An Introduction to Functional Program Derivation

Shin-Cheng Mu

2007 Formosan Summer School
on Logic, Language, and Computation
July 2–13, 2007

Part I
The Expand/Reduce Transformation

So I Was Asked...

- "So, you write programs, right? Then what happens?"
- 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?"

1 Prelude

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.
-

-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	$up \uparrow right$

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: even “simple” programs are not 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 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!

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

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

2 Preliminaries

2.1 Functions

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. $square\,3$.
- Given $f :: \alpha \to \beta$ and $g :: \beta \to \gamma$, their composition $g \cdot f :: \alpha \to \gamma$ is defined by $(g \cdot f)\,a = g\,(f\,a)$.

Recursively Defined Functions

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

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

  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:

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

2.2 Data Structures

Natural Numbers and Lists

- Natural numbers: $N = 0 \mid 1 + N$.
  - 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:

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

Some Notes on Notations

- In this lecture we use a Haskell-like notation. In OCaml, the function $fact$ is defined as:

```
let rec fact = function
    | 0 -> 1
    | n -> n * fact(n - 1);;
```

- The two types for trees would be defined as:

```
type 'a iTree =
    Null | Node of 'a * 'a iTree * 'a iTree
type 'a eTree =
    Tip of 'a | Bin of 'a eTree * 'a eTree
```

- Lists are denoted by `1::(2::(3::[])) = [1;2;3]`.

3 The Expand/Reduce Transformation

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.

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

3.1 Example: Sum of Squares

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:

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

  We have therefore constructed a recursive definition of $sumsq$:

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

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.

3.2 Proof by Induction

Proving Auxiliary Properties

- Our pattern of program derivation:

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

- 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 $x\colon xs$ as well.

Appending Two Lists

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

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

- E.g.

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

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

Sum Distributes into Append

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

$$
\begin{array}{ll}
 & sum\,[\,] + sum\,ys \\
= & \quad \{ \text{ Definition of } sum \ \} \\
 & 0 + sum\,ys \\
= & \quad \{ \text{ Arithmetic } \} \\
 & sum\,ys \\
= & \quad \{ \text{ By definition of } (\mathbin{+\!\!+}),\ [\,] \mathbin{+\!\!+} ys = ys \ \} \\
 & sum\,([\,] \mathbin{+\!\!+} ys)
\end{array}
$$

Sum Distributes into Append, the Inductive Case

Case $x : xs$:

$$
\begin{array}{ll}
 & sum\,(x : xs) + sum\,ys \\
= & \quad \{ \text{ Definition of } sum \} \\
 & (x + sum\,xs) + sum\,ys \\
= & \quad \{ \ (+) \text{ is associative: } (a+b)+c = a+(b+c) \ \} \\
 & x + (sum\,xs + sum\,ys) \\
= & \quad \{ \text{ Induction Hypothesis } \} \\
 & x + sum(xs \mathbin{+\!\!+} ys) \\
= & \quad \{ \text{ Definition of } sum \ \} \\
 & sum(x : (xs \mathbin{+\!\!+} ys)) \\
= & \quad \{ \text{ Definition of } (\mathbin{+\!\!+}) \ \} \\
 & sum((x : xs) \mathbin{+\!\!+} ys)
\end{array}
$$

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:

  $$(xs \mathbin{+\!\!+} ys) \mathbin{+\!\!+} zs \quad = \quad xs \mathbin{+\!\!+} (ys \mathbin{+\!\!+} zs).$$

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

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

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

  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:

$$data\, iTree\, \alpha = Null \mid Node\, \alpha\, (iTree\, \alpha)\, (iTree\, \alpha)$$

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 $Node\, a\, t\, u$.

3.3 Accumulating Parameter

Example: Reversing a List

- The function $reverse$ is defined by:

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

  E.g. $reverse\,[1,2,3,4] = ((([\,] +\!\!+ [4]) +\!\!+ [3]) +\!\!+ [2]) +\!\!+ [1] = [4,3,2,1]$.

- But how about its time complexity? Since $(+\!\!+)$ 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:

$$rcat\, xs\, ys \quad = \quad reverse\, xs +\!\!+ ys.$$

- 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 $[\,]$:

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

Reversing a List, Inductive Case

Case $x: xs$:

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

Linear-Time List Reversal

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

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

  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

$$
\begin{array}{ll}
& reverse\,[1,2,3,4] \\
= & rcat\,[1,2,3,4]\,[\,] \\
= & rcat\,[2,3,4]\,[1] \\
= & rcat\,[3,4]\,[2,1] \\
= & rcat\,[4]\,[3,2,1] \\
= & rcat\,[\,]\,[4,3,2,1] \\
= & [4,3,2,1]
\end{array}
$$

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

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

Tail Recursion

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

$$\begin{aligned} f\,x_1\,\ldots\,x_n &= \{\text{base case}\} \\ f\,x_1\,\ldots\,x_n &= f\,x_1'\,\ldots\,x_n' \end{aligned}$$

- 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:

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

Functional: We initialise $rcat$ by:

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

and try to derive a faster version of $rcat$ satisfying (1).

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

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

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

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 $ssp\,xs\,n = sumsq\,xs + n$.
- Initialisation: $sumsq\,xs = ssp\,xs\,0$.
- Construct ssp:

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

Notes on Compatibility with OCaml

Some of the functions we've mentioned, or will mention, have their equivalents defined in `module List`:

```
val hd : 'a list -> 'a
val tl : 'a list -> 'a list
val length : 'a list -> int
val append : 'a list -> 'a list -> 'a list
val concat : 'a list list -> 'a list
val map : ('a -> 'b) -> 'a list -> 'b list
```

3.4 Tupling

Steep Lists

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

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

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

$$steepsum\,xs \quad = \quad (steep\,xs, sum\,xs),$$

  and manage to synthesise a quick definition of $steepsum$, we can implement $steep$ by $steep = fst \cdot steepsum$.

- We again proceed by case analysis. Trivially,

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

Deriving for the Non-Empty Case

For the case for non-empty inputs.

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

Synthesised Program

- We have thus come up with:

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

  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.
- Now that we are talking about inductive proofs again, 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.

4 Folds

A Common Pattern We've Seen Many Times…

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

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

Replacing Constructors

- $$\begin{array}{lcl} foldr\,f\,e\,[\,] &=& e \\ foldr\,f\,e\,(x:\,xs) &=& f\,x\,(foldr\,f\,e\,xs) \end{array}$$
- One way to look at $foldr\,(\oplus)\,e$ is that it replaces $[\,]$ with e and $(:)$ with $(\oplus)$.

$$\begin{array}{cl} & foldr\,(\oplus)\,e\,[1,2,3,4] \\ = & foldr\,(\oplus)\,e\,(1:(2:(3:(4:[\,])))) \\ = & 1 \oplus (2 \oplus (3 \oplus (4 \oplus e))) \end{array}$$

- $sum = foldr\,(+)\,0$
- $length = foldr\,(\lambda x\,n.1 + n)\,0$
- $map\,f = foldr\,(\lambda x\,xs.f\,x:\,xs)\,[\,]$
- One can see that $id = foldr\,(:)\,[\,]$.

Notes on Notation

- Both $f\,x\,y$ and $x \oplus y$ denote a function applied to x and y successively. We use the prefix and infix notation alternatively whenever appropriate.
- The notation $\lambda x.expr$ denotes an anonymous function. In OCaml it may be written `fun x -> expr`.

Notes on Compatibility with OCaml

In `module List` there is a function `fold_right`, but the order of arguments is different. Our *foldr* can be defined by:

```
let rec foldr f a lst = match lst with
  | [] -> a
  | x::xs -> f x (foldr f a xs);;
```

Some example usage:

```
  let sum = foldr (fun x y -> x + y) 0;;
  let len = foldr (fun x y -> 1 + y) 0;;
  let map f = foldr (fun x lst -> (f x)::lst) [];;
```

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:
  - $$\begin{aligned} id\,[\,] &= [\,], \\ id\,(x:xs) &= x : id\,xs. \end{aligned}$$
  - $id = foldr\,(:)\,[\,]$.

4.1 The Fold-Fusion Theorem

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 1 (Fold-Fusion)**.** Given $f :: \alpha \to \beta \to \beta$, $e :: \beta$, $h :: \beta \to \gamma$, and $g :: \alpha \to \gamma \to \gamma$, 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.

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

Sum of Squares, Again

- Consider $\mathit{sum} \cdot \mathit{map}\,\mathit{square}$ again. This time we use the fact that $\mathit{map}\,f = \mathit{foldr}\,(\mathit{mf}\,f)\,[\,]$, where $\mathit{mf}\,f\,x\,\mathit{xs} = f\,x : \mathit{xs}$.

- $\mathit{sum} \cdot \mathit{map}\,\mathit{square}$ is a fold, if we can find a ssq such that $\mathit{sum}\,(\mathit{mf}\,\mathit{square}\,x\,\mathit{xs}) = \mathit{ssq}\,x\,(\mathit{sum}\,\mathit{xs})$. Let us try:

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

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

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:

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

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

4.2 More Useful Functions Defined as Folds

Longest Prefix

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

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

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

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

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

All Prefixes

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

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

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

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

All Suffixes

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

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

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

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

Scan

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

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

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

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

4.3 Finally, Solving Maximum Segment Sum

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.

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

- Therefore, mss is specified by:

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

The Derivation!

We reason:

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

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

Maximum Prefix Sum

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

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

The fold fusion works because:

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

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

Maximum Prefix Sum, 2nd Fold Fusion

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

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

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

$$
\begin{array}{ll}
 & max\,(0 : map\,(x+)\,xs) \\
= & 0 \uparrow max\,(map\,(x+)\,xs) \\
= & 0 \uparrow (x + max\,xs)
\end{array}
$$

Back to Maximum Segment Sum

We reason:

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

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 $step\, x\, (m,y) = ((0 \uparrow (x+y)) \uparrow m, 0 \uparrow (x+y))$.

4.4 Folds on Trees

Folds on Trees

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

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

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

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

- The fold for $eTree$, is given by:

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

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 $eTree$:

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

5 Unfolds

Unfolds Generate Data Structures

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

$$\begin{array}{lcl} \mathit{unfoldr}\, p\, f\, s & = & \textbf{if}\, p\, s\, \textbf{then}\, [\,]\, \textbf{else} \\ & & \;\textbf{let}\, (x, s') = f\, s\, \textbf{in}\, x : \mathit{unfoldr}\, p\, f\, s'. \end{array}$$

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

5.1 Unfold on Lists

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 $\mathit{fromto}\, m\, n$ 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 $\mathit{null}\, \mathit{xs}$ yields true iff $\mathit{xs} = [\,]$.

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 $(\mathit{xs}, \mathit{ys})$, the call $\mathit{merge}\, (\mathit{xs}, \mathit{ys})$ merges them into one sorted list:

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

5.2 Folds v.s. Unfolds

Folds and Unfolds

- Folds and unfolds are dual concepts. Folds consume data structure, while unfolds build data structures.
- List constructors have types: $(:) :: \alpha \to [\alpha] \to [\alpha]$ and $[] :: [\alpha]$; in $\mathit{fold}\,f\,e$, the arguments have types: $f :: \alpha \to \beta \to \beta$ and $e :: \beta$.
- List deconstructors have types: $\langle hd, tl \rangle :: [\alpha] \to (\alpha, [\alpha])$; in $\mathit{unfoldr}\,p\,f$, the argument f has type $\beta \to (\alpha, \beta)$.
- 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 $\mathit{map}\,f \cdot \mathit{iterate}\,f = \mathit{iterate}\,f\,(f\,x)$, we show that for all possible *observations*, the *lhs* equals the *rhs*.

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

$$\begin{array}{ll} & tl\,(\mathit{map}\,f\,(\mathit{iterate}\,f\,x)) \\ = & tl\,(f\,x : \mathit{map}\,f\,(\mathit{iterate}\,f\,(f\,x))) \\ = & \quad \{\mathit{hypothesis}\} \\ & tl\,(f\,x : \mathit{iterate}\,f\,(f\,(f\,x))) \\ = & tl\,(\mathit{iterate}\,f\,(f\,x)) \end{array}$$

  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:

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

And for externally labelled binary trees we define:

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

6 Hylomorphism

Unflattening a Tree

- Recall the function $\mathit{flatteneT} :: \mathit{eTree}\,\alpha \to [\alpha]$, defined as a fold, flattening a tree into a list. Let us consider doing the reverse.
- Assume that we have the following functions:
  - $\mathit{single}\,xs = \mathit{true}$ iff xs contains only one element.
  - $\mathit{half} :: [\alpha] \to ([\alpha],[\alpha])$ 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:

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

6.1 A Museum of Sorting Algorithms

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?

$$\mathit{msort} \quad = \quad \mathit{foldeT}\,\mathit{merge}_c\,\mathit{id} \cdot \mathit{unflatteneT}$$

- This is mergesort!

Quicksort as a Hylomorphism

- Let $\mathit{partition}$ be defined by:

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

- Recall the function $\mathit{flatteniT}$ 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 symmetricity:

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

  Both are defined as a fold after an unfold.

Insertion Sort and Selection Sort

- Insertion sort can be defined by an fold:

$$isort = foldr\ insert\ [\,],$$

where $insert$ is specified by

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

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

$$ssort = unfoldr\ null\ select,$$

where $select$ is specified by

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

6.2 Hylomorphism and Recursion

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

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

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 $fold \cdot unfold$. 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 $fold \cdot unfold$,

- 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!

7 Wrapping Up

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!

References

The following list of references is certainly not complete, but may serve as a starting point if you are interested in related topics.

Functional and Relational Program Derivation

Backhouse, R. C., 2002. Galois connections and fixed point calculus. In: Backhouse, R. C., Crole, R., Gibbons, J. (Eds.), Algebraic and Coalgebraic Methods in the Mathematics of Program Construction. No. 2297 in Lecture Notes in Computer Science. Springer-Verlag, pp. 89–148.

Backhouse, R. C., Crole, R., Gibbons, J. (Eds.), 2002. Algebraic and Coalgebraic Methods in the Mathematics of Program Construction. Springer-Verlag.

Backhouse, R. C., de Bruin, P., Malcolm, G., Voermans, E., van der Woude, J., 1991. Relational catamorphisms. In: Möller, B. (Ed.), Proceedings of the IFIP TC2/WG2.1 Working Conference on Constructing Programs. Elsevier Science Publishers, pp. 287–318.

Bird, R. S., 1989. Lectures on constructive functional programming. In: Broy, M. (Ed.), Constructive Methods in Computing Science. No. 55 in NATO ASI Series F. Springer-Verlag, pp. 151–216.

Bird, R. S., 1996. Functional algorithm design. Science of Computer Programming 26, 15–31.

Bird, R. S., 1998. Introduction to Functional Programming using Haskell. Prentice Hall.

Bird, R. S., de Moor, O., 1997. Algebra of Programming. International Series in Computer Science. Prentice Hall.

Bird, R. S., Gibbons, J., Mu, S.-C., January 2002. Algebraic methods for optimization problems. In: Backhouse, R. C., Crole, R., Gibbons, J. (Eds.), Algebraic and Coalgebraic Methods in the Mathematics of Program Construction. No. 2297 in Lecture Notes in Computer Science. Springer-Verlag, pp. 281–307.

Burstall, R. M., Darlington, J., 1977. A transformational system for developing recursive programs. Journal of the ACM 24 (1), 44–67.

Chin, W.-N., Hu, Z., 2002. Towards a modular program derivation via fusion and tupling. In: The First ACM SIGPLAN Conference on Generators and Components. No. 2487 in Lecture Notes in Computer Science. pp. 140–155.

Gibbons, J., November 1994. An introduction to the Bird-Meertens formalism. New Zealand Formal Program Development Colloquium Seminar, Hamilton.

Gibbons, J., 2002. Calculating functional programs. In: Algebraic and Coalgebraic Methods in the Mathematics of Program Construction. No. 2297 in Lecture Notes in Computer Science. Springer Berlin / Heidelberg, pp. 201–239.

Gibbons, J., Hutton, G., Altenkirch, T., April 2001. When is a function a fold or an unfold? In: Corradini, A., Lenisa, M., Montanari, U. (Eds.), Proceedings of the 4th International Workshop on Coalgebraic Methods in Computer Scienc. No. 44.1 in Electronic Notes in Theoretical Computer Science. Elsevier Science Publishers.

Hu, Z., Iwasaki, H., Takeichi, M., September 1996. Construction of list homomorphisms via tupling and fusion. In: 21st International Symposium on Mathematical Foundation of Computer Science. No. 1113 in Lecture Notes in Computer Science. Springer-Verlag, Cracow, pp. 407–418.

Hu, Z., Iwasaki, H., Takeichi, M., 1996. Deriving structural hylomorphisms from recursive definitions. In: Proceedings of the 1996 ACM SIGPLAN International Conference on Functional Programming. pp. 73–82.

Interesting Cases of Program Derivation

Bird, R. S., 1997. On building trees with minimum height. Journal of Functional Programming 7 (4), 441–445.

Bird, R. S., July 2001. Maximum marking problems. Journal of Functional Programming 11 (4), 411–424, journal of Functional Programming.

Bird, R. S., Gibbons, J., 2003. Arithmetic coding with folds and unfolds. In: Jeuring, J. T., Peyton Jones, S. (Eds.), Advanced Functional Programming (AFP 2002). No. 2638 in Lecture Notes in Computer Science. Springer-Verlag, pp. 1–26.

Bird, R. S., Gibbons, J., Jones, G., September 1999. Program optimisation, naturally. Symposium in Celebration of the work of C.A.R. Hoare.

Bird, R. S., Mu, S.-C., 2004. Inverting the Burrows-Wheeler transform. Journal of Functional Programming 14 (6), 603–612.

de Moor, O., Gibbons, J., September 1999. Bridging the algorithm gap: A linear-time functional program for paragraph formatting. Science of Computer Programming 35 (1), revised version of Technical Report CMS-TR-97-03, School of Computing and Mathematical Sciences, Oxford Brookes University.

Gibbons, J., May 1999. A pointless derivation of radixsort. Journal of Functional Programming 9 (3), 339–346.

Gibbons, J., Jones, G., 1993. Linear-time breadth-first tree algorithms: an exercise in the arithmetic of folds and zips. Tech. rep., University of Auckland, university of Auckland Computer Science Report No. 71, and IFIP Working Group 2.1 working paper 705 WIN-2.

Gries, D., 1989. The maximum-segment-sum problem. In: Dijkstra, E. W. (Ed.), Formal Development Programs and Proofs. University of Texas at Austin Year of Programming Series. Addison-Wesley, pp. 33–36.

Gries, D., van de Snepscheut, J. L., 1990. Inorder traversal of a binary tree and its inversion. In: Dijkstra, E. W. (Ed.), Formal Development of Programs and Proofs. Addison Wesley, pp. 37–42.

Hinze, R., May 2000. Perfect trees and bit-reversal permutations. Journal of Functional Programming 10 (3), 305–317.

Hinze, R., July 2002. Constructing tournament representations: an exercise in pointwise relational programming. In: Boiten, E., Möller, B. (Eds.), Sixth International Conference on Mathematics of Program Construction. No. 2386 in Lecture Notes in Computer Science. Springer-Verlag, Dagstuhl, Germany, pp. 131–147.

Hinze, R., 2004. An algebra of scans. In: Seventh International Conference on Mathematics of Program Construction. No. 3125 in Lecture Notes in Computer Science. pp. 186–210.

Hinze, R., January 2005. Church numerals, twice! Journal of Functional Programming 15 (1), 1–13.

Hutton, G., 2002. The countdown problem. Journal of Functional Programming 12 (6), 609–616.

Jeuring, J. T., 1991. Incremental algorithms on lists. In: van Leeuwen, J. (Ed.), Proceedings SION Computing Science in the Netherlands. pp. 315–335.

Jeuring, J. T., 1994. The derivation of on-line algorithms, with an application to finding palindromes. Algorithmica 11, 146–184.

Knapen, E., 23 November 1993. Relational programming, program inversion, and the derivation of parsing algorithms. Master's thesis, Eindhoven University of Technology.

Mu, S.-C., Bird, R. S., 2003. Rebuilding a tree from its traversals: a case study of program inversion. In: Ohori, A. (Ed.), Programming Languages and Systems. Proceedings. No. 2895 in Lecture Notes in Computer Science. Springer-Verlag, pp. 265–282.

Mu, S.-C., Bird, R. S., 2003. Theory and applications of inverting functions as folds. Science of Computer Programming (Special Issue for Mathematics of Program Construction) 51, 87–116.

Sasano, I., Hu, Z., Takeichi, M., Ogawa, M., September 2000. Make it practical: a generic linear-time algorithm for solving maximum-weightsum problems. In: Proceedings of the 2000 ACM SIGPLAN International Conference on Functional Programming. ACM Press, pp. 137–149.

Zantema, H., 1992. Longest segment problems. Science of Computer Programming 18 (1), 39–66.

Fold and Fold Fusion

Gibbons, J., Hutton, G., Altenkirch, T., April 2001. When is a function a fold or an unfold? In: Corradini, A., Lenisa, M., Montanari, U. (Eds.), Proceedings of the 4th International Workshop on Coalgebraic Methods in Computer Scienc. No. 44.1 in Electronic Notes in Theoretical Computer Science. Elsevier Science Publishers.

Hutton, G., July 1999. A tutorial on the universality and expressiveness of fold. Journal of Functional Programming 9 (4), 355–372.

Meijer, E., Fokkinga, M., Paterson, R., 1991. Functional programming with bananas, lenses, envelopes, and barbed wire. In: Proceedings of the 5th ACM conference on Functional programming languages and computer architecture. pp. 124–144.

Meijer, E., Hutton, G., 1995. Bananas in space: extending fold and unfold to exponential types. In: Proceedings of the Seventh International Conference on Functional Programming Languages and Computer Architecture. pp. 324–333.

Meijer, E., Jeuring, J. T., 1995. Merging monads and folds for functional programming. In: Advanced Functional Programming (AFP 1995). No. 925 in Lecture Notes in Computer Science. Springer-Verlag, pp. 228–266.

Takano, A., Meijer, E., 1995. Shortcut deforestation in calculational form. In: Proceedings of the Seventh International Conference on Functional Programming Languages and Computer Architecture. pp. 306–313.

Unfold and Coinduction

Gibbons, J., Hutton, G., April-May 2005. Proof methods for corecursive programs. Fundamenta Informaticae 66 (4), 353–366.

Gibbons, J., Jones, G., 1998. The under-appreciated unfold. In: Proceedings of the third ACM SIGPLAN international conference on Functional programming SIGPLAN international conference on Functional programming. ACM Press, pp. 273–279.

Gordon, A., 1995. A tutorial on co-induction and functional programming. In: Functional Programming, Glasgow 1994. pp. 78–95.

Jacobs, B., Rutten, J., 1997. A tutorial on (co)algebras and (co)induction. Bulletin of the European Association for Theoretical Computer Science 62, 222–259.

Procedural Program Derivation

Back, R. J. R., 1988. A calculus of refinements for program derivations. Acta Informatica 25, 593–624.

Back, R. J. R., von Wright, J., 1992. Combining angels, demons and miracles in program specifications. Theoretical Computer Science 100, 365–383.

Dijkstra, E. W., 1976. A Discipline of Programming. Prentice Hall.

Kaldewaij, A., 1990. Programming: the Derivation of Algorithms. Prentice Hall.