Constructing tournament representations: An exercise in pointwise relational programming

Ralf Hinze

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

Abstract. List or set comprehensions are a wonderful means to define nondeterministic or relational programs. Despite their beauty, comprehensions are somewhat underused in program calculation. The purpose of this paper is to remind the program-calculation community that comprehensions provide a convenient language for specifying and deriving nondeterministic programs in a pointwise manner. We illustrate the style of reasoning by re-solving the well-known problem of constructing *tournament representations*: Given a sequence x of integers, construct a heap whose inorder traversal is x itself.

1 Introduction

One attractive feature of pure functional languages such as Haskell [19] is that they are close to the language of reasoning. The programmer can use the programming language she is familiar with also for proving properties of programs or for calculating programs. When deriving a program from a specification she can check some or all of the intermediate steps simply by executing them (at least in principle).

In program calculation, however, there is often a need to go beyond the world of functions: nondeterministic problems, for instance, are most easily specified in terms of *relations*. Even deterministic problems that enjoy deterministic solutions sometimes benefit from a relational setting. The problem of constructing tournament representations, which we consider in this paper, falls into this category.

At first sight, the generalization from functions to relations does away with the aforementioned benefits of pure functional languages, but not quite. Using *monads* [16, 17] (in particular, the set monad) and *monad comprehensions* [27] (in particular, set comprehensions) one can easily embed relations into a pure functional language. As a simple example, consider the converse of list catenation ('[ ]' and ':' are Haskell's list constructors).

$$\begin{array}{lcl} \mathit{split} & :: & \forall a\,.\,[\,a\,] \to \mathit{Set}\;([\,a\,],[\,a\,]) \\ \mathit{split}\;z & = & \{\,([\,],z)\,\} \\ & \cup & \{\,(a:x,y) \mid a:z' \leftarrow \{\,z\,\},(x,y) \leftarrow \mathit{split}\;z'\,\} \end{array}$$

The call *split* z yields all pairs of lists whose catenation is the list z itself. Here, a relation is defined as a *set-valued function*. In what follows we will use the term relation as a synonym for set-valued function. The set comprehension in the second line nicely describes the behaviour of *split* when the first result list is non-empty. In general, a set comprehension consists of a head and a body, which in turn is a sequence of *generators* of the form $p \leftarrow e$. The left arrow can be read as set membership (pronounced: is drawn from) and the comma separating the generators can be seen as a conjunction (commas are also used for pairs). It is important to note, however, that a generator binds the variables that appear to the left of the arrow, see Sec. 2.

Now, the point of this paper is that set comprehension syntax not only provides a succinct notation for defining nondeterministic functions but that it is also suitable for specifying and deriving relational programs. We will support this claim by re-solving the well-known problem of constructing tournament representations [26], which has been repeatedly considered in the literature [9, 5, 15, 1, 20, 7]. The derivation, which constitutes the major part of the paper, is structured into four successive steps, each of which yields an executable Haskell program with a decreasing amount of nondeterminism. Though we use Haskell as a target language, we will work in the world of sets and total functions. In particular, lists and trees are always finite and fully defined. All the derivations with the notable exception of the second step are conducted in a pointwise style. The calculations are quite detailed as no intermediate steps are omitted.

2 Notation

Let us introduce the notation by means of a simple derivation: we calculate the inverse of list catenation. Formally, we are seeking a set-valued function *split* that satisfies

$$(x, y) \leftarrow \mathit{split}\ z \quad \equiv \quad x \mathbin{+\!\!+} y = z.$$

The derivation, which is based on fold-unfold transformations [6], proceeds as follows.

$$\begin{array}{ll}
 & (x, y) \leftarrow \mathit{split}\ z \\
\equiv & \quad \{\ \text{specification of } \mathit{split}\ \} \\
 & x \mathbin{+\!\!+} y = z \\
\equiv & \quad \{\ x \text{ has type } [a]\ \} \\
 & ([\,] \mathbin{+\!\!+} y = z,\ x = [\,]) \vee ((a : x') \mathbin{+\!\!+} y = z,\ x = a : x') \\
\equiv & \quad \{\ \text{definition of `} \mathbin{+\!\!+} \text{'}\ \} \\
 & (y = z,\ x = [\,]) \vee (a : (x' \mathbin{+\!\!+} y) = z,\ x = a : x') \\
\equiv & \quad \{\ \text{introduce } z'\ \} \\
 & (y = z,\ x = [\,]) \vee (a : z' = z,\ x' \mathbin{+\!\!+} y = z',\ x = a : x') \\
\equiv & \quad \{\ \text{specification of } \mathit{split}\ \} \\
 & (y = z,\ x = [\,]) \vee (a : z' = z,\ (x', y) \leftarrow \mathit{split}\ z',\ x = a : x')
\end{array}$$

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

$$e_1 \leftarrow e_2 \equiv q \quad \equiv \quad e_2 = \{\, e_1 \mid q \,\} \tag{1}$$

The derived programs (if they are set-valued) will usually take the form on the right side. The left side is, however, more convenient for conducting calculations.

Applying the comprehension principle we obtain the following equation.

$$\mathit{split}\ z \quad = \quad \{\,(x, y) \mid (y = z,\ x = [\,]) \vee (a : z' = z,\ (x', y) \leftarrow \mathit{split}\ z',\ x = a : x')\,\}$$

The set comprehension on the right is slightly more general than what is currently available in Haskell as it involves a disjunction and equations. However, we can easily eliminate the disjunction using the following law.

$$\{\, e \mid q \vee r \,\} \quad = \quad \{\, e \mid q \,\} \cup \{\, e \mid r \,\} \tag{2}$$

Furthermore, an equation of the form $p = e$ where p is a pattern can be replaced by a generator:

$$p = e \quad \equiv \quad p \leftarrow \{\, e \,\}. \tag{3}$$

If we additionally inline simple generators of the form $x \leftarrow \{\, e \,\}$ where x is a variable, we obtain the program listed in Sec. 1.

Before we proceed, let us briefly explain why we can use the derived equation as a definition. In principle, we have to show that the equation has a least (or maybe a unique) solution and that this solution satisfies the original specification. The first part is easy to establish by appealing to the Knaster-Tarski theorem [14, 24] (using the pointwise ordering on functions and the inclusion ordering on sets). In the sequel, we will take the existence of least fixed points for granted. For the second part, we can reorder the derivation above so that it constitutes an inductive proof (inductive on the result list x) showing that an *arbitrary* solution of the equation is equal to *split*. Note that this implies that the equation has, in fact, a *unique* solution.

Set comprehensions need not be a primitive concept. They can be given a precise semantics via the following identities:

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

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

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

Like λ- and **case**-expressions, generators are binding constructs: the generator $p \leftarrow e$ *binds* the variables that appear in p.[1] Consequently, equations that appear

[1] By contrast, in Zermelo type theory [25] the set comprehension itself constitutes the binding construct.

in comprehensions must also be binding constructs. In line with rule 3 we agree upon that an equation binds the variables of the left (the variables on the right must be bound at an outer level).

It remains to give a semantics to general equations of the form $e_1 = e_2$, which appear, for instance, in the specification of *split* and along the derivation. The idea is simply that the variables on the left are bound to all combinations of values which satisfy the equation. Formally, let $\{x_1, \ldots, x_n\}$ be the free variables of e_1, then $e_1 = e_2$ serves as an abbreviation for

$$x_1 \leftarrow T_1, \ldots, x_n \leftarrow T_n, e_1 \texttt{==} e_2 \ ,$$

where T_i is the type of x_i and '`==`' is the test for equality. Since we are working in the world of sets, we view a type simply as a set, possibly given by an inductive definition. Of course, the goal of our program calculations is to eliminate general equations in favour of generators, so the above is 'merely' a precise semantics for the initial specification and the intermediate steps.

3 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 [26]. Note that the tournament representation is unique if the given integers are distinct. If the sequence contains duplicates, then there are several heaps that satisfy the condition above. We do not, however, make any additional assumptions and allow ties to be broken in arbitrary ways.

In order to specify the problem formally we require the notions of binary tree, heap and inorder traversal.

We represent *binary trees* (trees for short) by the following data type.

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

Note that the type of trees is parametric in the type of labels.

A tree is said to be a *heap* if the label of each node is at most the labels of its descendants. To check the heap property it suffices to compare each label to its immediate descendants.

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

The helper function *top* returns the topmost element of a tree.

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

Here and in what follows it is convenient to assume the existence of extremal elements ($-\infty$ and ∞), which must not appear in the given integer sequence. The final Haskell program will do without, however.

The function *list* yields the inorder traversal of a given tree.

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

Returning to our problem, we are seeking a function that is the right-inverse of *list* and whose results satisfy the heap property. As we have mentioned before, even though the final program will be deterministic the derivation requires or at least benefits from a relational setting. Consequently, we specify the desired program as a set-valued function $\mathit{tournament} :: [\mathit{Int}] \to \mathit{Set}\ (\mathit{Tree}\ \mathit{Int})$ that satisfies

$$t \leftarrow \mathit{tournament}\ x \;\equiv\; \mathit{list}\ t = x,\ \mathit{heap}\ t.$$

Before we proceed, let us slightly generalize the problem. The task of constructing tournament representations is closely related to precedence parsing. Both problems differ only in the relations '$\geqslant$' and '$\leqslant$', on which the *heap* predicate is based. Thus, in order to keep the derivation sufficiently general, we abstract away from the type of integers and from the integer orderings.

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

Here, *Elem* is some type of elements and '$\gtrdot$' and '$\lessdot$' are some predicates on integers. The predicates must satisfy certain properties, which we will infer in the course of the derivation. The properties are signalled by the hint '**assumption**'.

4 Step 1: tupling

Most likely, *tournament* (or rather, one of its helper functions) will be defined recursively. Furthermore, the recursive invocations will work on (contiguous) subparts of the original sequence. So, as a first step, we generalize *tournament* to a function $\mathit{build} :: [\mathit{Elem}] \to \mathit{Set}\ (\mathit{Tree}\ \mathit{Elem}, [\mathit{Elem}])$ that satisfies

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

This generalisation is an instance of a well-known technique of program optimization called *tupling* [3] and constitutes the main inventive step of the derivation.

Before tackling *build* let us first express *tournament* in terms of *build*.

$$\begin{array}{cl} & t \leftarrow \mathit{tournament}\ x \\ \equiv & \quad \{\ \text{specification of } \mathit{tournament}\ \} \\ & \mathit{list}\ t = x,\ \mathit{heap}\ t \\ \equiv & \quad \{\ \text{`}[\,]\text{' is the unit of `}+\!\!+\text{'}\ \} \\ & \mathit{list}\ t +\!\!+ [\,] = x,\ \mathit{heap}\ t \\ \equiv & \quad \{\ \text{specification of } \mathit{build}\ \} \\ & (t, [\,]) \leftarrow \mathit{build}\ x \end{array}$$

Thus, *tournament* can be defined as follows.

$$\mathit{tournament}\ x \quad = \quad \{\, t \mid (t, [\,]) \leftarrow \mathit{build}\ x \,\}$$

The derivation of *build* proceeds almost mechanically.

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

To avoid writing a long disjunction we conduct two subproofs. **Case** $t = E$:

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

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

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

To summarize, we have shown that the specification satisfies the following equation.

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

In fact, the equation even has a *unique* solution. Again, this can be demonstrated by reordering the steps, so that the derivation above becomes an inductive proof (inductive on the constructed tree t) showing that an arbitrary solution of the equation satisfies the specification (which, by its form, has a unique solution).

Furthermore, if we replace sets by lists and set comprehensions by list comprehensions, then this equation constitutes an executable Haskell program. (Additionally, we must replace the equation $a : x_2 = x_1$ by the generator $a : x_2 \leftarrow [\, x_1 \,]$.) Of course, there is little point in doing so as the program is hopelessly inefficient. Note in this respect that the construction of the left subtree is 'pure guesswork' as *build* passes its argument list x unchanged to the first recursive call. Clearly, further massage is necessary.

5 Step 2: turning top-down into bottom-up

An obvious idea for improving *build* is to promote the tests, that is, $top\ l \gtrdot a \lessdot top\ r$, into the generation of the trees. This step can be simplified considerably if we first eliminate the left-recursive call to *build*, which is what we will do next.

At this point, it is preferable to switch temporarily to a point-free style—left-recursion elimination is purely structural and the program structure is obscured by data variables. Now, using the identities of Sec. 2 *build* can be put into the form

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

for suitable functions a and b. To obtain a point-free definition we eliminate the data variable x by lifting '$\cup$' to the function level—$(f \cup g)\ n = f\ n \cup g\ n$—and by replacing monadic application by monadic composition—$(f \diamond g)\ n = f\ n \triangleright g$. We obtain the recursion equation:

$$build \;=\; a \cup build \diamond b,$$

which has the unique[2] solution $a \diamond b^*$, where $(-)^*$ denotes the reflexive, transitive closure of a relation. Now, 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

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

[2] We can show uniqueness in this more general setting using the *unique extension property* [2]: $f = a \cup f \diamond b$ has a unique solution if the relation b is well-founded. In our case, this condition is satisfied as the length of the element list is strictly decreasing.

or reverting back to pointwise style:

$$
\begin{array}{lcl}
\mathit{build}\ x & = & \mathit{loop}\ (E, x) \\
\mathit{loop}\ (l, x_1) & = & \{(l, x_1)\} \\
 & \cup & \{(t, z) \mid a : x_2 = x_1, \\
 & & \qquad\qquad (r, y) \leftarrow \mathit{loop}\ (E, x_2), \\
 & & \qquad\qquad \mathit{top}\ l \gtrdot a \lessdot \mathit{top}\ r, \\
 & & \qquad\qquad (t, z) \leftarrow \mathit{loop}\ (N\ l\ a\ r, y)\}.
\end{array}
$$

Additionally, we have replaced $\mathit{build}\ x_2$ by $\mathit{loop}\ (E, x_2)$.

The above transformation has, in effect, turned a top-down program into a bottom-up one. The original definition of *build* constructed a tournament representation from the root to the leaves whereas the helper function *loop* starts at the leftmost leaf and works its way up to the root.

6 Step 3: promoting the tests

We are now in a position that we can easily promote the tests $\mathit{top}\ l \gtrdot a \lessdot \mathit{top}\ r$ into the generation of the trees. In fact, the first half of the condition, that is, $\mathit{top}\ l \gtrdot a$ can be readily applied since *loop* receives the left subtree l as an argument and a is the first element of its list argument. It remains to propagate $a \lessdot \mathit{top}\ r$ motivating the following specification.

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

Note that the specified function *loop-to* maintains an *invariant*: if the topmost label of its tree argument is at least a given bound, then this property also holds for the tree returned by *loop-to*. Thus, using guards we can nicely express pre- and postconditions and invariants. Furthermore, it is important to note that the specification cannot be satisfied for arbitrary relations '$\gtrdot$' and '$\lessdot$'. Consider the case where $p \lessdot \mathit{top}\ l$ is false but the expression on the right has a solution. Rather pleasantly, the derivation below will produce suitable conditions on '$\gtrdot$' and '$\lessdot$'.

The derivation of a program for *loop-to* proceeds as follows.

$$
\begin{array}{ll}
 & p \lessdot \mathit{top}\ l,\ (t, z) \leftarrow \mathit{loop\text{-}to}\ p\ (l, x_1) \\
\equiv & \quad \{\ \text{specification of } \mathit{loop\text{-}to}\ \} \\
 & (t, z) \leftarrow \mathit{loop}\ (l, x_1),\ p \lessdot \mathit{top}\ t \\
\equiv & \quad \{\ \text{definition of } \mathit{loop}\ \} \\
 & ((t, z) \leftarrow \{(l, x_1)\},\ p \lessdot \mathit{top}\ t) \\
 & \qquad \vee\ (a : x_2 = x_1,\ (r, y) \leftarrow \mathit{loop}\ (E, x_2),\ \mathit{top}\ l \gtrdot a \lessdot \mathit{top}\ r, \\
 & \qquad\qquad\quad (t, z) \leftarrow \mathit{loop}\ (N\ l\ a\ r, y),\ p \lessdot \mathit{top}\ t)
\end{array}
$$

Again, we split the proof into two two subproofs. First disjunction:

$$
\begin{array}{ll}
 & (t, z) \leftarrow \{(l, x_1)\},\ p \lessdot \mathit{top}\ t \\
\equiv & \quad \{\ \text{sets}\ \} \\
 & p \lessdot \mathit{top}\ l,\ (t, z) = (l, x_1).
\end{array}
$$

Second disjunction:

$$
\begin{array}{ll}
 & a : x_2 = x_1,\ (r, y) \leftarrow \mathit{loop}\ (E, x_2),\ \mathit{top}\ l \gtrdot a \lessdot \mathit{top}\ r, \\
 & \quad (t, z) \leftarrow \mathit{loop}\ (N\ l\ a\ r, y),\ p \lessdot \mathit{top}\ t \\
\equiv & \quad \{ \text{ specification of } \mathit{loop\text{-}to} \ \} \\
 & a : x_2 = x_1,\ (r, y) \leftarrow \mathit{loop}\ (E, x_2),\ \mathit{top}\ l \gtrdot a \lessdot \mathit{top}\ r, \\
 & \quad p \lessdot \mathit{top}\ (N\ l\ a\ r),\ (t, z) \leftarrow \mathit{loop\text{-}to}\ p\ (N\ l\ a\ r, y) \\
\equiv & \quad \{ \text{ definition of } \mathit{top} \text{ and rearranging } \} \\
 & a : x_2 = x_1,\ \mathit{top}\ l \gtrdot a,\ p \lessdot a,\ (r, y) \leftarrow \mathit{loop}\ (E, x_2),\ a \lessdot \mathit{top}\ r, \\
 & \quad (t, z) \leftarrow \mathit{loop\text{-}to}\ p\ (N\ l\ a\ r, y) \\
\equiv & \quad \{ \text{ specification of } \mathit{loop\text{-}to} \ \} \\
 & a : x_2 = x_1,\ \mathit{top}\ l \gtrdot a,\ p \lessdot a,\ a \lessdot \mathit{top}\ E,\ (r, y) \leftarrow \mathit{loop\text{-}to}\ a\ (E, x_2), \\
 & \quad (t, z) \leftarrow \mathit{loop\text{-}to}\ p\ (N\ l\ a\ r, y) \\
\equiv & \quad \{ \text{ definition of } \mathit{top} \text{ and } \textbf{assumption}\ e \lessdot \infty \ \} \\
 & a : x_2 = x_1,\ \mathit{top}\ l \gtrdot a,\ p \lessdot a,\ (r, y) \leftarrow \mathit{loop\text{-}to}\ a\ (E, x_2), \\
 & \quad (t, z) \leftarrow \mathit{loop\text{-}to}\ p\ (N\ l\ a\ r, y) \\
\equiv & \quad \{ \ \textbf{assumption}\ i \lessdot j \ \wedge\ k \gtrdot j \ \implies\ i \lessdot k \ \} \\
 & p \lessdot \mathit{top}\ l,\ a : x_2 = x_1,\ \mathit{top}\ l \gtrdot a,\ p \lessdot a,\ (r, y) \leftarrow \mathit{loop\text{-}to}\ a\ (E, x_2), \\
 & \quad (t, z) \leftarrow \mathit{loop\text{-}to}\ p\ (N\ l\ a\ r, y).
\end{array}
$$

The last step requires that the relations '$\gtrdot$' and '$\lessdot$' are related by a *zig-zag transitivity* law. Loosely speaking, the law expresses that the left subtree of the right subtree is also a legal immediate right subtree.

It remains to express *build* in terms of *loop-to*.

$$
\begin{array}{ll}
 & (t, y) \leftarrow \mathit{build}\ x \\
\equiv & \quad \{ \text{ definition of } \mathit{build} \ \} \\
 & (t, y) \leftarrow \mathit{loop}\ (E, x) \\
\equiv & \quad \{ \ \textbf{assumption}\ -\infty \lessdot e \ \} \\
 & (t, y) \leftarrow \mathit{loop}\ (E, x),\ -\infty \lessdot \mathit{top}\ t \\
\equiv & \quad \{ \text{ specification of } \mathit{loop\text{-}to} \ \} \\
 & -\infty \lessdot \mathit{top}\ E, (t, y) \leftarrow \mathit{loop\text{-}to}\ (-\infty)\ (E, x) \\
\equiv & \quad \{ \ \textbf{assumption}\ -\infty \lessdot e \ \} \\
 & (t, y) \leftarrow \mathit{loop\text{-}to}\ (-\infty)\ (E, x)
\end{array}
$$

To summarize, we have calculated the following recursion equation.

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

As usual, we can reorder the derivation to obtain an inductive proof showing that each solution of the equation satisfies the specification. This time we induct on the length of the list argument (note that this requires showing that the output list is always a suffix of the input list). The recursion equation can be easily turned into a respectable Haskell program. However, there is still ample room for improvement.

7 Step 4: strengthening

Recall that *build* considers all prefixes of its argument list. Thus, it produces many (intermediate) results which are eventually discarded by *tournament*. The purpose of this section is to calculate a variant of *loop-to* which consumes as many elements as possible building maximal subtrees. It is convenient to introduce a function

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

which returns the first element of a list. Assuming that $p \lessdot \mathit{top}\ l$ the desired function (called $\mathit{loop\text{-}to}'$) can be specified as follows.

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

The precondition guarantees that the first element of the argument list is a legal predecessor of l. Likewise, the postcondition ensures that the first element of the remaining list is a legal predecessor of the tree labelled with p. If the relation '$\gtrdot$' is the converse of '$\lessdot$', then these conditions imply that the constructed trees are maximal. However, even if the relations are not converses of each other, we know at least that the initial call to $\mathit{loop\text{-}to}'$ with $p = -\infty$ (see below) consumes the complete input sequence as $-\infty \gtrdot \mathit{hd}\ y$ implies $y = [\,]$ (assuming that $-\infty$ is the least element).

The calculation proceeds as follows.

$$\begin{array}{cl} & \mathit{top}\ l \gtrdot \mathit{hd}\ x,\ (t, z) \leftarrow \mathit{loop\text{-}to}'\ p\ (l, x) \\ \equiv & \quad \{\text{ specification of } \mathit{loop\text{-}to}' \ \} \\ & (t, z) \leftarrow \mathit{loop\text{-}to}\ p\ (l, x),\ p \gtrdot \mathit{hd}\ z \\ \equiv & \quad \{\ x \text{ has type } [\mathit{Elem}]\ \} \\ & ([\,] = x, (t, z) \leftarrow \mathit{loop\text{-}to}\ p\ (l, [\,]),\ p \gtrdot \mathit{hd}\ z) \\ & \qquad \vee\ (a : x_2 = x, (t, z) \leftarrow \mathit{loop\text{-}to}\ p\ (l, a : x_2),\ p \gtrdot \mathit{hd}\ z) \end{array}$$

As we are working towards a program we conduct a case analysis on the input list. **Case** $x = [\,]$:

$$\begin{array}{ll} & (t,z) \leftarrow \mathit{loop\text{-}to}\ p\ (l,[\,]),\ p \geqslant \mathit{hd}\ z \\ \equiv & \quad \{\ \text{definition of } \mathit{loop\text{-}to}\ \} \\ & (t,z) \leftarrow \{(l,[\,])\},\ p \geqslant \mathit{hd}\ z \\ \equiv & \quad \{\ \text{definition of } \mathit{hd} \text{ and } \textbf{assumption } e \geqslant -\infty\ \} \\ & \mathit{top}\ l \geqslant \mathit{hd}\ x, (t,z) = (l,[\,]) \end{array}$$

Case $x = a : x_2$:

$$\begin{array}{ll} & (t,z) \leftarrow \mathit{loop\text{-}to}\ p\ (l, a : x_2),\ p \geqslant \mathit{hd}\ z \\ \equiv & \quad \{\ \text{definition of } \mathit{loop\text{-}to}\ \} \\ & ((t,z) \leftarrow \{(l, a : x_2)\},\ p \geqslant \mathit{hd}\ z) \\ & \quad \vee\ (\mathit{top}\ l \geqslant a,\ p \lessdot a,\ (r,y) \leftarrow \mathit{loop\text{-}to}\ a\ (E, x_2), \\ & \qquad\qquad (t,z) \leftarrow \mathit{loop\text{-}to}\ p\ (N\ l\ a\ r, y),\ p \geqslant \mathit{hd}\ z) \end{array}$$

As usual, we split the proof into two subproofs. First disjunction:

$$\begin{array}{ll} & (t,z) \leftarrow \{(l, a : x_2)\},\ p \geqslant \mathit{hd}\ z \\ \equiv & \quad \{\ \text{definition of } \mathit{hd}\ \} \\ & (t,z) = (l, a : x_2),\ p \geqslant a \\ \equiv & \quad \{\ p \lessdot \mathit{top}\ l \text{ and } \textbf{assumption } i \lessdot j\ \wedge\ i \geqslant k \implies j \geqslant k\ \} \\ & \mathit{top}\ l \geqslant a, (t,z) = (l, a : x_2),\ p \geqslant a \\ \equiv & \quad \{\ \text{definition of } \mathit{hd}\ \} \\ & \mathit{top}\ l \geqslant \mathit{hd}\ x, (t,z) = (l, a : x_2),\ p \geqslant a \end{array}$$

The *zig-zag transitivity* law that is required in the second but last step is dual to the one of the previous section: it expresses that the right subtree of the left subtree is also a legal immediate left subtree.

Second disjunction:

$$\begin{array}{ll} & \mathit{top}\ l \geqslant a,\ p \lessdot a,\ (r,y) \leftarrow \mathit{loop\text{-}to}\ a\ (E, x_2), \\ & \qquad (t,z) \leftarrow \mathit{loop\text{-}to}\ p\ (N\ l\ a\ r, y),\ p \geqslant \mathit{hd}\ z \\ \equiv & \quad \{\ \text{specification of } \mathit{loop\text{-}to}'\ \} \\ & \mathit{top}\ l \geqslant a,\ p \lessdot a,\ (r,y) \leftarrow \mathit{loop\text{-}to}\ a\ (E, x_2), \\ & \qquad \mathit{top}\ (N\ l\ a\ r) \geqslant \mathit{hd}\ y,\ (t,z) \leftarrow \mathit{loop\text{-}to}'\ p\ (N\ l\ a\ r, y) \\ \equiv & \quad \{\ \text{definition of } \mathit{top}\ \} \\ & \mathit{top}\ l \geqslant a,\ p \lessdot a,\ (r,y) \leftarrow \mathit{loop\text{-}to}\ a\ (E, x_2), \\ & \qquad a \geqslant \mathit{hd}\ y,\ (t,z) \leftarrow \mathit{loop\text{-}to}'\ p\ (N\ l\ a\ r, y) \\ \equiv & \quad \{\ \text{specification of } \mathit{loop\text{-}to}'\ \} \\ & \mathit{top}\ l \geqslant a,\ p \lessdot a,\ \mathit{top}\ E \geqslant \mathit{hd}\ x_2,\ (r,y) \leftarrow \mathit{loop\text{-}to}'\ a\ (E, x_2), \end{array}$$

$$
\begin{array}{ll}
& \quad (t,z) \leftarrow \textit{loop-to}'\ p\ (N\ l\ a\ r, y) \\
\equiv & \{\ \text{definition of } \textit{top} \text{ and } \textbf{assumption}\ \infty \gtrdot e\ \} \\
& \textit{top}\ l \gtrdot a,\ p \lessdot a,\ (r,y) \leftarrow \textit{loop-to}'\ a\ (E, x_2), \\
& \quad (t,z) \leftarrow \textit{loop-to}'\ p\ (N\ l\ a\ r, y) \\
\equiv & \{\ \text{definition of } \textit{hd}\ \} \\
& \textit{top}\ l \gtrdot \textit{hd}\ x,\ p \lessdot a,\ (r,y) \leftarrow \textit{loop-to}'\ a\ (E, x_2), \\
& \quad (t,z) \leftarrow \textit{loop-to}'\ p\ (N\ l\ a\ r, y)
\end{array}
$$

We can now define *tournament* directly in terms of $\textit{loop-to}'$.

$$
\begin{array}{ll}
& t \leftarrow \textit{tournament}\ x \\
\equiv & \{\ \text{definition of } \textit{tournament}\ \} \\
& (t,[\,]) \leftarrow \textit{build}\ x \\
\equiv & \{\ \text{definition of } \textit{build}\ \} \\
& (t,[\,]) \leftarrow \textit{loop-to}\ (-\infty)\ (E, x) \\
\equiv & \{\ \text{definition of } \textit{hd} \text{ and } \textbf{assumption}\ -\infty \gtrdot e \equiv e = -\infty\ \} \\
& (t,y) \leftarrow \textit{loop-to}\ (-\infty)\ (E,x),\ -\infty \gtrdot \textit{hd}\ y \\
\equiv & \{\ \text{specification of } \textit{loop-to}'\ \} \\
& \textit{top}\ E \gtrdot \textit{hd}\ x, (t,y) \leftarrow \textit{loop-to}'\ (-\infty)\ (E,x) \\
\equiv & \{\ \text{definition of } \textit{top} \text{ and } \textbf{assumption}\ \infty \gtrdot e\ \} \\
& (t,y) \leftarrow \textit{loop-to}'\ (-\infty)\ (E,x)
\end{array}
$$

As an aside, note that $\textit{hd}\ y = -\infty \equiv y = [\,]$, which is implicitly used in the third step, holds because the input does not contain $-\infty$ as an element.

To summarize, we have calculated the following program for constructing tournament representations (the proof that $\textit{loop-to}'$ satisfies the specification uses the same induction scheme as in the previous section).

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

This program satisfies the original specification under the proviso that $-\infty$ is the least element (∞ is no longer needed),

$$-\infty \lessdot e \gtrdot -\infty,$$

$$-\infty \gtrdot e \equiv e = -\infty,$$

and that the orderings satisfy the zig-zag transitivity laws,

$$i \lessdot j \ \wedge\ k \gtrdot j \implies i \lessdot k,$$

$$i \lessdot j \ \wedge\ i \gtrdot k \implies j \gtrdot k.$$

8 A Haskell program

The derived program is still nondeterministic but the final step to a deterministic Haskell program is a small one. To start with, we instantiate the abstract relations '$\gtrdot$' and '$\lessdot$' setting $(\gtrdot) = (\geqslant)$ and $(\lessdot) = (<)$:

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

Since the relations are exclusive, we can now replace the disjoint union in the second equation by a conditional as justified by the following calculation.

$$\begin{array}{ll}
 & \{e_1 \mid p,\ q_1\} \cup \{e_2 \mid \neg\, p,\ q_2\} \\
\equiv & \quad \{\ \text{set comprehensions}\ \} \\
 & (\textbf{if}\ p\ \textbf{then}\ \{e_1 \mid q_1\}\ \textbf{else}\ \varnothing) \cup (\textbf{if}\ \neg\, p\ \textbf{then}\ \{e_2 \mid q_2\}\ \textbf{else}\ \varnothing) \\
\equiv & \quad \{\ \textbf{if}\ \neg\, c\ \textbf{then}\ a\ \textbf{else}\ b = \textbf{if}\ c\ \textbf{then}\ b\ \textbf{else}\ a\ \} \\
 & (\textbf{if}\ p\ \textbf{then}\ \{e_1 \mid q_1\}\ \textbf{else}\ \varnothing) \cup (\textbf{if}\ p\ \textbf{then}\ \varnothing\ \textbf{else}\ \{e_2 \mid q_2\}) \\
\equiv & \quad \{\ \text{union distributes over conditionals}\ \} \\
 & \textbf{if}\ p\ \textbf{then}\ \{e_1 \mid q_1\} \cup \varnothing\ \textbf{else}\ \varnothing \cup \{e_2 \mid q_2\} \\
\equiv & \quad \{\ s \cup \varnothing = s = \varnothing \cup s\ \} \\
 & \textbf{if}\ p\ \textbf{then}\ \{e_1 \mid q_1\}\ \textbf{else}\ \{e_2 \mid q_2\}
\end{array}$$

Applying this transformation we get

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

Note that we have saved half of the comparisons as compared to the program of Sec. 6. Furthermore, note that $\mathit{loop\text{-}to}'$ has exactly one solution for each combination of arguments. Thus, to obtain a deterministic program we simply switch from the set monad to the identity monad effectively replacing set comprehensions by **let**-bindings.

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

$$
\begin{array}{lcl}
\mathit{tournament} & :: & \forall a \,.\, (\mathit{Ord}\ a) \Rightarrow [\,a\,] \to \mathit{Tree}\ a \\
\mathit{tournament}\ x & = & \mathit{loop}\ E\ x \\
\mathit{loop} & :: & \forall a \,.\, (\mathit{Ord}\ a) \Rightarrow \mathit{Tree}\ a \to [\,a\,] \to \mathit{Tree}\ a \\
\mathit{loop}\ l\ [\,] & = & l \\
\mathit{loop}\ l\ (a : x) & = & \mathbf{let}\ (r, y) = \mathit{loop\text{-}to}\ a\ E\ x\ \mathbf{in}\ \mathit{loop}\ (N\ l\ a\ r)\ y \\
\mathit{loop\text{-}to} & :: & \forall a \,.\, (\mathit{Ord}\ a) \Rightarrow a \to \mathit{Tree}\ a \to [\,a\,] \to (\mathit{Tree}\ a, [\,a\,]) \\
\mathit{loop\text{-}to}\ p\ l\ [\,] & = & (l, [\,]) \\
\mathit{loop\text{-}to}\ p\ l\ \mathit{as}@(a : x) & & \\
\quad\mid p \geqslant a & = & (l, \mathit{as}) \\
\quad\mid \mathit{otherwise} & = & \mathbf{let}\ (r, y) = \mathit{loop\text{-}to}\ a\ E\ x\ \mathbf{in}\ \mathit{loop\text{-}to}\ p\ (N\ l\ a\ r)\ y
\end{array}
$$

Fig. 1. A Haskell program for constructing tournament representations.

It is not hard to see that the derived Haskell program takes linear time and space, which is optimal for the given problem.

We have assumed throughout that we are working with an abstract type *Elem* of elements. Using Haskell's type classes [10] we can nicely capture this abstraction generalizing the type of integers to an arbitrary instance of the classes *Ord* and *Bounded*: the first class provides the ordering relation; the second provides the extremal element $-\infty$. Actually, if we are willing to accept some duplication of code, we can even remove the dependence on *Bound* by specializing $\mathit{loop\text{-}to}'\ p$ for $p = -\infty$:

$$\mathit{loop}'\ (l, x) \quad = \quad \mathit{fst}\ (\mathit{loop\text{-}to}'\ (-\infty)\ (l, x)).$$

The final program that incorporates this generalization is displayed in Fig. 1. (Additionally, we have renamed and curried loop' and $\mathit{loop\text{-}to}'$.)

Before we review related work, let us consider two variations of the problem.

Strict heaps A minor twist is to require the heap to be *strict*: the label of each node must be strictly smaller than the labels of its descendants. For this variant we simply instantiate the abstract relations to strict orderings: $(\gtrdot) = (>)$ and $(\lessdot) = (<)$. In this case, *tournament* has *at most* one solution. Therefore, we cannot refine *Set* to the identity monad but must use the *Maybe* monad instead. The details of rewriting the program of Sec. 7 are left to the reader.

Precedence parsing The problem of constructing tournament representations closely corresponds to the problem of precedence parsing. In fact, a solution to both problems was first given in the context of parsing by Floyd [8].

In Haskell, an operator can be assigned a *precedence* and an *associativity*. The higher the precedence the more tightly binds the operator. Conflicts between operators of equal precedence are resolved using associativity: left associativity causes grouping to the left, right associativity accordingly to the right. Sequences of non-associative operators of equal precedence are not allowed.

If we represent operators by the data types

$$\begin{aligned} \textbf{data}\ \mathit{Assoc} &= L \mid N \mid R \\ \textbf{data}\ \mathit{Op} &= \mathit{Op}\ \mathit{Assoc}\ \mathit{Int}, \end{aligned}$$

we can define the relations '$\gtrdot$' and '$\lessdot$' as follows:

$$\begin{aligned} \mathit{Op}\ a\ p \gtrdot \mathit{Op}\ L\ p' &= p \geqslant p' \\ \mathit{Op}\ a\ p \gtrdot \mathit{Op}\ N\ p' &= p > p' \\ \mathit{Op}\ a\ p \gtrdot \mathit{Op}\ R\ p' &= p > p' \\ \mathit{Op}\ L\ p \lessdot \mathit{Op}\ a'\ p' &= p < p' \\ \mathit{Op}\ N\ p \lessdot \mathit{Op}\ a'\ p' &= p < p' \\ \mathit{Op}\ R\ p \lessdot \mathit{Op}\ a'\ p' &= p \leqslant p'. \end{aligned}$$

The minimal element is given by $\mathit{Op}\ L\ (-\infty)$.

Since there may be still several expression trees for a given sequence of operators, we must refine *Set* to the *List* monad. The details are again left to the reader.

The program of Sec. 7 does not consider the operands of operators. However, since expression trees are *full* binary trees—each node has either no or two subtrees—operands can be easily added at a later stage. Alternatively, operands can be defined as elements of highest precedence.

9 Related work

Tournament representations were introduced by Vuillemin [26] as a special case of *cartesian trees*. (A cartesian tree consists of points in the plane such that the x-part is a binary search tree and the y-part is a binary heap. This data structure is also known as a *treap*.) Both data structures have a number of applications in computational geometry and adaptive sorting [9, 15].

The first derivation of a heap construction function is due to Bird [5]; several authors have subsequently presented alternative approaches [1, 20, 7]. The main idea of most solutions is to represent the tournament tree by its *left spine*, the sequence of pennants (topped binary trees) on the path from the leftmost leaf to the root. This representation change turns a top-down *data structure* into a bottom-up one and nicely corresponds to the second step of our derivation, where we converted a top-down *algorithm* into a bottom-up one.

The derivation that is closest in spirit to ours is the one by Augusteijn [1]. He conducts similar steps but misses the optimization introduced in Sec. 7. Interestingly, Augusteijn employs a pointwise style for left-recursion elimination, which is based on a rather specific theorem. We feel that the point-free argument of Sec. 5 is more elegant.

We have mentioned before that the problem of constructing tournament representations is closely related to precedence parsing. Though the work in this area [8] predates the papers above, this relationship is hardly recognized. Precedence parsing as an instance of bottom-up parsing uses a stack, which contains the recognized prefix of a sentential form. In the case of an expression grammar this stack corresponds to the right spine representation of a tree. Our algorithm can be seen as a stack-free implementation of the parsing algorithm, where the stack is implicitly represented by the recursion stack, see also [21, 13].

List comprehensions, which are due to Burstall and Darlington, were incorporated in several non-strict functional languages such as KRC, Miranda, and Haskell. Wadler generalized list comprehensions to monad comprehensions [27]. List and set comprehensions also appear in several textbooks on program derivation, most notably [4, 18], but they seem to play only a minor rôle in actual derivations.

An alternative approach to pointwise relational programming was recently put forward by de Moor and Gibbons [7]. They propose to use a nondeterministic functional language that includes relational combinators such as converse and choice. The use of choice allows for a much tighter integration of the relational and the functional world at the cost of weakening β- and η-conversion to inequalities. In a sense, our approach is closer to pure functional languages such as Haskell, which require an embedding of the relational part, whereas de Moor's and Gibbons's calculus is closer to functional logic languages such as Curry [11]. In fact, there are several crosslinks to logic programming, for instance, embeddings of Prolog into Haskell [12, 22] and fold-unfold systems for logic programs [23], which we plan to explore in the future.

Acknowledgements

I am grateful to the five anonymous referees, who went over and above the call of duty, providing numerous constructive comments for the revision of this paper. I am particularly indebted to Ernie Cohen for valuable suggestions regarding presentation and for spotting two errors in the derivation of *loop-to* and *loop-to*$'$.

References

1. Lex Augusteijn. An alternative derivation of a binary heap construction function. In R.S. Bird, C.C. Morgan, and J.C.P Woodcock, editors, *Second International Conference on the Mathematics of Program Construction, Oxford*, volume 669 of *Lecture Notes in Computer Science*, pages 368–374. Springer, 1992.
2. Roland Backhouse. Galois connections and fixed point calculus, 2001. Lecture Notes.
3. Richard Bird. *Introduction to Functional Programming using Haskell*. Prentice Hall Europe, London, 2nd edition, 1998.
4. Richard Bird and Oege de Moor. *Algebra of Programming*. Prentice Hall Europe, London, 1997.
5. Richard S. Bird. Lectures on constructive functional programming. In Manfred Broy, editor, *Constructive Methods in Computer Science*. Springer-Verlag, 1988.
6. R. Burstall and J. Darlington. A tranformation system for developing recursive programs. *Journal of the ACM*, 24(1):44–67, 1977.

7. Oege de Moor and Jeremy Gibbons. Pointwise relational programming. In T. Rus, editor, *Proceedings of Algebraic Methodology and Software Technology (AMAST 2000), Iowa*, volume 1816 of *Lecture Notes in Computer Science*, pages 371–390. Springer-Verlag, May 2000.
8. R.W. Floyd. Syntactic analysis and operator precedence. *Journal of the ACM*, 10(3):316–333, 1963.
9. H.N. Gabow, J.L. Bentley, and R.E. Tarjan. Scaling and related techniques for geometry problems. In *Proceedings of the 16th Annual ACM Symposium on Theory of Computing*, pages 135–143, 1984.
10. Cordelia V. Hall, Kevin Hammond, Simon L. Peyton Jones, and Philip L. Wadler. Type classes in Haskell. *ACM Transactions on Programming Languages and Systems*, 18(2):109–138, March 1996.
11. Michael Hanus, editor. *Curry—An Integrated Functional Logic Language (Version 0.7.1)*, June 2000.
12. Ralf Hinze. Prolog's control constructs in a functional setting — Axioms and implementation. *International Journal of Foundations of Computer Science*, 12(2):125–170, 2001.
13. Ralf Hinze and Ross Paterson. Derivation of a typed functional LR parser, 2002. in preparation.
14. B. Knaster. Un théorème sur les fonctions d'ensembles. *Annales de la Societé Polonaise de Mathematique*, 6:133–134, 1928.
15. Christos Levcopoulos and Ola Petersson. Heapsort—adapted for presorted files. In F. Dehne, J.-R. Sack, and N. Santoro, editors, *Algorithms and Data Structures*, volume 382 of *Lecture Notes in Computer Science*, pages 499–509. Springer, 1989.
16. Eugenio Moggi. An abstract view of programming languages. Technical Report ECS-LFCS-90-113, Department of Computer Science, Edinburgh University, 1990.
17. Eugenio Moggi. Notions of computation and monads. *Information and Computation*, 93(1):55–92, 1991.
18. Helmut A. Partsch. *Specification and Transformation of Programs—A Formal Approach to Software Development*. Texts and Monographs in Computer Science. Springer-Verlag, Berlin, 1990.
19. Simon Peyton Jones and John Hughes, editors. *Haskell 98 — A Non-strict, Purely Functional Language*, February 1999. Available from `http://www.haskell.org/definition/`.
20. Berry Schoenmakers. Inorder traversal of a binary heap and its inversion in optimal time and space. In R.S. Bird, C.C. Morgan, and J.C.P Woodcock, editors, *Second International Conference on the Mathematics of Program Construction, Oxford*, volume 669 of *Lecture Notes in Computer Science*, pages 291–301. Springer, 1992.
21. Michael Sperber and Peter Thiemann. Generation of LR parsers by partial evaluation. *ACM Transactions on Programming Languages and Systems*, 22(3):224–264, 2000.
22. J.M. Spivey and S. Seres. Embedding Prolog in Haskell. In Erik Meijer, editor, *Proceedings of the 3rd Haskell Workshop, Paris, France*, September 1999. The proceedings appeared as a technical report of Universiteit Utrecht, UU-CS-1999-28.
23. H. Tamaki and T. Sato. Unfold/fold transformations of logic programs. In *Second International Conference on Logic Programming*, pages 127–138, 1984.
24. A. Tarski. A lattice-theoretic fixpoint theorem and its applications. *Pacific Journal of Mathematics*, 5:285–309, 1955.
25. Paul Taylor. *Practical Foundations of Mathematics*. Number 59 in Cambridge Studies in Advanced Mathematics. Cambridge University Press, 1999.

26. Jean Vuillemin. A unifying look at data structures. *Communications of the ACM*, 23:229–239, 1980.
27. Philip Wadler. Comprehending monads. In *Proceedings of the 1990 ACM Conference on LISP and Functional Programming, Nice*, pages 61–78. ACM-Press, June 1990.