined:

$$\begin{array}{lcl} \mathit{fact}\,0 & = & 1, \\ \mathit{fact}\,(n+1) & = & (n+1) \times \mathit{fact}\,n. \end{array}$$

  As a simplified view, we take *fact* as the *least* set satisfying the equations above.
  - As a result, any *total* function satisfying the equations above is *fact*. *This is a long story cut short, however!*
- Applying *fact* to a value:

$$\begin{array}{cl} & \mathit{fact}\,3 \\ = & 3 \times \mathit{fact}\,2 \\ = & 3 \times 2 \times \mathit{fact}\,1 \\ = & 3 \times 2 \times \mathit{fact}\,1 \\ = & 3 \times 2 \times 1 \times 1. \end{array}$$

Natural Numbers and Lists

- Natural numbers: $data\, N = 0 \mid 1 + N$.
  - E.g. 3 can be seen as being composed out of $1 + (1 + (1 + 0))$.
- Lists: $data\, [a] \;=\; [\,] \mid a : [a]$.
  - A list with three items 1, 2, and 3 is constructed by $1 : (2 : (3 : [\,]))$, abbreviated as $[1, 2, 3]$.
  - $hd\, (x : xs) = x$.
  - $tl\, (x : xs) = xs$.
- Noticed some similarities?

Binary Trees

For this course, we will use two kinds of binary trees: internally labelled trees, and externally labelled ones:

- $\mathit{data}\ \mathit{iTree}\ \alpha\ =\ \mathit{Null}\ |\ \mathit{Node}\ \alpha\ (\mathit{iTree}\ \alpha)\ (\mathit{iTree}\ \alpha)$.
  - E.g. $\mathit{Node}\ 3\ (\mathit{Node}\ 2\ \mathit{Null}\ \mathit{Null})\ (\mathit{Node}\ 1\ \mathit{Null}\ (\mathit{Node}\ 4\ \mathit{Null}\ \mathit{Null}))$.
- $\mathit{data}\ \mathit{eTree}\ \alpha\ =\ \mathit{Tip}\ a\ |\ \mathit{Bin}\ (\mathit{eTree}\ \alpha)\ (\mathit{eTree}\ \alpha)$.
  - E.g. $\mathit{Bin}\ (\mathit{Bin}\ (\mathit{Tip}\ 1)\ (\mathit{Tip}\ 2))\ (\mathit{Tip}\ 3)$.

Sum and Map

- The function *sum* adds up the numbers in a list:

$$\begin{array}{lcl} sum & :: & [Int] \rightarrow Int \\ sum\,[\,] & = & 0 \\ sum\,(x : xs) & = & x + sum\,xs. \end{array}$$

  - E.g. $sum\,[7, 9, 11] = 27$.

- The function *map f* takes a list and builds a new list by applying *f* to every item in the input:

$$\begin{array}{lcl} map & :: & (\alpha \rightarrow \beta) \rightarrow [\alpha] \rightarrow [\beta] \\ map\,f\,[\,] & = & [\,] \\ map\,f\,(x : xs) & = & f\,x : map\,f\,xs. \end{array}$$

  - E.g. $map\,square\,[3, 4, 6] = [9, 16, 36]$.

Sum of Squares

- Given a sequence $a_1, a_2, \ldots, a_n$, compute $a_1^2 + a_2^2 + \ldots + a_n^2$.
  Specification: $sumsq = sum \cdot map\,square$.
- The spec. builds an intermediate list. Can we eliminate it?
- The input is either empty or not. When it is empty:

$$sumsq\,[\,]$$

Sum of Squares

- Given a sequence a_1,a_2,...,a_n, compute $a_1^2 + a_2^2 + \ldots + a_n^2$.
  Specification: $sumsq = sum \cdot map\ square$.
- The spec. builds an intermediate list. Can we eliminate it?
- The input is either empty or not. When it is empty:

$$
\begin{aligned}
& sumsq\ [\,] \\
= \quad & \{ \text{ definition of } sumsq \ \} \\
& (sum \cdot map\ square)\ [\,]
\end{aligned}
$$

Sum of Squares

- Given a sequence $a_1, a_2, \ldots, a_n$, compute $a_1^2 + a_2^2 + \ldots + a_n^2$. Specification: $sumsq = sum \cdot map\ square$.
- The spec. builds an intermediate list. Can we eliminate it?
- The input is either empty or not. When it is empty:

$$
\begin{aligned}
& sumsq\,[\,] \\
= & \quad \{ \text{ definition of } sumsq \text{ } \} \\
& (sum \cdot map\ square)\,[\,] \\
= & \quad \{ \text{ function composition } \} \\
& sum\,(map\ square\,[\,])
\end{aligned}
$$

Sum of Squares

- Given a sequence $a_1, a_2, \ldots, a_n$, compute $a_1^2 + a_2^2 + \ldots + a_n^2$.
  Specification: $\mathit{sumsq} = \mathit{sum} \cdot \mathit{map}\ \mathit{square}$.
- The spec. builds an intermediate list. Can we eliminate it?
- The input is either empty or not. When it is empty:

$$
\begin{aligned}
& \mathit{sumsq}\ [\,] \\
= & \quad \{ \text{ definition of } \mathit{sumsq} \ \} \\
& (\mathit{sum} \cdot \mathit{map}\ \mathit{square})\ [\,] \\
= & \quad \{ \text{ function composition } \} \\
& \mathit{sum}\ (\mathit{map}\ \mathit{square}\ [\,]) \\
= & \quad \{ \text{ definition of } \mathit{map} \ \} \\
& \mathit{sum}\ [\,]
\end{aligned}
$$

Sum of Squares

- Given a sequence $a_1, a_2, \ldots, a_n$, compute $a_1^2 + a_2^2 + \ldots + a_n^2$. Specification: $sumsq = sum \cdot map\ square$.
- The spec. builds an intermediate list. Can we eliminate it?
- The input is either empty or not. When it is empty:

$$
\begin{array}{ll}
 & sumsq\ [\,] \\
= & \quad \{ \text{ definition of } sumsq \ \} \\
 & (sum \cdot map\ square)\ [\,] \\
= & \quad \{ \text{ function composition } \} \\
 & sum\ (map\ square\ [\,]) \\
= & \quad \{ \text{ definition of } map \ \} \\
 & sum\ [\,] \\
= & \quad \{ \text{ definition of } sum \ \} \\
 & 0.
\end{array}
$$

Sum of Squares, the Inductive Case

- Consider the case when the input is not empty:

$$sumsq\ (x : xs)$$

Sum of Squares, the Inductive Case

- Consider the case when the input is not empty:

$$
\begin{array}{ll}
 & \mathit{sumsq}\,(x : \mathit{xs}) \\
= & \quad \{\ \text{definition of } \mathit{sumsq}\ \} \\
 & \mathit{sum}\,(\mathit{map}\,\mathit{square}\,(x : \mathit{xs}))
\end{array}
$$

Sum of Squares, the Inductive Case

- Consider the case when the input is not empty:

$$
\begin{array}{ll}
 & sumsq\,(x : xs) \\
= & \quad \{ \text{ definition of } sumsq \ \} \\
 & sum\,(map\;square\,(x : xs)) \\
= & \quad \{ \text{ definition of } map \ \} \\
 & sum\,(square\;x : map\;square\;xs)
\end{array}
$$

Sum of Squares, the Inductive Case

- Consider the case when the input is not empty:

$$
\begin{aligned}
& \mathit{sumsq}\,(x : xs) \\
= \quad & \{\ \text{definition of } \mathit{sumsq}\ \} \\
& \mathit{sum}\,(\mathit{map}\,\mathit{square}\,(x : xs)) \\
= \quad & \{\ \text{definition of } \mathit{map}\ \} \\
& \mathit{sum}\,(\mathit{square}\,x : \mathit{map}\,\mathit{square}\,xs) \\
= \quad & \{\ \text{definition of } \mathit{sum}\ \} \\
& \mathit{square}\,x + \mathit{sum}\,(\mathit{map}\,\mathit{square}\,xs)
\end{aligned}
$$

Sum of Squares, the Inductive Case

- Consider the case when the input is not empty:

$$
\begin{aligned}
& \mathit{sumsq}\,(x : xs) \\
= & \quad \{ \text{ definition of } \mathit{sumsq} \ \} \\
& \mathit{sum}\,(\mathit{map}\,\mathit{square}\,(x : xs)) \\
= & \quad \{ \text{ definition of } \mathit{map} \ \} \\
& \mathit{sum}\,(\mathit{square}\,x : \mathit{map}\,\mathit{square}\,xs) \\
= & \quad \{ \text{ definition of } \mathit{sum} \ \} \\
& \mathit{square}\,x + \mathit{sum}\,(\mathit{map}\,\mathit{square}\,xs) \\
= & \quad \{ \text{ definition of } \mathit{sumsq} \ \} \\
& \mathit{square}\,x + \mathit{sumsq}\,xs.
\end{aligned}
$$

We have therefore constructed a recursive definition of sumsq:

$$
\begin{aligned}
\mathit{sumsq}\,[\,] &= 0 \\
\mathit{sumsq}\,(x : xs) &= \mathit{square}\,x + \mathit{sumsq}\,xs.
\end{aligned}
$$

Unfold/Fold Transformation

- Perhaps the most intuitive, yet still handy, style of functional program derivation.
- Keep unfolding the definition of functions, apply necessary rules, and finally fold the definition back.
- It works under the assumption that a function satisfying the derived equations *is* the function defined by the equations.
- In this course, we use the terms “fold” and “unfold” for another purpose. Therefore we refer to this technique as the expand/reduce transformation.

Proving Auxiliary Properties

- Our style of program derivation:

$$\begin{array}{ll} & \textit{expression} \\ = & \quad \{\text{some property}\} \\ & \ldots . \end{array}$$

- Some of the properties are rather obvious. Some needs to be proved separately.
- In this section we will practice perhaps the most fundamental proof technique, which is still very useful.

The Induction Principle

- Recall the so called "mathematical induction". To prove that a property p holds for all natural numbers, we need to show:
  - that p holds for 0, and
  - if p holds for n, it holds for $n + 1$ as well.
- We can do so because the set of natural numbers is an *inductive type*.
- The type of *finite* lists is an inductive types too. Therefore the property p holds for all finite lists if
  - property p holds for $[\,]$, and
  - if p holds for xs, it holds for $x : xs$ as well.

Appending Two Lists

- The function $(+\!\!+)$ appends two lists into one:

$$
\begin{array}{lcl}
(+\!\!+) & :: & [a] \to [a] \to [a] \\
[\,] +\!\!+ ys & = & ys \\
(x : xs) +\!\!+ ys & = & x : (xs +\!\!+ ys).
\end{array}
$$

- E.g.

$$
\begin{array}{ll}
& [1,2] +\!\!+ [3,4] \\
= & 1 : ([2] +\!\!+ [3,4]) \\
= & 1 : (2 : ([\,] +\!\!+ [3,4])) \\
= & 1 : (2 : [3,4]) \\
= & [1,2,3,4].
\end{array}
$$

- The time it takes to compute $xs +\!\!+ ys$ is proportional to the length of x.

Sum Distributes into Append

Example: let us show that $\mathit{sum}\,(\mathit{xs} +\!\!+\, \mathit{ys}) = \mathit{sum}\,\mathit{xs} + \mathit{sum}\,\mathit{ys}$, for finite lists xs and ys.
Case $[\,]$:

$$\mathit{sum}\,[\,] + \mathit{sum}\,\mathit{ys}$$

Sum Distributes into Append

Example: let us show that $sum\,(xs +\!\!+\, ys) = sum\,xs + sum\,ys$, for finite lists xs and ys.
Case $[\,]$:

$$
\begin{aligned}
& sum\,[\,] + sum\,ys \\
= \quad & \{\text{ definition of } sum \text{ }\} \\
& 0 + sum\,ys
\end{aligned}
$$

Sum Distributes into Append

Example: let us show that $\mathit{sum}\,(\mathit{xs} \mathbin{+\!\!+} \mathit{ys}) = \mathit{sum}\,\mathit{xs} + \mathit{sum}\,\mathit{ys}$, for finite lists xs and ys.
Case $[\,]$:

$$
\begin{array}{ll}
 & \mathit{sum}\,[\,] + \mathit{sum}\,\mathit{ys} \\
= & \quad \{\ \text{definition of } \mathit{sum}\ \} \\
 & 0 + \mathit{sum}\,\mathit{ys} \\
= & \quad \{\ \text{arithmetic}\ \} \\
 & \mathit{sum}\,\mathit{ys}
\end{array}
$$

Sum Distributes into Append

Example: let us show that $sum\,(xs ++ ys) = sum\,xs + sum\,ys$, for finite lists xs and ys.
Case $[\,]$:

$$
\begin{array}{ll}
 & sum\,[\,] + sum\,ys \\
= & \quad \{ \text{ definition of } sum \text{ } \} \\
 & 0 + sum\,ys \\
= & \quad \{ \text{ arithmetic } \} \\
 & sum\,ys \\
= & \quad \{ \text{ by definition of } (++),\ [\,] ++ ys = ys \text{ } \} \\
 & sum\,([\,] ++ ys).
\end{array}
$$

Sum Distributes into Append, the Inductive Case

Case $x : xs$:

$$sum\,(x : xs) + sum\,ys$$

Sum Distributes into Append, the Inductive Case

Case $x : xs$:

$$
\begin{aligned}
& \mathit{sum}\,(x : xs) + \mathit{sum}\,ys \\
= \quad & \{\ \text{definition of } \mathit{sum} \} \\
& (x + \mathit{sum}\,xs) + \mathit{sum}\,ys
\end{aligned}
$$

Sum Distributes into Append, the Inductive Case

Case $x : xs$:

$$
\begin{aligned}
& \mathit{sum}\,(x : xs) + \mathit{sum}\,ys \\
= \quad & \{\ \text{definition of } \mathit{sum} \} \\
& (x + \mathit{sum}\,xs) + \mathit{sum}\,ys \\
= \quad & \{\ (+) \text{ is associative: } (a + b) + c = a + (b + c)\ \} \\
& x + (\mathit{sum}\,xs + \mathit{sum}\,ys)
\end{aligned}
$$

Sum Distributes into Append, the Inductive Case

Case $x : xs$:

$$
\begin{aligned}
& \mathit{sum}\,(x : xs) + \mathit{sum}\,ys \\
= \quad & \{ \text{ definition of } \mathit{sum} \} \\
& (x + \mathit{sum}\,xs) + \mathit{sum}\,ys \\
= \quad & \{ (+) \text{ is associative: } (a + b) + c = a + (b + c) \} \\
& x + (\mathit{sum}\,xs + \mathit{sum}\,ys) \\
= \quad & \{ \text{ induction hypothesis } \} \\
& x + \mathit{sum}(xs +\!\!+\, ys)
\end{aligned}
$$

Sum Distributes into Append, the Inductive Case

Case $x : xs$:

$$
\begin{aligned}
& sum\,(x : xs) + sum\,ys \\
= \quad & \{ \text{ definition of } sum \} \\
& (x + sum\,xs) + sum\,ys \\
= \quad & \{ \ (+) \text{ is associative: } (a + b) + c = a + (b + c) \ \} \\
& x + (sum\,xs + sum\,ys) \\
= \quad & \{ \text{ induction hypothesis } \} \\
& x + sum(xs +\!\!+\, ys) \\
= \quad & \{ \text{ definition of } sum \ \} \\
& sum(x : (xs +\!\!+\, ys))
\end{aligned}
$$

Sum Distributes into Append, the Inductive Case

Case $x : xs$:

$$
\begin{aligned}
& \mathit{sum}\,(x : xs) + \mathit{sum}\,ys \\
= \quad & \{ \text{ definition of } \mathit{sum} \} \\
& (x + \mathit{sum}\,xs) + \mathit{sum}\,ys \\
= \quad & \{ \ (+) \text{ is associative: } (a + b) + c = a + (b + c) \ \} \\
& x + (\mathit{sum}\,xs + \mathit{sum}\,ys) \\
= \quad & \{ \text{ induction hypothesis } \} \\
& x + \mathit{sum}(xs +\!\!+\, ys) \\
= \quad & \{ \text{ definition of } \mathit{sum} \ \} \\
& \mathit{sum}(x : (xs +\!\!+\, ys)) \\
= \quad & \{ \text{ definition of } (+\!\!+) \ \} \\
& \mathit{sum}((x : xs) +\!\!+\, ys).
\end{aligned}
$$

Some Properties to be Proved

The following properties are left as exercises for you to prove. We will make use of some of them in the lecture.

- Concatenation is associative:

  $$(xs +\!\!+ \; ys) +\!\!+ \; zs \quad = \quad xs +\!\!+ \; (ys +\!\!+ \; zs).$$

  (Note that the right-hand side is in general faster than the left-hand side.)

- The function *concat* concatenates a list of lists:

  $$\begin{aligned} concat\,[\,] &\quad = \quad [\,], \\ concat\,(xs : xss) &\quad = \quad xs +\!\!+ \; concat\; xss. \end{aligned}$$

  E.g. $concat\,[[1,2],[3,4],[5]] = [1,2,3,4,5]$. We have $sum \cdot concat = sum \cdot map\, sum$.

Inductive Proofs on Trees

Recall the datatype:

$$\mathit{data}\ \mathit{iTree}\ \alpha \ = \ \mathit{Null} \mid \mathit{Node}\ \alpha\ (\mathit{iTree}\ \alpha)\ (\mathit{iTree}\ \alpha).$$

What is the induction principle for *iTree*?
A property *p* holds for all finite *iTree*s if ...

Inductive Proofs on Trees

Recall the datatype:

$$\mathit{data}\ \mathit{iTree}\,\alpha \;=\; \mathit{Null} \mid \mathit{Node}\,\alpha\,(\mathit{iTree}\,\alpha)\,(\mathit{iTree}\,\alpha).$$

What is the induction principle for *iTree*?
A property *p* holds for all finite *iTree*s if ...

- the property *p* holds for *Null*, and
- for all *a*,*t*,and *u*, if *p* holds for *t* and *u*, then *p* holds for *Node a t u*.

Example: Reversing a List

- The function *reverse* is defined by:

$$\begin{array}{lcl} reverse\,[\,] & = & [\,], \\ reverse\,(x : xs) & = & reverse\,xs \mathbin{+\!\!+} [x]. \end{array}$$

  E.g.
  $reverse\,[1, 2, 3, 4] = ((([\,] \mathbin{+\!\!+} [4]) \mathbin{+\!\!+} [3]) \mathbin{+\!\!+} [2]) \mathbin{+\!\!+} [1] = [4, 3, 2, 1]$.
- But how about its time complexity? Since $(\mathbin{+\!\!+})$ is $O(n)$, it takes $O(n^2)$ time to revert a list this way.
- Can we make it faster?

Introducing an Accumulating Parameter

- Let us consider a generalisation of *reverse*. Define:

$$rcat\ xs\ ys \quad = \quad reverse\ xs \mathbin{+\!\!+} ys.$$

- If we can construct a fast implementation of *rcat*, we can implement *reverse* by:

$$reverse\ xs \quad = \quad rcat\ xs\ [\,].$$

Reversing a List, Base Case

Let us use our old trick of Expand/Reduce transformation.
Consider the case when xs is $[\,]$:

$$rcat\ [\,]\ ys$$

Reversing a List, Base Case

Let us use our old trick of Expand/Reduce transformation.
Consider the case when xs is $[\,]$:

$$
\begin{array}{ll}
 & rcat\,[\,]\,ys \\
= & \quad \{ \text{ definition of } rcat \text{ } \} \\
 & reverse\,[\,] +\!\!+\, ys
\end{array}
$$

Reversing a List, Base Case

Let us use our old trick of Expand/Reduce transformation.
Consider the case when xs is $[\,]$:

$$
\begin{array}{ll}
 & rcat\ [\,]\ ys \\
= & \quad \{\ \text{definition of } rcat\ \} \\
 & reverse\ [\,] \mathbin{+\!\!+} ys \\
= & \quad \{\ \text{definition of } reverse\ \} \\
 & [\,] \mathbin{+\!\!+} ys
\end{array}
$$

Reversing a List, Base Case

Let us use our old trick of Expand/Reduce transformation.
Consider the case when xs is $[\,]$:

$$
\begin{aligned}
& \mathit{rcat}\,[\,]\,\mathit{ys} \\
= \quad & \{ \text{ definition of } \mathit{rcat} \} \\
& \mathit{reverse}\,[\,] \mathbin{+\!\!+} \mathit{ys} \\
= \quad & \{ \text{ definition of } \mathit{reverse} \} \\
& [\,] \mathbin{+\!\!+} \mathit{ys} \\
= \quad & \{ \text{ definition of } (\mathbin{+\!\!+}) \} \\
& \mathit{ys}.
\end{aligned}
$$

Reversing a List, Inductive Case

Case $x : xs$:

$$rcat\,(x : xs)\,ys$$

Reversing a List, Inductive Case

Case $x : xs$:

$$
\begin{array}{ll}
 & rcat\,(x : xs)\,ys \\
= & \quad \{\ \text{definition of } rcat\ \} \\
 & reverse\,(x : xs) +\!\!+\, ys
\end{array}
$$

Reversing a List, Inductive Case

Case $x : xs$:

$$
\begin{aligned}
& rcat\,(x : xs)\,ys \\
= \quad & \{\ \text{definition of } rcat\ \} \\
& reverse\,(x : xs) \mathbin{+\!\!+} ys \\
= \quad & \{\ \text{definition of } reverse\ \} \\
& (reverse\,xs \mathbin{+\!\!+} [x]) \mathbin{+\!\!+} ys
\end{aligned}
$$

Reversing a List, Inductive Case

Case $x : xs$:

$$
\begin{array}{ll}
 & rcat\,(x : xs)\,ys \\
= & \quad \{ \text{ definition of } rcat \ \} \\
 & reverse\,(x : xs) +\!\!+\, ys \\
= & \quad \{ \text{ definition of } reverse \ \} \\
 & (reverse\,xs +\!\!+ [x]) +\!\!+\, ys \\
= & \quad \{ \text{ since } (xs +\!\!+ ys) +\!\!+ zs = xs +\!\!+ (ys +\!\!+ zs) \ \} \\
 & reverse\,xs +\!\!+ ([x] +\!\!+ ys)
\end{array}
$$

Reversing a List, Inductive Case

Case $x : xs$:

$$
\begin{aligned}
& \mathit{rcat}\,(x : xs)\,ys \\
= \quad & \{\ \text{definition of } \mathit{rcat}\ \} \\
& \mathit{reverse}\,(x : xs) \mathbin{+\!\!+} ys \\
= \quad & \{\ \text{definition of } \mathit{reverse}\ \} \\
& (\mathit{reverse}\,xs \mathbin{+\!\!+} [x]) \mathbin{+\!\!+} ys \\
= \quad & \{\ \text{since } (xs \mathbin{+\!\!+} ys) \mathbin{+\!\!+} zs = xs \mathbin{+\!\!+} (ys \mathbin{+\!\!+} zs)\ \} \\
& \mathit{reverse}\,xs \mathbin{+\!\!+} ([x] \mathbin{+\!\!+} ys) \\
= \quad & \{\ \text{definition of } \mathit{rcat}\ \} \\
& \mathit{rcat}\,xs\,(x : ys).
\end{aligned}
$$

Linear-Time List Reversal

- We have therefore constructed an implementation of *rcat*:

$$\begin{aligned} rcat\,[\,]\,ys &= ys \\ rcat\,(x : xs)\,ys &= rcat\,xs\,(x : ys), \end{aligned}$$

  which runs in linear time!
- A generalisation of *reverse* is easier to implement than *reverse* itself? How come?
- If you try to understand *rcat* operationally, it is not difficult to see how it works.
  - The partially reverted list is *accumulated* in *ys*.
  - The initial value of *ys* is set by $reverse\,xs = rcat\,xs\,[\,]$.
  - Hmm... it is like a *loop*, isn't it?

Tracing Reverse

$$
\begin{aligned}
& \mathit{reverse}\,[1,2,3,4] \\
= \;& \mathit{rcat}\,[1,2,3,4]\,[\,] \\
= \;& \mathit{rcat}\,[2,3,4]\,[1] \\
= \;& \mathit{rcat}\,[3,4]\,[2,1] \\
= \;& \mathit{rcat}\,[4]\,[3,2,1] \\
= \;& \mathit{rcat}\,[\,]\,[4,3,2,1] \\
= \;& [4,3,2,1]
\end{aligned}
$$

$$
\begin{aligned}
\mathit{reverse}\,\mathit{xs} &= \mathit{rcat}\,\mathit{xs}\,[\,] \\
\mathit{rcat}\,[\,]\,\mathit{ys} &= \mathit{ys} \\
\mathit{rcat}\,(x : \mathit{xs})\,\mathit{ys} &= \mathit{rcat}\,\mathit{xs}\,(x : \mathit{ys})
\end{aligned}
$$

$$
\begin{aligned}
& \mathit{xs}, \mathit{ys} \leftarrow \mathit{XS}, [\,]; \\
& \mathit{while}\ \mathit{xs} \neq [\,]\ \mathit{do} \\
& \qquad \mathit{xs}, \mathit{ys} \leftarrow \mathit{tl}\,\mathit{xs}, \mathit{hd}\,\mathit{xs} : \mathit{ys}; \\
& \mathit{return}\ \mathit{ys};
\end{aligned}
$$

Tail Recursion

- Tail recursion: a special case of recursion in which the last operation is the recursive call.

$$\begin{array}{lcl} f\ x_1\ \ldots\ x_n & = & \{\text{base case}\} \\ f\ x_1\ \ldots\ x_n & = & f\ x'_1\ \ldots\ x'_n \end{array}$$

- To implement general recursion, we need to keep a stack of return addresses. For tail recursion, we do not need such a stack.
- Tail recursive definitions are like loops. Each x_i is updated to x'_i in the next iteration of the loop.
- The first call to f sets up the initial values of each x_i.

Accumulating Parameters

- To efficiently perform a computation (e.g. *reverse xs*), we introduce a generalisation with an extra parameter, e.g.:

$$rcat\ xs\ ys \quad = \quad reverse\ xs \mathbin{+\!\!+} ys.$$

- Try to derive an efficient implementation of the generalised function. The extra parameter is usually used to "accumulate" some results, hence the name.
  - To make the accumulation work, we usually need some kind of associativity.
- A technique useful for, but not limited to, constructing tail-recursive definition of functions.

Loop Invariants

To implement *reverse*, we introduce *rcat* such that:

$$\mathit{rcat}\ \mathit{xs}\ \mathit{ys} \quad = \quad \mathit{reverse}\ \mathit{xs} \mathbin{+\!\!+} \mathit{ys}. \tag{1}$$

Functional:
We initialise *rcat* by:

$$\mathit{reverse}\ \mathit{xs} \quad = \quad \mathit{rcat}\ \mathit{xs}\ [\,],$$

and try to derive a faster version of *rcat* satisfying (1):

$$\begin{aligned} \mathit{rcat}\ [\,]\ \mathit{ys} &\quad = \quad \mathit{ys} \\ \mathit{rcat}\ (x : \mathit{xs})\ \mathit{ys} &\quad = \quad \mathit{rcat}\ \mathit{xs}\ (y : \mathit{ys}). \end{aligned}$$

Procedural:
We initialise the loop, and try to derive a loop body maintaining a *loop invariant* related to (1).

$$\begin{array}{l} \mathit{xs}, \mathit{ys} \leftarrow \mathit{XS}, [\,]; \\ \{\mathit{reverse}\ \mathit{XS} = \mathit{reverse}\ \mathit{xs} \mathbin{+\!\!+} \mathit{ys}\} \\ \mathit{while}\ \mathit{xs} \neq [\,]\ \mathit{do} \\ \quad \mathit{xs}, \mathit{ys} \leftarrow \mathit{tl}\ \mathit{xs}, \mathit{hd}\ \mathit{xs} : \mathit{ys}; \\ \mathit{return}\ \mathit{ys}; \end{array}$$

Accumulating Parameter: Another Example

- Recall the "sum of squares" problem:

$$\begin{array}{lcl} sumsq\,[\,] & = & 0 \\ sumsq\,(x : xs) & = & square\,x + sumsq\,xs. \end{array}$$

  The program still takes linear space (for the stack of return addresses). Let us construct a tail recursive auxiliary function.

- Introduce $ssp\,xs\,n =$.
- Initialisation: $sumsq\,xs =$.
- Construct ssp:

Accumulating Parameter: Another Example

- Recall the "sum of squares" problem:

$$\begin{array}{lcl} \mathit{sumsq}\,[\,] & = & 0 \\ \mathit{sumsq}\,(x : \mathit{xs}) & = & \mathit{square}\,x + \mathit{sumsq}\,\mathit{xs}. \end{array}$$

  The program still takes linear space (for the stack of return addresses). Let us construct a tail recursive auxiliary function.

- Introduce $\mathit{ssp}\,\mathit{xs}\,n = \mathit{sumsq}\,\mathit{xs} + n$.
- Initialisation: $\mathit{sumsq}\,\mathit{xs} =$.
- Construct ssp:

Accumulating Parameter: Another Example

- Recall the "sum of squares" problem:

$$\begin{aligned} \mathit{sumsq}\,[\,] &= 0 \\ \mathit{sumsq}\,(x : xs) &= \mathit{square}\,x + \mathit{sumsq}\,xs. \end{aligned}$$

  The program still takes linear space (for the stack of return addresses). Let us construct a tail recursive auxiliary function.

- Introduce $\mathit{ssp}\,xs\,n = \mathit{sumsq}\,xs + n$.
- Initialisation: $\mathit{sumsq}\,xs = \mathit{ssp}\,xs\,0$.
- Construct ssp:

Accumulating Parameter: Another Example

- Recall the "sum of squares" problem:

$$\begin{array}{lcl} \mathit{sumsq}\,[\,] & = & 0 \\ \mathit{sumsq}\,(x : xs) & = & \mathit{square}\,x + \mathit{sumsq}\,xs. \end{array}$$

  The program still takes linear space (for the stack of return addresses). Let us construct a tail recursive auxiliary function.

- Introduce $\mathit{ssp}\,xs\,n = \mathit{sumsq}\,xs + n$.
- Initialisation: $\mathit{sumsq}\,xs = \mathit{ssp}\,xs\,0$.
- Construct ssp:

$$\begin{array}{lcl} \mathit{ssp}\,[\,]\,n & = & 0 + n \;=\; n \\ \mathit{ssp}\,(x : xs)\,n & = & (\mathit{square}\,x + \mathit{sumsq}\,xs) + n \\ & = & \mathit{sumsq}\,xs + (\mathit{square}\,x + n) \\ & = & \mathit{ssp}\,xs\,(\mathit{square}\,x + n). \end{array}$$

Steep Lists

- A *steep list* is a list in which every element is larger than the sum of those to its right:

$$\begin{aligned} steep\,[\,] &= true \\ steep\,(x : xs) &= steep\,xs \wedge x > sum\,xs. \end{aligned}$$

- The definition above, if executed directly, is an $O(n^2)$ program. Can we do better?
- Just now we learned to construct a generalised function which takes more input. This time, we try the dual technique: to construct a function returning more results.

Generalise by Returning More

- Recall that $\mathit{fst}\,(a, b) = a$ and $\mathit{snd}\,(a, b) = b$.
- It is hard to quickly compute steep alone. But if we define

  $$\mathit{steepsum}\ \mathit{xs} \quad = \quad (\mathit{steep}\ \mathit{xs}, \mathit{sum}\ \mathit{xs}),$$

  and manage to synthesise a quick definition of $\mathit{steepsum}$, we can implement steep by $\mathit{steep} = \mathit{fst} \cdot \mathit{steepsum}$.
- We again proceed by case analysis. Trivially,

  $$\mathit{steepsum}\,[\,] \quad = \quad (\mathit{true}, 0).$$

Deriving for the Non-Empty Case

For the case for non-empty inputs:

$$\mathit{steepsum}\,(x : xs)$$

Deriving for the Non-Empty Case

For the case for non-empty inputs:

$$
\begin{array}{ll}
 & \mathit{steepsum}\,(x : xs) \\
= & \quad \{\ \text{definition of } \mathit{steepsum}\ \} \\
 & (\mathit{steep}\,(x : xs), \mathit{sum}\,(x : xs))
\end{array}
$$

Deriving for the Non-Empty Case

For the case for non-empty inputs:

$$
\begin{aligned}
& \mathit{steepsum}\,(x : xs) \\
= \quad & \{ \text{ definition of } \mathit{steepsum} \text{ } \} \\
& (\mathit{steep}\,(x : xs), \mathit{sum}\,(x : xs)) \\
= \quad & \{ \text{ definitions of } \mathit{steep} \text{ and } \mathit{sum} \text{ } \} \\
& (\mathit{steep}\,xs \wedge x > \mathit{sum}\,xs, x + \mathit{sum}\,xs)
\end{aligned}
$$

Deriving for the Non-Empty Case

For the case for non-empty inputs:

$$
\begin{aligned}
& \mathit{steepsum}\,(x : xs) \\
= \quad & \{\ \text{definition of } \mathit{steepsum}\ \} \\
& (\mathit{steep}\,(x : xs), \mathit{sum}\,(x : xs)) \\
= \quad & \{\ \text{definitions of } \mathit{steep} \text{ and } \mathit{sum}\ \} \\
& (\mathit{steep}\,xs \wedge x > \mathit{sum}\,xs, x + \mathit{sum}\,xs) \\
= \quad & \{\ \text{extracting sub-expressions involving } xs\ \} \\
& \textbf{let}\,(b, y) = (\mathit{steep}\,xs, \mathit{sum}\,xs) \\
& \textbf{in}\,(b \wedge x > y, x + y)
\end{aligned}
$$

Deriving for the Non-Empty Case

For the case for non-empty inputs:

$$
\begin{aligned}
& \mathit{steepsum}\,(x : xs) \\
= \quad & \{ \text{ definition of } \mathit{steepsum} \ \} \\
& (\mathit{steep}\,(x : xs), \mathit{sum}\,(x : xs)) \\
= \quad & \{ \text{ definitions of } \mathit{steep} \text{ and } \mathit{sum} \ \} \\
& (\mathit{steep}\,xs \land x > \mathit{sum}\,xs, x + \mathit{sum}\,xs) \\
= \quad & \{ \text{ extracting sub-expressions involving } xs \ \} \\
& \mathbf{let}\,(b, y) = (\mathit{steep}\,xs, \mathit{sum}\,xs) \\
& \mathbf{in}\,(b \land x > y, x + y) \\
= \quad & \{ \text{ definition of } \mathit{steepsum} \ \} \\
& \mathbf{let}\,(b, y) = \mathit{steepsum}\ xs \\
& \mathbf{in}\,(b \land x > y, x + y).
\end{aligned}
$$

Synthesised Program

- We have thus come up with:

$$\begin{array}{lcl} \mathit{steep} & = & \mathit{fst} \cdot \mathit{steepsum} \\ \mathit{steepsum}\,[\,] & = & (\mathit{true}, 0) \\ \mathit{steepsum}\,(x : \mathit{xs}) & = & \mathbf{let}\,(b, y) = \mathit{steepsum}\ \mathit{xs} \\ & & \mathbf{in}\,(b \wedge x > y, x + y), \end{array}$$

  which runs in $O(n)$ time.
- Again we observe the phenomena that a more general function is easier to implement.
- It is actually common in indutive proofs, too. To prove a theorem, we sometimes have to generalise it so that we have a stronger inductive hypothesis.
- Talking about inductive proofs again, in the next lecture let us see a general pattern for induction.

Summary for the First Day

- Program derivation: constructing programs from their specifications, through formal reasoning.
- Expand/reduce transformation: the most fundamental kind of program derivation — expand the definitions of functions, and reduce them back when necessary.
- Most of the properties we need during the reasoning, for this course, can be proved by induction.
- Accumulating parameters: sometimes a more general program is easier to construct.
  - Sometimes used to construct loops. Closely related to loop invariants in procedural program derivation.
  - Usually relies on some associtivity property to work.
- Tupling: a dual technique often used to generalise a function so that we can derive a quicker recursive definition.
- Like it so far? More fun tomorrow!

Part II

Fold, Unfold, and Hylomorphism

From Yesterday...

- Expand/reduce transformation: the most basic kind of program derivation. Expand the definitions of functions, and reduce them back when necessary.
- Proof by induction.
- Accumulating parameter: a handy technique for, among other purposes, deriving tail recursive functions.
- Tupling: a dual technique often used to generalise a function so that we can derive a quicker recursive definition.
- Today we will be dealing with slightly abstract concepts.

A Common Pattern We've Seen Many Times...

- $$\begin{array}{lcl} \mathit{sum}\,[\,] & = & 0 \\ \mathit{sum}\,(x : xs) & = & x + \mathit{sum}\,xs \end{array}$$
- $$\begin{array}{lcl} \mathit{length}\,[\,] & = & 0 \\ \mathit{length}\,(x : xs) & = & 1 + \mathit{length}\,xs \end{array}$$
- $$\begin{array}{lcl} \mathit{map}\,f\,[\,] & = & [\,] \\ \mathit{map}\,f\,(x : xs) & = & f\,x : \mathit{map}\,f\,xs \end{array}$$
- This pattern is extracted and called *foldr*:

$$\begin{array}{lcl} \mathit{foldr}\,f\,e\,[\,] & = & e, \\ \mathit{foldr}\,f\,e\,(x : xs) & = & f\,x\,(\mathit{foldr}\,f\,e\,xs). \end{array}$$

Replacing Constructors

- $$\begin{array}{lcl} \mathit{foldr}\ f\ e\ [\,] & = & e \\ \mathit{foldr}\ f\ e\ (x : xs) & = & f\ x\ (\mathit{foldr}\ f\ e\ xs) \end{array}$$
- One way to look at $\mathit{foldr}\ (\oplus)\ e$ is that it replaces $[\,]$ with e and $(:)$ with $(\oplus)$:
$$\begin{array}{ll} & \mathit{foldr}\ (\oplus)\ e\ [1,2,3,4] \\ = & \mathit{foldr}\ (\oplus)\ e\ (1 : (2 : (3 : (4 : [\,])))) \\ = & 1 \oplus (2 \oplus (3 \oplus (4 \oplus e))). \end{array}$$
- $\mathit{sum} = \mathit{foldr}\ (+)\ 0$.
- $\mathit{length} = \mathit{foldr}\ (\lambda x\ n.1 + n)\ 0$.
- $\mathit{map}\ f = \mathit{foldr}\ (\lambda x\ xs.f\ x : xs)\ [\,]$.
- One can see that $\mathit{id} = \mathit{foldr}\ (:)\ [\,]$.

Some Trivial Folds on Lists

- Function *max* returns the maximum element in a list:
  - $$\begin{array}{lcl} max\,[\,] & = & -\infty, \\ max\,(x : xs) & = & x \uparrow max\,xs. \end{array}$$
- Function *prod* returns the product of a list:
  - $$\begin{array}{lcl} prod\,[\,] & = & 1, \\ prod\,(x : xs) & = & x \times prod\,xs. \end{array}$$
- Function *and* returns the conjunction of a list:
  - $$\begin{array}{lcl} and\,[\,] & = & true, \\ and\,(x : xs) & = & x \wedge and\,xs. \end{array}$$
- Lets emphasise again that *id* on lists is a fold:
  - $$\begin{array}{lcl} id\,[\,] & = & [\,], \\ id\,(x : xs) & = & x : id\,xs. \end{array}$$

Some Trivial Folds on Lists

- Function *max* returns the maximum element in a list:
  - $$\begin{aligned} max\,[\,] &= -\infty, \\ max\,(x : xs) &= x \uparrow max\,xs. \end{aligned}$$
  - $max = foldr\,(\uparrow)\,{-\infty}$.
- Function *prod* returns the product of a list:
  - $$\begin{aligned} prod\,[\,] &= 1, \\ prod\,(x : xs) &= x \times prod\,xs. \end{aligned}$$
- Function *and* returns the conjunction of a list:
  - $$\begin{aligned} and\,[\,] &= true, \\ and\,(x : xs) &= x \wedge and\,xs. \end{aligned}$$
- Lets emphasise again that *id* on lists is a fold:
  - $$\begin{aligned} id\,[\,] &= [\,], \\ id\,(x : xs) &= x : id\,xs. \end{aligned}$$

Some Trivial Folds on Lists

- Function *max* returns the maximum element in a list:
  - $$\begin{aligned} \mathit{max}\,[\,] &= -\infty, \\ \mathit{max}\,(x : xs) &= x \uparrow \mathit{max}\,xs. \end{aligned}$$
  - $\mathit{max} = \mathit{foldr}\,(\uparrow)\,{-\infty}$.
- Function *prod* returns the product of a list:
  - $$\begin{aligned} \mathit{prod}\,[\,] &= 1, \\ \mathit{prod}\,(x : xs) &= x \times \mathit{prod}\,xs. \end{aligned}$$
  - $\mathit{prod} = \mathit{foldr}\,(\times)\,1$.
- Function *and* returns the conjunction of a list:
  - $$\begin{aligned} \mathit{and}\,[\,] &= \mathit{true}, \\ \mathit{and}\,(x : xs) &= x \wedge \mathit{and}\,xs. \end{aligned}$$
- Lets emphasise again that *id* on lists is a fold:
  - $$\begin{aligned} \mathit{id}\,[\,] &= [\,], \\ \mathit{id}\,(x : xs) &= x : \mathit{id}\,xs. \end{aligned}$$

Some Trivial Folds on Lists

- Function *max* returns the maximum element in a list:
  - $$\begin{aligned} max\,[\,] &= -\infty, \\ max\,(x : xs) &= x \uparrow max\,xs. \end{aligned}$$
  - $max = foldr\,(\uparrow)\,{-\infty}$.
- Function *prod* returns the product of a list:
  - $$\begin{aligned} prod\,[\,] &= 1, \\ prod\,(x : xs) &= x \times prod\,xs. \end{aligned}$$
  - $prod = foldr\,(\times)\,1$.
- Function *and* returns the conjunction of a list:
  - $$\begin{aligned} and\,[\,] &= true, \\ and\,(x : xs) &= x \wedge and\,xs. \end{aligned}$$
  - $and = foldr\,(\wedge)\,true$.
- Lets emphasise again that *id* on lists is a fold:
  - $$\begin{aligned} id\,[\,] &= [\,], \\ id\,(x : xs) &= x : id\,xs. \end{aligned}$$

Some Trivial Folds on Lists

- Function *max* returns the maximum element in a list:
  - $$\begin{aligned} max\,[\,] &= -\infty, \\ max\,(x : xs) &= x \uparrow max\, xs. \end{aligned}$$
  - $max = foldr\,(\uparrow)\,\text{-}\infty$.
- Function *prod* returns the product of a list:
  - $$\begin{aligned} prod\,[\,] &= 1, \\ prod\,(x : xs) &= x \times prod\, xs. \end{aligned}$$
  - $prod = foldr\,(\times)\,1$.
- Function *and* returns the conjunction of a list:
  - $$\begin{aligned} and\,[\,] &= true, \\ and\,(x : xs) &= x \wedge and\, xs. \end{aligned}$$
  - $and = foldr\,(\wedge)\,true$.
- Lets emphasise again that *id* on lists is a fold:
  - $$\begin{aligned} id\,[\,] &= [\,], \\ id\,(x : xs) &= x : id\, xs. \end{aligned}$$
  - $id = foldr\,(:)\,[\,]$.

Why Folds?

- The same reason we kept talking about *patterns* in design.
- Control abstraction, procedure abstraction, data abstraction, . . . can programming patterns be abstracted too?
- Program structure becomes an entity we can talk about, reason about, and reuse.
  - We can describe algorithms in terms of fold, unfold, and other recognised patterns.
  - We can prove properties about folds,
  - and apply the proved theorems to all programs that are folds, either for compiler optimisation, or for mathematical reasoning.
- Among the theorems about folds, the most important is probably the *fold-fusion* theorem.

The Fold-Fusion Theorem

The theorem is about when the composition of a function and a fold can be expressed as a fold.

Theorem (Fold-Fusion)

Given $f :: \alpha \rightarrow \beta \rightarrow \beta$, $e :: \beta$, $h :: \beta \rightarrow \gamma$, and $g :: \alpha \rightarrow \gamma \rightarrow \gamma$, we have:

$$h \cdot \mathit{foldr}\ f\ e \quad = \quad \mathit{foldr}\ g\ (h\ e),$$

if $h\ (f\ x\ y) = g\ x\ (h\ y)$ for all x and y.

For program derivation, we are usually given h, f, and e, from which we have to construct g.

Tracing an Example

Let us try to get an intuitive understand of the theorem:

$$
\begin{array}{ll}
 & h\,(\mathit{foldr}\; f\; e\; [a, b, c]) \\
= & \quad \{\ \text{definition of } \mathit{foldr}\ \} \\
 & h\,(f\; a\,(f\; b\,(f\; c\; e)))
\end{array}
$$

Tracing an Example

Let us try to get an intuitive understand of the theorem:

$$
\begin{array}{ll}
 & h\,(\mathit{foldr}\ f\ e\ [a, b, c]) \\
= & \quad \{\ \text{definition of } \mathit{foldr}\ \} \\
 & h\,(f\ a\,(f\ b\,(f\ c\ e))) \\
= & \quad \{\ \text{since } h\,(f\ x\ y) = g\ x\,(h\ y)\ \} \\
 & g\ a\,(h\,(f\ b\,(f\ c\ e)))
\end{array}
$$

Tracing an Example

Let us try to get an intuitive understand of the theorem:

$$
\begin{aligned}
& h\,(foldr\; f\; e\,[a, b, c]) \\
= & \quad \{ \text{ definition of } foldr \text{ } \} \\
& h\,(f\; a\,(f\; b\,(f\; c\; e))) \\
= & \quad \{ \text{ since } h\,(f\; x\; y) = g\; x\,(h\; y) \text{ } \} \\
& g\; a\,(h\,(f\; b\,(f\; c\; e))) \\
= & \quad \{ \text{ since } h\,(f\; x\; y) = g\; x\,(h\; y) \text{ } \} \\
& g\; a\,(g\; b\,(h\,(f\; c\; e)))
\end{aligned}
$$

Tracing an Example

Let us try to get an intuitive understand of the theorem:

$$
\begin{array}{ll}
 & h\,(foldr\ f\ e\,[a,b,c]) \\
= & \quad \{\ \text{definition of } foldr\ \} \\
 & h\,(f\ a\,(f\ b\,(f\ c\ e))) \\
= & \quad \{\ \text{since } h\,(f\ x\ y) = g\ x\,(h\ y)\ \} \\
 & g\ a\,(h\,(f\ b\,(f\ c\ e))) \\
= & \quad \{\ \text{since } h\,(f\ x\ y) = g\ x\,(h\ y)\ \} \\
 & g\ a\,(g\ b\,(h\,(f\ c\ e))) \\
= & \quad \{\ \text{since } h\,(f\ x\ y) = g\ x\,(h\ y)\ \} \\
 & g\ a\,(g\ b\,(g\ c\,(h\ e)))
\end{array}
$$

Tracing an Example

Let us try to get an intuitive understand of the theorem:

$$
\begin{aligned}
& h\,(foldr\;f\;e\;[a,b,c]) \\
= \quad & \quad \{\text{ definition of } foldr \;\} \\
& h\,(f\;a\,(f\;b\,(f\;c\;e))) \\
= \quad & \quad \{\text{ since } h\,(f\;x\;y) = g\;x\,(h\;y)\;\} \\
& g\;a\,(h\,(f\;b\,(f\;c\;e))) \\
= \quad & \quad \{\text{ since } h\,(f\;x\;y) = g\;x\,(h\;y)\;\} \\
& g\;a\,(g\;b\,(h\,(f\;c\;e))) \\
= \quad & \quad \{\text{ since } h\,(f\;x\;y) = g\;x\,(h\;y)\;\} \\
& g\;a\,(g\;b\,(g\;c\,(h\;e))) \\
= \quad & \quad \{\text{ definition of } foldr \;\} \\
& foldr\;g\,(h\;e)\,[a,b,c].
\end{aligned}
$$

Sum of Squares, Again

- Consider $sum \cdot map\, square$ again. This time we use the fact that $map\, f = foldr\, (mf\ f)\, [\,]$, where $mf\ f\, x\, xs = f\, x : xs$.
- $sum \cdot map\, square$ is a fold, if we can find a ssq such that $sum\, (mf\ square\, x\, xs) = ssq\, x\, (sum\, xs)$. Let us try:

$$
\begin{aligned}
& sum\, (mf\ square\, x\, xs) \\
= & \quad \{\ \text{definition of } mf\ \} \\
& sum\, (square\, x : xs) \\
= & \quad \{\ \text{definition of } sum\ \} \\
& square\, x + sum\, xs \\
= & \quad \{\ \text{let } ssq\, x\, y = square\, x + y\ \} \\
& ssq\, x\, (sum\, xs).
\end{aligned}
$$

Therefore, $sum \cdot map\, square = foldr\, ssq\, 0$.

More on Folds and Fold-fusion

- Compare the proof with the one yesterday. They are essentially the same proof.
- Fold-fusion theorem abstracts away the common parts in this kind of inductive proofs, so that we need to supply only the "important" parts.
- Tupling can be seen as a kind of fold-fusion. The derivation of *steepsum*, for example, can be seen as fusing:

$$\mathit{steepsum} \cdot \mathit{id} \quad = \quad \mathit{steepsum} \cdot \mathit{foldr}\ (:)\ [\,].$$

- Not every function can be expressed as a fold. For example, *tl* is not a fold!

Longest Prefix

- The function call $\mathit{takeWhile}\ p\ \mathit{xs}$ returns the longest prefix of xs that satisfies p:

$$\begin{aligned} \mathit{takeWhile}\ p\ [\,] &= [\,], \\ \mathit{takeWhile}\ p\ (x : \mathit{xs}) &= \textbf{if}\ p\ x\ \textbf{then}\ x : \mathit{takeWhile}\ p\ \mathit{xs} \\ &\quad\ \textbf{else}\ [\,]. \end{aligned}$$

- E.g. $\mathit{takeWhile}\ (\leq 3)\ [1,2,3,4,5] = [1,2,3]$.
- It can be defined by a fold:

$$\begin{aligned} \mathit{takeWhile}\ p &= \mathit{foldr}\ (\mathit{tke}\ p)\ [\,], \\ \mathit{tke}\ p\ x\ \mathit{xs} &= \textbf{if}\ p\ x\ \textbf{then}\ x : \mathit{xs}\ \textbf{else}\ [\,]. \end{aligned}$$

- Its dual, $\mathit{dropWhile}\ (\leq 3)\ [1,2,3,4,5] = [4,5]$, is not a fold.

All Prefixes

- The function *inits* returns the list of all prefixes of the input list:

$$\begin{aligned} \mathit{inits}\,[\,] &= [[\,]], \\ \mathit{inits}\,(x : xs) &= [\,] : \mathit{map}\,(x :)\,(\mathit{inits}\,xs). \end{aligned}$$

- E.g. $\mathit{inits}\,[1,2,3] = [[\,],[1],[1,2],[1,2,3]]$.
- It can be defined by a fold:

$$\begin{aligned} \mathit{inits} &= \mathit{foldr}\;\mathit{ini}\,[[\,]], \\ \mathit{ini}\;x\;xss &= [\,] : \mathit{map}\,(x :)\;xss. \end{aligned}$$

All Suffixes

- The function *tails* returns the list of all suffixes of the input list:

$$\begin{array}{lcl} \mathit{tails}\,[\,] & = & [\,], \\ \mathit{tails}\,(x : xs) & = & \mathbf{let}\,(ys : yss) = \mathit{tails}\,xs \\ & & \mathbf{in}\,(x : ys) : ys : yss. \end{array}$$

- E.g. $\mathit{tails}\,[1,2,3] = [[1,2,3],[2,3],[3],[\,]]$.
- It can be defined by a fold:

$$\begin{array}{lcl} \mathit{tails} & = & \mathit{foldr}\;\mathit{til}\;[[\,]], \\ \mathit{til}\;x\,(ys : yss) & = & (x : ys) : ys : yss. \end{array}$$

Scan

- $scanr\ f\ e = map\ (foldr\ f\ e) \cdot tails$.
- E.g.

$$
\begin{aligned}
& scanr\ (+)\ 0\ [1,2,3] \\
= \quad & map\ sum\ (tails\ [1,2,3]) \\
= \quad & map\ sum\ [[1,2,3],[2,3],[3],[\,]] \\
= \quad & [6,5,3,0].
\end{aligned}
$$

- Of course, it is slow to actually perform $map\ (foldr\ f\ e)$ separately. By fold-fusion, we get a faster implementation:

$$
\begin{aligned}
scanr\ f\ e \quad &= \quad foldr\ (sc\ f)\ [e], \\
sc\ f\ x\ (y : ys) \quad &= \quad f\ x\ y : y : ys.
\end{aligned}
$$

Specifying Maximum Segment Sum

- Finally we have introduced enough concepts to tackle the maximum segment sum problem!
- A segment can be seen as a prefix of a suffix.
- The function *segs* computes the list of all the segments.

$$segs \quad = \quad concat \cdot map\, inits \cdot tails\,.$$

- Therefore, *mss* is specified by:

$$mss \quad = \quad max \cdot map\, sum \cdot segs\,.$$

The Derivation!

We reason:

$$max \cdot map\ sum \cdot concat \cdot map\ inits \cdot tails$$

The Derivation!

We reason:

$$
\begin{aligned}
& \mathit{max} \cdot \mathit{map}\,\mathit{sum} \cdot \mathit{concat} \cdot \mathit{map}\,\mathit{inits} \cdot \mathit{tails} \\
= & \quad \{\ \text{since } \mathit{map}\,f \cdot \mathit{concat} = \mathit{concat} \cdot \mathit{map}\,(\mathit{map}\,f)\ \} \\
& \mathit{max} \cdot \mathit{concat} \cdot \mathit{map}\,(\mathit{map}\,\mathit{sum}) \cdot \mathit{map}\,\mathit{inits} \cdot \mathit{tails}
\end{aligned}
$$

The Derivation!

We reason:

$$
\begin{aligned}
& \mathit{max} \cdot \mathit{map}\,\mathit{sum} \cdot \mathit{concat} \cdot \mathit{map}\,\mathit{inits} \cdot \mathit{tails} \\
= \quad & \{ \text{ since } \mathit{map}\,f \cdot \mathit{concat} = \mathit{concat} \cdot \mathit{map}\,(\mathit{map}\,f) \ \} \\
& \mathit{max} \cdot \mathit{concat} \cdot \mathit{map}\,(\mathit{map}\,\mathit{sum}) \cdot \mathit{map}\,\mathit{inits} \cdot \mathit{tails} \\
= \quad & \{ \text{ since } \mathit{max} \cdot \mathit{concat} = \mathit{max} \cdot \mathit{map}\,\mathit{max} \ \} \\
& \mathit{max} \cdot \mathit{map}\,\mathit{max} \cdot \mathit{map}\,(\mathit{map}\,\mathit{sum}) \cdot \mathit{map}\,\mathit{inits} \cdot \mathit{tails}
\end{aligned}
$$

The Derivation!

We reason:

$$
\begin{aligned}
& \mathit{max} \cdot \mathit{map\,sum} \cdot \mathit{concat} \cdot \mathit{map\,inits} \cdot \mathit{tails} \\
= \quad & \{ \text{ since } \mathit{map\,f} \cdot \mathit{concat} = \mathit{concat} \cdot \mathit{map}\,(\mathit{map\,f}) \ \} \\
& \mathit{max} \cdot \mathit{concat} \cdot \mathit{map}\,(\mathit{map\,sum}) \cdot \mathit{map\,inits} \cdot \mathit{tails} \\
= \quad & \{ \text{ since } \mathit{max} \cdot \mathit{concat} = \mathit{max} \cdot \mathit{map\,max} \ \} \\
& \mathit{max} \cdot \mathit{map\,max} \cdot \mathit{map}\,(\mathit{map\,sum}) \cdot \mathit{map\,inits} \cdot \mathit{tails} \\
= \quad & \{ \text{ since } \mathit{map\,f} \cdot \mathit{map\,g} = \mathit{map}\,(f \cdot g) \ \} \\
& \mathit{max} \cdot \mathit{map}\,(\mathit{max} \cdot \mathit{map\,sum} \cdot \mathit{inits}) \cdot \mathit{tails}.
\end{aligned}
$$

Recall the definition $\mathit{scanr}\ f\ e = \mathit{map}\,(\mathit{foldr}\ f\ e) \cdot \mathit{tails}$. If we can transform $\mathit{max} \cdot \mathit{map\,sum} \cdot \mathit{inits}$ into a fold, we can turn the algorithm into a scan, which has a faster implementation.

Maximum Prefix Sum

Concentrate on $\mathit{max} \cdot \mathit{map}\,\mathit{sum} \cdot \mathit{inits}$:

$$
\begin{array}{ll}
& \mathit{max} \cdot \mathit{map}\,\mathit{sum} \cdot \mathit{inits} \\
= & \quad \{ \text{ definition of } \mathit{init}, \ \mathit{ini}\,x\,\mathit{xss} = [\,] : \mathit{map}\,(x:\,)\,\mathit{xss} \ \} \\
& \mathit{max} \cdot \mathit{map}\,\mathit{sum} \cdot \mathit{foldr}\,\mathit{ini}\,[[\,]]
\end{array}
$$

Maximum Prefix Sum

Concentrate on $max \cdot map\, sum \cdot inits$:

$$
\begin{aligned}
& max \cdot map\, sum \cdot inits \\
= \quad & \{ \text{ definition of } init,\ ini\ x\ xss = [\,] : map\,(x:\,)\,xss \ \} \\
& max \cdot map\, sum \cdot foldr\ ini\ [[\,]] \\
= \quad & \{ \text{ fold fusion, see below } \} \\
& max \cdot foldr\ zplus\ [0].
\end{aligned}
$$

The fold fusion works because:

$$
\begin{aligned}
& map\, sum\,(ini\ x\ xss) \\
= \quad & map\, sum\,([\,] : map\,(x:\,)\,xss) \\
= \quad & 0 : map\,(sum \cdot (x:\,))\,xss \\
= \quad & 0 : map\,(x+)\,(map\, sum\, xss).
\end{aligned}
$$

Define $zplus\ x\ xss = 0 : map\,(x+)\,xss$.

Maximum Prefix Sum, 2nd Fold Fusion

Concentrate on $\mathit{max} \cdot \mathit{map}\,\mathit{sum} \cdot \mathit{inits}$:

$$
\begin{aligned}
& \mathit{max} \cdot \mathit{map}\,\mathit{sum} \cdot \mathit{inits} \\
= \quad & \{ \text{ definition of } \mathit{init},\ \mathit{ini}\,x\,\mathit{xss} = [\,] : \mathit{map}\,(x:\,)\,\mathit{xss} \ \} \\
& \mathit{max} \cdot \mathit{map}\,\mathit{sum} \cdot \mathit{foldr}\,\mathit{ini}\,[[\,]] \\
= \quad & \{ \text{ fold fusion, } \mathit{zplus}\,x\,\mathit{xss} = 0 : \mathit{map}\,(x+)\,\mathit{xss} \ \} \\
& \mathit{max} \cdot \mathit{foldr}\,\mathit{zplus}\,[0] \\
= \quad & \{ \text{ fold fusion, let } \mathit{zmax}\,x\,y = 0 \uparrow (x + y) \ \} \\
& \mathit{foldr}\,\mathit{zmax}\,0.
\end{aligned}
$$

The fold fusion works because $\uparrow$ distributes into $(+)$:

$$
\begin{aligned}
& \mathit{max}\,(0 : \mathit{map}\,(x+)\,\mathit{xs}) \\
= \quad & 0 \uparrow \mathit{max}\,(\mathit{map}\,(x+)\,\mathit{xs}) \\
= \quad & 0 \uparrow (x + \mathit{max}\,\mathit{xs}).
\end{aligned}
$$

Back to Maximum Segment Sum

We reason:

$$
\begin{aligned}
& \mathit{max} \cdot \mathit{map}\,\mathit{sum} \cdot \mathit{concat} \cdot \mathit{map}\,\mathit{inits} \cdot \mathit{tails} \\
= & \quad \{ \text{ since } \mathit{map}\,f \cdot \mathit{concat} = \mathit{concat} \cdot \mathit{map}\,(\mathit{map}\,f) \ \} \\
& \mathit{max} \cdot \mathit{concat} \cdot \mathit{map}\,(\mathit{map}\,\mathit{sum}) \cdot \mathit{map}\,\mathit{inits} \cdot \mathit{tails} \\
= & \quad \{ \text{ since } \mathit{max} \cdot \mathit{concat} = \mathit{max} \cdot \mathit{map}\,\mathit{max} \ \} \\
& \mathit{max} \cdot \mathit{map}\,\mathit{max} \cdot \mathit{map}\,(\mathit{map}\,\mathit{sum}) \cdot \mathit{map}\,\mathit{inits} \cdot \mathit{tails} \\
= & \quad \{ \text{ since } \mathit{map}\,f \cdot \mathit{map}\,g = \mathit{map}\,(f \cdot g) \ \} \\
& \mathit{max} \cdot \mathit{map}\,(\mathit{max} \cdot \mathit{map}\,\mathit{sum} \cdot \mathit{inits}) \cdot \mathit{tails}
\end{aligned}
$$

Back to Maximum Segment Sum

We reason:

$$
\begin{aligned}
& \mathit{max} \cdot \mathit{map}\,\mathit{sum} \cdot \mathit{concat} \cdot \mathit{map}\,\mathit{inits} \cdot \mathit{tails} \\
= \quad & \{ \text{ since } \mathit{map}\,f \cdot \mathit{concat} = \mathit{concat} \cdot \mathit{map}\,(\mathit{map}\,f) \text{ } \} \\
& \mathit{max} \cdot \mathit{concat} \cdot \mathit{map}\,(\mathit{map}\,\mathit{sum}) \cdot \mathit{map}\,\mathit{inits} \cdot \mathit{tails} \\
= \quad & \{ \text{ since } \mathit{max} \cdot \mathit{concat} = \mathit{max} \cdot \mathit{map}\,\mathit{max} \text{ } \} \\
& \mathit{max} \cdot \mathit{map}\,\mathit{max} \cdot \mathit{map}\,(\mathit{map}\,\mathit{sum}) \cdot \mathit{map}\,\mathit{inits} \cdot \mathit{tails} \\
= \quad & \{ \text{ since } \mathit{map}\,f \cdot \mathit{map}\,g = \mathit{map}\,(f \cdot g) \text{ } \} \\
& \mathit{max} \cdot \mathit{map}\,(\mathit{max} \cdot \mathit{map}\,\mathit{sum} \cdot \mathit{inits}) \cdot \mathit{tails} \\
= \quad & \{ \text{ reasoning in the previous slides } \} \\
& \mathit{max} \cdot \mathit{map}\,(\mathit{foldr}\,\mathit{zmax}\,0) \cdot \mathit{tails}
\end{aligned}
$$

Back to Maximum Segment Sum

We reason:

$$
\begin{aligned}
& \mathit{max} \cdot \mathit{map}\,\mathit{sum} \cdot \mathit{concat} \cdot \mathit{map}\,\mathit{inits} \cdot \mathit{tails} \\
= & \quad \{ \text{ since } \mathit{map}\,f \cdot \mathit{concat} = \mathit{concat} \cdot \mathit{map}\,(\mathit{map}\,f) \ \} \\
& \mathit{max} \cdot \mathit{concat} \cdot \mathit{map}\,(\mathit{map}\,\mathit{sum}) \cdot \mathit{map}\,\mathit{inits} \cdot \mathit{tails} \\
= & \quad \{ \text{ since } \mathit{max} \cdot \mathit{concat} = \mathit{max} \cdot \mathit{map}\,\mathit{max} \ \} \\
& \mathit{max} \cdot \mathit{map}\,\mathit{max} \cdot \mathit{map}\,(\mathit{map}\,\mathit{sum}) \cdot \mathit{map}\,\mathit{inits} \cdot \mathit{tails} \\
= & \quad \{ \text{ since } \mathit{map}\,f \cdot \mathit{map}\,g = \mathit{map}\,(f \cdot g) \ \} \\
& \mathit{max} \cdot \mathit{map}\,(\mathit{max} \cdot \mathit{map}\,\mathit{sum} \cdot \mathit{inits}) \cdot \mathit{tails} \\
= & \quad \{ \text{ reasoning in the previous slides } \} \\
& \mathit{max} \cdot \mathit{map}\,(\mathit{foldr}\,\mathit{zmax}\,0) \cdot \mathit{tails} \\
= & \quad \{ \text{ introducing } \mathit{scanr} \ \} \\
& \mathit{max} \cdot \mathit{scanr}\,\mathit{zmax}\,0.
\end{aligned}
$$

Maximum Segment Sum in Linear Time!

- We have derived $mss = max \cdot scanr\ zmax\ 0$, where $zmax\ x\ y = 0 \uparrow (x + y)$.
- The algorithm runs in linear time, but takes linear space.
- A tupling transformation eliminates the need for linear space.

  $$mss \quad = \quad fst \cdot maxhd \cdot scanr\ zmax\ 0$$

  where $maxhd\ xs = (max\ xs, hd\ xs)$. We omit this last step in the lecture.
- The final program is $mss = fst \cdot foldr\ step\ (0, 0)$, where $step\ x\ (m, y) = ((0 \uparrow (x + y)) \uparrow m, 0 \uparrow (x + y))$.

Folds on Trees

- Folds are not limited to lists. In fact, every datatype with so-called "regular base functors" induces a fold.
- Recall some datatypes for trees:

$$\begin{array}{lcl} \mathit{data}\ \mathit{iTree}\ \alpha & = & \mathit{Null} \mid \mathit{Node}\ a\ (\mathit{iTree}\ \alpha)\ (\mathit{iTree}\ \alpha); \\ \mathit{data}\ \mathit{eTree}\ \alpha & = & \mathit{Tip}\ a \mid \mathit{Bin}\ (\mathit{eTree}\ \alpha)\ (\mathit{eTree}\ \alpha). \end{array}$$

- The fold for iTree, for example, is defined by:

$$\begin{array}{lcl} \mathit{foldiT}\ f\ e\ \mathit{Null} & = & e, \\ \mathit{foldiT}\ f\ e\ (\mathit{Node}\ a\ t\ u) & = & f\ a\ (\mathit{foldiT}\ f\ e\ t)\ (\mathit{foldiT}\ f\ e\ u). \end{array}$$

- The fold for eTree, is given by:

$$\begin{array}{lcl} \mathit{foldeT}\ f\ g\ (\mathit{Tip}\ x) & = & g\ x, \\ \mathit{foldeT}\ f\ g\ (\mathit{Bin}\ t\ u) & = & f\ (\mathit{foldeT}\ f\ g\ t)\ (\mathit{foldeT}\ f\ g\ u). \end{array}$$

Some Simple Functions on Trees

- to compute the size of an *iTree*:

$$sizeiTree \quad = \quad foldiT\,(\lambda x\,m\,n.m + n + 1)\,0.$$

- To sum up labels in an *eTree*:

$$sumeTree \quad = \quad foldeT\,(+)\,id.$$

- To compute a list of all labels in an *iTree* and an *eTree*:

$$\begin{aligned} flatteniT \quad &= \quad foldiT\,(\lambda x\,xs\,ys.xs \mathbin{+\!\!+} [x] \mathbin{+\!\!+} ys)\,[\,], \\ flatteneT \quad &= \quad foldeT\,(\mathbin{+\!\!+})\,(\lambda x.[x]). \end{aligned}$$

Unfolds Generate Data Structures

- While folds consumes a data structure, *unfolds* builds data structures.
- Unfold on lists is defined by:

$$\begin{aligned} \mathit{unfoldr}\ p\ f\ s &= \mathbf{if}\ p\ s\ \mathbf{then}\ [\,]\ \mathbf{else} \\ &\quad\ \mathbf{let}\ (x, s') = f\ s\ \mathbf{in}\ x : \mathit{unfoldr}\ p\ f\ s'. \end{aligned}$$

  The value s is a "seed" to generate a list with. Function p checks the seed to determines whether to stop. If not, function f is used to generate an element and the next seed.

Some Useful Functions Defined as Unfolds

- For brevity let us introduce the "split" notation. Given functions $f :: \alpha \to \beta$ and $g :: \alpha \to \gamma$, $\langle f, g \rangle :: \alpha \to (\beta, \gamma)$ is a function defined by:

  $$\langle f, g \rangle\, a = (f\, a, g\, a).$$

- The function call *fromto m n* builds a list $[n, n + 1, \ldots, m]$:

  $$\mathit{fromto}\ m \quad = \quad \mathit{unfoldr}\ (\geq m)\ \langle \mathit{id}, (1+) \rangle.$$

- The function tails^+ is like *tails*, but returns non-empty tails only:

  $$\mathit{tails}^+ \quad = \quad \mathit{unfoldr}\ \mathit{null}\ \langle \mathit{id}, \mathit{tl} \rangle,$$

  where *null xs* yields *true* iff $xs = [\,]$.

Unfolds May Build Infinite Data Structures

- The function call $\mathit{from}\ n$ builds the infinitely long list $[n, n+1, \ldots]$:

$$\mathit{from} \quad = \quad \mathit{unfoldr}\ (\mathit{const}\ \mathit{false})\ \langle \mathit{id}, (1+) \rangle .$$

- More generally, $\mathit{iterate}\ f\ x$ builds an infinitely long list $[x, f\ x, f\ (f\ x) \ldots]$:

$$\mathit{iterate}\ f \quad = \quad \mathit{unfoldr}\ (\mathit{const}\ \mathit{false})\ \langle \mathit{id}, f \rangle .$$

  We have $\mathit{from} = \mathit{iterate}\ (1+)$.

Merging as an Unfold

- Given two *sorted* lists (xs, ys), the call $merge\,(xs, ys)$ merges them into one sorted list:

$$
\begin{array}{lcl}
merge & = & unfoldr\ null2\ mrg \\
null2\,(xs, ys) & = & null\ xs \wedge null\ ys \\
mrg\,([\,], y : ys) & = & (y, ([\,], ys)) \\
mrg\,(x : xs, [\,]) & = & (x, (xs, [\,])) \\
mrg\,(x : xs, y : ys) & = & \mathbf{if}\ x \leq y\ \mathbf{then}\ (x, (xs, y : ys)) \\
 & & \quad \mathbf{else}\ (y, (x : xs, ys)).
\end{array}
$$

Folds and Unfolds

- Folds and unfolds are dual concepts. Folds consume data structure, while unfolds build data structures.
- List constructors have types: $(:) :: \alpha \rightarrow [\alpha] \rightarrow [\alpha]$ and $[\,] :: [\alpha]$; in $\mathit{fold}\ f\ e$, the arguments have types: $f :: \alpha \rightarrow \beta \rightarrow \beta$ and $e :: \beta$.
- List deconstructors have types: $\langle hd, tl \rangle :: [\alpha] \rightarrow (\alpha, [\alpha])$; in $\mathit{unfoldr}\ p\ f$, the argument f has type $\beta \rightarrow (\alpha, \beta)$.
- They do not look exactly symmetrical yet. But that is just because our notations are not general enough.

Folds v.s. Unfolds

- Folds are defined on inductive datatypes. All inductive datatypes are finite, and emit inductive proofs. Folds basically captures induction on the input.
- As we have seen, unfolds may generate infinite data structures.
  - They are related to *coinductive* datatypes.
  - Proof by induction does not (trivially) work for coinductive data in general. We need to instead use *coinductive proof*.

A Sketch of A Coinductive Proof

To prove that $map\ f \cdot iterate\ f = iterate\ f\ (f\ x)$, we show that for all possible *observations*, the *lhs* equals the *rhs*.

- $hd \cdot map\ f \cdot iterate\ f = hd \cdot iterate\ f\ (f\ x)$. Trivial.
- $tl \cdot map\ f \cdot iterate\ f = tl \cdot iterate\ f\ (f\ x)$:

$$
\begin{aligned}
& tl\ (map\ f\ (iterate\ f\ x)) \\
= \;& tl\ (f\ x : map\ f\ (iterate\ f\ (f\ x))) \\
= \;& \quad \{hypothesis\} \\
& tl\ (f\ x : iterate\ f\ (f\ (f\ x))) \\
= \;& tl\ (iterate\ f\ (f\ x)).
\end{aligned}
$$

The hypothesis looks a bit shaky: isn't it circular reasoning? We need to describe it in a more rigourous setting to establish its validity. This is out of the scope of this lecture.

Unfolds on Trees

Unfolds can also be extended to trees. For internally labelled binary trees we define:

$$\begin{array}{lcl} \mathit{unfoldiT}\ p\ f\ s & = & \textbf{if}\ p\ s\ \textbf{then}\ \mathit{Null}\ \textbf{else} \\ & & \quad \textbf{let}\ (x, s_1, s_2) = f\ s \\ & & \quad \textbf{in}\ \mathit{Node}\ x\ (\mathit{unfoldiT}\ p\ f\ s_1) \\ & & \qquad\qquad\qquad (\mathit{unfoldiT}\ p\ f\ s_2). \end{array}$$

And for externally labelled binary trees we define:

$$\begin{array}{lcl} \mathit{unfoldeT}\ p\ f\ g\ s & = & \textbf{if}\ p\ s\ \textbf{then}\ \mathit{Tip}\ (g\ s)\ \textbf{else} \\ & & \quad \textbf{let}\ (s_1, s_2) = f\ s \\ & & \quad \textbf{in}\ \mathit{Bin}\ (\mathit{unfoldeT}\ p\ f\ g\ s_1) \\ & & \qquad\qquad (\mathit{unfoldeT}\ p\ f\ g\ s_2). \end{array}$$

Unflattening a Tree

- Recall the function $\mathit{flatteneT} :: \mathit{eTree}\,\alpha \rightarrow [\alpha]$, defined as a fold, flattening a tree into a list. Let us consider doing the reverse.
- Assume that we have the following functions:
  - $\mathit{single}\,\mathit{xs} = \mathit{true}$ iff xs contains only one element.
  - $\mathit{half} :: [\alpha] \rightarrow ([\alpha],[\alpha])$ split a list of length n into two lists of lengths roughly half of n.
- The function $\mathit{unflatteneT}$ builds a tree out of a list:

$$\begin{array}{lcl} \mathit{unflattenT} & :: & [\alpha] \rightarrow \mathit{eTree}\,[\alpha], \\ \mathit{unflattenT} & = & \mathit{unfoldeT}\;\mathit{single}\;\mathit{half}\;\mathit{id}\,. \end{array}$$

Mergesort as a Hylomorphism

- Recall the function *merge* merging a pair of sorted lists into one sorted list. Assume that it has a *curried* variant $merge_c$.
- What does this function do?

$$msort \quad = \quad foldeT\ merge_c\ id \cdot unflatteneT$$

- This is mergesort!

Quicksort as a Hylomorphism

- Let *partition* be defined by:

$$\mathit{partition}\,(x : \mathit{xs}) \quad = \quad (x, \mathit{filter}\,(\leq x)\,\mathit{xs}, \mathit{filter}\,(> x)\,\mathit{xs}).$$

- Recall the function *flatteniT* flattening an *iTree*, defined by a fold.
- Quicksort can be defined by:

$$\mathit{qsort} \quad = \quad \mathit{flatteniT} \cdot \mathit{unfoldiT}\ \mathit{null}\ \mathit{partition}.$$

- Compare and notice some symmetry:

$$\begin{aligned} \mathit{qsort} &= \mathit{flatteniT} \cdot \mathit{partitioniT}, \\ \mathit{msort} &= \mathit{mergeeT} \cdot \mathit{unflatteneT}. \end{aligned}$$

  Both are defined as a fold after an unfold.

Insertion Sort and Selection Sort

- Insertion sort can be defined by an fold:

  $$\mathit{isort} = \mathit{foldr}\ \mathit{insert}\ [\,],$$

  where *insert* is specified by

  $$\mathit{insert}\ x\ \mathit{xs} = \mathit{takeWhile}\ (< x)\ \mathit{xs} +\!\!+ [x] +\!\!+ \mathit{dropWhile}\ (< x)\ \mathit{xs}.$$

- Selection sort, on the other hand, can be naturally seen as an unfold:

  $$\mathit{ssort} = \mathit{unfoldr}\ \mathit{null}\ \mathit{select},$$

  where *select* is specified by

  $$\mathit{select}\ \mathit{xs} = (\mathit{max}\ \mathit{xs}, \mathit{xs} - [\mathit{max}\ \mathit{xs}]).$$

Hylomorphism

- A fold after an unfold is called a *hylomorphism*.
- The unfold phase expands a data structure, while the fold phase reduces it.
- The divide-and-conquer pattern, for example, can be modelled by hylomorphism on trees.
- To avoid generating an intermediate tree, the fold and the unfold can be fused into a recursive function. E.g. let $hyloiT\ f\ e\ p\ g = foldiT\ f\ e \cdot unfoldiT\ p\ g$, we have

$$\begin{array}{lcl} hyloiT\ f\ e\ p\ g\ s & = & \mathbf{if}\ p\ s\ \mathbf{then}\ e\ \mathbf{else} \\ & & \quad \mathbf{let}\ (x, s_1, s_2) = g\ s \\ & & \quad \mathbf{in}\ f\ x\ (hyloiT\ f\ e\ p\ g\ s_1) \\ & & \qquad\qquad (hyloiT\ f\ e\ p\ g\ s_2). \end{array}$$

Hylomorphism and Recursion

Okay, we can express hylomorphisms using recursion. But let us look at it the other way round.

- Imagine a programming in which you are *not* allowed to write explicit recursion. You are given only folds and unfolds for algebraic datatypes[1].
- When you do need recursion, define a datatype capturing the pattern of recursion, and split the recursion into a fold and an unfold.
- This way, we can express any recursion by hylomorphisms!

Therefore, the hylomorphism is a concept as expressive as recursive functions (and, therefore, the Turing machine) — if we are allowed to have hylomorphisms, that is.

[1] Built from regular base functors, if that makes any sense.

Folds Take Inductive Types

- So far, we have assumed that it is allowed to write $\mathit{fold} \cdot \mathit{unfold}$. However, let us not forget that they are defined on different types!
- Folds takes inductive types.
  - If we use folds only, everything terminates (a good property!).
  - Recall that we assume a simple model of functions between sets.
  - On the downside, of course, not every program can be written in terms of folds.

Unfolds Return Coinductive Types

Unfolds returns coinductive types.

- We can generate infinite data structure.
- But if we are allowed to use only unfolds, every program still terminates because there is no “consumer” to infinitely process the infinite data.
- Not every program can be written in terms of unfolds, either.

Hylomorphism, Recursion and Termination

If we allow $\mathit{fold} \cdot \mathit{unfold}$,

- we can now express *every* program computable by a Turing machine.
- But, we need a model assuming that inductive types and coinductive types coincide.
- Therefore, folds must prepare to accept infinite data.
- Therefore, some programs may fail to terminate!
- Which means that *partial functions* have emerged.
- Recursive equations may not have unique solutions.
- And everything we believe so far are not on a solid basis anymore!

Termination, Type Theory, Semantics ...

- One possible way out: instead of total function between sets, we move to *partial functions* between *complete partial orders*, and model what recursion means in this setting.
- There are also alternative approaches staying with functions and sets, but talk about when an equation has a unique solution.
- This is where all the following concepts and fields meet each other: unique solutions, termination, type theory, semantics, programming language theory, computability theory ... and a lot more!

What have we learned?

- To derive programs from specification, functional programming languages allows the expand/reduce transformation.
- A number of properties we need can be proved by induction.
- To capture recurring patterns in reasoning, we move to structural recursion: folds captures induction, while unfolds capture coinduction.
  - We gave lots of examples of the fold-fusion rule.
  - Unfolds are equally important, unfortunately we ran out of space.
- Hylomorphism is as expressive as you can get. However, it introduces non-termination. And that opens rooms for plenty of related research.

Where to Go from Here?

- The Functional Pearls column in Journal of Functional Proramming has lots of neat example of derivations.
- Procedural program derivation (basing on the weakest precondition calculus) is another important branch we did not talk about.
- There are plenty of literature about folds, and
- more recently, papers about unfolds and coinduction.
- You may be interested in theories about inductive types, coinductive types, and datatypes in general,
- and semantics, denotational and operational,
- which may eventually lead you to category theory!

Part III

Procedural Program Derivation

From Day 1 and Day 2...

We have covered a lot about functional program derivation:

- Expand/reduce transformation, and proof by induction.
- Some derivation techniques: accumulating parameter, tupling.
- Folds and fold fusion.
- Unfolds and hylomorphism.

For something you can apply to your work in the "real world", we will talk about deriving procedural programs in the last part of this lecture.
Most of the materials are taken from Anne Kaldewaij's book *Programming: the Derivation of Algorithms*.

The Guarded Command Language

- A program computing the greatest common divisor:

$$
\begin{array}{l}
|[\ \textbf{con}\ A, B : int;\\
\quad \textbf{var}\ x, y : int;\\
\\
\quad x, y := A, B;\\
\quad \textbf{do}\ y < x \rightarrow x := x - y\\
\quad\quad [\!]\ x < y \rightarrow y := y - x\\
\quad \textbf{od}\\
\\
]|.
\end{array}
$$

- Notice: a section for declarations, followed by a section for statements.
- Assignments: $:=$; **do** denotes loops with guarded bodies.

The Guarded Command Language

- A program computing the greatest common divisor:

$$
\begin{array}{l}
|[\ \mathbf{con}\ A, B : \mathit{int};\ \{0 < A \wedge 0 < B\} \\
\quad \mathbf{var}\ x, y : \mathit{int}; \\
\\
\quad x, y := A, B; \\
\quad \mathbf{do}\ y < x \rightarrow x := x - y \\
\quad\quad [\!]\ x < y \rightarrow y := y - x \\
\quad \mathbf{od} \\
\quad \{x = y = \mathit{gcd}(A, B)\} \\
]|.
\end{array}
$$

- Notice: a section for declarations, followed by a section for statements.
- Assignments: $:=$; **do** denotes loops with guarded bodies.
- Assertions delimited in curly brackets.

Assertions

- The *state space* of a program is the states of all its variables.
  - E.g. the GCD program has state space $\mathbb{Z} \times \mathbb{Z}$.
- The *Hoare triple* $\{P\}S\{Q\}$, operationally, denotes that the statement S, when executed in a state satisfying P, *terminates* in a state satisfying Q.
  - E.g., $\{P\}S\{true\}$ expresses that S terminates.
  - $\{P\}S\{Q\}$ and $P_0 \Rightarrow P$ implies $\{P_0\}S\{Q\}$.
  - $\{P\}S\{Q\}$ and $Q \Rightarrow Q_0$ implies $\{P\}S\{Q_0\}$.
- Perhaps the simplest statement: $\{P\}skip\{Q\}$ iff. $P \Rightarrow Q$.

Verification v.s. Derivation

- Recall the relationship between verification and derivation:
  - Verification: given a program, prove that it is correct with respect to some specification.
  - Derivation: start from the specification, and attempt to construct *only* correct programs.
- For this course, verification is mostly about putting in the right assertions.
- We will talk about verification first, before moving on to derivation.

Substitution and Assignments

- $P[E/x]$: substituting occurrences of x in P for E.
  - E.g. $(x \leq 3)[x - 1/x] \equiv x - 1 \leq 3 \equiv x \leq 4$.
- Which is correct:
  1. $\{P\}x := E\{P[E/x]\}$, or
  2. $\{P[E/x]\}x := E\{P\}$?

Substitution and Assignments

- $P[E/x]$: substituting occurrences of x in P for E.
  - E.g. $(x \leq 3)[x-1/x] \equiv x - 1 \leq 3 \equiv x \leq 4$.
- Which is correct:
  1. $\{P\}x := E\{P[E/x]\}$, or
  2. $\{P[E/x]\}x := E\{P\}$?
- Answer: 2! For example:

$$
\begin{aligned}
& \{(x \leq 3)[x+1/x]\}x := x+1\{x \leq 3\} \\
\equiv\ & \{x+1 \leq 3\}x := x+1\{x \leq 3\} \\
\equiv\ & \{x \leq 2\}x := x+1\{x \leq 3\}.
\end{aligned}
$$

E.g. Swapping Booleans

- The $\equiv$ operator is defined by

$$\begin{array}{ll} \mathit{true} \equiv \mathit{true} = \mathit{true} & \quad \mathit{false} \equiv \mathit{true} = \mathit{false} \\ \mathit{true} \equiv \mathit{false} = \mathit{false} & \quad \mathit{false} \equiv \mathit{false} = \mathit{true} \end{array}$$

- $(a \equiv b) \equiv c = a \equiv (b \equiv c)$; $\mathit{true} \equiv a = a$.
- Verify:

$$\begin{array}{l} |[\ \mathbf{var}\ a, b : \mathit{bool}; \\ \quad \{a \equiv A \wedge b \equiv B\} \\ \quad a := a \equiv b; \\ \\ \quad b := a \equiv b; \\ \\ \quad a := a \equiv b; \\ \quad \{a \equiv B \wedge b \equiv A\} \\ ]|. \end{array}$$

E.g. Swapping Booleans

- The $\equiv$ operator is defined by

$$\begin{array}{ll} \mathit{true} \equiv \mathit{true} = \mathit{true} & \mathit{false} \equiv \mathit{true} = \mathit{false} \\ \mathit{true} \equiv \mathit{false} = \mathit{false} & \mathit{false} \equiv \mathit{false} = \mathit{true} \end{array}$$

- $(a \equiv b) \equiv c = a \equiv (b \equiv c)$; $\mathit{true} \equiv a = a$.
- Verify:

$$\begin{array}{l} |[\ \mathbf{var}\ a, b : \mathit{bool}; \\ \quad \{a \equiv A \wedge b \equiv B\} \\ \quad a := a \equiv b; \\ \\ \quad b := a \equiv b; \\ \quad \{a \equiv b \equiv B \wedge b \equiv A\} \\ \quad a := a \equiv b; \\ \quad \{a \equiv B \wedge b \equiv A\} \\ ]|. \end{array}$$

E.g. Swapping Booleans

- The $\equiv$ operator is defined by

$$\begin{array}{ll} \mathit{true} \equiv \mathit{true} = \mathit{true} & \mathit{false} \equiv \mathit{true} = \mathit{false} \\ \mathit{true} \equiv \mathit{false} = \mathit{false} & \mathit{false} \equiv \mathit{false} = \mathit{true} \end{array}$$

- $(a \equiv b) \equiv c = a \equiv (b \equiv c)$; $\mathit{true} \equiv a = a$.
- Verify:

$$\begin{array}{ll}
|[\ \mathbf{var}\ a, b : \mathit{bool}; & \\
\quad \{a \equiv A \wedge b \equiv B\} & \\
\quad a := a \equiv b; & \\
 & \{a \equiv a \equiv b \equiv B \wedge a \equiv b \equiv A\} \\
\quad b := a \equiv b; & \\
\quad \{a \equiv b \equiv B \wedge b \equiv A\} & \\
\quad a := a \equiv b; & \\
\quad \{a \equiv B \wedge b \equiv A\} & \\
]|. &
\end{array}$$

E.g. Swapping Booleans

- The $\equiv$ operator is defined by

$$\begin{array}{ll} true \equiv true = true & false \equiv true = false \\ true \equiv false = false & false \equiv false = true \end{array}$$

- $(a \equiv b) \equiv c = a \equiv (b \equiv c)$; $true \equiv a = a$.
- Verify:

$$\begin{array}{l} |[\ \mathbf{var}\ a, b : bool; \\ \quad \{a \equiv A \land b \equiv B\} \\ \quad a := a \equiv b; \\ \quad \{b \equiv B \land a \equiv b \equiv A\} \Rightarrow \{a \equiv a \equiv b \equiv B \land a \equiv b \equiv A\} \\ \quad b := a \equiv b; \\ \quad \{a \equiv b \equiv B \land b \equiv A\} \\ \quad a := a \equiv b; \\ \quad \{a \equiv B \land b \equiv A\} \\ ]|. \end{array}$$

E.g. Swapping Booleans

- The $\equiv$ operator is defined by

$$\begin{array}{ll} \mathit{true} \equiv \mathit{true} = \mathit{true} & \mathit{false} \equiv \mathit{true} = \mathit{false} \\ \mathit{true} \equiv \mathit{false} = \mathit{false} & \mathit{false} \equiv \mathit{false} = \mathit{true} \end{array}$$

- $(a \equiv b) \equiv c = a \equiv (b \equiv c)$; $\mathit{true} \equiv a = a$.
- Verify:

$$\begin{array}{l} |[\ \mathbf{var}\ a, b : \mathit{bool}; \\ \quad \{a \equiv A \wedge b \equiv B\} \Rightarrow \{b \equiv B \wedge a \equiv b \equiv b \equiv A\} \\ \quad a := a \equiv b; \\ \quad \{b \equiv B \wedge a \equiv b \equiv A\} \\ \quad b := a \equiv b; \\ \quad \{a \equiv b \equiv B \wedge b \equiv A\} \\ \quad a := a \equiv b; \\ \quad \{a \equiv B \wedge b \equiv A\} \\ ]|. \end{array}$$

E.g. Swapping Booleans

- The $\equiv$ operator is defined by

$$\begin{array}{ll} \mathit{true} \equiv \mathit{true} = \mathit{true} & \mathit{false} \equiv \mathit{true} = \mathit{false} \\ \mathit{true} \equiv \mathit{false} = \mathit{false} & \mathit{false} \equiv \mathit{false} = \mathit{true} \end{array}$$

- $(a \equiv b) \equiv c = a \equiv (b \equiv c)$; $\mathit{true} \equiv a = a$.
- Verify:

$$\begin{array}{l} [\![\ \mathbf{var}\ a, b : \mathit{bool}; \\ \quad \{a \equiv A \land b \equiv B\} \\ \quad a := a \equiv b; \\ \quad \{b \equiv B \land a \equiv b \equiv A\} \\ \quad b := a \equiv b; \\ \quad \{a \equiv b \equiv B \land b \equiv A\} \\ \quad a := a \equiv b; \\ \quad \{a \equiv B \land b \equiv A\} \\ ]\!]. \end{array}$$

Selection

- Selection takes the form $\mathbf{if}\ B_0 \rightarrow S_0 [\!] \ldots [\!] B_n \rightarrow S_n\ \mathbf{fi}$.
- Each B_i is called a *guard*; $B_i \rightarrow S_i$ is a *guarded command*.
- If none of the guards $B_0 \ldots B_n$ evaluate to true, the program aborts. Otherwise, one of the command with a true guard is chosen *non-deterministically* and executed.

Selection

- Selection takes the form **if** $B_0 \rightarrow S_0 [\!] \ldots [\!] B_n \rightarrow S_n$ **fi**.
- Each B_i is called a *guard*; $B_i \rightarrow S_i$ is a *guarded command*.
- If none of the guards $B_0 \ldots B_n$ evaluate to true, the program aborts. Otherwise, one of the command with a true guard is chosen *non-deterministically* and executed.
- To annotate an **if** statement:

$$
\begin{array}{l}
\{P\} \\
\textbf{if } B_0 \rightarrow \{P \wedge B_0\} S_0 \{Q\} \\
\ [\!]\ B_1 \rightarrow \{P \wedge B_1\} S_1 \{Q\} \\
\textbf{fi} \\
\{Q, Pf\},
\end{array}
$$

where $Pf :: P \Rightarrow B_0 \vee B_1$.

Binary Maximum

- Goal: to assign $x \uparrow y$ to z. By definition,
  $z = x \uparrow y \equiv (z = x \vee z = y) \wedge x \leq z \wedge y \leq z$.

Binary Maximum

- Goal: to assign $x \uparrow y$ to z. By definition,
  $z = x \uparrow y \equiv (z = x \vee z = y) \wedge x \leq z \wedge y \leq z$.
- Try $z := x$. We reason:

$$
\begin{array}{ll}
 & ((z = x \vee z = y) \wedge x \leq z \wedge y \leq z)[x/z] \\
\equiv & (x = x \vee x = y) \wedge x \leq x \wedge y \leq x \\
\equiv & y \leq x,
\end{array}
$$

  which hinted at using a guarded command: $y \leq x \rightarrow z := x$.

Binary Maximum

- Goal: to assign $x \uparrow y$ to z. By definition,
  $z = x \uparrow y \equiv (z = x \vee z = y) \wedge x \leq z \wedge y \leq z$.
- Try $z := x$. We reason:

$$\begin{array}{ll} & ((z = x \vee z = y) \wedge x \leq z \wedge y \leq z)[x/z] \\ \equiv & (x = x \vee x = y) \wedge x \leq x \wedge y \leq x \\ \equiv & y \leq x, \end{array}$$

  which hinted at using a guarded command: $y \leq x \rightarrow z := x$.
- Indeed:

$$\begin{array}{l} \{\mathit{true}\} \\ \textbf{if } y \leq x \rightarrow \{y \leq x\} z := x \{z = x \uparrow y\} \\ \;\, [\!]\; x \leq y \rightarrow \{x \leq y\} z := y \{z = x \uparrow y\} \\ \textbf{fi} \\ \{z = x \uparrow y\}. \end{array}$$

Loops

- Repetition takes the form **do** $B_0 \rightarrow S_0 \mathbin{[\!]} \ldots \mathbin{[\!]} B_n \rightarrow S_n$ **od**.
- If none of the guards $B_0 \ldots B_n$ evaluate to true, the loop terminates. Otherwise one of the commands is chosen non-deterministically, before the next iteration.

Loops

- Repetition takes the form $\mathbf{do}\ B_0 \rightarrow S_0 [\!] \ldots [\!] B_n \rightarrow S_n\ \mathbf{od}$.
- If none of the guards $B_0 \ldots B_n$ evaluate to true, the loop terminates. Otherwise one of the commands is chosen non-deterministically, before the next iteration.
- To annotate a loop (for partial correctness):

$$
\begin{array}{l}
\{P\} \\
\mathbf{do}\ B_0 \rightarrow \{P \wedge B_0\} S_0 \{P\} \\
\quad [\!]\ B_1 \rightarrow \{P \wedge B_1\} S_1 \{P\} \\
\mathbf{od} \\
\{Q, Pf\},
\end{array}
$$

where $Pf :: P \wedge \neg B_0 \wedge \neg B_1 \Rightarrow Q$.

Loops

- Repetition takes the form $\mathbf{do}\ B_0 \rightarrow S_0 [\!] \ldots [\!] B_n \rightarrow S_n\ \mathbf{od}$.
- If none of the guards $B_0 \ldots B_n$ evaluate to true, the loop terminates. Otherwise one of the commands is chosen non-deterministically, before the next iteration.
- To annotate a loop (for partial correctness):

$$\begin{array}{l} \{P\} \\ \mathbf{do}\ B_0 \rightarrow \{P \wedge B_0\} S_0 \{P\} \\ \quad [\!]\ B_1 \rightarrow \{P \wedge B_1\} S_1 \{P\} \\ \mathbf{od} \\ \{Q, Pf\}, \end{array}$$

  where $Pf :: P \wedge \neg B_0 \wedge \neg B_1 \Rightarrow Q$.
- P is called the *loop invariant*. Every loop should be constructed with an invariant in mind!

Linear-Time Exponentiation

$|[$ **con** $N\{0 \leq N\}$; **var** $x, n : int$;

$x, n := 1, 0$;

do $n \neq N \rightarrow$

$x, n := x + x, n + 1$

od

$\{x = 2^N \}$

$]|$

Linear-Time Exponentiation

$|[\ \mathbf{con}\ N\{0 \leq N\};\ \mathbf{var}\ x, n : int;$

$x, n := 1, 0;$
$\{x = 2^n \wedge n \leq N\}$
$\mathbf{do}\ n \neq N \rightarrow$

$\quad x, n := x + x, n + 1$

$\mathbf{od}$
$\{x = 2^N \qquad \}$
$]|$

Linear-Time Exponentiation

$|[\ \mathbf{con}\ N\{0 \leq N\}; \mathbf{var}\ x, n : int;$

$x, n := 1, 0;$

$\{x = 2^n \wedge n \leq N\}$

$\mathbf{do}\ n \neq N \rightarrow$

$x, n := x + x, n + 1$

$\mathbf{od}$

$\{x = 2^N, Pf2\}$

$]|$

Pf2:

$$x = 2^n \wedge n \leq N \wedge \neg(n \neq N) \Rightarrow x = 2^N$$

Linear-Time Exponentiation

$|[\,\mathbf{con}\ N\{0 \le N\};\ \mathbf{var}\ x, n : int;$

$\quad x, n := 1, 0;$
$\quad \{x = 2^n \wedge n \le N\}$
$\quad \mathbf{do}\ n \ne N \rightarrow$

$\qquad x, n := x + x, n + 1$
$\qquad \{x = 2^n \wedge n \le N, Pf1\}$
$\quad \mathbf{od}$
$\quad \{x = 2^N, Pf2\}$
$]|$

Pf2:

$$x = 2^n \wedge n \le N \wedge \neg(n \ne N) \Rightarrow x = 2^N$$

Linear-Time Exponentiation

$$
\begin{array}{l}
|[\ \mathbf{con}\ N\{0 \leq N\}; \mathbf{var}\ x, n : int; \\
\quad x, n := 1, 0; \\
\quad \{x = 2^n \wedge n \leq N\} \\
\quad \mathbf{do}\ n \neq N \rightarrow \\
\qquad \{x = 2^n \wedge n \leq N \wedge n \neq N\} \\
\qquad x, n := x + x, n + 1 \\
\qquad \{x = 2^n \wedge n \leq N, Pf1\} \\
\quad \mathbf{od} \\
\quad \{x = 2^N, Pf2\} \\
]|
\end{array}
$$

Pf2:

$$
\begin{array}{l}
x = 2^n \wedge n \leq N \wedge \neg(n \neq N) \\
\qquad \Rightarrow x = 2^N
\end{array}
$$

Linear-Time Exponentiation

$|[\,\mathbf{con}\ N\{0 \le N\};\ \mathbf{var}\ x, n : int;$

$\quad x, n := 1, 0;$
$\quad \{x = 2^n \wedge n \le N\}$
$\quad \mathbf{do}\ n \ne N \rightarrow$
$\qquad \{x = 2^n \wedge n \le N \wedge n \ne N\}$
$\qquad x, n := x + x, n + 1$
$\qquad \{x = 2^n \wedge n \le N, Pf1\}$
$\quad \mathbf{od}$
$\quad \{x = 2^N, Pf2\}$
$]|$

Pf1:

$$
\begin{aligned}
& (x = 2^n \wedge n \le N)[x + x, n + 1/x, n] \\
\equiv\ & x + x = 2^{n+1} \wedge n + 1 \le N \\
\equiv\ & x = 2^n \wedge n < N
\end{aligned}
$$

Pf2:

$$
\begin{aligned}
& x = 2^n \wedge n \le N \wedge \neg(n \ne N) \\
& \quad \Rightarrow x = 2^N
\end{aligned}
$$

Greatest Common Divisor

- Known: $gcd(x, x) = x$; $gcd(x, y) = gcd(x, x - y)$ if $x > y$.

Greatest Common Divisor

- Known: $gcd(x,x) = x$; $gcd(x,y) = gcd(x, x-y)$ if $x > y$.
-
$$
\begin{array}{l}
|[\ \textbf{con}\ A, B : int; \{0 < A \wedge 0 < B\} \\
\quad \textbf{var}\ x, y : int; \\
\\
\quad x, y := A, B; \\
\quad \{0 < x \wedge 0 < y \wedge gcd(x,y) = gcd(A,B)\} \\
\quad \textbf{do}\ y < x \rightarrow x := x - y \\
\quad\quad [\!]\ x < y \rightarrow y := y - x \\
\quad \textbf{od} \\
\quad \{x = gcd(A,B) \wedge y = gcd(A,B)\} \\
]|
\end{array}
$$

Greatest Common Divisor

- Known: $gcd(x,x) = x$; $gcd(x,y) = gcd(x, x-y)$ if $x > y$.

$$
\begin{array}{l}
|[\ \mathbf{con}\ A, B : int; \{0 < A \wedge 0 < B\} \\
\quad \mathbf{var}\ x, y : int; \\
\\
\quad x, y := A, B; \\
\quad \{0 < x \wedge 0 < y \wedge gcd(x,y) = gcd(A,B)\} \\
\quad \mathbf{do}\ y < x \rightarrow x := x - y \\
\quad\quad [\!]\ x < y \rightarrow y := y - x \\
\quad \mathbf{od} \\
\quad \{x = gcd(A,B) \wedge y = gcd(A,B)\} \\
]|
\end{array}
$$

-
$$
\begin{array}{cl}
 & (0 < x \wedge 0 < y \wedge gcd(x,y) = gcd(A,B))[x - y/x] \\
\equiv & 0 < x - y \wedge 0 < y \wedge gcd(x-y, y) = gcd(A,B) \\
\Leftarrow & 0 < x \wedge 0 < y \wedge gcd(x,y) = gcd(A,B) \wedge y < x
\end{array}
$$

A Weird Equilibrium

- Consider the following program:

$$
\begin{array}{l}
|[\ \textbf{var}\ x, y, z : \mathit{int};\\
\quad \{\mathit{true} \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \}\\
\quad \textbf{do}\ x < y \rightarrow x := x + 1\\
\quad\quad [\!]\ y < z \rightarrow y := y + 1\\
\quad\quad [\!]\ z < x \rightarrow z := z + 1\\
\quad \textbf{od}\\
\quad \{x = y = z\}\\
]|.
\end{array}
$$

- If it terminates at all, we do have $x = y = z$. But why does it terminate?

A Weird Equilibrium

- Consider the following program:

$$
\begin{array}{l}
|[\ \mathbf{var}\ x, y, z : int; \\
\quad \{true, bnd = 3 \times (x \uparrow y \uparrow z) - (x + y + z)\} \\
\quad \mathbf{do}\ x < y \rightarrow x := x + 1 \\
\quad\quad [\!]\ y < z \rightarrow y := y + 1 \\
\quad\quad [\!]\ z < x \rightarrow z := z + 1 \\
\quad \mathbf{od} \\
\quad \{x = y = z\} \\
]|.
\end{array}
$$

- If it terminates at all, we do have $x = y = z$. But why does it terminate?
  1. $bnd \geq 0$, and $bnd = 0$ implies none of the guards are true.
  2. $\{x < y \wedge bnd = t\}x := x + 1\{bnd < t\}$.

Repetition

To annotate a loop for *total correctness*:

$$\begin{array}{l}
\{P, bnd = t\} \\
\textbf{do}\ B_0 \rightarrow \{P \wedge B_0\} S_0 \{P\} \\
\quad [\!]\ B_1 \rightarrow \{P \wedge B_1\} S_1 \{P\} \\
\textbf{od} \\
\{Q\},
\end{array}$$

we have got a list of things to prove:

Repetition

To annotate a loop for *total correctness*:

$$\begin{array}{l} \{P, bnd = t\} \\ \textbf{do}\ B_0 \rightarrow \{P \wedge B_0\} S_0 \{P\} \\ \quad [\!]\ B_1 \rightarrow \{P \wedge B_1\} S_1 \{P\} \\ \textbf{od} \\ \{Q\}, \end{array}$$

we have got a list of things to prove:

1. $B \wedge \neg B_0 \wedge \neg B_1 \Rightarrow Q$,

Repetition

To annotate a loop for *total correctness*:

$$
\begin{array}{l}
\{P, bnd = t\} \\
\textbf{do}\ B_0 \rightarrow \{P \wedge B_0\} S_0 \{P\} \\
\quad [\!]\ B_1 \rightarrow \{P \wedge B_1\} S_1 \{P\} \\
\textbf{od} \\
\{Q\},
\end{array}
$$

we have got a list of things to prove:

1. $B \wedge \neg B_0 \wedge \neg B_1 \Rightarrow Q$,
2. for all i, $\{P \wedge B_i\} S_i \{P\}$,

Repetition

To annotate a loop for *total correctness*:

$$
\begin{array}{l}
\{P, bnd = t\} \\
\mathbf{do}\ B_0 \rightarrow \{P \wedge B_0\} S_0 \{P\} \\
\quad [\!]\ B_1 \rightarrow \{P \wedge B_1\} S_1 \{P\} \\
\mathbf{od} \\
\{Q\},
\end{array}
$$

we have got a list of things to prove:

1. $B \wedge \neg B_0 \wedge \neg B_1 \Rightarrow Q$,
2. for all i, $\{P \wedge B_i\} S_i \{P\}$,
3. $P \wedge (B_1 \vee B_2) \Rightarrow t \geq 0$,

Repetition

To annotate a loop for *total correctness*:

$$
\begin{array}{l}
\{P, bnd = t\} \\
\textbf{do } B_0 \rightarrow \{P \wedge B_0\} S_0 \{P\} \\
\quad [\!] \; B_1 \rightarrow \{P \wedge B_1\} S_1 \{P\} \\
\textbf{od} \\
\{Q\},
\end{array}
$$

we have got a list of things to prove:

1. $B \wedge \neg B_0 \wedge \neg B_1 \Rightarrow Q$,
2. for all i, $\{P \wedge B_i\} S_i \{P\}$,
3. $P \wedge (B_1 \vee B_2) \Rightarrow t \geq 0$,
4. for all i, $\{P \wedge B_i \wedge t = C\} S_i \{t < C\}$.

E.g. Linear-Time Exponentiation

- What is the bound function?

$$
\begin{array}{l}
|[\ \mathbf{con}\ N\{0 \leq N\};\ \mathbf{var}\ x, n : int; \\
\\
\quad x, n := 1, 0; \\
\quad \{x = 2^n \wedge n \leq N \qquad\qquad \} \\
\quad \mathbf{do}\ n \neq N \rightarrow \\
\qquad x, n := x + x, n + 1 \\
\quad \mathbf{od} \\
\quad \{x = 2^N\} \\
]|
\end{array}
$$

E.g. Linear-Time Exponentiation

- What is the bound function?

$$
\begin{array}{l}
|[\,\mathbf{con}\ N\{0 \leq N\};\mathbf{var}\ x, n : int;\\
\\
\quad x, n := 1, 0;\\
\quad \{x = 2^n \wedge n \leq N, N - n\}\\
\quad \mathbf{do}\ n \neq N \rightarrow\\
\qquad x, n := x + x, n + 1\\
\quad \mathbf{od}\\
\quad \{x = 2^N\}\\
]|
\end{array}
$$

- $x = 2^n \wedge n \wedge n \neq N \Rightarrow N - n \geq 0$,
- $\{\ldots \wedge N - n = t\}x, n := x + x, n - 1\{N - n < t\}$.

E.g. Greatest Common Divisor

- What is the bound function?

$$
\begin{array}{l}
|[\ \mathbf{con}\ A, B : int; \{0 < A \wedge 0 < B\} \\
\quad \mathbf{var}\ x, y : int; \\
\\
\quad x, y := A, B; \\
\quad \{0 < x \wedge 0 < y \wedge gcd(x, y) = gcd(A, B) \qquad\qquad \} \\
\quad \mathbf{do}\ y < x \rightarrow x := x - y \\
\quad\quad [\!]\ x < y \rightarrow y := y - x \\
\quad \mathbf{od} \\
\quad \{x = gcd(A, B) \wedge y = gcd(A, B)\} \\
]|
\end{array}
$$

E.g. Greatest Common Divisor

- What is the bound function?

$$
\begin{array}{l}
|[\ \mathbf{con}\ A, B : int; \{0 < A \wedge 0 < B\} \\
\quad \mathbf{var}\ x, y : int; \\
\\
\quad x, y := A, B; \\
\quad \{0 < x \wedge 0 < y \wedge gcd(x, y) = gcd(A, B), bnd = |x - y|\} \\
\quad \mathbf{do}\ y < x \rightarrow x := x - y \\
\quad\quad [\!] \ x < y \rightarrow y := y - x \\
\quad \mathbf{od} \\
\quad \{x = gcd(A, B) \wedge y = gcd(A, B)\} \\
]|
\end{array}
$$

- $\ldots \Rightarrow |x - y| \geq 0$,
- $\{\ldots 0 < y \wedge y < x \wedge |x - y| = t\}x := x - y\{|x - y| < t\}$.

Deriving Programs from Specifications

- From such a specification:

  $$\begin{array}{l} |[\ \mathbf{con}\ \mathit{declarations};\\ \quad \{\mathit{preconditions}\}\\ \quad \mathit{prog}\\ \quad \{\mathit{postcondition}\}\\ ]| \end{array}$$

  we hope to derive *prog*.
- We usually work backwards from the post condition.
- The techniques we are about to learn is mostly about constructing loops and loop invariants.

Conjunctive Postconditions

- When the post condition has the form $P \wedge Q$, one may take one of the conjuncts as the invariant and the other as the guard:
  - $\{P\}\mathbf{do}\,\neg Q \to S\,\mathbf{od}\{P \wedge Q\}$.
- E.g. consider the specficication:

$$
\begin{array}{l}
|[\ \mathbf{con}\ A, B : int; \{0 \leq A \wedge 0 \leq B\} \\
\quad \mathbf{var}\ q, r : int; \\
\quad divmod \\
\quad \{q = A\,\mathbf{div}\,B \wedge r = A\,\mathbf{mod}\,B\} \\
]|.
\end{array}
$$

- The post condition expands to $R :: A = q \times B + r \wedge 0 \leq r \wedge r < B$.

Computing the Quotient and the Remainder

Let try $A = q \times B + r \wedge 0 \leq r$ as the invariant and $\neg(r < B)$.

$\{P :: A = q \times B + r \wedge 0 \leq r\}$
do $B \leq r \rightarrow$

od
$\{P \wedge r < B\}$

Computing the Quotient and the Remainder

Let try $A = q \times B + r \wedge 0 \leq r$ as the invariant and $\neg(r < B)$.

- P is established by $q, r := 0, A$.

$q, r := 0, A;$
$\{P :: A = q \times B + r \wedge 0 \leq r\}$
do $B \leq r \rightarrow$

od
$\{P \wedge r < B\}$

Computing the Quotient and the Remainder

Let try $A = q \times B + r \wedge 0 \leq r$ as the invariant and $\neg(r < B)$.

$q, r := 0, A;$
$\{P :: A = q \times B + r \wedge 0 \leq r\}$
do $B \leq r \rightarrow$

od
$\{P \wedge r < B\}$

- P is established by $q, r := 0, A$.
- Choose r as the bound.

Computing the Quotient and the Remainder

Let try $A = q \times B + r \wedge 0 \leq r$ as the invariant and $\neg(r < B)$.

$q, r := 0, A;$
$\{P :: A = q \times B + r \wedge 0 \leq r\}$
do $B \leq r \rightarrow$

$\quad r := r - B$

od
$\{P \wedge r < B\}$

- P is established by $q, r := 0, A$.
- Choose r as the bound.
- Since $B > 0$, try $r := r - B$:

$$\begin{aligned} & P[r - B/r] \\ \equiv\ & A = q \times B + r - B \wedge 0 \leq r - B \\ \equiv\ & A = (q-1) \times B + r \wedge B \leq r. \end{aligned}$$

Computing the Quotient and the Remainder

Let try $A = q \times B + r \wedge 0 \leq r$ as the invariant and $\neg(r < B)$.

$q, r := 0, A;$
$\{P :: A = q \times B + r \wedge 0 \leq r\}$
do $B \leq r \rightarrow$
$\quad r := r - B$
od
$\{P \wedge r < B\}$

- P is established by $q, r := 0, A$.
- Choose r as the bound.
- Since $B > 0$, try $r := r - B$:

$$\begin{aligned} & P[r - B/r] \\ \equiv\ & A = q \times B + r - B \wedge 0 \leq r - B \\ \equiv\ & A = (q-1) \times B + r \wedge B \leq r. \end{aligned}$$

- $$\begin{aligned} & (A = (q-1)B + r \wedge B \leq r) \\ \Leftarrow\ & A = q \times B + r \wedge B \leq r \end{aligned}$$

Computing the Quotient and the Remainder

Let try $A = q \times B + r \wedge 0 \leq r$ as the invariant and $\neg(r < B)$.

$q, r := 0, A;$
$\{P :: A = q \times B + r \wedge 0 \leq r\}$
do $B \leq r \rightarrow$
  $q := q + 1;$
  $r := r - B$
od
$\{P \wedge r < B\}$

- P is established by $q, r := 0, A$.
- Choose r as the bound.
- Since $B > 0$, try $r := r - B$:
  $$\begin{aligned} & P[r - B/r] \\ \equiv\ & A = q \times B + r - B \wedge 0 \leq r - B \\ \equiv\ & A = (q-1) \times B + r \wedge B \leq r. \end{aligned}$$
- $$\begin{aligned} & (A = (q-1)B + r \wedge B \leq r)[q+1/q] \\ \Leftarrow\ & A = q \times B + r \wedge B \leq r \end{aligned}$$

Quantifications

- Given associative $\oplus$ with identity e, we denote $x\, m \oplus x\,(m+1) \ldots \oplus x\,(n-1)$ by $(\oplus i : m{\leq}i{<}n : x\, i)$.
- $(\oplus i : n{\leq}i{<}n : x\, i) = e$.
- $(\oplus i : m{\leq}i{<}n{+}1 : x\, i) = (\oplus i : m{\leq}i{<}n : x\, i) \oplus x\, n$ if $m \leq n$.
- E.g.
  - $(+i : 3{\leq}i{<}5 : i^2) = 3^2 + 4^2 = 25$.
  - $(+i, j : 3{\leq}i{\leq}j{<}5 : i \times j) = 3 \times 3 + 3 \times 4 + 4 \times 4$.
  - $(\wedge i : 2{\leq}i{<}9 : \mathit{odd}\ i \Rightarrow \mathit{prime}\ i) = \mathit{true}$.
  - $(\uparrow i : 1{\leq}i{<}7 : -i^2 + 5i) = 6$ (when $i = 2$ or 3).
- As a convention, $(+i : 0{\leq}i{<}n : x\, i)$ is written $(\Sigma i : 0{\leq}i{<}n : x\, i)$.

Summing Up an Array

$|[\, \mathbf{con}\ N : int; \{0 \leq N\}\ f : \mathbf{array}\ [0..N)\ \mathbf{of}\ int;$

$\{x = (\Sigma i : 0 \leq i < n : f\ i) \qquad , bnd : N - n\}$
$\mathbf{do}\ n \neq N \rightarrow \qquad \mathbf{od}$
$\{x = (\Sigma i : 0 \leq i < N : f\ i)\}$
$]|$

Summing Up an Array

$$
\begin{array}{l}
|[\ \mathbf{con}\ N : int; \{0 \leq N\}\ f : \mathbf{array}\ [0..N)\ \mathbf{of}\ int; \\
\quad n, x := 0, 0; \\
\quad \{x = (\Sigma i : 0 \leq i < n : f\ i) \qquad\qquad , bnd : N - n\} \\
\quad \mathbf{do}\ n \neq N \rightarrow \qquad\qquad\qquad\qquad \mathbf{od} \\
\quad \{x = (\Sigma i : 0 \leq i < N : f\ i)\} \\
]|
\end{array}
$$

Summing Up an Array

$$
\begin{array}{l}
|[\,\textbf{con}\ N : int; \{0 \leq N\}\ f : \textbf{array}\ [0..N)\ \textbf{of}\ int; \\
\quad n, x := 0, 0; \\
\quad \{x = (\Sigma i : 0{\leq}i{<}n : f\ i) \qquad\qquad , bnd : N - n\} \\
\quad \textbf{do}\ n \neq N \rightarrow \qquad\qquad n := n + 1\ \textbf{od} \\
\quad \{x = (\Sigma i : 0{\leq}i{<}N : f\ i)\} \\
]|
\end{array}
$$

- Use $N - n$ as bound, try incrementing n:

$$
\begin{array}{l}
(x = (\Sigma i : 0{\leq}i{<}n : f\ i) \qquad\qquad)[n + 1/n] \\
\equiv\ x = (\Sigma i : 0{\leq}i{<}n + 1 : f\ i) \\
\\
\equiv\ x = (\Sigma i : 0{\leq}i{<}n : f\ i) + f\ n
\end{array}
$$

Summing Up an Array

$$
\begin{array}{l}
|[\ \mathbf{con}\ N : int; \{0 \le N\}\ f : \mathbf{array}\ [0..N)\ \mathbf{of}\ int; \\
\quad n, x := 0, 0; \\
\quad \{x = (\Sigma i : 0 \le i < n : f\ i) \wedge 0 \le n, bnd : N - n\} \\
\quad \mathbf{do}\ n \ne N \to \qquad\qquad\qquad n := n + 1\ \mathbf{od} \\
\quad \{x = (\Sigma i : 0 \le i < N : f\ i)\} \\
]|
\end{array}
$$

- Use $N - n$ as bound, try incrementing n:

$$
\begin{array}{l}
(x = (\Sigma i : 0 \le i < n : f\ i) \wedge 0 \le n)[n + 1/n] \\
\equiv\ x = (\Sigma i : 0 \le i < n + 1 : f\ i) \wedge 0 \le n + 1 \\
\Leftarrow x = (\Sigma i : 0 \le i < n + 1 : f\ i) \wedge 0 \le n \\
\equiv\ x = (\Sigma i : 0 \le i < n : f\ i) + f\ n \wedge 0 \le n
\end{array}
$$

Summing Up an Array

$$
\begin{array}{l}
|[\ \textbf{con}\ N : int; \{0 \leq N\}\ f : \textbf{array}\ [0..N)\ \textbf{of}\ int; \\
\quad n, x := 0, 0; \\
\quad \{x = (\Sigma i : 0{\leq}i{<}n : f\ i) \wedge 0 \leq n, bnd : N - n\} \\
\quad \textbf{do}\ n \neq N \rightarrow \qquad\qquad n := n + 1\ \textbf{od} \\
\quad \{x = (\Sigma i : 0{\leq}i{<}N : f\ i)\} \\
]|
\end{array}
$$

- Use $N - n$ as bound, try incrementing n:

$$
\begin{array}{l}
(x = (\Sigma i : 0{\leq}i{<}n : f\ i) \wedge 0 \leq n)[n+1/n] \\
\equiv\ x = (\Sigma i : 0{\leq}i{<}n+1 : f\ i) \wedge 0 \leq n+1 \\
\Leftarrow x = (\Sigma i : 0{\leq}i{<}n+1 : f\ i) \wedge 0 \leq n \\
\equiv\ x = (\Sigma i : 0{\leq}i{<}n : f\ i) + f\ n \wedge 0 \leq n \\
(x = (\Sigma i : 0{\leq}i{<}n : f\ i) + f\ n \wedge 0 \leq n)
\end{array}
$$

-

$$\Leftarrow\ x = (\Sigma i : 0{\leq}i{<}n : f\ i) \wedge 0 \leq n$$

Summing Up an Array

$$
\begin{array}{l}
|[\ \textbf{con}\ N : int; \{0 \leq N\}\ f : \textbf{array}\ [0..N)\ \textbf{of}\ int; \\
\quad n, x := 0, 0; \\
\quad \{x = (\Sigma i : 0 {\leq} i {<} n : f\ i) \wedge 0 \leq n, bnd : N - n\} \\
\quad \textbf{do}\ n \neq N \rightarrow x := x + f\ n;\ \ n := n + 1\ \textbf{od} \\
\quad \{x = (\Sigma i : 0 {\leq} i {<} N : f\ i)\} \\
]|
\end{array}
$$

- Use $N - n$ as bound, try incrementing n:

$$
\begin{array}{l}
(x = (\Sigma i : 0 {\leq} i {<} n : f\ i) \wedge 0 \leq n)[n + 1/n] \\
\equiv\ x = (\Sigma i : 0 {\leq} i {<} n + 1 : f\ i) \wedge 0 \leq n + 1 \\
\Leftarrow x = (\Sigma i : 0 {\leq} i {<} n + 1 : f\ i) \wedge 0 \leq n \\
\equiv\ x = (\Sigma i : 0 {\leq} i {<} n : f\ i) + f\ n \wedge 0 \leq n
\end{array}
$$

- $$
\begin{array}{l}
(x = (\Sigma i : 0 {\leq} i {<} n : f\ i) + f\ n \wedge 0 \leq n)[x + f\ n/x] \\
\equiv\ x + f\ n = (\Sigma i : 0 {\leq} i {<} n : f\ i) + f\ n \wedge 0 \leq n \\
\Leftarrow\ x = (\Sigma i : 0 {\leq} i {<} n : f\ i) \wedge 0 \leq n
\end{array}
$$

Fibonacci

Recall: $fib\,0 = 0$, $fib\,1 = 1$, and $fib\,(n+2) = fib\,n + fib\,(n+1)$.

$$
\begin{array}{l}
|[\ \mathbf{con}\ N : int; \{0 \le N\}\ \ \mathbf{var}\ x \quad : int; \\
\quad n, x \quad := 0, 0 \quad ; \\
\quad \{x = fib\,n \wedge 0 \le n \le N \qquad\qquad\qquad \} \\
\quad \mathbf{do}\ n \ne N \rightarrow \qquad\qquad\qquad n := n + 1 \\
\quad \mathbf{od} \\
\quad \{x = fib\,N\}\ ]|
\end{array}
$$

- Inv. is established by $n, x := 0, 0$.
- $$\begin{array}{ll} & (x = fib\,n \wedge 0{\le}n{\le}N \qquad\qquad)[n{+}1/n] \\ \equiv & x = fib\,(n{+}1) \wedge 0{\le}n{<}N \end{array}$$

Fibonacci

Recall: $fib\, 0 = 0$, $fib\, 1 = 1$, and $fib\,(n+2) = fib\, n + fib\,(n+1)$.

$$
\begin{array}{l}
|[\ \mathbf{con}\ N : int; \{0 \le N\}\ \ \mathbf{var}\ x : int; \\
\quad n, x := 0, 0\ ; \\
\quad \{x = fib\, n \wedge 0 \le n \le N\} \\
\quad \mathbf{do}\ n \ne N \to \qquad n := n + 1 \\
\quad \mathbf{od} \\
\quad \{x = fib\, N\}\ ]|
\end{array}
$$

- Inv. is established by $n, x := 0, 0$.
- $(x = fib\, n \wedge 0 \le n \le N)[n{+}1/n]$
  $\equiv\ x = fib\,(n{+}1) \wedge 0 \le n < N$
- $(x = fib\,(n{+}1) \wedge \ldots)$
  $\Leftarrow\ x = fib\, n \wedge \ldots$

Fibonacci

Recall: $fib\,0 = 0$, $fib\,1 = 1$, and $fib\,(n+2) = fib\,n + fib\,(n+1)$.

$$
\begin{array}{l}
|[\ \mathbf{con}\ N : int; \{0 \leq N\}\ \ \mathbf{var}\ x, y : int; \\
\quad n, x, y := 0, 0, 1; \\
\quad \{x = fib\,n \wedge 0 \leq n \leq N \wedge\ y = fib\,(n+1)\} \\
\quad \mathbf{do}\ n \neq N \rightarrow \qquad\qquad n := n+1 \\
\quad \mathbf{od} \\
\quad \{x = fib\,N\}\ ]|
\end{array}
$$

- Inv. is established by $n, x := 0, 0$.
- $$\begin{array}{l} (x = fib\,n \wedge 0{\leq}n{\leq}N \wedge y = fib\,(n{+}1))[n{+}1/n] \\ \equiv\ x = fib\,(n{+}1) \wedge 0{\leq}n{<}N \wedge y = fib\,(n{+}2) \end{array}$$
- $$\begin{array}{l} (x = fib\,(n{+}1) \wedge \ldots \wedge y = fib\,(n{+}2)) \\ \\ \Leftarrow\ x = fib\,n \wedge \ldots \wedge y = fib\,(n{+}1) \end{array}$$

Fibonacci

Recall: $fib\,0 = 0$, $fib\,1 = 1$, and $fib\,(n+2) = fib\,n + fib\,(n+1)$.

$$
\begin{array}{l}
|[\ \mathbf{con}\ N : int; \{0 \le N\}\ \ \mathbf{var}\ x, y : int; \\
\quad n, x, y := 0, 0, 1; \\
\quad \{x = fib\,n \wedge 0 \le n \le N \wedge\ \ y = fib\,(n+1)\} \\
\quad \mathbf{do}\ n \ne N \to\ \ x, y := y, x + y;\quad n := n + 1 \\
\quad \mathbf{od} \\
\quad \{x = fib\,N\}\ ]|
\end{array}
$$

- Inv. is established by $n, x := 0, 0$.
- $$\begin{array}{l} (x = fib\,n \wedge 0{\le}n{\le}N \wedge y = fib\,(n{+}1))[n{+}1/n] \\ \equiv\ x = fib\,(n{+}1) \wedge 0{\le}n{<}N \wedge y = fib\,(n{+}2) \end{array}$$
- $$\begin{array}{l} (x = fib\,(n{+}1) \wedge \ldots \wedge y = fib\,(n{+}2))[y, x+y/x, y] \\ \equiv\ y = fib\,(n{+}1) \wedge \ldots \wedge x + y = fib\,(n{+}2) \\ \Leftarrow\ x = fib\,n \wedge \ldots \wedge y = fib\,(n{+}1) \end{array}$$

Using Associativity

- Consider again computing A^B. Notice that:

$$\begin{array}{rcll} x^0 & = & 1 & \\ x^y & = & 1 \times (x \times x)^{y\ \mathbf{div}\ 2} & \text{if } \mathit{even}\ y, \\ & = & x \times x^{y-1} & \text{if } \mathit{odd}\ y. \end{array}$$

- Starting from A^B, we can use the properties above to keep "shifting some value to the left" until we have $x_1 \times \ldots \times 1$.
- Also notice that we need $\times$ to be associative.

Using Associativity

- In general, to achieve $r = f\,X$ where

$$\begin{array}{lcll} f\,x & = & a & \text{if } b\,x, \\ f\,x & = & g\,x \oplus f\,(h\,x) & \text{if } \neg b\,x. \end{array}$$

for associative $\oplus$ with identity e, we may:

$$\begin{array}{l} x, r := X, e; \\ \{r \oplus f\,x = f\,X\} \\ \mathbf{do}\ \neg b\,x \rightarrow x, r := h\,x, r \oplus g\,x\ \mathbf{od}; \\ \{r \oplus a = f\,X\} \\ r := r \oplus a. \end{array}$$

Using Associativity

- In general, to achieve $r = f\,X$ where

  $$\begin{array}{lcll} f\,x & = & a & \text{if } b\,x, \\ f\,x & = & g\,x \oplus f\,(h\,x) & \text{if } \neg b\,x. \end{array}$$

  for associative $\oplus$ with identity e, we may:

  $$\begin{array}{l} x, r := X, e; \\ \{r \oplus f\,x = f\,X\} \\ \mathbf{do}\ \neg b\,x \to x, r := h\,x, r \oplus g\,x\ \mathbf{od}; \\ \{r \oplus a = f\,X\} \\ r := r \oplus a. \end{array}$$

- Verify:

  $$\begin{array}{l} (r \oplus f\,x = f\,X)[h\,x, r \oplus g\,x / x, r] \\ \equiv\ (r \oplus g\,x) \oplus f\,(h\,x) = f\,X \\ \equiv\ r \oplus (g\,x \oplus f\,(h\,x)) = f\,X \\ \equiv\ r \oplus f\,x = f\,X. \end{array}$$

Fast Exponentiation

- To achieve $r = A^B$, choose invariant $r \times x^y = A^B$:

$$
\begin{array}{l}
r, x, y := 1, A, B; \\
\{r \times x^y = A^B \wedge 0 \leq y, bnd = y\} \\
\mathbf{do}\ y \neq 0 \wedge even\, y \rightarrow x, y := x \times x, y\ \mathbf{div}\ 2 \\
\quad [\!]\ y \neq 0 \wedge odd\ y \rightarrow r, y := r \times x, y - 1 \\
\mathbf{od} \\
\{r \times x^y = A^B \wedge y = 0\}.
\end{array}
$$

Fast Exponentiation

- To achieve $r = A^B$, choose invariant $r \times x^y = A^B$:

$$
\begin{array}{l}
r, x, y := 1, A, B; \\
\{r \times x^y = A^B \wedge 0 \leq y, bnd = y\} \\
\mathbf{do}\ y \neq 0 \wedge even\, y \rightarrow x, y := x \times x, y\ \mathbf{div}\ 2 \\
\quad [\!]\ y \neq 0 \wedge odd\, y \rightarrow r, y := r \times x, y - 1 \\
\mathbf{od} \\
\{r \times x^y = A^B \wedge y = 0\}.
\end{array}
$$

- Verify the second branch, for example:

$$
\begin{array}{l}
(r \times x^y = A^B)[r \times x, y - 1/r, y] \\
\equiv\ (r \times x) \times x^{y-1} = A^B \\
\equiv\ r \times (x \times x^{y-1}) = A^B \\
\Leftarrow\ r \times x^y = A^B \wedge y < 0.
\end{array}
$$

Specification

$$
\begin{array}{l}
|[\ \textbf{con}\ N : int; \{0 \leq N\}\ \ f : \textbf{array}\ [0..N)\ \textbf{of}\ int; \\
\quad \textbf{var}\ r \quad : int; \\
\\
\quad \{r = (\uparrow p, q : 0 \leq p \leq q \leq N : sum\ p\ q)\} \\
]|
\end{array}
$$

- $sum\ p\ q = \Sigma i : p \leq i < q : f\ i$.

Specification

$$
\begin{array}{l}
|[\ \mathbf{con}\ N : int; \{0 \le N\}\ \ f : \mathbf{array}\ [0..N)\ \mathbf{of}\ int; \\
\quad \mathbf{var}\ r, n : int; \\
\\
\quad n, r := 0, 0; \\
\quad \{r = (\uparrow p, q : 0{\le}p{\le}q{\le}n : sum\ p\ q) \wedge 0 \le n \le N\} \\
\quad \mathbf{do}\ n \ne N \rightarrow \\
\qquad\qquad \ldots; n := n + 1 \\
\quad \mathbf{od} \\
\quad \{r = (\uparrow p, q : 0{\le}p{\le}q{\le}N : sum\ p\ q)\} \\
]|
\end{array}
$$

- $sum\ p\ q = \Sigma i : p{\le}i{<}q : f\ i$.
- Replacing constant N by variable n, use an up-loop.

Strengthening the Invariant

- Let $P_0 \equiv r = (\uparrow p, q : 0 \leq p \leq q \leq n : sum\ p\ q)$.

$$
\begin{array}{l}
n, r \quad := 0, 0 \quad ; \\
\{P_0 \land 0 \leq n \leq N \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \} \\
\mathbf{do}\ n \neq N \rightarrow \\
\qquad \ldots ; n := n + 1 \\
\mathbf{od} \\
\{r = (\uparrow p, q : 0 \leq p \leq q \leq N : sum\ p\ q)\}
\end{array}
$$

$$
\begin{array}{ll}
& (r = (\uparrow p, q : 0 \leq p \leq q \leq n : sum\ p\ q) \land 0 \leq n \leq N)[n+1/n] \\
\equiv & r = (\uparrow p, q : 0 \leq p \leq q \leq n+1 : sum\ p\ q) \land 0 \leq n+1 \leq N \\
\equiv & r = (\uparrow p, q : 0 \leq p \leq q \leq n : sum\ p\ q) \ \uparrow \\
& \quad (\uparrow p, q : 0 \leq p \leq n+1 : sum\ p\ (n+1)) \\
& \land\ 0 \leq n+1 \leq N
\end{array}
$$

Strengthening the Invariant

- Let $P_0 \equiv r = (\uparrow p, q : 0{\leq}p{\leq}q{\leq}n : sum\, p\, q)$.

$$
\begin{array}{l}
n, r, s := 0, 0, 0; \\
\{P_0 \land 0 \leq n \leq N \land s = (\uparrow p : 0{\leq}p{\leq}n : sum\, p\, n)\} \\
\mathbf{do}\ n \neq N \rightarrow \\
\qquad \ldots; n := n + 1 \\
\mathbf{od} \\
\{r = (\uparrow p, q : 0{\leq}p{\leq}q{\leq}N : sum\, p\, q)\}
\end{array}
$$

$$
\begin{array}{ll}
& (r = (\uparrow p, q : 0{\leq}p{\leq}q{\leq}n : sum\, p\, q) \land 0{\leq}n{\leq}N)[n + 1/n] \\
\equiv & r = (\uparrow p, q : 0{\leq}p{\leq}q{\leq}n + 1 : sum\, p\, q) \land 0{\leq}n + 1{\leq}N
\end{array}
$$

- $$
\begin{array}{ll}
\equiv & r = (\uparrow p, q : 0{\leq}p{\leq}q{\leq}n : sum\, p\, q)\ \uparrow \\
& \quad (\uparrow p, q : 0{\leq}p{\leq}n{+}1 : sum\, p\, (n{+}1)) \\
& \land\ 0{\leq}n + 1{\leq}N
\end{array}
$$

- Let's introduce $P_1 \equiv s = (\uparrow p : 0{\leq}p{\leq}n : sum\, p\, n)$.

Constructing the Loop Body

- Known: $P_0 \equiv r = (\uparrow p, q : 0 \le p \le q \le n : sum\ p\ q)$,
- $P_1 \equiv s = (\uparrow p : 0 \le p \le n : sum\ p\ n)$,
- $P_0[n+1/n] \equiv r = (\uparrow p, q : 0 \le p \le q \le n : sum\ p\ q) \uparrow (\uparrow p : 0 \le p \le n+1 : sum\ p\ (n+1))$.
- Therefore, a possible strategy would be:

$$
\begin{array}{l}
\{P_0 \wedge P_1 \ldots\} \\
s := ?; \\
\{P_0 \wedge P_1[n+1/n] \ldots\} \\
r := r \uparrow s; \\
\{P_0[n+1/n] \wedge P_1[n+1/n] \ldots\} \\
n := n + 1 \\
\{P_0 \wedge P_1 \cdots\}
\end{array}
$$

Updating the Prefix Sum

Recall $P_1 \equiv s = (\uparrow p : 0 \leq p \leq n : sum\; p\; n)$.

$$
\begin{array}{ll}
 & (\uparrow p : 0 \leq p \leq n : sum\; p\; n)[n+1/n] \\
= & \uparrow p : 0 \leq p \leq n{+}1 : sum\; p\; (n{+}1) \\
= & (\uparrow p : 0 \leq p \leq n : sum\; p\; (n{+}1)) \uparrow sum\; (n{+}1)\; (n{+}1) \\
= & (\uparrow p : 0 \leq p \leq n : sum\; p\; (n{+}1)) \uparrow 0 \\
= & (\uparrow p : 0 \leq p \leq n : (sum\; p\; n + f\; n)) \uparrow 0 \\
= & ((\uparrow p : 0 \leq p \leq n : sum\; p\; n) + f\; n) \uparrow 0
\end{array}
$$

Thus, $\{P_1\} s := ?\{P_1[n+1/n]\}$ is satisfied by $s := (s + f\; n) \uparrow 0$.

Derived Program

$$
\begin{array}{l}
|[\ \textbf{con}\ N : int; \{0 \leq N\}\ \ f : \textbf{array}\ [0..N)\ \textbf{of}\ int; \\
\quad \textbf{var}\ r, s, n : int; \\
\\
\quad n, r, s := 0, 0, 0; \\
\quad \{P_0 \wedge P_1 \wedge 0 \leq n \leq N, bnd : N - n\} \\
\quad \textbf{do}\ n \neq N \rightarrow \\
\qquad\qquad s := (s + f\ n) \uparrow 0; \\
\qquad\qquad r := r \uparrow s; \\
\qquad\qquad n := n + 1 \\
\quad \textbf{od} \\
\quad \{r = (\uparrow 0 \leq p \leq q \leq N : s : um\ p\ q)\} \\
]|
\end{array}
$$

- $P_0 \equiv r = (\uparrow 0 \leq p \leq q \leq n : s : um\ p\ q)$.
- $P_1 \equiv s = (\uparrow 0 \leq p \leq n : s : um\ p\ n)$.

What have we learned?

- Procedural program derivation by backwards reasoning.
- Key to procedural program derivation: every loop shall be built with an invariant and a bound in mind.
- Some techniques to construct loop invariants:
  - taking conjuncts as invariants;
  - replacing constants by variables;
  - strengthening the invariant;
  - tail invariants.
- Some of them are closely related to techniques we introduced in Day 1 and Day 2, e.g. tupling and accumulating parameters.

What's Missing?

- Side-effects strictly forbidden in expressions.
- That means *aliasing* could cause disasters,
- which in turn makes call-by-reference dangerous.
  - Extra care must be taken when we introduce subroutines.
- And, no pointers. Which means that we have problem talking about complex data structures.
  - In contrast, functional program derivation is essentially built on a theory of data structure.
  - Rescue: separation logic, to talk about when data structure is shared.

Where to Go from Here?

- Early issues of Science of Computer Programming have regular columns for program derivation.
- Books and papers by Dijkstra, Gries, Back, Backhouse, etc.
- You might not actually derive programs, but knowledge learnt here can be applied to program verification.
  - Plenty of tools around for program verification basing on pre/post-conditions. Some of them will be taught in the next summer school.
- You might never derive any more programs for the rest of your life. But the next time you need a loop, you will know better how to construct it and why it works.