Constructing tournament representations: An exercise in pointwise relational programming

RALF HINZE

Institut für Informatik III, Universität Bonn
Römerstraße 164, 53117 Bonn, Germany
Email: ralf@informatik.uni-bonn.de
Homepage: http://www.informatik.uni-bonn.de/~ralf

July, 2002

(Pick the slides at .../~ralf/talks.html#T32.)

Prologue

We all know and love functional programming.

☞ A functional language can be used for proving properties of programs or for calculating programs (fold-unfold transformations).

Prologue

However, there is often a need to go beyond the world of functions:

- nondeterministic problems are most easily specified in terms of *relations*;
- even deterministic problems that enjoy deterministic solutions may benefit from a relational setting.

☞ Problems about functions of real variables are sometimes solved more easily in the complex plane.

A problem: tournament representations

Here is the problem:

> Given a sequence x of integers, construct a heap whose inorder traversal is x itself.

This heap is a so-called *tournament representation* of x.

NB. The tournament representation is unique iff the given integers are distinct.

A problem: tournament representations

$$\textbf{data}\ \mathit{Tree}\ a\ =\ E \mid N\ (\mathit{Tree}\ a)\ a\ (\mathit{Tree}\ a)$$

$$\begin{array}{lcl} \mathit{heap} & :: & \mathit{Tree}\ \mathit{Int} \to \mathit{Bool} \\ \mathit{heap}\ E & = & \mathit{True} \\ \mathit{heap}\ (N\ l\ a\ r) & = & \mathit{heap}\ l\ \wedge\ \mathit{top}\ l \geqslant a \leqslant \mathit{top}\ r\ \wedge\ \mathit{heap}\ r \end{array}$$

$$\begin{array}{lcl} \mathit{top} & :: & \mathit{Tree}\ \mathit{Int} \to \mathit{Int} \\ \mathit{top}\ E & = & \infty \\ \mathit{top}\ (N\ l\ a\ r) & = & a \end{array}$$

$$\begin{array}{lcl} \mathit{list} & :: & \forall a\,.\,\mathit{Tree}\ a \to [a] \\ \mathit{list}\ E & = & [\,] \\ \mathit{list}\ (N\ l\ a\ r) & = & \mathit{list}\ l +\!\!+ [a] +\!\!+ \mathit{list}\ r \end{array}$$

Skipping ahead: a Haskell solution

Our goal is to derive the following deterministic Haskell solution.

$$
\begin{array}{lcl}
\mathit{tournament} & :: & [\mathit{Int}] \to \mathit{Tree}\ \mathit{Int} \\
\mathit{tournament}\ x & = & \mathbf{let}\ (t, y) = \mathit{loop\text{-}to}'\ (-\infty)\ E\ x\ \mathbf{in}\ t \\
\mathit{loop\text{-}to}'\ p\ (l, [\,]) & = & (l, [\,]) \\
\mathit{loop\text{-}to}'\ p\ (l, a : x) & & \\
\quad \mid p \geqslant a & = & (l, a : x) \\
\quad \mid \mathit{otherwise} & = & \mathbf{let}\ (r, y) = \mathit{loop\text{-}to}'\ a\ (E, x)\ \mathbf{in}\ \mathit{loop\text{-}to}'\ p\ (N\ l\ a\ r, y)
\end{array}
$$

Relational calculus

The artist's approach to program derivation: switch to the world of relations.

Specification: we seek a function $tournament :: [Int] \rightarrow Tree\ Int$ such that

$$tournament \subseteq heap? \cdot list^{\circ}.$$

Here, $heap\,? :: Tree\ Int \rightarrow Tree\ Int$ is a subset of identity and $list^{\circ}$ is the converse of $list$.

Characteristics: point-free reasoning, derivations employ universal properties of operators.

Relational calculus

> [...] the calculus of relations has gained a good deal of notoriety for the apparently enormous number of operators and laws that one has to memorise in order to do proofs effectively.

[Richard Bird and Oege de Moor, *Algebra of Programming*, Prentice Hall Europe, London, 1997].

Pointwise versus point-free

Pointwise:

$$(x \mathbin{+\!\!+} y) \mathbin{+\!\!+} z \;=\; x \mathbin{+\!\!+} (y \mathbin{+\!\!+} z).$$

Point-free:

$$cat \cdot (cat * id) \;=\; cat \cdot (id * cat) \cdot assocr.$$

[Oege de Moor and Jeremy Gibbons, *Pointwise relational programming*, AMAST 2000, LNCS 1816, May 2000].

Functional calculus and set comprehensions

The craftsman's approach to program derivation: stay in the world of functions and model relations by *set-valued functions*.

Specification: we seek a function $tournament :: [Int] \rightarrow Set\ (Tree\ Int)$ such that

$$t \leftarrow tournament\ x \ \equiv \ list\ t = x,\ heap\ t.$$

Here, $e_1 \leftarrow e_2$ is set comprehension notation (is drawn from).

Characteristics: pointwise reasoning; derivations are based on fold-unfold transformations.

Step 1: tupling

We generalize $tournament$ to a function $build :: [Int] \to Set\ (Tree\ Int, [Int])$ that satisfies

$$(t, y) \leftarrow build\ x \ \equiv \ list\ t +\!\!+ y = x,\ heap\ t.$$

Step 1: tupling

The derivation of $build$ proceeds almost mechanically.

$$
\begin{array}{ll}
 & (t, y) \leftarrow build\ x \\
\equiv & \quad \{\ \text{specification of } build\ \} \\
 & list\ t +\!\!+ y = x,\ heap\ t \\
\equiv & \quad \{\ t \text{ has type } Tree\ Int\ \} \\
 & (list\ E +\!\!+ y = x,\ heap\ E,\ t = E) \\
 & \qquad \vee\ (list\ (N\ l\ a\ r) +\!\!+ y = x,\ heap\ (N\ l\ a\ r),\ t = N\ l\ a\ r)
\end{array}
$$

We conduct two subproofs.

Step 1: tupling

Case $t = E$:

$$
\begin{array}{ll}
 & list\ E +\!\!+ y = x,\ heap\ E \\
\equiv & \quad \{ \text{ definition of } list \text{ and } heap \ \} \\
 & [\,] +\!\!+ y = x \\
\equiv & \quad \{ \text{ definition of `}+\!\!+\text{'} \ \} \\
 & y = x.
\end{array}
$$

Step 1: tupling

Case $t = N\ l\ a\ r$:

$$
\begin{array}{ll}
 & list\ (N\ l\ a\ r) +\!\!+ y = x,\ heap\ (N\ l\ a\ r) \\
\equiv & \quad \{\ \text{definition of } list \text{ and } heap\ \} \\
 & list\ l +\!\!+ [a] +\!\!+ list\ r +\!\!+ y = x,\ heap\ l,\ top\ l \geqslant a \leqslant top\ r,\ heap\ r \\
\equiv & \quad \{\ \text{introduce } x_1 \text{ and rearrange}\ \} \\
 & list\ l +\!\!+ x_1 = x,\ heap\ l,\ [a] +\!\!+ list\ r +\!\!+ y = x_1,\ top\ l \geqslant a \leqslant top\ r,\ heap\ r \\
\equiv & \quad \{\ \text{specification of } build\ \} \\
 & (l, x_1) \leftarrow build\ x,\ [a] +\!\!+ list\ r +\!\!+ y = x_1,\ top\ l \geqslant a \leqslant top\ r,\ heap\ r \\
\equiv & \quad \{\ \text{introduce } x_2 \text{ and rearrange}\ \} \\
 & (l, x_1) \leftarrow build\ x,\ [a] +\!\!+ x_2 = x_1,\ list\ r +\!\!+ y = x_2,\ heap\ r,\ top\ l \geqslant a \leqslant top\ r \\
\equiv & \quad \{\ \text{specification of } build\ \} \\
 & (l, x_1) \leftarrow build\ x,\ [a] +\!\!+ x_2 = x_1,\ (r, y) \leftarrow build\ x_2,\ top\ l \geqslant a \leqslant top\ r \\
\equiv & \quad \{\ \text{definition of `}+\!\!+\text{'}\ \} \\
 & (l, x_1) \leftarrow build\ x,\ a : x_2 = x_1,\ (r, y) \leftarrow build\ x_2,\ top\ l \geqslant a \leqslant top\ r.
\end{array}
$$

Comprehension principle

We can turn an equivalence into an equation by applying the following *comprehension principle*.

$$(e_1 \leftarrow e_2 \equiv q) \;\equiv\; (e_2 = \{\, e_1 \mid q \,\})$$

Step 1—summary

We have shown that the specification satisfies the following equation.

$$
\begin{array}{lcl}
build & :: & [Int] \to Set\ (Tree\ Int, [Int]) \\
build\ x & = & \{(E, x)\} \\
 & \cup & \{(N\ l\ a\ r, y) \mid (l, x_1) \leftarrow build\ x, \\
 & & \qquad a : x_2 = x_1, \\
 & & \qquad (r, y) \leftarrow build\ x_2, \\
 & & \qquad top\ l \geqslant a \leqslant top\ r\}
\end{array}
$$

In fact, the specification is even the *unique* solution of the equation. We can reorder the derivation into an inductive proof showing that an arbitrary solution of the equation satisfies the specification.

NB. This is almost an executable Haskell program.

Set comprehensions

Set comprehensions can be given a precise semantics via the following identities:

$$
\begin{array}{lcl}
\{\, e \mid \epsilon \,\} & = & return\ e \\
\{\, e \mid b, q \,\} & = & \mathbf{if}\ b\ \mathbf{then}\ \{\, e \mid q \,\}\ \mathbf{else}\ \varnothing \\
\{\, e \mid p \leftarrow s, q \,\} & = & s \triangleright \lambda x \rightarrow \mathbf{case}\ x\ \mathbf{of}\ p \rightarrow \{\, e \mid q \,\}; \_ \rightarrow \varnothing,
\end{array}
$$

where $return$ and '$\triangleright$' are unit and bind of the set monad:

$$
\begin{array}{lcl}
return\ a & = & \{\, a \,\} \\
s \triangleright f & = & \bigcup \{\, f\ a \mid a \leftarrow s \,\}.
\end{array}
$$

Step 2: turning top-down into bottom-up

Observation: the construction of the left subtree is 'pure guesswork'.
Goal: eliminate the left-recursive call to $build$.

Using the above identities $build$ can be put into the form

$$build\ x\ =\ a\ x\ \cup\ build\ x \triangleright b,$$

for suitable functions a and b.

In a point-free style:

$$build\ =\ a\ \cup\ build \diamond b.$$

Here, $(f\ \cup\ g)\ n = f\ n\ \cup\ g\ n$ and $(f \diamond g)\ n = f\ n \triangleright g$.

Step 2: turning top-down into bottom-up

The equation $build = a \cup build \diamond b$ has the unique solution $a \diamond b^*$, where $(-)^*$ is the reflexive, transitive closure of a relation.

The closure operator $(-)^*$ can be defined either as the least fixed point of a left-recursive or of a right-recursive equation:

$$\begin{aligned} e^* &= return \cup e^* \diamond e \\ e^* &= return \cup e \diamond e^*. \end{aligned}$$

Consequently, an equivalent definition of $build$ is

$$\begin{aligned} build &= a \diamond loop \\ loop &= return \cup b \diamond loop. \end{aligned}$$

Step 2—summary

$$
\begin{array}{lcl}
build\ x & = & loop\ (E, x) \\
loop\ (l, x_1) & = & \{(l, x_1)\} \\
 & \cup & \{(t, z) \mid a : x_2 = x_1, \\
 & & \quad (r, y) \leftarrow loop\ (E, x_2), \\
 & & \quad top\ l \geqslant a \leqslant top\ r, \\
 & & \quad (t, z) \leftarrow loop\ (N\ l\ a\ r, y)\}
\end{array}
$$

NB. The transformation has turned a top-down program into a bottom-up one.

Step 3: promoting the tests

Goal: promote the tests $top\ l \geqslant a \leqslant top\ r$ into the generation of the trees.

We specify $loop\text{-}to$:

$$p \leqslant top\ l,\ (t, y) \leftarrow loop\text{-}to\ p\ (l, x) \equiv (t, y) \leftarrow loop\ (l, x),\ p \leqslant top\ t.$$

NB. The specified function $loop\text{-}to$ maintains an *invariant*.

Step 3—summary

$$
\begin{array}{lcl}
\mathit{build}\ x & = & \mathit{loop\text{-}to}\ (-\infty)\ (E, x) \\
\mathit{loop\text{-}to}\ p\ (l, x_1) & = & \{(l, x_1)\} \\
 & \cup & \{(t, z) \mid a : x_2 = x_1, \\
 & & \quad \mathit{top}\ l \geqslant a,\ p \leqslant a, \\
 & & \quad (r, y) \leftarrow \mathit{loop\text{-}to}\ a\ (E, x_2), \\
 & & \quad (t, z) \leftarrow \mathit{loop\text{-}to}\ p\ (N\ l\ a\ r, y)\}
\end{array}
$$

Step 4: strengthening

Observation: $build$ considers all prefixes of its argument.
Goal: calculate a variant of $loop\text{-}to$ which consumes as many elements as possible building maximal subtrees.

Assuming that $p \leqslant top\ l$ we specify $loop\text{-}to'$:

$$top\ l \geqslant hd\ x,\ (t, y) \leftarrow loop\text{-}to'\ p\ (l, x) \equiv (t, y) \leftarrow loop\text{-}to\ p\ (l, x),\ p \geqslant hd\ y.$$

Here, hd is given by:

$$\begin{array}{lcl} hd & :: & [Int] \to Int \\ hd\ [\,] & = & -\infty \\ hd\ (a : x) & = & a. \end{array}$$

Step 4—summary

$$
\begin{array}{lcl}
\mathit{tournament}\ x & = & \{\, t \mid (t, y) \leftarrow \mathit{loop\text{-}to}'\ (-\infty)\ (E, x)\,\} \\
\mathit{loop\text{-}to}'\ p\ (l, [\,]) & = & \{\, (l, [\,])\,\} \\
\mathit{loop\text{-}to}'\ p\ (l, a : x_2) & = & \{\, (l, a : x_2) \mid p \geqslant a\,\} \\
 & \cup & \{\, (t, z) \mid p \leqslant a, \\
 & & \qquad (r, y) \leftarrow \mathit{loop\text{-}to}'\ a\ (E, x_2), \\
 & & \qquad (t, z) \leftarrow \mathit{loop\text{-}to}'\ p\ (N\ l\ a\ r, y)\,\}
\end{array}
$$

NB. The derived program is still nondeterministic.
NB. The control structure is a combination of recursion and iteration.

Epilogue

- Set comprehensions provide a convenient language for specifying and deriving nondeterministic programs in a pointwise manner.
- They also allow for point-free arguments.
- The task of constructing tournament representations is closely related to precedence parsing.