Functional-Program Inversion, with an application to Parser Construction

0 Introduction

As far as I know, E.W. Dijkstra was the first to report [6] that it is sometimes convenient to specify a program as the inverse of some other program, particularly so if that other program can be constructed more easily. This technique is attractive when the required inversion of the program is not too difficult. Program inversion has been studied mainly for sequential programs; for relatively recent discussions we refer to [5, 10].

In this paper we investigate program inversion in a functional setting, with an application to parser design: expressions can be viewed as linear representations of trees, and parsing can be viewed as an inverse process, namely of reconstructing a tree from its linear representation.

We will give special attention to tail-recursive function definitions, because in tail-recursive definitions function argument and function value stand, in a way, on equal footings; as we will see, this makes inversion easier.

The following example gives an impression of the kind of reasoning that provided the inspiration for the theorems to follow. We consider a datatype T of (finite) binary trees, defined recursively by:

$$\begin{array}{rl} & \langle\,\rangle \in T \\ \langle s,t\rangle & \in T \quad \text{, for all } s,t \text{ in } T \ . \end{array}$$

Such trees can be represented by lists in various ways, one of which is given by the function L, of type $T \to \mathcal{L}_*(\{0,1\})$, defined as follows:

$$\begin{array}{lcl} L{\cdot}\langle\,\rangle & = & [0] \\ L{\cdot}\langle s,t\rangle & = & [1] +\!\!+ L{\cdot}s +\!\!+ L{\cdot}t \end{array}$$

Function L maps a tree onto a list of 0's and 1's; as we will see, these digits serve to make the representation unambiguous, such that every list represents at most one tree. A parser for these lists is a (partial) function P, of type $\mathcal{L}_*(\{0,1\}) \to T$, with the property that P is an inverse of L:

$$(\forall s : s \in T : P{\cdot}(L{\cdot}s) = s) \quad .$$

A tail-recursive implementation[0] of L is, where parameter p has type $\mathcal{L}_*(T)$:

[0] The derivation of this definition is irrelevant here; in Section 3 we shall encounter a similar derivation.

$$
\begin{array}{lll}
(0) & L{\cdot}s & = F{\cdot}[\,]{\cdot}[\,s\,] \\
(1) & F{\cdot}x{\cdot}[\,] & = x \\
(2) & F{\cdot}x{\cdot}(\langle\,\rangle \triangleright p) & = F{\cdot}(x \triangleleft 0){\cdot}p \\
(3) & F{\cdot}x{\cdot}(\langle s,t\rangle \triangleright p) & = F{\cdot}(x \triangleleft 1){\cdot}(s \triangleright t \triangleright p)
\end{array}
$$

Due to the parameter patterns, rules (1) through (3) can be viewed as *rewrite rules*; by reading them from right to left, we obtain a new function P, defined in terms of a new function G –explanation follows–:

$$
\begin{array}{lll}
(4) & P{\cdot}x & = G{\cdot}x{\cdot}[\,] \\
(5) & G{\cdot}[\,]{\cdot}[\,s\,] & = s \\
(6) & G{\cdot}(x \triangleleft 0){\cdot}p & = G{\cdot}x{\cdot}(\langle\,\rangle \triangleright p) \\
(7) & G{\cdot}(x \triangleleft 1){\cdot}(s \triangleright t \triangleright p) & = G{\cdot}x{\cdot}(\langle s,t\rangle \triangleright p)
\end{array}
$$

Rule (4) is inspired by rule (1), rule (5) is inspired by rule (0), and rules (6) and (7) are obtained by reading rules (2) and (3) from right to left. Notice that rules (6) and (7) are not mutually conflicting, due to the digits 0 and 1.

Now it is not so bad an idea to surmise that this function P could be an inverse of function L. As a matter of fact, it is! To prove this we need a few theorems to capture the patterns underlying this transformation. The one theorem deals with tail recursion, whereas the other two deal with case analysis. This is the subject of Section 2; first, however, we study the inversion of simple, linearly recursive definitions.

1 Pattern driven program inversion

A linearly recursive function definition can often be transformed into a definition for an inverse of that function, by naively reading the definition "backwards". When the definition contains parameter patterns, its mere shape provides heuristic guidance for the transformation. We illustrate this technique by means of a few simple examples. Undoubtedly will it be possible to formulate a theorem capturing the essence of the technique for a large class of function definitions; yet, we shall not explore this here: ad hoc correctness proofs are usually straightforward, and the technique is mainly relevant for its heuristic value.

1.0 A simple example

We consider the type $\mathcal{L}_*(\{\,0..9\,\})$ –$\mathcal{L}10$ for short– of (finite) lists of decimal digits. Lists of this type represent natural numbers, according to the rules

of the decimal number system. The number represented by digit list x is $val{\cdot}x$; in view of the properties of the decimal system, like $569 = 56*10+9$, a definition for val is easily written down:

$$\begin{array}{lcl} val{\cdot}[\,] & = & 0 \\ val{\cdot}(x \lhd d) & = & val{\cdot}x * 10 + d \end{array}$$

This representation is not unique: '7' and '007', for example, denote the same number; similarly, the number 'zero' is represented accurately by the empty list, but in every day life we prefer the nonempty string '0'.

For any natural number we may ask for a digit string whose value is that number. So, we ask for a function str, of type $\mathsf{Nat} \to \mathcal{L}10$, mapping numbers onto the digit lists representing those numbers; a definition for str may not be as obvious, but it can be *specified* by requiring that val is its inverse:

$$(\forall m :: val{\cdot}(str{\cdot}m) = m) \quad .$$

A rather naive way to obtain a definition for str is to read the definition for val "backwards": because $val{\cdot}[\,] = 0$ we suspect that $str{\cdot}0 = [\,]$ should hold too, and because $val{\cdot}x*10+d$ is an instance of the more general $n*10+d$ we suspect that $str{\cdot}(n*10+d) = str{\cdot}n \lhd d$ should be viable as well. Therefore, we guess that this might be a useful definition for str:

$$\begin{array}{lcl} str{\cdot}0 & = & [\,] \\ str{\cdot}(n*10+d) & = & str{\cdot}n \lhd d \end{array}$$

To verify this we now derive a definition for str, in the "normal" way. The general strategy is that an equation of the shape $x :\ f{\cdot}x = E$ can be solved by rewriting its right-hand side E into an expression of the shape $f{\cdot}F$, whence we conclude that $x = F$ solves the equation. The right-hand sides of the definition of val are expressions of the shapes 0 and $n*10+d$. Because the argument of str is also a value of val, we use these shapes to guide the derivation of a definition for str, that is, we deal with the cases $m := 0$ and $m := n*10+d$ separately:

$$\begin{array}{ll} & val{\cdot}(str{\cdot}0) \\ = & \quad \{\ \text{specification of } str\ \} \\ & 0 \\ = & \quad \{\ \text{definition of } val\ \} \\ & val{\cdot}[\,] \quad , \end{array}$$

and:

$$
\begin{array}{ll}
 & val\cdot(str\cdot(n*10+d)) \\
= & \quad \{ \text{ specification of } str \ \} \\
 & n*10+d \\
= & \quad \{ \text{ Induction Hypothesis (see below), to reintroduce } val \ \} \\
 & val\cdot(str\cdot n)*10+d \\
= & \quad \{ \text{ definition of } val \ \} \\
 & val\cdot(str\cdot n \triangleleft d) \quad ,
\end{array}
$$

from which we obtain the following definition for str:

$$
\begin{array}{lll}
 & str\cdot 0 & = [\,] \\
(8) & str\cdot(n*10+d) & = str\cdot n \triangleleft d
\end{array}
$$

The appeal to the Induction Hypothesis is only correct if $n < n*10+d$, which for natural n and d is equivalent to $0 < n*10+d$. Hence, rule (8) is only applicable to positive arguments, and it must be guarded by $0 < n*10+d$.

The definition thus obtained is almost the same as the one we obtained in the naive way. The only difference lies in the guard: apparently, it is a rule of the game that the right-hand side expressions of the original definition must be distinguishable. After elimination of the parameter patterns the definition for str becomes:

$$
\begin{array}{lll}
str\cdot m & = & \mathsf{if}\ \ m{=}0 \ \rightarrow\ [\,] \\
 & & [\!]\ \ m{>}0 \ \rightarrow\ str\cdot n \triangleleft d \\
 & & \qquad\qquad \mathsf{whr}\ \ n = m\,\mathsf{div}\,10\ \ \&\ \ d = m\,\mathsf{mod}\,10\ \ \mathsf{end} \\
 & & \mathsf{fi}
\end{array}
$$

1.1 The role of parameter patterns

In the previous subsection we used as definition for function val:

$$
\begin{array}{ll}
val\cdot[\,] & = 0 \\
val\cdot(x \triangleleft d) & = val\cdot x*10+d
\end{array}
$$

This definition contains so-called *parameter patterns*, which make it possible to view the definition as two independent rewrite rules. This definition actually is an abbreviation of:

$$
\begin{array}{lll}
val\cdot y & = & \mathsf{if}\ \ y{=}[\,] \ \rightarrow\ 0 \\
 & & [\!]\ \ y{\neq}[\,] \ \rightarrow\ val\cdot x*10+d\ \ \mathsf{whr}\ \ x,d\colon\ y = x \triangleleft d\ \ \mathsf{end} \\
 & & \mathsf{fi}
\end{array}
$$

That the use of parameter patterns causes no inconsistencies follows from two properties of the list constructors, namely the property:

$$x \lhd d \neq [\,] \quad , \text{ for all } x, d \quad ,$$

so, the two rules of the definition can be distinguished, and the property that the equation:

$$x, d\colon\ y \,=\, x \lhd d$$

has a solution for every nonempty list y, that is, operator $\lhd$ has an inverse. Because every nonempty list is of the shape $x \lhd d$, the above defines $val{\cdot}x$ for every list x.

These properties are not specific for the list datatype, as the proper use of parameter patterns always brings about these two requirements: the different cases in a definition must be distinguishable and the equations implied by the patterns must have solutions. The former requirement is needed to make the definition deterministic, the latter is needed to guarantee that the function is indeed defined everywhere on its intended domain.

If we can also distinguish the *right-hand* side expressions of such a definition, and if these right-hand sides themselves constitute "meaningful patterns", then a definition for the function's inverse can be obtained in an almost mechanical way. For our example, we do so by observing that the right-hand side expressions, 0 and $val{\cdot}x * 10 + d$ indeed can be distinguished, if we restrict the latter to positive values. Moreover, the equation $n, d\colon\ m \,=\, n * 10 + d$ can be solved. Its solution even is unique under the additional (type) requirement $0 \leq d < 10$. Without performing any calculation we obtain the same definition as derived in the previous subsection:

$$\begin{array}{lcl}
str{\cdot}m & = & \mathsf{if}\ \ m{=}0 \ \rightarrow\ [\,] \\
 & & [\!]\ \ m{>}0 \ \rightarrow\ str{\cdot}n \lhd d \\
 & & \qquad\qquad \mathsf{whr}\ \ n = m \,\mathsf{div}\, 10\ \ \&\ \ d = m \,\mathsf{mod}\, 10\ \ \mathsf{end} \\
 & & \mathsf{fi}
\end{array}$$

1.2 The Lines/Unlines Problem

The following problem is taken from [2]. For some set B and some value $\diamond$ not in B we are given a function *unlines*, of type $\mathcal{L}_*(\mathcal{L}_*(B)) \rightarrow \mathcal{L}_*(B \cup \{\diamond\})$. Function *unlines* is defined on nonempty lists only; here we assume $s \in \mathcal{L}_*(B)$, so $\neg(\diamond \in s)$, and $p \in \mathcal{L}_*(\mathcal{L}_*(B))$:

$$\begin{array}{lcl}
unlines{\cdot}(s \rhd [\,]) & = & s \\
unlines{\cdot}(s \rhd p) & = & s +\!\!+ [\diamond] +\!\!+ unlines{\cdot}p \quad , \text{ for } p\colon\ p \neq [\,]
\end{array}$$

The problem is to derive a definition for an inverse of *unlines*, that is, a function *lines*, satisfying:

$$(\forall p : p \neq [\,] : \mathit{lines} \cdot (\mathit{unlines} \cdot p) = p) \quad .$$

The right-hand side expressions in *unlines*'s definition can be distinguished: $\diamond$ occurs in the one but not in the other. A straightforward inversion of the definition of *unlines* yields as correct (but not very efficient) definition for *lines*:

$$\begin{array}{lcll} \mathit{lines} \cdot s & = & s \triangleright [\,] & \text{, for } s : \neg(\diamond \in s) \\ \mathit{lines} \cdot (s +\!\!+ [\diamond] +\!\!+ x) & = & s \triangleright \mathit{lines} \cdot x & \text{, for } s : \neg(\diamond \in s) \end{array}$$

To eliminate the awkward pattern $s +\!\!+ [\diamond] +\!\!+ x$ (and to improve the efficiency accordingly), we distinguish $s = [\,]$ and $s \neq [\,]$, for both rules of this definition. With $s := [\,]$ we obtain as instances:

$$\begin{array}{lcl} \mathit{lines} \cdot [\,] & = & [\,] \triangleright [\,] \\ \mathit{lines} \cdot ([\diamond] +\!\!+ x) & = & [\,] \triangleright \mathit{lines} \cdot x \end{array}$$

Furthermore, with $s := b \triangleright s$, we derive:

$$\begin{array}{ll} & \mathit{lines} \cdot (b \triangleright s) \\ = & \quad \{\ \mathit{lines}\ \} \\ & (b \triangleright s) \triangleright [\,] \\ = & \quad \{\ \text{introduction of an operator } \oplus\text{, so as to rearrange terms}\ \} \\ & b \oplus (s \triangleright [\,]) \\ = & \quad \{\ \mathit{lines}\ \} \\ & b \oplus (\mathit{lines} \cdot s) \quad , \end{array}$$

so we obtain $\mathit{lines} \cdot (b \triangleright s) = b \oplus (\mathit{lines} \cdot s)$. This derivation is correct provided that we define the new operator $\oplus$ by:

$$(\forall b, s, p :: b \oplus (s \triangleright p) = (b \triangleright s) \triangleright p) \quad .$$

Similarly, we derive:

$$\begin{array}{ll} & \mathit{lines} \cdot (b \triangleright s +\!\!+ [\diamond] +\!\!+ x) \\ = & \quad \{\ \mathit{lines}\ \} \\ & (b \triangleright s) \triangleright \mathit{lines} \cdot x \\ = & \quad \{\ \oplus\ \} \end{array}$$

$$
\begin{array}{ll}
 & b \oplus (s \triangleright \mathit{lines} \cdot x) \\
= & \quad \{\ \mathit{lines}\ \} \\
 & b \oplus (\mathit{lines} \cdot (s +\!\!+ [\diamond] +\!\!+ x))\ \ .
\end{array}
$$

From these two derivations it follows that the cases for $\mathit{lines} \cdot (b \triangleright s)$ and for $\mathit{lines} \cdot (b \triangleright s +\!\!+ [\diamond] +\!\!+ x)$ can be combined into a single rule for $\mathit{lines} \cdot (b \triangleright x)$. Thus, we obtain as new, and more efficient, definition for lines –where $b \neq \diamond$–:

$$
\begin{array}{lcl}
\mathit{lines} \cdot [\,] & = & [\,] \triangleright [\,] \\
\mathit{lines} \cdot (\diamond \triangleright x) & = & [\,] \triangleright \mathit{lines} \cdot x \\
\mathit{lines} \cdot (b \triangleright x) & = & b \oplus (\mathit{lines} \cdot x) \\
\\
b \oplus (s \triangleright p) & = & (b \triangleright s) \triangleright p
\end{array}
$$

2 Tail recursion

At first sight it may seem strange that a function with a tail-recursive definition should have an inverse at all: the typical shape of a tail-recursive rule is $F \cdot x = F \cdot (f \cdot x)$, so function F thus defined cannot possibly have a inverse. Nevertheless, if F is only used on a subspace of its domain, it may have an inverse on that subset. As we will see, this is often the case.

As the prototype of tail recursion we first study definitions of this shape:

$$
\begin{array}{lcl}
F \cdot x & = & \mathsf{if}\ \ \neg b \cdot x \ \ \rightarrow \ \ x \\
 & & []\ \ \ b \cdot x \ \ \rightarrow \ \ F \cdot (f \cdot x) \\
 & & \mathsf{fi}
\end{array}
$$

By means of *tail fusion* more general forms are obtained, that is, for any function h this definition can be transformed easily into a tail-recursive definition for $h \circ F$, here called $F1$, by simply inserting an application of h in the base case:

$$
\begin{array}{lcl}
F1 \cdot x & = & \mathsf{if}\ \ \neg b \cdot x \ \ \rightarrow \ \ h \cdot x \\
 & & []\ \ \ b \cdot x \ \ \rightarrow \ \ F1 \cdot (f \cdot x) \\
 & & \mathsf{fi}
\end{array}
$$

We present a few theorems for the simple case first, after which we discuss applications to the more general case; finally, we illustrate the technique with two simple examples.

2.0 Three little theorems

For some fixed type X, boolean functions b and c on X, and functions f and g of type $X \to X$, we consider functions F and G, also of type $X \to X$, with the following tail-recursive definitions:

$$\begin{array}{lcl} F\cdot x & = & \mathsf{if}\ \neg b\cdot x \ \to\ x \\ & & [\!]\ \ b\cdot x \ \to\ F\cdot(f\cdot x) \\ & & \mathsf{fi} \end{array}$$

$$\begin{array}{lcl} G\cdot x & = & \mathsf{if}\ \neg c\cdot x \ \to\ x \\ & & [\!]\ \ c\cdot x \ \to\ G\cdot(g\cdot x) \\ & & \mathsf{fi} \end{array}$$

In the proof of the following theorem we assume that F is well-defined and we use mathematical induction on its domain. Formally, this presupposes the existence of a well-founded ordering $<$ on X with the property:

$$(\forall x ::\ b\cdot x \Rightarrow f\cdot x < x) \quad .$$

This guarantees that F is well-defined. That function G is also (sufficiently) well-defined then follows from the theorem, which is why we do not need to formulate this separately.

All function values of F satisfy $\neg b$, which is a property we shall need later.

Lemma 0: $(\forall x ::\ \neg b\cdot(F\cdot x))$.

proof: by straightforward mathematical induction on x.
□

Because generally F is not injective we cannot simply require G to be F's inverse, but we do have the following, weaker theorem. Fortunately, this theorem turns out to be quite useful.

Theorem 1: $(\forall x ::\ G\cdot(F\cdot x) = G\cdot x)$, provided that:

(9) $(\forall x ::\ b\cdot x \Rightarrow c\cdot(f\cdot x))$
(10) $(\forall x ::\ b\cdot x \Rightarrow g\cdot(f\cdot x) = x)$

proof: by mathematical induction on x:

$$
\begin{array}{ll}
 & G\cdot(F\cdot x) \\
= & \quad \{ \text{ case } \neg b\cdot x\colon \text{ definition of } F \ \} \\
 & G\cdot x \quad ,
\end{array}
$$

and:

$$
\begin{array}{ll}
 & G\cdot(F\cdot x) \\
= & \quad \{ \text{ case } b\cdot x\colon \text{ definition of } F \ \} \\
 & G\cdot(F\cdot(f\cdot x)) \\
= & \quad \{ \ f\cdot x<x\colon \text{ induction hypothesis } \} \\
 & G\cdot(f\cdot x) \\
= & \quad \{ \ b\cdot x\colon (9)\,, \text{ definition of } G \ \} \\
 & G\cdot(g\cdot(f\cdot x)) \\
= & \quad \{ \ b\cdot x\colon (10) \ \} \\
 & G\cdot x \quad .
\end{array}
$$

□

Corollary 2: $(\forall x :: \neg c\cdot x \Rightarrow G\cdot(F\cdot x) = x)$

□

In words: on the subset (of X) defined by $\neg c$ function G is the inverse of F.

* * *

Very often, functions f and g are defined by case analysis on their domain X. The following development takes this into account.

For some boolean functions β_i and γ_i on X, and functions φ_i and ψ_i of type $X \to X$, where i ranges over some finite set, we now assume that f and g satisfy:

(11) $(\forall i, x :: \ \beta_i\cdot x \Rightarrow f\cdot x = \varphi_i\cdot x)$
(12) $(\forall i, x :: \ \gamma_i\cdot x \Rightarrow g\cdot x = \psi_i\cdot x)$
(13) $(\forall x :: \ b\cdot x \Rightarrow (\exists i :: \beta_i\cdot x))$

This represents the case that f and g have been defined by case analysis. Notice that condition (13) is a usual requirement for a function defined by case analysis: on its domain of use –in this case: $b-$, at least one of its guards must be `true`. Surprisingly enough we do not need the corresponding condition for g in our proofs.

The following two little theorems provide the necessary refinements for the above premisses (9) and (10).

Theorem 3: $(\forall x ::\ b{\cdot}x \Rightarrow c{\cdot}(f{\cdot}x))$, provided that:

(14) $(\forall i, x ::\ \gamma_i{\cdot}x \Rightarrow c{\cdot}x)$
(15) $(\forall i, x ::\ \beta_i{\cdot}x \Rightarrow \gamma_i{\cdot}(\varphi_i{\cdot}x))$

proof: assuming $b{\cdot}x$ and, using (13), $\beta_i{\cdot}x$ we derive:

$$
\begin{array}{ll}
 & c{\cdot}(f{\cdot}x) \\
\Leftarrow & \quad \{\ (14)\ \} \\
 & \gamma_i{\cdot}(f{\cdot}x) \\
\equiv & \quad \{\ \beta_i{\cdot}x\colon (11)\ \} \\
 & \gamma_i{\cdot}(\varphi_i{\cdot}x) \\
\equiv & \quad \{\ \beta_i{\cdot}x\colon (15)\ \} \\
 & \mathsf{true}\ .
\end{array}
$$

□

Theorem 4: $(\forall x ::\ b{\cdot}x \Rightarrow g{\cdot}(f{\cdot}x) = x)$, provided that:

(15) $(\forall i, x ::\ \beta_i{\cdot}x \Rightarrow \gamma_i{\cdot}(\varphi_i{\cdot}x))$
(16) $(\forall i, x ::\ \beta_i{\cdot}x \Rightarrow \psi_i{\cdot}(\varphi_i{\cdot}x) = x)$

proof: assuming $b{\cdot}x$ and, using (13), $\beta_i{\cdot}x$ we derive:

$$
\begin{array}{ll}
 & g{\cdot}(f{\cdot}x) \\
= & \quad \{\ \beta_i{\cdot}x\colon (11)\ \} \\
 & g{\cdot}(\varphi_i{\cdot}x) \\
= & \quad \{\ \beta_i{\cdot}x\colon (15), (12)\ \} \\
 & \psi_i{\cdot}(\varphi_i{\cdot}x) \\
= & \quad \{\ \beta_i{\cdot}x\colon (16)\ \} \\
 & x\ .
\end{array}
$$

□

2.1 Pragmatics

A function like F is often used to define another function, H (say), in the following way:

$$H \;=\; h \circ F \circ p \quad ,$$

where h and p are functions such that p "wraps up" H's argument and h "unwraps" the result. For instance, F may be a generalization of H, and both its argument and its value may contain elements that are irrelevant for the special case in which it is used; if this is so, p *embeds* H's argument into the larger domain of F, whereas h *projects* F's value back onto the smaller range of H. (So, h and p provide the *interface* between the generalization and its use; in Section 2.2 we give examples.) Function H now has type $W \leftarrow V$, provided h has type $W \leftarrow X$ and p has type $X \leftarrow V$; in what follows dummy v ranges over V.

The inverse, if any, of function H must of course satisfy:

$$H^{-1} \;=\; p^{-1} \circ F^{-1} \circ h^{-1} \quad ,$$

so, H^{-1} exists if p^{-1}, F^{-1}, and h^{-1} exist. By Corollary 2, $G\cdot(F\cdot x) = x$, for all x satisfying $\neg c\cdot x$. If functions h and p have inverses k and q, then the inverse of H is function K, defined by:

$$K \;=\; q \circ G \circ k \quad .$$

Theorem 5: $(\forall v ::\; K\cdot(H\cdot v) \;=\; v)$, provided that:

(17) $H \;=\; h \circ F \circ p$
(18) $K \;=\; q \circ G \circ k$
(19) $(\forall v ::\; \neg c\cdot(p\cdot v)\,)$
(20) $(\forall v ::\; q\cdot(p\cdot v) = v\,)$
(21) $(\forall x ::\; \neg b\cdot x \Rightarrow k\cdot(h\cdot x) = x\,)$

proof: by straightforward calculation:

$$\begin{array}{ll} & K\cdot(H\cdot v) \\ = & \quad \{\ (17) \text{ and } (18)\ \} \\ & q\cdot(G\cdot(k\cdot(h\cdot(F\cdot(p\cdot v))))) \\ = & \quad \{\ (21)\text{, using lemma 0}\ \} \\ & q\cdot(G\cdot(F\cdot(p\cdot v))) \\ = & \quad \{\ \text{Corollary 2, using } (19)\ \} \end{array}$$

$$
\begin{array}{ll}
 & q\cdot(p\cdot v) \\
= & \quad \{\ (20)\ \} \\
 & v\ .
\end{array}
$$

□

By means of tail fusion, functions h and q can be incorporated into the definitions of F and G. That is, we define functions F' and G' by:

$$
\begin{array}{lcl}
F' & = & h \circ F \\
G' & = & q \circ G
\end{array}
$$

Then, F' and G' admit tail-recursive definitions as well, and we obtain:

$$H^{-1} \ = \ K \ ,$$

where:

$$
\begin{array}{lcl}
H & = & F' \circ p \\
F'\cdot x & = & \mathsf{if}\ \neg b\cdot x \ \rightarrow \ h\cdot x \\
 & & [] \quad b\cdot x \ \rightarrow \ F'\cdot(f\cdot x) \\
 & & \mathsf{fi} \\
K & = & G' \circ k \\
G'\cdot x & = & \mathsf{if}\ \neg c\cdot x \ \rightarrow \ q\cdot x \\
 & & [] \quad c\cdot x \ \rightarrow \ G'\cdot(g\cdot x) \\
 & & \mathsf{fi}
\end{array}
$$

* * *

In practical applications functions often have more than one parameter. These parameters can be treated as a single parameter by *tupling* them. For example, for function F with 3 parameters we can introduce a new function F' with:

$$F\cdot x\cdot y\cdot z \ = \ F'\cdot\langle x, y, z\rangle \ .$$

2.2 Two simple examples

As a first example we consider the well-known tail-recursive definition for function rev, such that $R\cdot x\cdot y = rev\cdot x +\!\!+ y$:

$$
\begin{array}{lcl}
rev\cdot x & = & R\cdot x\cdot[\,] \\
R\cdot[\,]\cdot y & = & y \\
R\cdot(b \triangleright x)\cdot y & = & R\cdot x\cdot(b \triangleright y)
\end{array}
$$

In terms of the nomenclature of Theorem 5 this amounts to (with $H := rev$):

$$rev \;=\; h \circ F \circ p \quad \text{, with:}$$

$$\begin{array}{lcl} p{\cdot}x & = & \langle x, [\,]\,\rangle \\ F{\cdot}\langle\,[\,], y\rangle & = & \langle\,[\,], y\rangle \\ F{\cdot}\langle b \triangleright x, y\rangle & = & F{\cdot}\langle x, b \triangleright y\rangle \\ h{\cdot}\langle x, y\rangle & = & y \end{array}$$

$$rev^{-1} \;=\; q \circ G \circ k \quad \text{, with:}$$

$$\begin{array}{lcl} k{\cdot}y & = & \langle\,[\,], y\rangle \\ G{\cdot}\langle x, [\,]\,\rangle & = & \langle x, [\,]\,\rangle \\ G{\cdot}\langle x, b \triangleright y\rangle & = & F{\cdot}\langle b \triangleright x, y\rangle \\ q{\cdot}\langle x, y\rangle & = & x \end{array}$$

In this case, functions G and F only differ in the order of the elements of the pairs: $F{\cdot}\langle x, y\rangle = G{\cdot}\langle y, x\rangle$. Hence, it is easily verified that: $h \circ F \circ p = q \circ G \circ k$, from which we conclude: $rev = rev^{-1}$.

* * *

As a second example we recall the definition of function L from Section 0:

$$\begin{array}{lcl} L{\cdot}s & = & F{\cdot}[\,]{\cdot}[\,s\,] \\ F{\cdot}x{\cdot}[\,] & = & x \\ F{\cdot}x{\cdot}(\langle\,\rangle \triangleright p) & = & F{\cdot}(x \triangleleft 0){\cdot}p \\ F{\cdot}x{\cdot}(\langle s, t\rangle \triangleright p) & = & F{\cdot}(x \triangleleft 1){\cdot}(s \triangleright t \triangleright p) \end{array}$$

Due to the parameter patterns, rules (1) through (3) can be viewed as *rewrite rules*; by reading them from right to left, we obtain a new function P, defined in terms of a new function G –explanation follows–:

$$\begin{array}{lcl} P{\cdot}x & = & G{\cdot}x{\cdot}[\,] \\ G{\cdot}[\,]{\cdot}[\,s\,] & = & s \\ G{\cdot}(x \triangleleft 0){\cdot}p & = & G{\cdot}x{\cdot}(\langle\,\rangle \triangleright p) \\ G{\cdot}(x \triangleleft 1){\cdot}(s \triangleright t \triangleright p) & = & G{\cdot}x{\cdot}(\langle s, t\rangle \triangleright p) \end{array}$$

Tja, hoe leg ik dat uit? De eerste twee regels in de definitie van G zijn verkregen uit de eerste twee regels in de definitie van F, maar dan wel verwisseld. De laatste twee regels in de definitie van G zijn "gewoon" verkregen door de corresponderende regels in de definitie van F van links naar rechts te lezen. Als ik het netjes wil doen moet ik laten zien dat deze *symbol dynamics* een correcte toepassing is van het voorafgaande stukje theorie. Hier moet ik nog even naar kijken.

3 Heap reconstruction

3.0 Introduction

A *heap* is a binary tree whose nodes have been labelled with integers; moreover, in a heap these integers are subjected to a so-called *heap condition* expressing that the integer in each node is the minimum of all integers in the subtree with that node as its root.

We consider the problem of reconstructing heaps from their in-order traversals. We do so by program inversion: it is easier to define the function mapping trees onto their in-order traversals, and the mapping of in-order traversals onto heaps is just the inverse of this function. For binary trees in general this inverse is not unique, because different trees may have the same in-order traversals, but we will see that every heap has a unique list as its in-order traversal.

This problem has also been discussed by other authors [9, 5, 10], also by means of program inversion, but in an imperative setting. Here we use functional programming only, but the final program happens to be tail-recursive: if so desired, it can be transscribed into an (iterative) sequential program. Of the imperative solutions the treatment in [8] best resembles the derivation given here, but they only consider the problem of reconstructing trees, not heaps, from their in-order traversals. In the next section we extend our solution to a full-fledged expression parser.

3.1 Trees, traversals, and parsers

We consider a datatype T of labelled binary trees. By means of the heap condition we shall restrict T to heaps, but we postpone this for a while. Type T is defined recursively as follows, where B denotes the type of the values in the nodes – the "labels" –:

$$\begin{aligned} \langle\,\rangle &\in T \\ \langle r, b, s\rangle &\in T \quad \text{, for all } r, s \text{ in } T \text{ and } b \text{ in } B \ . \end{aligned}$$

Here $\langle\,\rangle$ represents an (unlabelled) *leaf* whereas $\langle r, b, s\rangle$ is a *composite* tree with subtrees r and s and with b for the label of its root; thus, only the tree's *internal* nodes are labelled.

Throughout this section dummies b, c denote elements of B, r, s, t denote trees, and x, y, z range over $\mathcal{L}_*(B)$.

The in-order traversal of a tree r is the list $L{\cdot}r$, where function L, of type $T \to \mathcal{L}_*(B)$, is defined by:

$$\begin{array}{lcl} L{\cdot}\langle\,\rangle & = & [\,] \\ L{\cdot}\langle r, b, s\rangle & = & L{\cdot}r +\!\!+ [b] +\!\!+ L{\cdot}s \end{array}$$

A *parser* is a function that reconstructs a tree from its in-order traversal. Because different trees may have the same in-order traversals, function L is not injective and it has no inverse. On the other hand, every list of integers is the in-order traversal of at least one tree; so, L is surjective and many functions exist for which L itself is an inverse. Therefore, we specify a parser to be a function P that has L as its inverse:

$$L{\cdot}(P{\cdot}x) \;=\; x \quad , \quad \text{for all } x \in \mathcal{L}_*(B) \quad .$$

This specification admits many solutions, but when we introduce the heap condition, P turns out to be unique and L turns out to be a bijection from the heaps to $\mathcal{L}_*(B)$. The problem now is to design an efficient definition for P.

* * *

A naive solution is obtained by a straightforward application of the inversion technique from Section 1; that is, inspired by the shape of the right-hand sides of L's definition we obtain as a tentative definition for P:

$$\begin{array}{lcl} P{\cdot}[\,] & = & \langle\,\rangle \\ P{\cdot}(x +\!\!+ [b] +\!\!+ y) & = & \langle P{\cdot}x\,, [b]\,, P{\cdot}y\rangle \end{array}$$

The pattern $x +\!\!+ [b] +\!\!+ y$ in this definition can be eliminated –the equation $x, b, y\colon x +\!\!+ [b] +\!\!+ y \;=\; z$ can be solved– but this yields an inefficient solution. By means of the instantiation $x := [\,]$ we could, of course, obtain as an efficient but very special case:

$$P{\cdot}(b \triangleright y) \;=\; \langle\langle\,\rangle, b, P{\cdot}y\rangle \quad ,$$

but this must be rejected as too premature a restriction of our freedom, in view of the, as yet unknown, heap condition.

* * *

Therefore, we transform the definition of L into a more manageable form, by generalizing it. By looking at formulae like:

$$\begin{array}{lcl} L{\cdot}r & = & L{\cdot}r \\ L{\cdot}\langle r,b,s\rangle & = & L{\cdot}r +\!\!+ [b] +\!\!+ L{\cdot}s \\ L{\cdot}\langle\langle r,b,s\rangle,c,t\rangle & = & L{\cdot}r +\!\!+ [b] +\!\!+ L{\cdot}s +\!\!+ [c] +\!\!+ L{\cdot}t \end{array}$$

and so on ,

we observe that, apart from the first $L{\cdot}r$, elements of B and trees occur in *pairs*. To capture the underlying pattern we introduce a binary operator $\oplus$ defined by:

$$x \oplus \langle b,t\rangle \;=\; x +\!\!+ [b] +\!\!+ L{\cdot}t \quad ,$$

as a result of which we have, for example:

$$\begin{array}{lcl} L{\cdot}r & = & L{\cdot}r \\ L{\cdot}\langle r,b,s\rangle & = & L{\cdot}r \oplus \langle b,s\rangle \\ L{\cdot}\langle\langle r,b,s\rangle,c,t\rangle & = & L{\cdot}r \oplus \langle b,s\rangle \oplus \langle c,t\rangle \end{array}$$

With $\otimes$ for the right-folded version of $\oplus$ we obtain:

$$\begin{array}{lcl} L{\cdot}r & = & L{\cdot}r \otimes [\,] \\ L{\cdot}\langle r,b,s\rangle & = & L{\cdot}r \otimes [\,\langle b,s\rangle\,] \\ L{\cdot}\langle\langle r,b,s\rangle,c,t\rangle & = & L{\cdot}r \otimes [\,\langle b,s\rangle,\langle c,t\rangle\,] \end{array} \tag{22}$$

Because $+\!\!+$ associates with $\oplus$ –that is: $x +\!\!+ (y \oplus p) = (x +\!\!+ y) \oplus p$– it associates with $\otimes$ as well: $\otimes$ *inherits* this property from $\oplus$.

Thus, with ps a list of pairs of the shape $\langle b,s\rangle$, the expression $L{\cdot}r \otimes ps$ is a meaningful generalization of $L{\cdot}r$, for we have $L{\cdot}r = L{\cdot}r \otimes [\,]$. From the definition of L and the properties of $\otimes$ and $\oplus$ we obtain the following recurrence relations for $L{\cdot}r \otimes ps$:

$$\begin{array}{lcl} L{\cdot}\langle\,\rangle \otimes [\,] & = & [\,] \\ L{\cdot}\langle\,\rangle \otimes (\langle b,s\rangle \triangleright ps) & = & [b] +\!\!+ (L{\cdot}s \otimes ps) \\ L{\cdot}\langle r,b,s\rangle \otimes ps & = & L{\cdot}r \otimes (\langle b,s\rangle \triangleright ps) \end{array} \tag{23} \tag{24}$$

The first two of these rules would lend themselves very well for systematic inversion, if it were not for the third, tail-recursive rule. For the sake of homogeneity we now carry the transformation one step further, to obtain a completely tail-recursive definition. This is simple as we only have to do away with the operation $[b]+\!\!+$, and $+\!\!+$ is associative. So, we introduce a function G with specification:

$$G{\cdot}x{\cdot}r{\cdot}ps \;=\; x +\!\!+ L{\cdot}r \otimes ps \quad , \quad \text{for all } x,r,ps \quad .$$

This yields:

$$L{\cdot}r \;=\; G{\cdot}[\,]{\cdot}r{\cdot}[\,] \quad ,$$

In $G{\cdot}x{\cdot}r{\cdot}ps$ variable x is an integer list, r is a tree, and ps is a list of pairs $\langle b, s\rangle$, for integer b and tree s. A (tail-recursive) definition for G is:

$$\begin{array}{lll} & G{\cdot}x{\cdot}\langle\,\rangle{\cdot}[\,] & = \; x \\ (25) & G{\cdot}x{\cdot}\langle\,\rangle{\cdot}(\langle b, s\rangle \triangleright ps) & = \; G{\cdot}(x \triangleleft b){\cdot}s{\cdot}ps \\ (26) & G{\cdot}x{\cdot}\langle s, c, t\rangle{\cdot}ps & = \; G{\cdot}x{\cdot}s{\cdot}(\langle c, t\rangle \triangleright ps) \end{array}$$

The inversion technique for tail-recursive definitions requires that the right-hand sides of (25) and (26) can be distinguished. In the right-hand expression $G{\cdot}(x \triangleleft b){\cdot}s{\cdot}ps$ the first argument of G is a nonempty list $x \triangleleft b$; to make explicit what we need, we now refine rule (26) by treating the instances $x := [\,]$ and $x := x \triangleleft b$ as separate cases. Similarly, we refine rule (25) by separating $ps := [\,]$ and $ps := \langle c, t\rangle \triangleright ps$:

$$\begin{array}{lll} & G{\cdot}x{\cdot}\langle\,\rangle{\cdot}[\,] & = \; x \\ & G{\cdot}x{\cdot}\langle\,\rangle{\cdot}(\langle b, s\rangle \triangleright [\,]) & = \; G{\cdot}(x \triangleleft b){\cdot}s{\cdot}[\,] \\ (27) & G{\cdot}x{\cdot}\langle\,\rangle{\cdot}(\langle b, s\rangle \triangleright \langle c, t\rangle \triangleright ps) & = \; G{\cdot}(x \triangleleft b){\cdot}s{\cdot}(\langle c, t\rangle \triangleright ps) \\ (28) & G{\cdot}(x \triangleleft b){\cdot}\langle s, c, t\rangle{\cdot}ps & = \; G{\cdot}(x \triangleleft b){\cdot}s{\cdot}(\langle c, t\rangle \triangleright ps) \\ & G{\cdot}[\,]{\cdot}\langle s, c, t\rangle{\cdot}ps & = \; G{\cdot}[\,]{\cdot}s{\cdot}(\langle c, t\rangle \triangleright ps) \end{array}$$

In this version of the definition the only "conflict" occurs between the right-hand sides of rules (27) and (28), so all we must do now is find conditions to distinguish them; once we have obtained such conditions we can apply the Inversion Theorem. For this purpose we need the heap condition.

3.2 The heap condition

We now assume that set B is totally ordered; we denote this ordering by means of the common symbols "$\leq$" (and "$<$"); so, either $b \leq c$ or $c \leq b$ (or both) holds for all $b, c \in B$.

For $b \in B$ and tree s we use $b \leq s$ (and similarly $b < s$) as a shorthand[1] for $(\forall c : c \in s : b \leq c)$, that is, we have:

$$\begin{array}{lll} b \leq \langle\,\rangle & \equiv & \mathsf{true} \\ b \leq \langle s, c, t\rangle & \equiv & b \leq s \;\wedge\; b \leq c \;\wedge\; b \leq t \end{array}$$

[1] This convention is not without danger! For example, this new relation is not transitive.

Tree r is a heap if it satisfies $H{\cdot}r$, where predicate H is defined by:

$$\begin{array}{lcl} H{\cdot}\langle\,\rangle & \equiv & \mathsf{true} \\ H{\cdot}\langle r,b,s\rangle & \equiv & H{\cdot}r \;\wedge\; H{\cdot}s \;\wedge\; b\leq r \;\wedge\; b<s \end{array}$$

A *parser* now is a function that reconstructs a heap –instead of just a tree– from its in-order traversal; as we will see, this heap happens to be unique. So, a parser is now a function P, of type $\mathcal{L}_*(B)\rightarrow T$, with specification:

$$L{\cdot}(P{\cdot}x)=x \;\wedge\; H{\cdot}(P{\cdot}x) \quad , \quad \text{for all } x \quad .$$

3.3 The inversion

Now we investigate how the properties of H can exploited to distinguish the above rules (27) and (28). This is not difficult, but it is somewhat tedious.

First of all, the definition of H is such that all subtrees of a heap are heaps themselves. As a consequence, because we are only interested in $L{\cdot}r$ for heaps r, it is a precondition of the functions L and G that all variables of type T in our formulae are heaps; we shall use this tacitly.

Secondly, we recall property (26):

$$G{\cdot}x{\cdot}\langle s,c,t\rangle{\cdot}ps \;=\; G{\cdot}x{\cdot}s{\cdot}(\langle c,t\rangle \triangleright ps) \quad .$$

The precondition $H{\cdot}\langle s,c,t\rangle$ implies $c<t$. Therefore, the pair $\langle c,t\rangle$ in the right-hand side satisfies $c<t$ and, hence, it is a precondition of $G{\cdot}x{\cdot}r{\cdot}ps$ that *every* pair $\langle c,t\rangle$ in the list ps satisfies $c<t$. This, in turn, is useful when we reconsider rule (25):

$$G{\cdot}x{\cdot}\langle\,\rangle{\cdot}(\langle b,s\rangle \triangleright ps) \;=\; G{\cdot}(x\triangleleft b){\cdot}s{\cdot}ps \quad ,$$

from which we infer that G also has as precondition that every (relevant) expression of the shape $G{\cdot}(x\triangleleft b){\cdot}s{\cdot}ps$ satisfies $b<s$. In this way we obtain for rule (28):

$$(28) \qquad G{\cdot}(x\triangleleft b){\cdot}\langle s,c,t\rangle{\cdot}ps \;=\; G{\cdot}(x\triangleleft b){\cdot}s{\cdot}(\langle c,t\rangle \triangleright ps) \quad , \mathsf{with}: \;\; b<c$$

Thirdly and finally, by applying (26) twice, with $s:=\langle r,b,s\rangle$, we obtain:

$$G{\cdot}x{\cdot}\langle\,\langle r,b,s\rangle,c,t\,\rangle{\cdot}ps \;=\; G{\cdot}x{\cdot}r{\cdot}(\langle b,s\rangle \triangleright \langle c,t\rangle \triangleright ps) \quad .$$

Here $H{\cdot}\langle\,\langle r,b,s\rangle,c,t\,\rangle$ implies that $c\leq b$, which gives rise to the additional precondition that all *successive* pairs $\langle b,s\rangle$ and $\langle c,t\rangle$ in ps satify $c\leq b$. From this we conclude for rule (27):

$$(27)\quad G{\cdot}x{\cdot}\langle\,\rangle{\cdot}(\langle b,s\rangle \triangleright \langle c,t\rangle \triangleright ps) \;=\; G{\cdot}(x \triangleleft b){\cdot}s{\cdot}(\langle c,t\rangle \triangleright ps) \quad ,\ \mathsf{with}:\ c \leq b$$

The two conditions $b<c$ and $c\leq b$ are each other's complement, so they represent disjoint cases and this case analysis is exhaustive. By rewriting the rules from the definition of G from right to left we now obtain a definition for a new function F:

$$\begin{array}{lll}
 & F{\cdot}[\,]{\cdot}s{\cdot}[\,] = s & \\
(29) & F{\cdot}(x \triangleleft b){\cdot}s{\cdot}[\,] = F{\cdot}x{\cdot}\langle\,\rangle{\cdot}(\langle b,s\rangle \triangleright [\,]) & \\
(30) & F{\cdot}(x \triangleleft b){\cdot}s{\cdot}(\langle c,t\rangle \triangleright ps) = F{\cdot}x{\cdot}\langle\,\rangle{\cdot}(\langle b,s\rangle \triangleright \langle c,t\rangle \triangleright ps) & ,\ \mathsf{if}\ c\leq b \\
(31) & F{\cdot}(x \triangleleft b){\cdot}s{\cdot}(\langle c,t\rangle \triangleright ps) = F{\cdot}(x \triangleleft b){\cdot}\langle s,c,t\rangle{\cdot}ps & ,\ \mathsf{if}\ b<c \\
(32) & F{\cdot}[\,]{\cdot}s{\cdot}(\langle c,t\rangle \triangleright ps) = F{\cdot}[\,]{\cdot}\langle s,c,t\rangle{\cdot}ps &
\end{array}$$

According to Theorem 5 we now have $G{\cdot}[\,]{\cdot}(F{\cdot}x{\cdot}\langle\,\rangle{\cdot}[\,]){\cdot}[\,] = x$. Hence, because $L{\cdot}r = G{\cdot}[\,]{\cdot}r{\cdot}[\,]$, we define:

$$P{\cdot}x \;=\; F{\cdot}x{\cdot}\langle\,\rangle{\cdot}[\,] \quad ,$$

and we obtain $L{\cdot}(P{\cdot}x) = x$, as required.

4 An expression parser

In their simplests possible form, expressions are composed from "values" and "binary operators". The trees we studied in the previous section are a very abstract rendering of expressions, where leaves – instances of the empty tree $\langle\,\rangle$ – represent the values and where labels of internal nodes – the values b in $\langle r,b,s\rangle$ – represent the *priorities* of the binary operators. Thus, a heap reflects the structure of an expression according to the priorities of its operators.

This can be used as the basis for the construction of a full-fledged expression parser. We will do so in two steps: first, we introduce the values, by labelling the leaves as well. Second, we introduce parentheses to represent (as linear expressions) those trees that are not heaps: because of the priorities of the operators, an expression like $2+3*5$ represents the heap $\langle\langle 2\rangle,+,\langle\langle 3\rangle,*,\langle 5\rangle\rangle\rangle$, and the non-heap $\langle\langle 2\rangle,*,\langle\langle 3\rangle,+,\langle 5\rangle\rangle\rangle$ can only be represented by the parenthesized expression $2*(3+5)$.

4.0 Values

To account for values, which will be just integers, we *redefine* the type of trees in the following way:

$$\begin{array}{rl} \langle n \rangle \in T & \text{, for all } n \text{ in } \mathsf{Int} \text{ .} \\ \langle r, b, s \rangle \in T & \text{, for all } r, s \text{ in } T \text{ and } b \text{ in } B \text{ .} \end{array}$$

A tree $\langle n \rangle$ now is a leaf labelled with value n; for our purposes, the type of n, integer, actually is irrelevant, except that we assume it to be disjoint from B: values and operators must be distinguishable, and here the binary operators are (represented by) elements of B. Hence, we assume $\mathsf{Int} \cap B = \phi$.

To avoid unnecessary case analysis we retain the empty tree, $\langle\,\rangle$, but the subtrees in $\langle r, b, s \rangle$ will never be $\langle\,\rangle$. Therefore, we do not include $\langle\,\rangle$ in type T be add it "as an afterthought", so to speak: in what follows parameters s, t have type $T \cup \{\langle\,\rangle\}$.

The definition of function L is now extended accordingly, thus:

$$\begin{array}{lcl} L{\cdot}\langle\,\rangle & = & [\,] \\ L{\cdot}\langle n \rangle & = & [n] \\ L{\cdot}\langle r, b, s \rangle & = & L{\cdot}r +\!\!+ [b] +\!\!+ L{\cdot}s \end{array}$$

The tail-recursive definition of our generalized function G now becomes:

$$\begin{array}{lll} & G{\cdot}x{\cdot}\langle\,\rangle{\cdot}[\,] & = x \\ & G{\cdot}x{\cdot}\langle\,\rangle{\cdot}(\langle b, s \rangle \triangleright [\,]) & = G{\cdot}(x \triangleleft b){\cdot}s{\cdot}[\,] \\ & G{\cdot}x{\cdot}\langle\,\rangle{\cdot}(\langle b, s \rangle \triangleright \langle c, t \rangle \triangleright ps) & = G{\cdot}(x \triangleleft b){\cdot}s{\cdot}(\langle c, t \rangle \triangleright ps) \\ & G{\cdot}(x \triangleleft b){\cdot}\langle s, c, t \rangle{\cdot}ps & = G{\cdot}(x \triangleleft b){\cdot}s{\cdot}(\langle c, t \rangle \triangleright ps) \\ (33) & G{\cdot}(x \triangleleft n){\cdot}\langle s, c, t \rangle{\cdot}ps & = G{\cdot}(x \triangleleft n){\cdot}s{\cdot}(\langle c, t \rangle \triangleright ps) \\ & G{\cdot}[\,]{\cdot}\langle s, c, t \rangle{\cdot}ps & = G{\cdot}[\,]{\cdot}s{\cdot}(\langle c, t \rangle \triangleright ps) \\ (34) & G{\cdot}x{\cdot}\langle n \rangle{\cdot}ps & = G{\cdot}(x \triangleleft n){\cdot}\langle\,\rangle{\cdot}ps \end{array}$$

Rule (34) represents the new kind of leaves; the use of $\langle\,\rangle$ in its right-hand side is possible because we have retained the empty tree. Rule (33) emerges as an additional variant of the case $x \neq [\,]$, as a consequence of rule (34). The right-hand sides of (33) and (34) are distinguishable from all other right-hand sides, because we have required n and b to be distinguishable; in addition, they are mutually distinguishable because, in (33), $s \neq \langle\,\rangle$. Hence, the inversion of this definition of G poses no additional problems.

4.1 Parenthesized expressions

On the one hand, not every parenthesized expression represents a heap; on the other hand, we wish to retain the heap condition to reflect the priorities of the binary operators. Therefore, we *weaken* the heap condition by exempting some subtrees from it. We do so by extending the type of trees with an additional class of elements, of the shape $\langle s \rangle$, for tree s:

$$\begin{array}{rl} \langle n \rangle \in T & \text{, for all } n \text{ in } \mathsf{Int} \ . \\ \langle s \rangle \in T & \text{, for all } s \text{ in } T \ . \\ \langle r, b, s \rangle \in T & \text{, for all } r, s \text{ in } T \text{ and } b \text{ in } B \ . \end{array}$$

So, $\langle s \rangle$ is a *leaf* with a tree s for its value: s is "wrapped up", so to speak. As in the previous case, the datatype of interest now is $T \cup \{ \langle\,\rangle \}$.

For this extended type of trees we now redefine the heap condition H in such a way that trees s in $\langle s \rangle$ are ignored, as becomes apparent in the new definition of $c \leq \langle s \rangle$; notice, however, that s itself still must be a heap:

$$\begin{array}{lcl} H \cdot \langle\,\rangle & \equiv & \mathsf{true} \\ H \cdot \langle n \rangle & \equiv & \mathsf{true} \\ H \cdot \langle s \rangle & \equiv & H \cdot s \\ H \cdot \langle r, b, s \rangle & \equiv & H \cdot r \ \wedge \ H \cdot s \ \wedge \ b \leq r \ \wedge \ b < s \end{array}$$

$$\begin{array}{lcl} c \leq \langle\,\rangle & \equiv & \mathsf{true} \\ c \leq \langle n \rangle & \equiv & \mathsf{true} \\ c \leq \langle s \rangle & \equiv & \mathsf{true} \\ c \leq \langle r, b, s \rangle & \equiv & c \leq r \ \wedge \ c \leq b \ \wedge \ c \leq s \end{array}$$

Again, the definition of function L must be extended accordingly, as follows. To avoid confusion with (the symbols in) our own expressions we use constants $\langle$ and $\rangle$ as symbols for the opening and closing parentheses:

$$\begin{array}{lcl} L \cdot \langle\,\rangle & = & [\,] \\ L \cdot \langle n \rangle & = & [n] \\ L \cdot \langle s \rangle & = & [\langle\,] +\!\!+ L \cdot s +\!\!+ [\,\rangle] \\ L \cdot \langle r, b, s \rangle & = & L \cdot r +\!\!+ [b] +\!\!+ L \cdot s \end{array}$$

From Section 3.1 we recall that function G is a generalization of L, specified by[2]:

$$G \cdot x \cdot s \cdot ps \ = \ x +\!\!+ L \cdot s \otimes ps \ ,$$

where $\otimes$ is folded $\oplus$ and where $\oplus$ is defined by:

$$x \oplus \langle b, t \rangle \ = \ x +\!\!+ [b] +\!\!+ L \cdot t \ .$$

To stay within this pattern as much as possible, so as to avoid unnecessary case analysis, we apply a *coding trick* to represent $\rangle$, in the following derivation for the case $G \cdot x \cdot \langle s \rangle \cdot ps$:

[2] remember that $+\!\!+$ associates with $\otimes$, hence the absence of parentheses.

$$
\begin{array}{ll}
 & G{\cdot}x{\cdot}\langle s\rangle{\cdot}ps \\
= & \quad \{ \text{ specification of } G \ \} \\
 & x +\!\!+ L{\cdot}\langle s\rangle \otimes ps \\
= & \quad \{ \text{ definition of } L \ \} \\
 & x +\!\!+ [\langle\!\!\!\cdot] +\!\!+ L{\cdot}s +\!\!+ [\cdot\!\!\!\rangle] \otimes ps \\
= & \quad \{ +\!\!+ \text{ and } \lhd \ \} \\
 & (x \lhd \langle\!\!\!\cdot) +\!\!+ L{\cdot}s +\!\!+ [\cdot\!\!\!\rangle] \otimes ps \\
= & \quad \{ \text{ coding trick: introduce } [\,] \text{ and apply the definition of } L \ \} \\
 & (x \lhd \langle\!\!\!\cdot) +\!\!+ L{\cdot}s +\!\!+ [\cdot\!\!\!\rangle] +\!\!+ L{\cdot}\langle\,\rangle \otimes ps \\
= & \quad \{ \text{ definition of } \oplus \text{ and } \otimes \ \} \\
 & (x \lhd \langle\!\!\!\cdot) +\!\!+ L{\cdot}s \otimes (\langle\cdot\!\!\!\rangle, \langle\,\rangle \rangle \triangleright ps) \\
= & \quad \{ \text{ specification of } G, \text{ by Ind. Hyp. } \} \\
 & G{\cdot}(x \lhd \langle\!\!\!\cdot){\cdot}s{\cdot}(\langle\cdot\!\!\!\rangle, \langle\,\rangle \rangle \triangleright ps) \quad .
\end{array}
$$

Thus, with this addition we obtain a new definition for G:

$$
\begin{array}{lll}
 & G{\cdot}x{\cdot}\langle\,\rangle{\cdot}[\,] & = \ x \\
 & G{\cdot}x{\cdot}\langle\,\rangle{\cdot}(\langle b,s\rangle \triangleright [\,]) & = \ G{\cdot}(x \lhd b){\cdot}s{\cdot}[\,] \\
(35) & G{\cdot}x{\cdot}\langle\,\rangle{\cdot}(\langle b,s\rangle \triangleright \langle c,t\rangle \triangleright ps) & = \ G{\cdot}(x \lhd b){\cdot}s{\cdot}(\langle c,t\rangle \triangleright ps) \\
(36) & G{\cdot}(x \lhd b){\cdot}\langle s,c,t\rangle{\cdot}ps & = \ G{\cdot}(x \lhd b){\cdot}s{\cdot}(\langle c,t\rangle \triangleright ps) \\
 & G{\cdot}(x \lhd n){\cdot}\langle s,c,t\rangle{\cdot}ps & = \ G{\cdot}(x \lhd n){\cdot}s{\cdot}(\langle c,t\rangle \triangleright ps) \\
 & G{\cdot}[\,]{\cdot}\langle s,c,t\rangle{\cdot}ps & = \ G{\cdot}[\,]{\cdot}s{\cdot}(\langle c,t\rangle \triangleright ps) \\
 & G{\cdot}x{\cdot}\langle n\rangle{\cdot}ps & = \ G{\cdot}(x \lhd n){\cdot}\langle\,\rangle{\cdot}ps \\
(37) & G{\cdot}x{\cdot}\langle s\rangle{\cdot}ps & = \ G{\cdot}(x \lhd \langle\!\!\!\cdot){\cdot}s{\cdot}(\langle\cdot\!\!\!\rangle, \langle\,\rangle \rangle \triangleright ps)
\end{array}
$$

* * *

We now consider the inversion of this definition. As a result of the coding trick, pairs of the shape $\langle\cdot\!\!\!\rangle, t\rangle$ may occur in list ps, but pairs of the shape $\langle\langle\!\!\!\cdot, t\rangle$ will not occur in ps: this is a precondition of G. In addition, it is a precondition of G that operators c in trees $\langle s,c,t\rangle$ are neither $\langle\!\!\!\cdot$ nor $\cdot\!\!\!\rangle$.

The right-hand side of the new rule (37) in G's definition is (potentially) in conflict with the right-hand sides of rules (35) and (36) only. For rule (35) this conflict is resolved because, due to G's precondition, we have $b \neq \langle\!\!\!\cdot$; for (36) the conflict is resolved because $c \neq \cdot\!\!\!\rangle$. Apparently, the guard to distinguish the new rule (37) from the old ones (35) and (36) is $b = \langle\!\!\!\cdot \ \wedge \ c = \cdot\!\!\!\rangle$.

Thus, without further difficulties we obtain as extended definition for our inverse function F:

$$
\begin{array}{lcl}
F\cdot[\,]\cdot s\cdot[\,] & = & s \\
F\cdot(x\triangleleft b)\cdot s\cdot[\,] & = & F\cdot x\cdot\langle\,\rangle\cdot(\langle b,s\rangle\triangleright[\,]) \\
F\cdot(x\triangleleft b)\cdot s\cdot(\langle c,t\rangle\triangleright ps) & = & \\
\quad \mathsf{if}\ \ b\neq\langle\ \wedge\ c\leq b & \rightarrow & F\cdot x\cdot\langle\,\rangle\cdot(\langle b,s\rangle\triangleright\langle c,t\rangle\triangleright ps) \\
\quad [\!]\ \ c\neq\rangle\ \wedge\ b<c & \rightarrow & F\cdot(x\triangleleft b)\cdot\langle s,c,t\rangle\cdot ps \\
\quad [\!]\ \ b=\langle\ \wedge\ c=\rangle & \rightarrow & F\cdot x\cdot\langle s\rangle\cdot ps \\
\quad \mathsf{fi} & & \\
F\cdot(x\triangleleft n)\cdot\langle\,\rangle\cdot ps & = & F\cdot(x\triangleleft n)\cdot\langle n\rangle\cdot ps \\
F\cdot(x\triangleleft n)\cdot s\cdot(\langle c,t\rangle\triangleright ps) & = & F\cdot(x\triangleleft n)\cdot\langle s,c,t\rangle\cdot ps\ \ ,\ \ \mathsf{if}\ s\neq\langle\,\rangle \\
F\cdot[\,]\cdot s\cdot(\langle c,t\rangle\triangleright ps) & = & F\cdot[\,]\cdot\langle s,c,t\rangle\cdot ps
\end{array}
$$

4.2 Tree homomorphisms

A tree homomorphism is a function on trees with the property that the functions's value for a composite tree is defined in terms of the (same) function's values for its subtrees only, that is, independent of other properties of those subtrees. As a result, if the (corresponding) subtrees in two composite trees have the same function value, then so do the composite trees themselves.

Examples of tree homomorphisms are: our function L, and (usually) all functions defining the value of an expression in terms of the values of its subexpressions. The heap condition H, on the other hand, is not a tree homomorphism. A *code generator* in a compiler also is a tree homomorphism, provided the code generated for a composite construct is composed from pieces of code for that construct's constituents.

In the above definition for our function F, trees are composed from other trees, for example in the rule

$$F\cdot(x\triangleleft n)\cdot s\cdot(\langle c,t\rangle\triangleright ps) \;=\; F\cdot(x\triangleleft n)\cdot\langle s,c,t\rangle\cdot ps\ \ ,$$

where $\langle s,c,t\rangle$ (in the right-hand side) is composed from s and the pair $\langle c,t\rangle$ (in the left-hand side). In this definition tree construction is the *only* operation on trees. As a result, the definition of F is compositional itself; hence, it can be easily fused with any tree homomorphism, giving rise to a function mapping the linear representation of a tree to the value of the homomorphism applied to that tree: this transformation amounts to nothing more than, first, replacing all variables of type T by variables of the new type (of the homomorphism's values) and, second, replacing the tree constructors by the corresponding operators of the homomorphism.

For example, if the homomorphism is a code generator our parser becomes a true compiler!

5 Epilogue

Wat heb ik hier nou van geleerd? Enerzijds is het een opwindende exercitie, en leuk om te observeren dat een complete compiler zo compact kan worden genoteerd. Anderzijds is het toch nogal bewerkelijk geweest. Maar misschien mag ik, met het citaat "There is no royal road to geometry" in gedachten, toch tevreden zijn. Tenslotte zijn heaps al geen triviale structuren meer.

References

[1] R.S. Bird, Ph. Wadler, *Introduction to Functional Programming*, Prentice Hall International, Hemel Hempstead (1988).

[2] R.S. Bird, *Lectures on Constructive Functional Programming*, in: [4].

[3] R.S. Bird et al. (eds.), *Mathematics of Program Construction*, LNCS 669, Springer-Verlag, Berlin (1993).

[4] M. Broy (ed.), *Constructive Methods in Computing Science*, Springer-Verlag, Berlin (1989).

[5] W. Chen, J.T. Udding, *Program inversion: more than fun!*, Science of Computer Programming 15 (1990), pp. 1-13.

[6] E.W. Dijkstra, *Program Inversion*, in his: *Selected Writings on Computing: a Personal Perspective*, Springer-Verlag, New York (1982), pp. 351-354.

[7] E.W. Dijkstra (ed.), *Formal Development of Programs and Proofs*, Addison-Wesley, Menlo Park (1990).

[8] D. Gries, J.L.A. van de Snepscheut, *Inorder Traversal of a Binary Tree and its Inversion*, in: [7].

[9] B. Schoenmakers, *Inorder Traversal of a Binary Heap and its Inversion in Optimal Time and Space*, in: [3].

[10] J.L.A. van de Snepscheut, *What Computing is All About*, Springer-Verlag, New York (1993).

Eindhoven, 17 march 2011

Rob R. Hoogerwoord
department of mathematics and computing science
Eindhoven University of Technology
postbus 513
5600 MB Eindhoven