Designing and Implementing Combinator Languages

S. Doaitse Swierstra
Pablo R. Azero Alcocer
Joao Saraiva

Doaitse Swierstra
Department of Computer Science
Utrecht University
P.O. Box 80.089
3508 TB Utrecht
The Netherlands
email: doaitse@cs.uu.nl
url: http://www.cs.uu.nl/~doaitse/

Pablo Azero
Department of Computer Science
Utrecht University
P.O. Box 80.089
3508 TB Utrecht
The Netherlands
email: pablo@cs.uu.nl
url: http://www.cs.uu.nl/~pablo/

João Saraiva
University of Minho & Utrecht University
P.O. Box 80.089
3508 TB Utrecht
The Netherlands
email: saraiva@cs.uu.nl
url: http://www.cs.uu.nl/~saraiva/

Contents

Chapter 1

Introduction

1.1 Defining Languages

Ever since the Computer Science community has discovered the concept of a programming language there has been a continuous quest for the ideal, all-encompassing programming language. On the other hand have we been overwhelmed by an everlasting flow of all kind of special purpose programming languages. Attempts to bridge this gap between only one language and infinitely many caused research into so-called extensible programming languages.

In a certain sense every programming language with name binding constructs is extensible. In these lectures we will show that it is the unique combination of *higher order functions*, an *advanced typing system* and the availability of *lazy evaluation* which makes Haskell one of the most promising candidates for the "ideal extensible language".

Before we start with giving many examples and guidelines of how to use the features just mentioned (to their and even a bit past their limits), we want to spend some time to explain what actually constitutes a programming language. A proper programming language description at least contains:

- a context free grammar describing the *concrete syntax* of the language, with a mapping to
- an underlying context-free grammar, describing the *abstract structure* of the language
- context sensitive conditions that capture the constraints which are not easily expressed at the context-free level, like correct name introduction and use and type checking; usually such context conditions can either be expressed in a compositional way directly, or they can be expressed in terms of a fixed-point of a function which itself may be computed in a compositional way; with *compositional* we mean here that a property of a construct can be expressed in terms of properties of its constituents.
- a mechanism of assigning a "meaning" to a construct; one of the most common ways of doing this is by giving a *denotational semantics*, which boils down to describing how a function which represents the meaning of that construct can be derived from the abstract program structure. If this is the case Haskell may be used for expressing this algorithm. But also Haskell types may be used for the domain of such algorithms. It is here that the fact that we can use higher order functions plays a

crucial role. Fot the domains and co-domians of the functions denoting the semantics we may use Haskell types again.

Of course one can design a new language from scratch by defining all the above components. languages however do have a lot in common like definitions, type systems, abstraction mechanism, IO-systems etc. It would be a lot of work to implement this anew for every new language, so if we could borrow this from some other language that would be nice.

1.2 Extending Languages

There are many ways in which one can extend an existing language:

- By far the most common way to extend a language is by including some form of macro processor. These extensions are almost all at the syntactic level and do not use any form of global analysis to steer their behaviour.
- By embedding a term-rewriting system, which makes it in principle possible to acquire information about parts of the program and to move this information to other places where it may be used. The disadvantage of this approach is that on the one hand the method is very elaborate, and on the other hand it is hard to keep track of what happens if two independently designed term rewriting systems are used to transform the same program text: composition of two confluent term-rewriting systems usually is not confluent.
- By giving access to an underlying interpreter, providing reflection. An almost endless world of possibilities appears. Unfortunately there is a price to be paid: as a consequence of constructs being analyzed dynamically one can in general not guarantee that the program will not terminate erroneously.

Besides these approaches there is a fourth one, which we call *embedding*, to be described in the next section.

1.3 Embedding languages

When we embed a language in another language we are not so much extending that other language, but we make it look as if this were the case; actually we do not extend the language at all. It is here that the concept of a combinator-language shows up: we use the already available mechanisms in the language for describing the components mentioned in section 1.1:

- for describing the concrete representation (or syntax if you prefer that term) of our extension we typically introduce new operators and functions. It would be nice if we had an underlying language with distfix operators (like **if**..**then**..**else**..**fi**) available, but in practice we can do quite well with a sufficient number of operator priorities and the possibility to define new infix operators.
- for the representation of the abstract syntax we may use Haskell data types, which nicely correspond to abstract syntax trees

- for describing context sensitive restrictions we will use catamorphisms, since they capture the notion of exploiting the compositional nature of our restrictions
- for describing the semantic domains we will again use Haskell types. The way they are composed is again by using catamorphisms.

We want to emphasize that this approach has been very fruitful and has already led to several nice combinator libraries.[2, 9, 3, 15].

As we will show it is not always an attractive job to code the catamorphisms needed, and thus we introduce a special notation for them based on attribute grammars: they can be seen as a way of defining catamorphisms in a more "programmer friendly" way.

Attribute grammars have traditionally been used for describing implementations of programming languages, and their appearance here should not come as a surprise. Using attribute grammars has always been limited by the need to choose a specific language for describing the semantic functions and a specific target language. Fortunately, as we will show, it is nowadays quite straightforward to use the attribute grammar based way of thinking when programming in the setting of a modern, lazily evaluated functional language: it is the declarative way of thinking in both formalisms that bridges the gap, and when using Haskell you get an attribute grammar evaluator for free.[5, 6]

Thinking in terms of attribute grammars is useful when writing complicated functions and their associated calls. By explicitly naming argument and result positions (by the introduction of attribute names), we are no longer restricted to the implicit positional argument passing enforced by conventional function definitions.

We will identify where Haskell, as it stands now, has strong points and where unessential limitations prohibit an even more profitable use of this approach.

1.4 Overview of the Lectures

In the chapter 2 we will describe a number of so-called circular programs. This introduction serves to make you more familiar with lazy evaluation, what can be done with it, and how to exploit it in a systematic way. It also serves to make you once more familiar with the algebraic approach to programming [13, 1], and how we can design programs by constructing algebras and combining them. Although this all works nicely when done in a systematic way, we will also show why this approach is extremely cumbersome if things you want to describe are getting more complicated: one needs to be a book-keeping genius to keep track of what you are writing and calculating and combining. In the course of this discussion it will become clear why the approach which solely relies on monads in attacking these problems will not work out as expected.

In chapter 3 we will solve the same example problems again, but now by taking an attribute grammar based approach.

Chapter a large case study in which we attack the pretty printing problem as described in [3]. Hughes defines a set of operators which may be used to describe the two-dimensional layout of documents, and especially documents that contain structured text that is to be formatted according to that structure. Designing this language has been a long standing testbed for program design techniques and we hope to show that when such problems are attacked in a step-wise fashion and with proper administrative support one may easily generate quite complicated programs, which many would not dare to write by hand.

In the last chapter we will show some of the consequences of our techniques when it is taken in its simplest form, and describe some program transformations, which finally may result in a large set of relatively small strict, pure functions. So even ML-programmers should be happy in the end if they continue to read on.

Chapter 2

Circular Programs

We start by developing a somewhat different way of looking at functional programs, and especially those programs that make heavy use of functions that recursively descend over data structures; in our case one may think about these data structures as abstract syntax trees. When computing a property of such a recursive object (i.e. a program in a new language) one may do so by defining two sets of functions: one set describing how to recursively visit the nodes of the tree, and a set of algebras describing what to compute at each node when visited.

One of the most important steps in this process is deciding what the carrier type of these algebras is to be. Once this step has been taken, the types will be a guideline in the further development of the algebra. We will see that such carrier types may be functions themselves, and that deciding on the type of such functions may not always be simple. In this section we will present a view on such recursive computations, that will enable us to "design" the carrier type in an incremental way. We will do so by constructing algebras out of other algebras. In this way we define the meaning of the language in a *semantically compositional* way.

2.1 The Rep_Min problem

One of the famous examples in which the power of lazy evaluation is demonstrated is the so-called *rep_min* problem ([12]). Many have wondered how this program actually achieves its goal, since at first sight it seems that it is impossible to actually compute anything with this program. We will use this problem, and its associated solutions, to build up understanding of a whole class of such programs.

In listing 1 we present the data type of interest, i.e. a `Tree` which in this case stands for simple binary trees, together with the associated signature. The *carrier type* of an algebra is the distinguished type that describes the objects of the algebra. Here it is represented by the type parameter of the signature type:

```
type Tree_Algebra a = (Int -> a, a -> a -> a)
```

and the associated evaluation function `cata_Tree`, that systematically replaces the constructors `Leaf` and `Bin` by their corresponding operations from the

```
data Tree = Leaf Int
          | Bin  Tree Tree

type Tree_Algebra a = (Int -> a, a -> a -> a)

cata_Tree :: Tree_Algebra a -> Tree -> a

cata_Tree alg@(leaf, _   ) (Leaf i)  = leaf i
cata_Tree alg@(_    , bin) (Bin l r) = bin (cata_Tree alg l)
                                           (cata_Tree alg r)
```

Listing 1: rm.start

algebra `alg` that is passed as an argument.

We now want to construct a function `rep_min :: Tree -> Tree` that returns a `Tree` with the same "shape" as its argument `Tree`, but with the values in its leaves replaced by the minimal value occurring in the original tree. In figure 2.1 an example of an argument and a result are given.

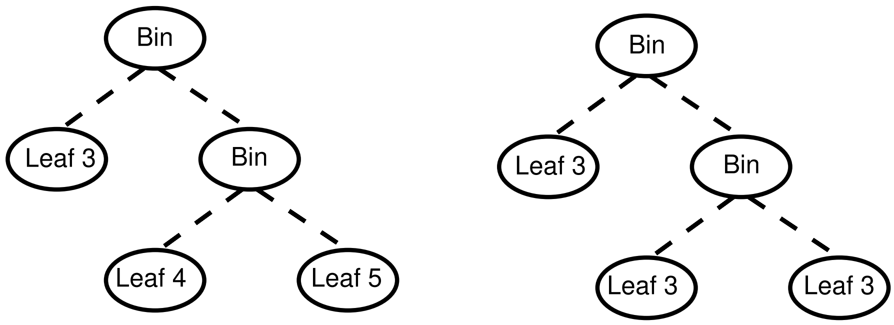


Figure 2.1: The function `rep_min`

2.1.1 Straightforward Solution

A very simple solution consists of a function in which `cata_Tree` is used twice: once for computing the minimal value of the leaf values, and once for constructing the resulting `Tree`. The function `replace_min` that solves the problem in this way is given in listing 2. Notice that the variable `m` is a global variable of the `rep_alg`-algebra, that is used in the tree constructing call of `cata_Tree`. In 2.2 we have shown the flow of the data in a recursive call of `cata_Tree`, when computing the minimal value. Although the previous solution as such is no problem, we will try to construct a solution that calls `cata_Tree` only once.

```
min_alg = (id, min::(Int->Int->Int))
replace_min t = cata_Tree rep_alg t
                where m = cata_Tree min_alg t
                      rep_alg = (const (Leaf m), Bin)
```

Listing 2: rm.sol1

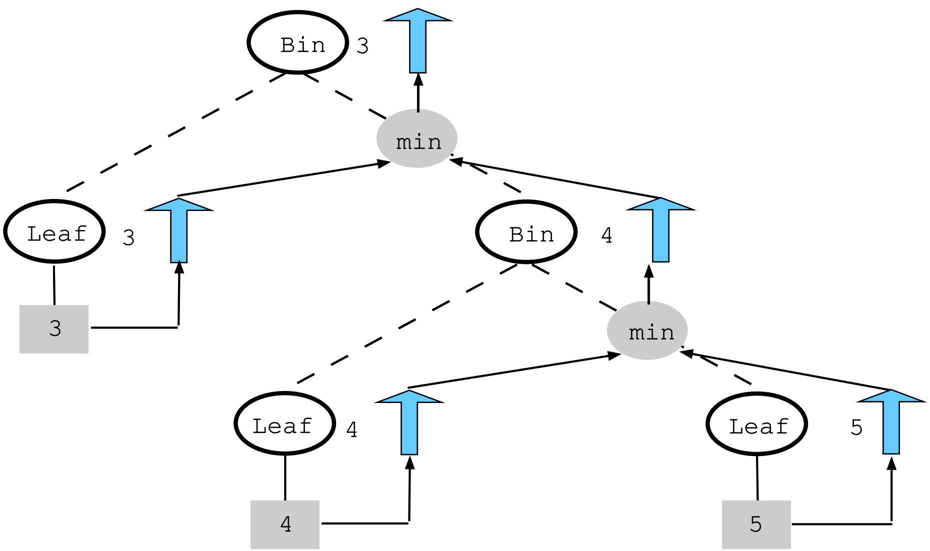


Figure 2.2: Computing the minimum value

```
rep_alg = (\ _         -> \m -> Leaf m
          ,\lfun rfun -> \m -> let lr = lfun m
                                   rr = rfun m
                               in  Bin lr rr
          )
replace_min' t = (cata_Tree rep_alg t) (cata_Tree min_alg t)
```

Listing 3: rm.sol2

2.1.2 Lambda Lifting

In an intermediate step to achieve this goal we first present program listing 3. In this program the global variable has been removed and instead the second call of `cata_Tree` now does not construct a `Tree` anymore, but instead *a tree constructing function* of type `Int -> Tree`, which takes the computed minimal value as an argument. Notice how we have emphasized the fact that a function is returned through some superfluous notation: an extra lambda was introduced in the definition of the functions constituting `rep_alg`. This process is done routinely by functional compilers and is known as *lambda-lifting*. In figure 2.3 we have shown the flow of information when this function is called. The down-arrows to the left of the non-terminals correspond to the parameters of the constructed function, and the up-arrow to the right the results of the call of the constructed functions. When we look at the top level node we see that the final value is a function that takes one argument, i.e. in our case the minimum value, and that returns a `Tree`. It is the call of `cata_Tree` that constructs this final function by pasting together the small functions found in the `rep_alg` algebra. These small functions can be identified as the small data flow graphs out of which this large graph is constructed.

In figure 2.4 the small data flow graphs corresponding to the functions making up `rep_alg` are given separately. We have labeled the nodes with the same names as the variables in the program.

2.1.3 Tupling Computations

We are now ready to make the step to a formulation in which `cata_Tree` is called only once. Note that in the last solution actually the two calls of `cata_Tree` don't interfere with each other. As a consequence we may perform both the computation of the tree constructing function and the minimal value in one go, by tupling the results of the computations. The solution is given in listing 4. First a function `tuple` is defined. This function takes two `Tree_Algebra`s as arguments and constructs a third one, which has as its carrier tuples of the carriers of the original algebra's. The resulting computation is shown in figure 2.5.

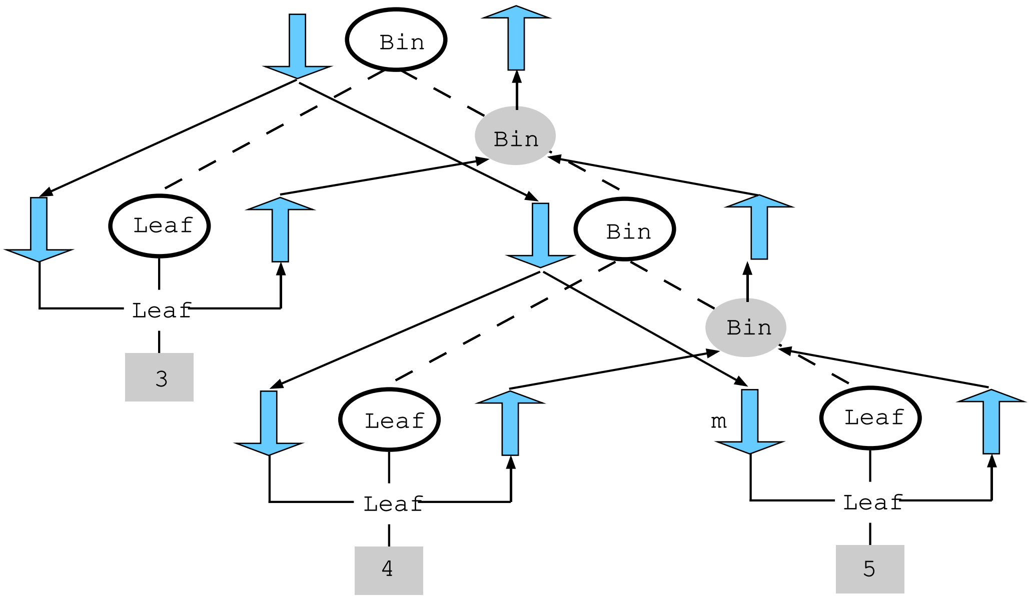


Figure 2.3: The flow of information when building the result

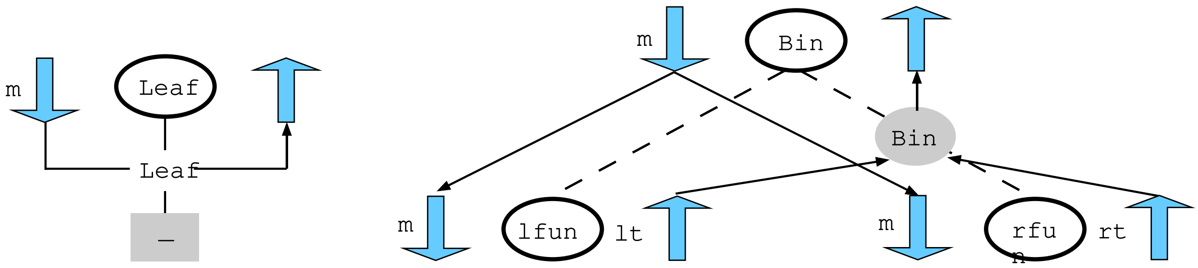


Figure 2.4: The building blocks

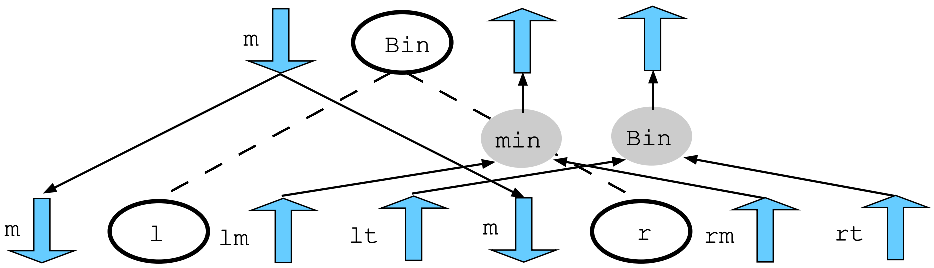


Figure 2.5: Tupling the computations

```
infix 9 `tuple`

tuple :: Tree_Algebra a -> Tree_Algebra b -> Tree_Algebra (a,b)
(leaf1, bin1) `tuple` (leaf2, bin2) = (\i   -> (leaf1  i, leaf2 i)
                                      ,\l r -> (bin1 (fst l) (fst r)
                                               ,bin2 (snd l) (snd r)
                                               )
                                      )
min_tup_rep :: Tree_Algebra (Int, Int -> Tree)
min_tup_rep =  (min_alg `tuple` rep_alg)

replace_min'' t = r m
                  where (m, r) = cata_Tree min_tup_rep t
```

Listing 4: rm.sol3

2.1.4 Merging Tupled Functions

In the next step we transform the type of the carrier set in the previous example, i.e. `(Int, Int->Tree)` into a, for this purpose equivalent, type `Int -> (Int, Tree)`. This transformation is not essential here, but we use it to demonstrate that if we compute a cartesian product of functions, we may transform that type into a new type in which we compute only one function, which takes as its arguments the cartesian product of all the arguments of the functions in the tuple, and returns as its result the cartesian product of the result types. In our example the computation of the minimal value may be seen as a function of type `()->Int`. As a consequence the argument of the new type is `((), Int)`, which is isomorphic to just `Int`, and the result type becomes `(Int, Tree)`.

We want to mention here too that the reverse is in general not true; given a function of type `(a, b) -> (c, d)`, it is in general not possible to split this function into two functions of type `a -> c` and `b -> d`, which together achieve the same effect. The new version is given in listing 5.

Notice how we have, in an attempt to make the different rôles of the parameters explicit, again introduced extra lambdas in the definition of the functions making up the algebra. The parameters after the second lambda are there because we construct values in a higher order carrier set. The parameters after the first lambda are there because we deal with a `Tree_Algebra`. A curious step which has been taken here is that part of the result, in our case the value `m`, is passed back as an argument to the result of `(cata_Tree merged_alg t)`. Lazy evaluation makes this work.

That such programs were possible came originally as a great surprise to many functional programmers, especially to those who used to program in LISP or ML, languages which requires arguments to be evaluated completely before the call is being evaluated (so-called *strict evaluation* in contrast to

```
merged_alg :: Tree_Algebra (Int -> (Int,Tree))
merged_alg = (\i        ->  \m -> (i, Leaf m)
             ,\lfun rfun -> \m -> let  (lm,lr) = lfun m
                                       (rm,rr) = rfun m
                                  in   (lm `min` rm
                                       , Bin lt rt
                                       )
             )

replace_min''' t = r
                   where (m, r) = (cata_Tree merged_alg t) m
```

Listing 5: rm.sol4

```
replace_min'''' t =
      r
      where (m, r) = tree t m
            tree (Leaf i)  = \m -> (i, Leaf m)
            tree (Bin l r) = \m -> let (lm, lt) = tree l m
                                       (rm, rt) = tree r m
                                   in (lm `min` rm, Bin lt rt)
```

Listing 6: rm.sol5

lazy evaluation). Because of this surprising behaviour this class of programs became known as *circular programs*. Notice however that there is nothing circular in this program. Each value is defined in terms of other values, and no value is defined in terms of itself (as e.g. in `ones=1:ones`).

Finally we give in listing 6 the version of this program in which the function `cata_Tree` has been unfolded. Thus we obtain the original solution (or problem) given in Bird[12].

2.2 Table Formatting

In this section we will treat a more complicated example, in which we show how to systematically design the algebra's involved.

Our final goal is to develop a program which recognizes and formats (possibly nested) HTML style tables, as described by the following grammar:

$$\begin{array}{lllll}
\mathit{table} & \rightarrow & \texttt{<TABLE>} & \mathit{rows} & \texttt{</TABLE>} \\
\mathit{rows} & \rightarrow & & \mathit{row}^{*} & \\
\mathit{row} & \rightarrow & \texttt{<TR>} & \mathit{elems} & \texttt{</TR>} \\
\mathit{elems} & \rightarrow & & \mathit{elem}^{*} & \\
\mathit{elem} & \rightarrow & \texttt{<TD>} & \mathit{string} \mid \mathit{table} & \texttt{</TD>}
\end{array}$$

An example of accepted input and the associated output is given in figure 2.6.

```
<TABLE>
  <TR><TD>the</TD>
      <TD>table</TD>
  </TR>
  <TR><TD><TABLE>
     <TR><TD>formatter</TD>
             <TD>in       </TD>
</TR>
        <TR> <TD>functional</TD>
 <TD>polytypic </TD>
</TR>
      </TABLE>
  </TD>
  <TD>style</TD>
  </TR>
</TABLE>
```

(a) HTML input

```
|-----------------------------|
|the                     |table|
|-----------------------------|
||--------------------||style|
||formatter |in       ||     |
||--------------------||     |
||functional|polytypic||     |
||--------------------||     |
|-----------------------------|
```

(b) Output

Figure 2.6: Table formatting

2.2.1 A Parser for Tables

Our first version of the table formatter parses the input and returns the abstract syntax tree. In subsequent sections we modify parts of it to compute the actual formatting. The program is written with so-called parser combinators [15] – here mostly defined as infix operators: functions which construct parsers out of more elementary parsers, completely analogous to the well-known recursive descent parsing technique. An example of the advantages of embedding a formalism (in our case context-free grammars) in a language which provides powerful abstraction techniques is that this automatically gives us an abstraction mechanism for the embedded language (in our case the context-free grammars). Since we already have a naming mechanism available we do not have to deal separately with the concept of *nonterminal*.

Parsing with combinators: giving structure

In the first chapter we have remarked that we may want to distinguish the concrte syntax and the abstract syntax. in this paper we will assume the availablity of a set of parsing combinators, that enables us to coinstruct such a mapping almost without effort.

Before we describe the structure of the combinator `taggedwith` that will be used to construct a parser for recognising HTML-tagged structures, we will briefly discuss the basic combinators used in its construction.

The types of the basic combinators used in this example are:

```
<*> :: Parser (a -> b) -> Parser a -> Parser b
<|> :: Parser a -> Parser a -> Parser a
<$> :: (a -> b) -> Parser a -> Parser b
succeed :: a -> Parser a
```

The type `Parser` is a quite complicated one,and is described in [15]. Here it suffices to know that a `Parser a` will recognise a sequence of tokens and terun a vluae of type `a` as the result of the parsing process. The *sequence* combinator `<*>`, composes two parsers sequentially. The meaning of the combined result is computed by applying the result of the first component to the second. The *choice* combinator `<|>` constructs a new parser which may perform the role of either argument parser. Finally, `succeed`, the parser that returns a parser that always succeeds (recognizes the empty string) and returns the argument of `succeed` as its semantic value. Sequence, choice and succeed form, together with elementary parsers for keywords (`pKey`), and parsers for lower case identifiers (`pVarid`) the basic constructors for parsers for context free languages.

A fifth combinator is defined for describing further processing of the semantic values returned by the parsers. It is the *application* defined as:

```
f <$> p = succeed f <*> p
```

Thus, it applies the function `f`, the so called *semantic function*, to the result of parser `p`. We will see how, by a careful combination of such semantic functions and parser combinators, we can prevent a parse tree from coming into existence at all [14, 10].

Now let us take a look at the program in figure listing 7, and take the combinator `taggedwith`. This combinator takes two arguments: a `String` providing the text of the tag and the `Parser` for the structure enclosed between the tags. Its semantics are: recognize the 'open' tag `s`, then (combinator `<*>`) recognize the structure `p`, then (again `<*>`) parse the 'close' tag. The combinators `<*->`, `<$->` and `<-*>` combine parsers, but throw away the result at the side of the `-`-symbol in their name. As a result of this the result of a a call `taggedwith s p` returns only the result recognized by the call of `p` inside its body.

The Kleene * in the two first grammar rules are realized by the combinator `pFoldr` (see figure listing 8). The first argument of `pFoldr` is a tuple of two values: `(zero,op) :: Alg_List`, an *algebra* that uniquely defines the homomorphism from the carrier set of the initial algebra to the carrier set of the argument algebra (in our case the type `b`). The second argument of `pFoldr` is

```
type Alg_List a b = ( a -> b -> b, b )

type Alg_Table table rows row elems elem
  = ( rows  -> table, Alg_List row  rows
    , elems -> row  , Alg_List elem elems
    , (String -> elem, table -> elem) )

taggedwith :: Eval a
           => String -> Parser Token a -> Parser Token a
taggedwith s p =  topen s <-*> p <*-> tclose s
  where topen  s = pKey "<"   <*-> pKey s <*-> pKey ">"
        tclose s = pKey "</"  <*-> pKey s <*-> pKey ">"

format_table ( sem_table, sem_rows, sem_row
             , sem_elems, (sem_selem,sem_telem) ) = pTable
 where
  pTable = sem_table <$> taggedwith "TABLE"
            (pFoldr sem_rows (taggedwith "TR"
               (sem_row <$> pFoldr sem_elems (taggedwith "TD"
                  (   sem_selem <$> pVarid
                  <|> sem_telem <$> pTable
            )  )  )          )               )
```

Listing 7: Parsing tables

```
pFoldr :: (Symbol s, Eval b)
       => Alg_List a b -> Parser s a -> Parser s b
pFoldr alg@(op,zero) p = pfm
  where pfm = op <$> p <*> pfm <|> succeed zero

-- Some useful algebras
init_list = ((:), [])
max_alg   = (max, 0)  -- Take the max element
sum_alg   = ((+), 0)  -- Sum all elements
```

Listing 8: List manipulation

a parser for p-structures.

pFoldr alg@(op,zero) works as follows: as long as it is possible to recognize a p-structure apply the p-parser and combine the results using the binary operator op. If no further elements can be recognized it returns zero as semantic value. As an example of its use take pFoldr sum_alg p_Integer, provided that pInteger has been defined to recognize integers. The expression would recognize a sequence of integers, and return their sum. You may note that because op is a binary operator, the actual parse result is a large expression which is constructed out of applications of op-calls and recognized elements, and the zero which is used when no further elements can be recognized. Because we work in a lazy language, the value of this expression will only be evaluated when it is actually needed, which will usually be in a test or at a strict argument position.

Now finally we can have a look at the function format_table. We see that it takes for each non-terminal an algebra describing how to construct the semantic value for that nonterminal out of the semantic values of the elements in its right hand side. From the type of Alg_table we see that it takes a set of carrier types as argument. As a result the whole parser is polymorphic in all these domains: all it does is recognising the structure of a table and composing the recognized elements once it is told how to compose them by the argument of type Alg_Table.

Simulating structure walks: adding semantics

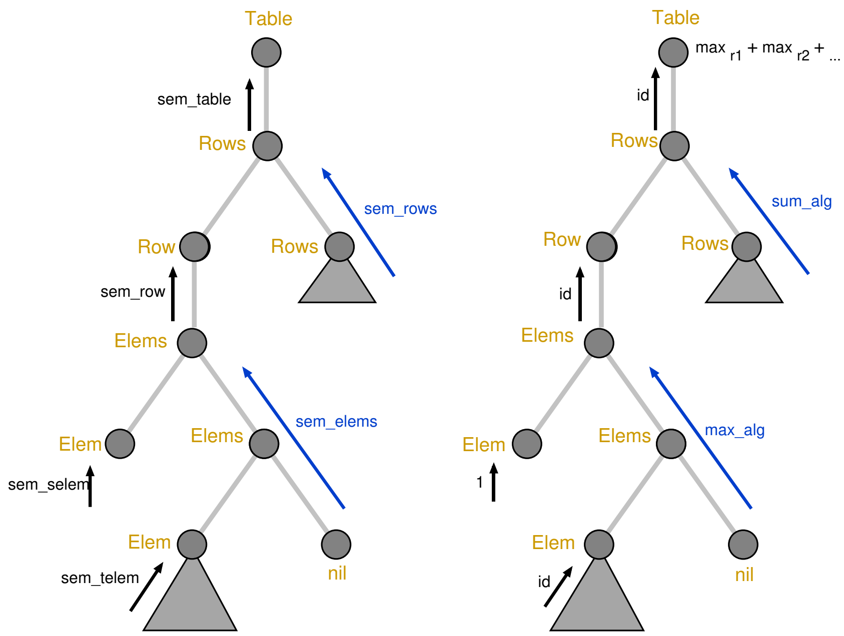
By providing different definitions for the algebras passed to the pFoldr-calls and for the sem_antic functions we may describe quite different results. With our first set of definitions we will describe the data structure holding the table as the result of the parsing process:

```
type Table = Rows
type Rows  = [ Row ]
type Row   = Elems
type Elems = [ Elem ]
data Elem  = SElem String | TElem Table
table = format_table (id,init_list,id,init_list,(SElem,TElem))
```

The type of the element returned by table is Table. We always look at the type of the first parameter of the table algebra. It is already possible in the previous functions to see the role played by the semantic functions and the list algebras – figure 2.7(a). The latter apply functions to the collected elements, and the former provide intermediate computations such as transforming data types, collecting intermediate values and computing new values. In the following sections we will focus on the systematic description of these functions. We only give a polymorphic collection of functions (the algebra for tables) corresponding to such computations.

2.2.2 Walks, trees: where are they?

In the previous section we have seen how we can use algebras to describe the construction of abstract syntax trees. All we are using these trees for is



(a) Building the AST (b) Computing the heights

Figure 2.7: Computations over trees

to compute the meaning of the recognized structure. As when using attribute grammars, we want to express this meaning in a compositional way: the meaning of a structured object is expressed in terms of its substructures. Expressed in a more mathematical style: we have to define a homomorphism from the initial algebra (the abstract syntax trees) to some other algebra (the meaning). Such homomorphisms have become known as catamorphisms [7]. An interesting consequence of trees being initial is that this function is completely defined by the target-algebra. Expressed in computer science terms this is just saying that the structure of the recursion follows directly from the data type definition; a fact well known to (functional) programmers and attribute grammar system users.

A direct consequence of this is that it is possible to compute the meaning of a structure directly, without going through an explicit tree-form representation: instead of referring to the initial algebra (constructed from the data type constructors) we use the meaning-algebra (constructed from the semantic functions) whenever we are performing a reduction (i.e. would construct a tree-node) in the parsing process.

2.2.3 Computing the Height

As a first step to a final solution let us define the functions for computing the height of a formatted table. Figure 2.7(b) depicts an attribute grammar view of the solution. We have one synthesized attribute `height`. The height of an element is the height of a simple element, `1`, or the height of a nested table.

The height of a row is the maximum of the heights of the elements of the row, and the height of a table is the sum of all the heights of the rows. This computational structure is actually what `pFoldr` is capturing: roll over the elements of the list, taking every element into account, accumulating a result. Thus the list algebra, in the parser known as `sem_elems`, for computing the height of a row is `max_alg`.

The height of the table is the sum of the heights of the rows. Again we can use a list algebra to express that computation, thus for `sem_rows` we pass `sum_alg`. The complete set of functions is

```
height_table = (id,sum_alg,id,max_alg,(const 1,id))
```

Note that `sem_table` and `sem_row` do not need special attention in this case: they only pass on their argument.

We observe the following relation between the set of functions defined and an attribute grammar: (a) the results of applying the semantic functions to the children nodes correspond to synthesized attributes and, (b) attribute computations are nicely described by algebras.

2.2.4 Computing the Widths

At the table level, the computation of widths deserves a bit of attention. We will not be able to deduce any maximum for the widths of the table until we have recognized the whole table. But instead of keeping the whole table, we can maintain a list with the maxima found thus far. When a new row is recognized, its width-values are to be compared with those of the accumulated list, taking the maxima of the columns' pair. But this is just applying an algebra to all the elements of a list, and thus obtaining a list. We introduce a *star* combinator:

```
star :: Alg_List a  b -> Alg_List [a] [b]
star (op, zero) = (zipWith op, repeat zero)
```

It takes an algebra, and returns an algebra which has as carrier set lists of elements of the original algebra. In this way, once we have defined the algebra for computing a maximum, `max_alg`, we can define an algebra for computing the pairwise maxima of two lists: `star max_alg` and this is what we need to compute the widths at the table level.

Now we want to combine the computations of the height and the width. Again, thinking in an attribute grammar style, we need another synthesized attribute. Because functions can only return a single value, we have to pair both results (height and width), and deliver them together. For the row, the width is the collection of widths of all the elements, thus `init_list`. What to return? A pair with the computation of heights and the computation of widths. Because of our algebraic style of programming, we can define an *algebra composition* combinator (also called *tupling* combinator), which takes two algebras and returns an algebra that computes a pair of values. In this way it is possible to structure the computations of the attributes even more. Note that the composition is at the semantic level and not only syntactic.

```
infixr <.>        -- infix binary operator, right associative
```

```
width_table
  = (sum, star max_alg, id, init_list, (length, id))

star :: Alg_List a  b -> Alg_List [a] [b]
star (op, zero) = (zipWith op, repeat zero)

hw_table = ( id ‘x‘ sum, sum_alg <.> star max_alg
           , id ‘x‘ id , max_alg <.> init_list
           , ( (const 1) ‘split‘ length , id ‘x‘ id ) )

f ‘x‘     g = h where h (u,v) = (f u, g v)
f ‘split‘ g = h where h u     = (f u, g u)

(<.>) :: Alg_List b a -> Alg_List b’ a’
      -> Alg_List (b,b’) (a,a’)
(f, e) <.> (f’, e’)
  =  ((\(x,x’) (xs,xs’) -> (f x xs, f’ x’ xs’)), (e,e’))
```

Listing 9: Computing heights and widths

```
(<.>) :: Alg_List b a -> Alg_List d c -> Alg_List (b,d) (a,c)
(f,a) <.> (g,c) = (\(x,y) (xs,ys) -> (f x xs, g y ys), (a,c))
```

Thus we use `max_alg <.> list_init` for synthesizing the height of the row paired with the list of widths of the elements of the row. We do the same at the table level and obtain the algebra `sum_alg <.> star max_alg`.

Finally, the result of the computation for a table must be a pair, but we obtain a list of widths from the application of `pFoldr`. Thus we need a further transformation `id ‘x‘ sum`. The *product* combinator `x` only applies its argument functions to the corresponding left and right elements of the pair. The new version of the program is shown in figure listing 9.

Let us note that: (a) we can compute several properties of a tree at the same time by tupling them, (b) computations for such tuples can be constructed out of computations for the elements of the tuples (`<.>`, `star`, `split` and `x`), and (c) the operators on algebras: composition and star, and `split` and product are independent of the problem at hand and could have been taken from a library.

Ex. 2.1 ▷ Exercise: Can you provide a composition operator for table algebra’s? ◁

2.2.5 Formatting

Once we have computed the width’s of all columns and the height of all rows we can start to work on the formatting of the table. We will first take the same approach as in the *rep-min* problem. Instead of computing the table

```
layout_table
  = ( bot_right . mk_table
    , v_compose <.> sum_alg <.> star max_alg
    , (apply2fst scnd) `split` snd
    , h_compose <.> max_alg <.> init_list
    , ( mk_box . ((:[]) `split` (const 1) `split` length)
      , mk_box
      )
    )

apply2fst = lift ($) fst

scnd = fst . snd
thid = snd . snd
```

Listing 10: Computing the formatted table

directly we will compute a function which, once it gets passed the widths of the columns, builds the table. This function will again be constructed using other functions which will construct a row, once they get passed the height of that row.

To format the table we do the following: elements are made to be the top-left element of a quarter plane (we call them `Box`es), extending to the east and the south, see figure 2.8. The table layout is constructed by placing these boxes beside and on top of each other. The code for the semantic functions and the algebras is shown in figure listing 10.

To simplify, we always place the element in the upper left corner of the box. Additional horizontal and vertical glue – blank text lines – are padded to the elements to fit in their actual layout space. All elements are furthermore equipped with a nice top left corner frame – delineating the quarter plane – as you can see in figure 2.8.

At the row level, elements are `h_compose`d, laying out one row of the table. The composition is done as follows: concatenate the next text line from each table, until there are no more lines. Because all the elements in the row have been filled with vertical glue at the end, this process also creates blank spaces if the element is not large enough to fill the vertical space.

Furthermore, because when the processing of a row has finished, the final height of the row is already known, and it can be applied to all the boxes, shaping the row horizontally. This ‘surgery’ is performed by `sem_row`, applying the computed height to the synthesized function (pattern captured by the function `apply2fst scnd`).

At the table level, the rows already formatted are `v_compose`d. This task is reduced to concatenating text lines. Finally, once all rows have been processed, the actual width of each column is known and thus, the table can be shaped

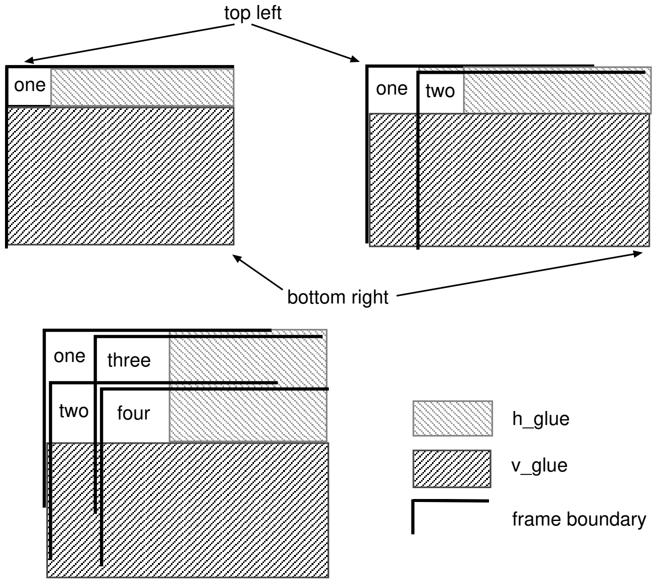


Figure 2.8: Superposition of boxes

vertically. This is done in `mk_table` with `apply2fst thid`. Then the grid is closed, with `bot_right` placing the bottom and right lines, and correcting of the actual size of the table. The implementation of box manipulation functions is given in figure listing 11.

Observe that the size of the boxes is flexible, but once we know the corresponding height and width it is possible to actually obtain the nicely formatted table. Even without noticing, we also put the grid in the table, placing the elements besides and on top of each other. We only need to take care of closing the grid, and providing each element with a top-left grid.

The simplicity of `h_compose` and `v_compose` is suspicious. Let us take a look inside `h_compose`. In terms of text elements it's only string manipulation, but let us take the attribute grammar view. At the `Elems` level we have the situation depicted in figure 2.9(a): an `Elems` cons node has two inherited attributes, the height and a list of widths, and one synthesized attribute, the layout of the element. The inherited attributes are passed down to its children, the height is distributed as it is (it is a global value for the row), but the widths have to be split element by element. The synthesized attributes are combined together using the `zipWith (++)` (but in general any `f`). Thus we have some patterns of attribute manipulation: pass down a global value (`fork`), pass down and split a composed value (`decons` if the value is a list and we want to decompose a list into its head and tail), combine inherited attributes (`<||>`) and combine synthesized attributes (`<=>>`), see figure 2.9(b).

Once more, thanks to the abstraction and higher orderness of the language, these patterns can be abstracted and used in a compositional way to express a computation of `h_compose`. The code of these combinators is shown in listing 12.

Note that there are no inherited attributes as such. We create partially

```
mk_box          = to_box ‘x‘ (+1) ‘x‘ (+1)
to_box t rh rw = map (take rw) . take rh . top_left . add_glue $ t
top_left  t     = map (’|’:) (h_line:t)

mk_table = (apply2fst thid) ‘split‘ scnd ‘split‘ (sum . thid)
bot_right (t,(h,w)) = (close_grid, (h + 1, w + 1))
  where close_grid = map (++"|") (t ++ [take w (’|’:h_line)])

h_compose = ( fork <||> decons <=>> zipWith (++), nil_table )
v_compose = ( lift (++)                           , nil_row   )


nil_table _ _ = repeat ""
nil_row   _   = []

h_glue        = repeat ’ ’
v_glue        = repeat h_glue
add_glue t    = map (++ h_glue) t ++ v_glue
h_line        = repeat ’-’
```

Listing 11: Functions for manipulating boxes

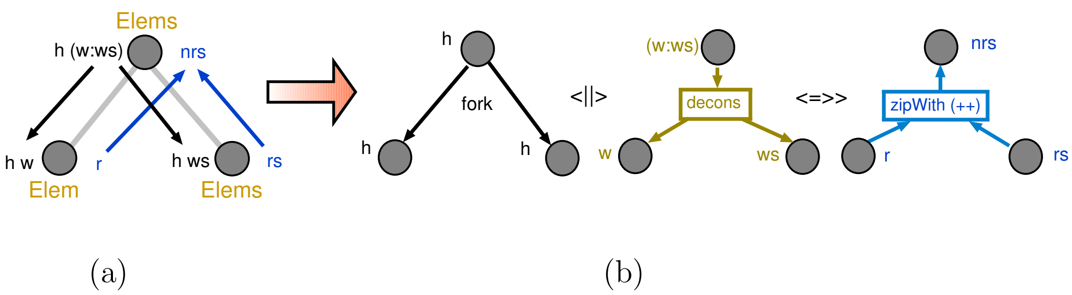


Figure 2.9: Attribute computation: (a) example (b) combining patterns

```
lift op f g = \x -> (f x) `op` (g x)
fork        = id   `split` id
decons      = head `split` tail

(<||>) :: (a->(b,c)) -> (d->(e,f)) -> a -> d -> ((b,e),(c,f))
forkl <||> forkr
  = \inh_l inh_r -> let (inh_ll,inh_lr) = forkl inh_l
                        (inh_rl,inh_rr) = forkr inh_r
                    in  ((inh_ll,inh_rl),(inh_lr,inh_rr))

(<=>>) :: (a -> d -> ((b,e),(c,f)))
       -> (g -> h -> k) -> (b -> e -> g) -> (c -> f -> h)
       -> a -> d -> k
fork <=>> merge_op
  = \fsyn_l fsyn_r -> \inh_l inh_r
   -> let ((inh_ll,inh_rl),(inh_lr,inh_rr)) = fork inh_l inh_r
          syn_l = fsyn_l inh_ll inh_rl
          syn_r = fsyn_r inh_lr inh_rr
      in  merge_op syn_l syn_r
```

Listing 12: Attribute computation combinators

parameterized functions and once we know the dependent value, we apply the function(s) to the value(s). Thus some attributes play a double role: they are synthesized (like the height of a row), but once their value has been computed they can be used in a subsequent computation; thus acting as inherited attributes.

We believe that this program clearly captures the notion of attribute grammar: a context free grammar is represented by the use of parser combinators, while attributes and attribute computations are expressed in terms of algebras and parameterized functions.

Furthermore, the program can be generalized rather straightforward to a polytypic function [4] because the constructors are general. Although not presented here, the algebra composition operator `<.>` can be defined for any arbitrary data type constructor *f*.

As a final remark we notice that probably the hardest part of the derivation was the design of the combinators `<||>`, `<=>>`, `fork` and `decons`, and using them to construct the data-flow patterns like in figure 2.9(a).

Chapter 3

Attribute Grammars

In the previous chapter we have developed programs for the *Rep-Min* and *Table-Formatting* problems. In both cases we computed a tuple of values as the result of a catamorphism and at least one of the elements of those tuples was a function which was at some point applied to another element of the tuple. In the Rep-Min example the tree constructing function was applied to the computed minimal value, and in the Table-Formatting example we had two occurrences of this phenomenon: the row-constructing function was applied to the computed maximum height of the row, and the table constructing function was applied to the list of computed row widths.

Since this pattern is quite common and the composition and invention of all the algebras was not so straightforward we will introduce an attribute grammar based notation, out of which we may easily generate equivalent definitions. The conclusion will be that we can design programs like Rep-Min and Table-Formatting by drawing pictures like the ones presented in the Rep-Min example. The price to be paid is that instead of having semantic compositionality, we have to fall back on syntactic compositionality. On the other hand we think that the advantages of our approach when developing combinator libraries, which are to be used by others only in their completed form, are so large that attempts to maintain semantic compositionality is simply too high.

We also hope to show that by taking the attribute grammar approach it will become much easier to extend the library or to make efficiency improvements. The next chapter in which we develop a set of pretty printing combinators in a sequence of steps is an example of the allowed flexibility. Since we anticipate that people will want to experiment with different implementations and designs we have tried to design our attribute grammar formalism in such a way that definitions can be easily changed and expanded without having to change the original program texts.

3.1 The Rep-Min Problem

In listing 13 we show the formulation of the Rep-Min problem, in our attribute grammar notation.

The first two **data** declarations introduce the grammar corresponding to the structure of our problem. The **attr** declarations specify the inherited and

```
DATA Tree
  | Leaf int:Int
  | Bin left, right : Tree

DATA Root | Root tree:Tree

-- the computation of the minal value

ATTR Tree [ -> m: Int]

SEM Tree
  | Leaf LHS.m = int
  | Bin  LHS.m = "left_m `min` right_m"

ATTR Tree [minval: Int <-]

SEM Tree
  | Bin   left.minval = lhs_minval
         right.minval = lhs_minval
-- the computation of the resulting tree

ATTR Tree [-> res: Tree]

SEM Tree
  | Leaf LHS.res = "Leaf lhs_minval"
  | Bin  LHS.res = "Bin left_res right_res"

-- use of the computed minimal value

ATTR Root [->res: Tree]
SEM Root
  | Root tree .minval  = tree_m
         LHS  .res     = tree_res
```

Listing 13: RepMin1.ag

synthesized attributes of the nonterminals. Attributes occurring before a `<-` are inherited attributes, corresponding to downward arrows in the pictures we have seen, and attributes following a `->` are synthesized attributes, corresponding to the upgoing arrows in the pictures. Declarations between `<-` and `->` introduce two attributes of the same name, one inherited and one synthesized. In the **sem** parts we specify the way attributes are computed out of other attributes. The actual definitions are pieces of Haskell text, which are neither parsed nor typechecked, and are copied literally into the generated program. References to other attributes in such rules follow a naming convention: a synthesized attribute `res` of a child `left` is e.g. referred to as `left_res`, whereas an inherited attribute `minval` is referred to as `lhs_minval`. In each semantic rule we have to specify what nonterminal (`SEM Tree`, what alternative (`|Leaf`), what component of the production (`LHS` or `left`)) and what attribute (`.res`) is specified by the rule.

If we put this text through our small system the code in listing 14 is generated.

Ex. 3.1 ▷ Exercise: Use the parser combinators together with the generated file to construct a solution for the Rep-Min problem, that reads a tree from a file, and writes the result into another file ◁.

One might wonder what progress has been made since both the input and the generated program are much longer than the original program in the previous section.

In the first place we have presented the input in the most elaborate form of our notation, thus making explicit all different components of the definition. Many abbreviations exist in order to cope with often occurring patterns of attribute use. A completely equivalent input is given in listing 15. Here we see that attributes may be declared together with introductions of new nonterminals, new alternatives or new semantic rules. Furthermore many straightforward so-called *copy rules* can be easily inferred by the system. It is the extension of the notation which makes things really work well. So is the attribute `minval` automatically made available in all nodes of the tree by the rule that if both a child and a father node have an inherited attribute with the same name, it is automatically passed on from the father to the child if no semantic rule has been defined (actually the rules for doing so are a bit more complicated, but this description will do for the time being). This rule captures the pattern that is normally associated with a reader monad. If we have however several attributes following this same pattern we do not have to introduce a new monad describing this jointly passing around of values.

Furthermore there are a lot of small but convenient conventions; if an element in the right hand side of a production is not explicitly named its name is constructed from the type by making the first letter into a lower case one. As a consequence we do not have to be creative in inventing a name for the value at a `Leaf`, it is just called `int`.

```
module repmin where

---------------------- Tree -------------------------
data Tree =Tree_Leaf Int | Tree_Bin Tree Tree
          deriving Show
-- semantic domains
type T_Tree =  Int->(Tree,Int )

-- funcs
sem_Tree_Leaf :: Int->T_Tree
sem_Tree_Leaf int lhs_minval  =  ((Leaf lhs_minval), int)
sem_Tree_Bin :: T_Tree->T_Tree->T_Tree
sem_Tree_Bin left right lhs_minval
 = let{ (left_res, left_m)  = left lhs_minval
   ;    (right_res, right_m)  = right lhs_minval
   }in  ((Bin left_res right_res), (left_m `min` right_m))

---------------------- Root -------------------------
data Root =Root_Root Tree
          deriving Show
-- semantic domains
type T_Root =  Tree

-- funcs
sem_Root_Root :: T_Tree->T_Root
sem_Root_Root tree  = let{ (tree_res, tree_m)  = tree tree_m}in  tree_res
```

Listing 14: RepMin

```
DATA Tree
  | Leaf Int
  | Bin left, right : Tree

SEM Tree [minval: Int <- -> m: Int res: Tree]
  | Leaf LHS.m   = int
            .res = "Leaf lhs_minval"
  | Bin  LHS.m   = "left_m `min` righth_m"
            .res = "Bin left_res right_res"


DATA Root [->res: Tree] | Root Tree
SEM  Root               | Root tree.minval = tree_m
```

Listing 15: RepMin2.ag

```
DATA Table | Table     Rows
DATA Rows  | Nil
           | Cons      Row      Rows
DATA Row   | Row       Elems
DATA Elems | Nil
           | Cons      Elem     Elems
DATA Elem  | Str       String
           | Tab       Table
```

Listing 16: TableData.ag

```
--< TableData.ag
ATTR Table  Row  Elem [ -> mh : Int ]

SEM Table
  | Table     LHS . mh = "rows_mh + 1"
ATTR Rows  [ -> mh USE "+"      "0": Int]
ATTR Elems [ -> mh USE "‘max‘" "0" : Int]
SEM Elem
  | Str       LHS . mh = "2"
  | Tab       LHS . mh = "table_mh + 1"
```

Listing 17: TableHeight.ag

3.2 The Table-Formatting Problem

In this section we will treat the Table-Formatting problem again, and do so again in a number of steps. Remember that in the previous chapter, when combining the algebra's we really had semantic compositionality: the algebra's could be defined separately, compiled separately in other modules, and be only composed at the very last moment, i.e. when we actually want to format a table.

Since we are in this chapter dealing with the generation of Haskell code (i.e. we use Haskell instead of C++ as our "assembly" language), we do not have to adhere so strict to the typing, naming and lexical rules of the language: we have much more freedom in designing the attribute grammar formalism in such a way that we may express ourselves in the most convenient way. To emphasize the compositional nature of our approach we split up the attribute grammar in many separate pieces of text, which are to be combined by the system.

We start of with the grammar in listing 16 that directly corresponds to the grammar presented before. In program (listing 17 we import the previous file (`--< TableData.ag`) and introduce for each nonterminal a synthesized

```
--< TableHeight.ag

-->type Rowwidths = [Int]
ATTR Table  Elem       [ -> mws: Int]
ATTR Rows   Row        [ -> mws: Rowwidths ]

SEM Table
  | Table     LHS . mws  = "lmw + 1"
              LOC . lmw  = "sum rows_mws"
SEM Rows
  | Nil       LHS . mws = "repeat zero"
  | Cons      LHS . mws = "zipWith max row_mws rows_mws"

ATTR Elems [-> mws USE ":" "[]" : Rowwidths ]
SEM Elem
  | Str       LHS . mws  = "length string + 1"
  | Tab       LHS . mws  = "table_mws + 1"
```

Listing 18: TableWidths.ag

attribute containing its minimal height in the formatted table. In listing 18 this version is extended further with the attributes and semantic functions for computing the minimal required widths; note how the tupling is now done implicitly by the system, and that we do not have to introduce special combinators to merge the height and the widths algebras into a combined one.

In listing 18 we see another small language element: lines preceded with `-->` are literally copied into the generated file. In this way small additional functions and type definitions can be passed on to the generated Haskell program, thus obviating the need to edit the generated file to contain `import ..` lines. Furthermore it is possible to provide, together with the introduction of a synthesized attribute a binary operator and a unit element (see the `USE ":" "[]"` phrase in the introduction of attribute `mws`). If no semantic rule is given for this attribute the attributes of the children with th same name are combined using the binary operator, and if no such attributes exist the unit element is taken as its value: in the case of nonterminal `Elems` all we want for the attribute `mws` is building up a list of the minimal widths of its `Elem`s and this is achieved through the `USE` phrase. Note that the `USE` construct is a typically polytypic, since it works for any kind of nonterminal (data type, functor).

In the next step the downwards distribution of the computed final heights and widths to the individual elements is described, so finally each element can be formatted according the actual size it occupies in the formatted table (listing 19). Here the advantages of the attribute grammar based formulation show up most clearly: we do not have to invent combinators for combining subcomputations and all we have to do is to indicate how the computed heights and

```
--< TableWidths.ag

ATTR Elems  [ ah : Int <- ]
SEM Row
  | Row       elems . ah  = "elems_mh"

ATTR Rows  Row  Elems [ aws : Rowwidths <- ]
SEM Table
  | Table     rows  . aws = "rows_mws"
SEM Elems
  | Cons      elems . aws = "tail lhs_aws"
```

Listing 19: TableDistr.ag

widths flow back into the abstract syntax tree. Finally we add the computation of the final formats, i.e. sequences of lines in listing 20 .

In listing 20 we see another small extension of the formalism, i.e. the `EXT` clause. The effect of this clause is in this case to extend the alternative `Cons` of nonterminal `Elems` with an extra element: `top_Left : Top_Left`. Although the nonterminal `Top_Left` has been introduced, it was not given productions. This is interpreted as an external nonterminal. It does not show up as a parameter in the generated semantic functions, but a call is generated. We now come to a final convention: if an inherited attribute has been declared an in the rule an attribute with that name would be allowed as a semantic function such semantic functions are generated automatically. So in listing 20 we actually have inserted a call to an external function, passing on some of the available attributes. extend the The final addition of some glue is given in listing 21 .

3.3 Comparison with Monadic Approach

As mentioned before many have tried to employ monads for capturing often occurring patterns of parameter passing and use. Unfortunately monads do in general not compose well. Recognizing this problem we have in our formalism taken a purely syntactic approach.

Reader Monads correspond in our formalism to an inherited attribute that is automatically passed on to all the elements in the tree by the copy rule generation process, provided they have indicated their interest in that value by declaring an inherited attribute, and provided all their parent types have done so too. Thus parameterizing a whole computation by a global value is now easily done. Furthermore this can be repeated as often as needed, so the effort for the programmer is almost nothing.

State Monads correspond to so-called chained attributes, i.e. pairs of an inherited and a synthesized attribute, that have the same name. In order to

```
--< TableDistr.ag

-->type Lines = [String]

ATTR Table Row Elems Elem [ -> lines : Lines]

SEM Table
  | Table     LHS . lines = "bot_right lmw rows_lines"

ATTR Rows [->lines USE "++" "[]" : Lines]

SEM Elems
  | Nil   LHS . lines = "repeat []"
  | Cons  LHS . lines = "zipWith (++) top_Left_ls elems_lines"
          LOC . haws  = "head lhs_aws"
SEM Elem
  | Str       LHS . lines = "[string]"

DATA Top_Left [ haws elem_mws lhs_ah elem_mh elem_lines <- -> ls ]
EXT Elems
  | Cons Top_Left
```

Listing 20: TableFormats.ag

```
--< TableFormats.ag
-->
-->-- -------------------------------------------------------
-->-- Additional layout functions ----------------------------
-->
-->sem_top_Left lines mh  ah  mw aw
-->  =  ("|" ++ hor_line (aw - 1))
-->  :  ["|" ++ l ++ hor_glue (aw-mw) | l <- lines]
-->  ++ ["|" ++ vg | vg <- ver_glue (aw - 1) (ah-mh)]
-->
-->bot_right mw lines = [ l ++ "|"
-->                     | l <- lines ++ ["|" ++ hor_line (mw - 1)]
-->                     ]
-->
-->hor_glue h   = take h (repeat ' ')
-->ver_glue h v = take v (repeat (hor_glue h))
-->hor_line n   = take n (repeat '-')
-->
-->-- -------------------------------------------------------
-->
```

Listing 21: TableFinal.ag

support the generation of the copy rules here too we now explain the complete process underlying the copy rule generation. Each element in the right hand side of the production has a context that steers the generation of non-specified semantic functions. For each attribute `at` for which no function is defined we first check whether there exists an element `elem` that defines a synthesized attribute `def` such that `at` = `elem_def`; this includes the inherited attributes of the father too (`lhs_def`). If this is the case that value is taken. If not it is checked whether its left hand side neighbour `l` has a synthesized attribute with name `at`. If it does `l_at` is taken, and if not the element one step further left is checked and so on. If nothing appropriate is found during this search finally the inherited attributes are checked. This rule also applies to the synthesized attributes of the left hand side, in which case the searching process starts at the last element of the right hand side.

So if we want to maintain e.g. a label counter, supplying new label numbers when generating code, we define the attribute `labels` to be both inherited and synthesized:

```
DATA Expr[<-labels: Int ->]
  | If ce,te,ee: Expr
SEM Expr
  | If ce.  labels = "lhs_labels +2"
```

In the generated code we now find:

```
sem_Expr_If ce te ee lhs_labels
 = let{ (ce_code, ce_labels)  = ce (lhs_labels +2)
   ;    (te_code, te_labels)  = te ce_labels
   ;    (ee_code, ee_labels)  = ee te_labels
   }in  ((ifthenelsecode ce_code te_code ee_code lhs_labels), ee_labels)
```

and we see that the `Labels`-value is nicely being passed on. Again this can be done for many attributes at the same time, without having to worry about the composition of those instances.

Writer Monads somehow correspond to synthesized attributes, which are composed with the `USE` clause.

Chapter 4

Pretty Printing

In this section we attack the pretty printing problem as described in [3, 11]. Pretty-printing deals with representing tree-based structures in a width-bounded area in a top-down, left to right order, and in such a way that the logical structure of the tree is clearly represented in the layout. In this chapter we develop a set of combinators for describing such layouts.

Suppose we want to pretty-print an `IF-THEN-ELSE-FI` structure. We may display it with different layouts as depicted in figure 4.1. The layout chosen will normally depend on the page width. Thus, with page width at least 31, layout a) is preferred, between 30 and 17 b) is chosen and in the range from 16 to 10 c) wins. Any attempt however to display inside a page less than 10 characters wide is bound to fail.

We define a layout to be *optimal* (nicest or prettiest) if it takes the least number of lines, while still not overflowing the right page margin. The examples in figure 4.1 are optimal for page widths 40, 28, and 15 respectively. Taking the second layout with respect to a page width of 35 is thus considered non optimal.

Our approach is based on the relation between the height and the width of a layout: higher when elements cannot be placed together horizontally because of the limited page width and wider otherwise. We prefer the wider solutions, since they will lead to a smaller overall height, as is evident in the examples of figure 4.1.

Since potentially many solutions have to be taken into account, this can be a cause of gross inefficiency. Fortunately we are saved by the fact that not all possible combinations have to be inspected. Of all the possible solutions

```
a. IF␣a>b␣THEN␣max=a␣ELSE␣max=b␣FI   height=1   width=31
b. IF␣a>b␣THEN␣max=a                  height=3   width=17
   ␣␣␣␣␣␣␣ELSE␣max=b
   FI
c. IF␣a>b                             height=4   width=10
   THEN␣max=a
   ELSE␣max=b
   FI
```

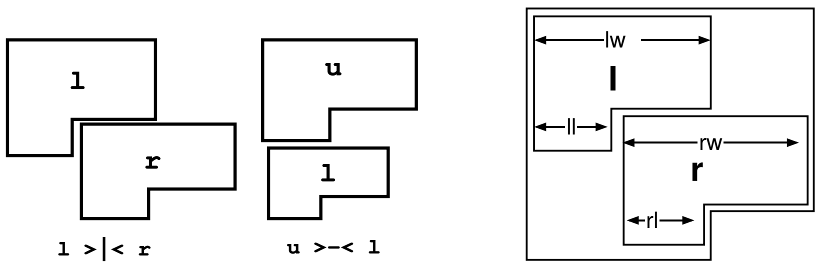
Figure 4.1: Pretty-printing an `IF-THEN-ELSE-FI structure`

with the same height, only a limited number of candidates has to be taken into account. Many combinations can be discarded from the overall computation by selecting only the narrowest solution for each height, and inspect only those candidate solutions that have at most the height of the final solution.

Taking the example of figure 4.1, we may want to describe such possible layouts as follows:

```
ite cond te ee
  =   ifc >|<  thenex >|< elseex  >|< fi
  >^< ifc >|< (thenex >-< elseex) >-< fi
  >^< ifc >-<  thenex >-< elseex  >-< fi
  where ifc    = text "if"   >|< cond
        thenex = text "then" >|< te
        elseex = text "else" >|< ee
fi     = text "fi"
```

where `text` converts strings into layouts, `>|<` places its two arguments beside each other, `>-<` places them above each other and `>^<` combines two possible layouts. Besides these combinators we also have `indent` that inserts a specific amount of white space in front of its argument and `empty` that represents the empty document and is a unit element for `>-<` and `>|<`. The effect of operations `>|<` and `>-<` is sketched in figure 4.2(a).



(a) Specification

(b) Implementation of `>|<`

Figure 4.2: Pretty-printing operations

Before going into implementation details we want to fix the interface (or the concrete grammar if you prefer) and the semantic domains involved:

```
infixr 2 >|<
infixr 1 >-<
infixr 0 >^<

empty  :: Formats
text   :: String  -> Formats
indent :: Int     -> Formats -> Formats
(>|<)  :: Formats -> Formats -> Formats
```

```
(>-<)  :: Formats -> Formats -> Formats
(>^<)  :: Formats -> Formats -> Formats
```

In the next sections we will, by improving on our search process, develop increasingly sophisticated versions of these combinators.

4.1 The general Approach

We start out by defining a basic set of combinators based on the context-free grammar of listing 22. We rely on the existence of a set of basic combinators which generate alternative layouts as sorted lists, assuming that their arguments are sorted lists too. Take for example the `>|<` combinator and assume

```
type Formats = [ Format ]
```

The combined solution is found by merging all lists that are obtained by putting an element from the left argument list besides all elements of the right argument list. Since we work in a lazy language the resulting list will be generated in an incremental way as need arises. The other operations are implemented in an analogous way. A detailed description of the implementation of the underlying basic machinery can be found in [8].

In the attribute grammar of listing 22 the specification of the pretty-printing operations is thus reduced to producing the appropriate basic function calls.

4.2 Improving Filtering

Since many potential candidates are taken into account, and every new choice point doubles the amount of work to be done, detecting solutions wider than the page width as early as possible will improve the efficiency of the process.

4.2.1 Filtering on the page width

Our first filter is based in the idea of communicating to each node the page width, preventing candidates wider than the page width to be constructed. Adding this filter to our first program is trivial: declaring an extra inherited attribute for all nonterminals introduced thus far (i.e. including the pseudo non-terminals that stand for external function calls), as you can see in listing 23. One might compare this with the effort to convert the program into monadic form in order to use a reader monad.

Since we want to be able to work with many different versions of our baisc combinators we tell the system to prefix all the calls with `pw_` as indicated by the phrase `PRE pw`.

A change in the underlying machinery is necessary because we now need to pass the width to be filtered on to the basic combinators, in which the actual combination process takes place. Take again the `>|<` operation depicted in figure 4.2(b). We now construct new solutions only when the width of the resulting layout (computed as `ll + rw`, where `ll` is the length of the last line of `l` and `rw` is the width of `r`) is less than the “global” page width `pw`.

```
-- Context-free grammar
DATA PP [ -> fmts : Formats ]
 | Empty
 | Text     String
 | Indent   Int       PP
 | Beside   left,     right : PP
 | Above    upper,    lower : PP
 | Choice   opta,     optb  : PP

-- Calling external functions
EXT PP
 | Empty    Empty_fmts    | Text     Text_fmts
 | Indent   Indent_fmts   | Beside   Beside_fmts
 | Above    Above_fmts    | Choice   Choice_fmts

-- Introducing external functions
DATA Empty_fmts  [                            -> fmts ]
DATA Text_fmts   [ string                  <- -> fmts ]
DATA Indent_fmts [ pP_fmts, int            <- -> fmts ]
DATA Beside_fmts [ right_fmts, left_fmts   <- -> fmts ]
DATA Above_fmts  [ lower_fmts, upper_fmts <- -> fmts ]
DATA Choice_fmts [ optb_fmts, opta_fmts    <- -> fmts ]
PRE sim

-- Display the solution found
DATA Root [ -> fmts : Output ]
  | Best PP

SEM Root
  | Best LHS.fmts = "putStr . display best $ pP_fmts"

-->type Output = IO ()
-->
```

Listing 22: Simple pretty-printer (`SPP.ag`)

```
--< SPP.ag

ATTR PP Root [ pw : T_PW <- ]

ATTR Empty_fmts  Text_fmts  Indent_fmts
     Beside_fmts Above_fmts Choice_fmts [ lhs_pw <- ]

PRE pw
```

Listing 23: Filtering with page width

4.2.2 Narrowing the Estimates Further

Actually the page width may be seen as an upper bound on the space available to all nodes. We want to improve on this bound by taking the context of the node into account. Once we know for for two nodes to be placed besides each other how much space each of them will take at least, and how much is available to both of them together, we may compute an estimate of how much space is at most available to each of them. The bound on the available space replaces the attribute `pw` and is called the `frame`.

Now have a look at the example in figure 4.3, and assume a page width of 20. At the root node we start with $(20, 20)$, which are the bounds on the total width and the length of the last line of the formats generated by that node,

Let us compute the frame of its left subtree `b`. Since the minimal width of the subtree `c` is 9, `b` has to fit inside a frame (20,11). Similar, since the end of the last line of the subtree `b` is at least 7 units from the left, the frame for the subtree `c` is (13,13). Since the frame (13,13) cannot accommodate the string `set of functions` the left alternative of the choice node can locally be discarded, and thus will not be combined elsewhere with other candidates only to be discarded as part of an impossible solution at the top of the computation.

In listing 24 we show how to compute the minimal space used by a node, which is needed to adjust the frames that are being inherited by its fellow nodes. In figure 4.4(a) we depict the attribute computations involved in the operation `>|<`.

In listing 25 the semantic functions for passing frames downwards are shown, and an illustration of the data flow for the operation `>|<` is shown in figure 4.4(b). Recall that we do not have to code all data flows, only the relevant computations are explicit. Copy rules involving passing information around are generated automatically by the MAG compiler as mentioned before in chapter 3. Also note that at the top level we are initiating the attribute computations with the frame `(lhs_pw,lhs_pw)`.

Finally in listing 26 we add the synthesis of the formats and an attribute for handling error conditions.

Ex. 4.1 ▷ Exercise: Note that up to now we do not need to compute the height of the document. Can you anticipate a situation where it is needed? Modify the

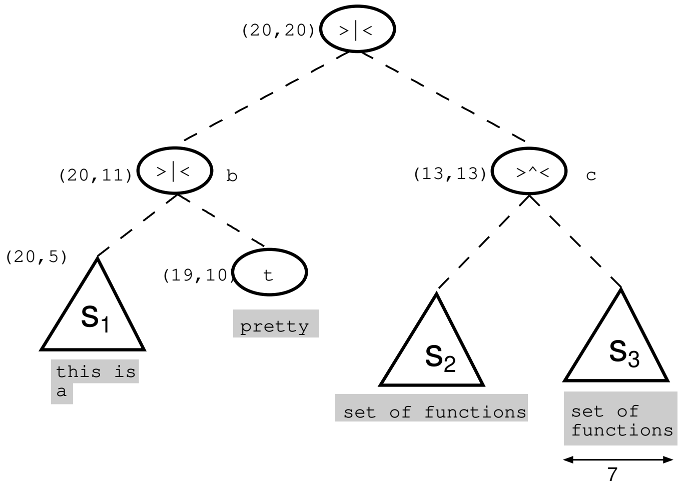
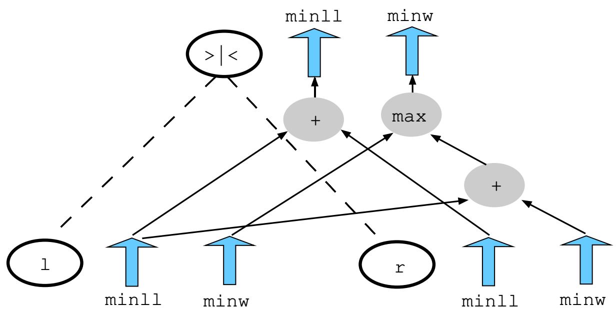


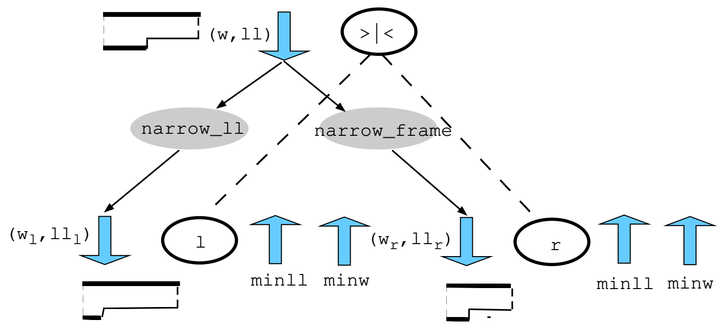
Figure 4.3: The frame bound

```
--< SPP.ag

SEM PP [ -> minw  USE " ‘max‘ " "0" : Int
            minll USE " ‘max‘ " "0" : Int ]
  | Text   LOC.minw  = "length string"
           LHS.minll = "minw"
  | Indent LHS.minw  = "int + pP_minw"
              .minll = "int + pP_minll"
  | Beside LHS.minw  = "left_minw ‘max‘
                        (left_minll + right_minw)"
              .minll = "left_minll + right_minll"
  | Choice LHS.minw  = "opta_minw ‘min‘ optb_minw"
              .minll = "opta_minll ‘min‘ optb_minll"
```

Listing 24: Computing min bounds: `FRPP.ag`



(a) Min bound attributes for `>|<`



(b) Frame attributes for `>|<`

Figure 4.4: Computing the frame

```
SEM PP [ frame : T_Frame <- ]
  | Indent pP    .frame = "narrow_frame int lhs_frame"
  | Beside left  .frame = "narrow_ll right_minw lhs_frame"
           right.frame = "narrow_frame left_minll lhs_frame"

SEM Root [ pw : T_PW <- ]
  | Best pP.frame = "(lhs_pw,0)"

-->narrow_frame i  (s,l) = (s-i, l-i)
-->narrow_ll    i  (s,l) = (s  , l-i)
```

Listing 25: Computing and communicating the frame: `FRPP.ag`(cont)

```
ATTR Empty_fmts  Text_fmts  Indent_fmts
     Beside_fmts Above_fmts Choice_fmts [ lhs_frame <- ]
PRE frame

-- Display the solution found
SEM Root
  | Best LHS.fmts
           := "putStr (if null pP_error then display best pP_fmts
                                        else pP_error)"

-- Error handling
SEM PP [ -> error USE "++" "[]": T_Error ]
  | Indent  LHS.error = "err (int < 0) 1"

TXT err
-->type T_Error = String
-->err cond message
-->  | not cond = ""
-->  | cond     = case message of
-->                 1 -> "negative indentation"
```

Listing 26: Error and formats: `FRPP.ag`(end) program `FRPP.ag` so that the computation of heights is included. ◁

Before starting to read the next section it is useful if you try hard to solve the following problem. The combinator `hv :: Formats -> Formats -> Formats` has the following behaviour:

```
? render (hv (text "aaaa") (text "bbbb")) 15
aaaabbbb
? render (hv (text "aaaa") (text "bbbb")) 7
aaaa
bbbb
?
```

The combinator places its arguments either vertically or horizontally, depending on the available frame.

Note that the type of `Formats` in our latest version of the combinators is:

```
type T_Formats = (Int, Int) -> (Error,Minw,Minll,OrigFormats)
type Minw      = Int
type Minll     = Int
type Error     = String
```

where `OrigFormats` is the type of the elements manipulated by the underlying machinery.

Ex. 4.2 ▷ Exercise: Write the combinator `hv`. ◁

4.3 Loss of Sharing in Computations

You may have given the following solution in the last exercise:

```
hv a b  = a >|< b >^< a >-< b
```

Unfortunately you have in this way given an very inefficient solution too. Why the previous definition of `hv` does not solve our problem? Because the arguments, `a` and `b`, of the expression are not plain values, but instead are functions to which secretly a frame is passed, and thus they are evaluated twice in `hv`. We have thus lost *sharing* as an unfortunate consequence of taking a higher order domain.

In order to get back the situation in which the computations are shared we have now to collect all the arguments that are passed at the different occurrences of the same expression. Fortunately we have a pleasant property of the filters and the generators: the program is thus far constructed in such a way that if we filter at some place with a value `v` and elsewhere with a value `w`, and `v<w`, then the solutions generated at the call with `w` may also be used at the place where the call with `v` is occurring. So if we manage to collect all the arguments of places where the same expression is occurring, we may compute the maximal value of the argument, and perform the call only once.

The problem is solved with the introduction of the following two new combinators:

```
--< FRPP.ag

DATA PPC [ -> fmts : Formats ]
  | Indent  Int    PPC
  | Beside  left , right: PPC
  | Above   upper, lower: PPC
  | Choice  opta , optb : PPC
  | Par

DATA PP
  | Apply  PPC  PPList

DATA PPList
  | Nil
  | Cons PP PPList
```

Listing 27: Extending the PP to PPC

- `par` acts as placeholder for a shared expression
- `app` is the takeholder binding held places to the shared expression, as a form of application

Given the new combinators we have to write the previous definition as:

```
hv a b  = (par >|< par >^< par >-< par) `app` [a, b]
example = hv (text "hello") (text "world!")
```

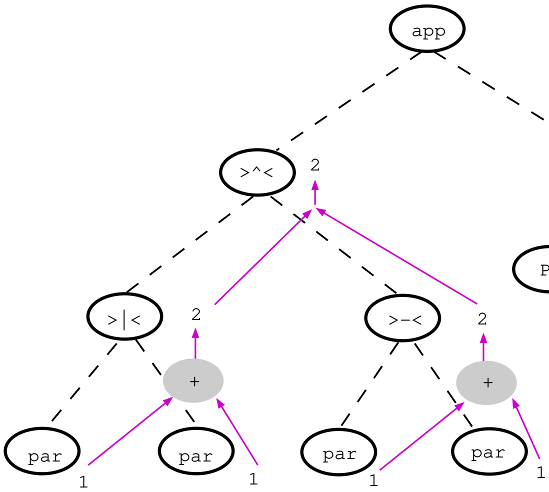
Now, not even knowing the actual value of `a` or `b`, we can still construct efficient combinators for pretty-printing structures.

4.3.1 Extending the grammar with Par an Apply

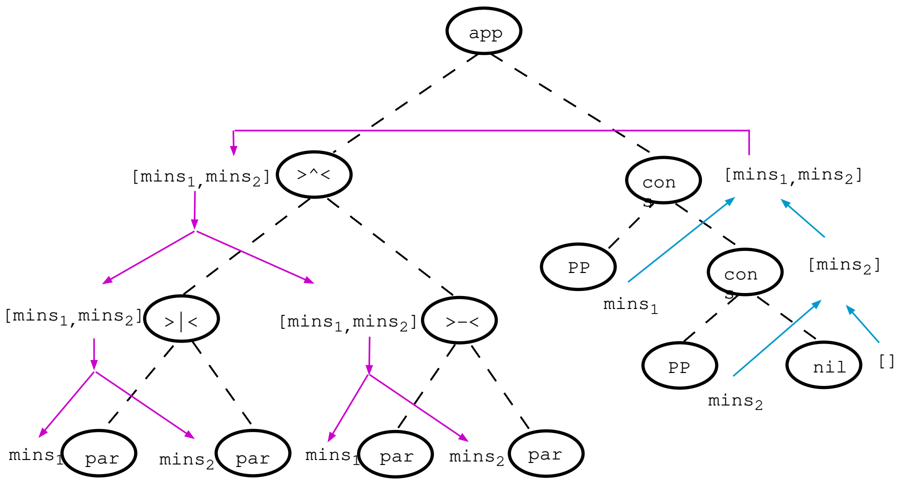
We introduce a different non-terminal in the grammar for those "complicated formats" as shown in listing 27. `text` and `empty` nodes are excluded since they can not contain placeholders.

For the implementation of the `par` and `apply` we proceed as follows:

- Compute for each `>-<` and `>|<` nodes the number (`numpars`) of `par` occurrences in both its subtrees (figure 4.5(a) and listing 28)
- Compute the minimal sizes of the arguments (`fillmins`) and distribute this information over the tree, using the `numpars` computed in the first step (figure 4.5(b) and listing 29)
- Now all sizes of all leaves have become available, we may compute the minimal sizes (`minll` and `minw`) of all nodes (in listing 30), which in their turn may be used to
- compute the frames for all nodes, which also will provide a frame to all the `par` nodes (in listing 30), which information (`reqs`) can be



(a) Collect number of `par`



(b) Collect `fillmins` and distribute them

Figure 4.5: Attribute computations with par and app

```
SEM PPC  [ -> reqs : T_Reqs  numpars : Int ]
  | Beside LHS.reqs     = "left_reqs ++ right_reqs"
              .numpars  = "left_numpars + right_numpars"
  | Above  LHS.reqs     = "upper_reqs ++ lower_reqs"
              .numpars  = "upper_numpars + lower_numpars"
  | Choice LHS.reqs     = "zipWith max opta_reqs optb_reqs"
              .numpars  = "opta_numpars"
  | Par    LHS.reqs     = "[lhs_frame]"
              .numpars  = "1"

TXT
-->type T_Reqs  = [T_Frame]
```

Listing 28: Collecting placeholders

- collected, and compared on the way up (figure 4.6(a) and listing 28), and
- be used at the right argument list of the `app` node to filter the list of solutions of the shared arguments , which
- lists have to be passed down (`fillfmts`) and distributed over the tree (figure 4.6(b) and listing 29)
- When these solutions have reached their final destinations the original computation can take place (figure 4.7 and listing 31).

`Choice` node only distribute the whole set to its children.

Note that because we keep track of the number of placeholders at each node it is possible to detect ill formed expressions: insufficient (or too many) arguments in the rhs of an `Apply` node, i.e. when the shape of the required argument structure does not match with the shape of the actual argument structure. A nicer solution is presented in a subsequent section, where we exploit the language features to solve the problem.

```
SEM PPC  [ fillfmts : T_Fills fillmins : T_Mins <- ]
  | Beside LOC  .e@(lfs,rfs) = "splitAt left_numpars lhs_fillfmts"
                .m@(lfm,rfm) = "splitAt left_numpars lhs_fillmins"
           left .fillfmts = "lfs"
                .fillmins = "lfm"
           right.fillfmts = "rfs"
                .fillmins = "rfm"
  | Above  LOC  .e@(ufs,lfs) = "splitAt upper_numpars lhs_fillfmts"
                .m@(ufm,lfm) = "splitAt upper_numpars lhs_fillmins"
           upper.fillfmts = "ufs"
                .fillmins = "ufm"
           lower.fillfmts = "lfs"
                .fillmins = "lfm"

SEM PP
  | Apply  pPC   . fillfmts = "pPList_fillfmts"
                 . fillmins = "pPList_fillmins"

SEM PPList [  reqs : T_Reqs                                <-
           -> fillfmts : T_Fills fillmins : T_Mins len : Int  ]
  | Nil  LHS    . fillfmts = "[]"
                . fillmins = "[]"
                . len      = "0"
  | Cons pP     . frame    = "head lhs_reqs"
         pPList . reqs     = "tail lhs_reqs"
         LHS    . fillfmts = "(pP_error,pP_fmts ):pPList_fillfmts"
                . fillmins = "(pP_minw ,pP_minll):pPList_fillmins"
                . len      = "pPList_len + 1"

TXT
-->type T_Fills = [(T_Error, Formats)]
-->type T_Mins  = [(Int, Int)]
```

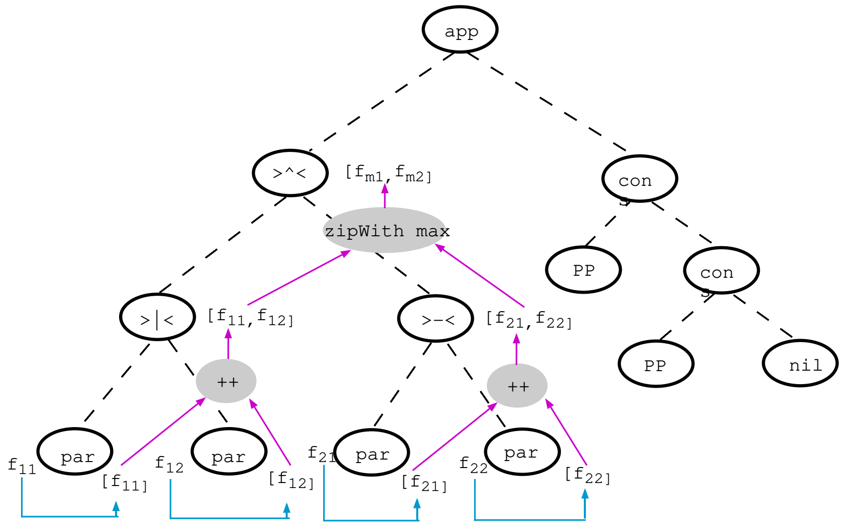
Listing 29: Collecting takeholders and distributing them

```
ATTR PPC [ frame : T_Frame <- ]
SEM PPC
  | Indent pPC   . frame  = "narrow_frame int lhs_frame"
  | Beside left  . frame  = "narrow_ll right_minw lhs_frame"
           right . frame  = "narrow_frame left_minll lhs_frame"

ATTR PPC [ -> minw, minll : Int ]
SEM PPC
  | Beside LHS   . minw   = "left_minw `max` (left_minll + right_minw)"
                 . minll  = "left_minll + right_minll"
  | Above  LHS   . minw   = "upper_minw `max` lower_minw"
  | Choice LHS   . minw   = "opta_minw `min` optb_minw"
                 . minll  = "opta_minll `min` optb_minll"

SEM PPC
  | Par  LOC . m@(minw,minll) = "head lhs_fillmins"
```

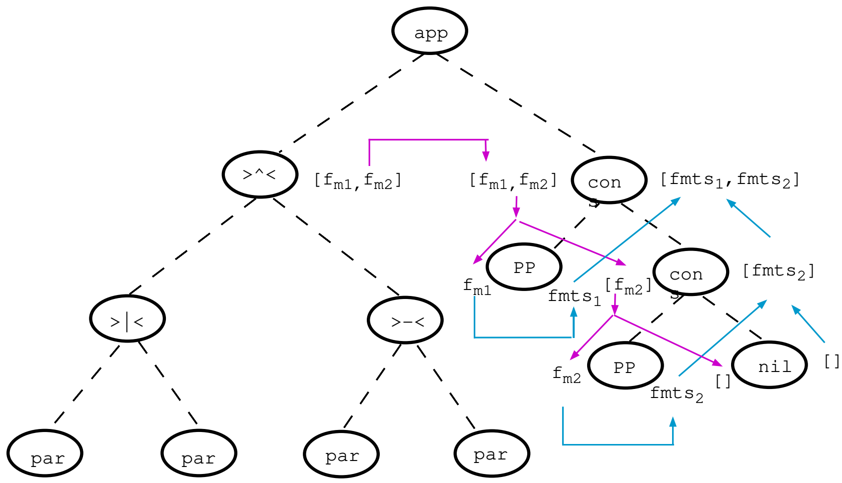
Listing 30: Computing the minimal values

```
SEM PPC
  | Indent  LHS.fmts  = "frame_indent_fmts lhs_frame int pPC_fmts"
  | Beside  LHS.fmts  = "frame_beside_fmts lhs_frame left_fmts right_fmts"
  | Above   LHS.fmts  = "frame_above_fmts lhs_frame upper_fmts lower_fmts"
  | Choice  LHS.fmts  = "frame_choice_fmts lhs_frame opta_fmts optb_fmts"
  | Par     LOC.e@(error,fmts) = "head lhs_fillfmts"

SEM PPC [ -> error USE " ++ " "[]" : T_Error ]
  | Indent  LHS . error = "err (int < 0) 1"
  | Choice  LHS . error = "err (length opta_reqs /= length optb_reqs) 3
                        ++ opta_error ++ optb_error"

SEM PP
  | Apply   LHS . error = "err (pPList_len /= length pPC_reqs) 2"

TXT err
-->                2 -> "incomplete parameter list"
-->                3 -> "incomplete parameter list in choice"
```

Listing 31: Producing the final formats and error messages

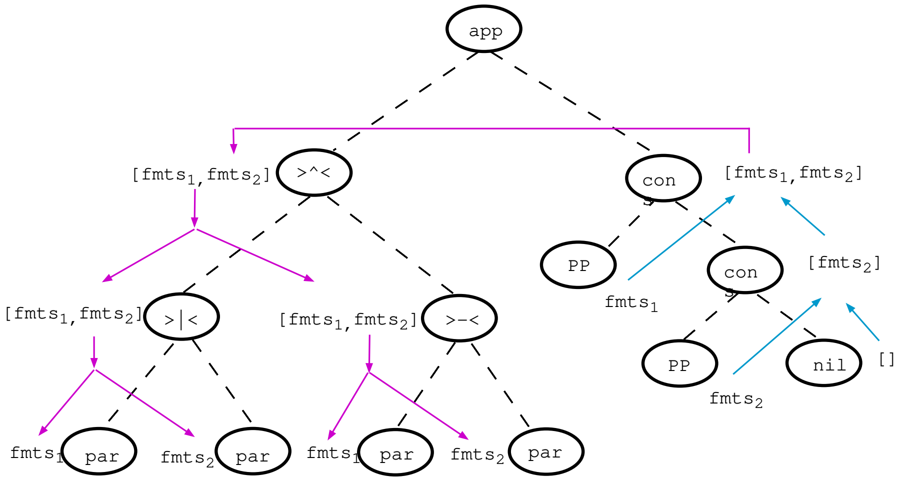


(a) Compute frames at par positions and collect them upwards

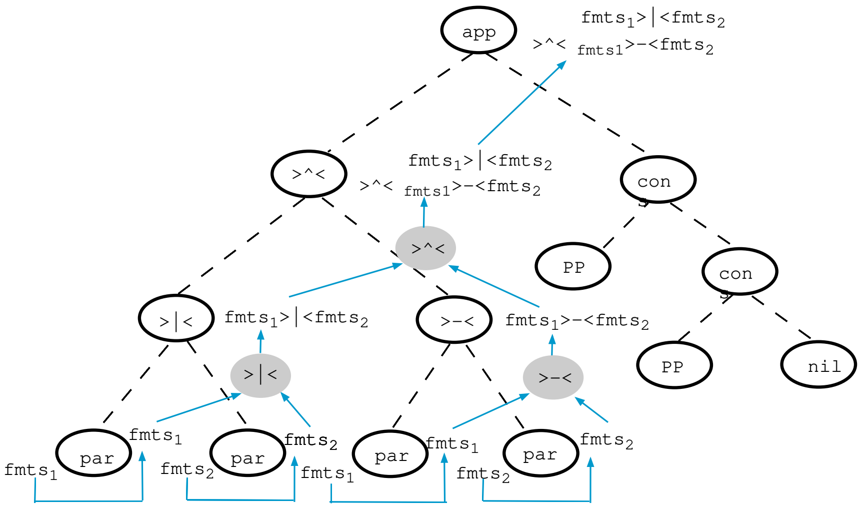


(b) Distribute frames and collect `fillfmts`

Figure 4.6: Attribute computations with par and app (cont.)



(a) Distribute formats



(b) Compute final result

Figure 4.7: Attribute computations with par and app (final)

Chapter 5

Strictification

5.1 Introduction

In a previous section we have remarked that it is always possible to combine a function `f:: a -> b` and a function `g:: c -> d` into a single function `fg:: (a, c) -> (b, d)` having the combined effect. It was this fact that enabled our small system to generate one large catamorphism walking over the tree "once", taking as its argument all inherited attributes and returning as its result all synthesized attributes. From a programmers point of view having this merging of all the separate functionalities into a single function made it quite easy to refer in one computation to results computed in another computation. One may wonder however whether also the reverse transformation is possible and what it might be good for.

For attribute grammars there exists a long tradition in optimizing their implementations in order to achieve efficiencies similar to hand written compilers. In this section we will present some of the analyses and the results of these with respect to our pretty printing combinators. The work we present here is well known in the attribute grammar world, but translates nicely into a functional setting. The overall effect will be that instead of having a single large function which, being lazily evaluated, manages to deal with dependencies from its results to its arguments, we will now construct a set of smaller functions which do not exhibit such behaviour, and can thus be evaluated in a strict way. This implementation technique was chosen in the course of a project in which we wanted to evaluate attribute grammars in an incremental way, using function caching. For this function caching to work well we implemented the transformations needed to convert the program into strict functions, since the memoisation of lazy functions, albeit possible, is not what we want to use at a large scale.

5.2 Pretty Printing Combinators Strictified

If we look at the type of the `Tree` catamorphism generated for the rep-min problem we see that it returns as result a function which takes the computed minimal value as an argument and returns a tuple containing the minimal value and the new tree as a result, so its type is `Int->(Int,Tree)`. When analyzing the overall dependencies between the argument and the result of this function however we may deduce that actually the first component of the result does not depend on the argument in any computation higher up in the tree (only the production `Root` in our case). If we augment the type with arrows indicating this dependency we get its *flow type*, which we have given in figure 5.2.

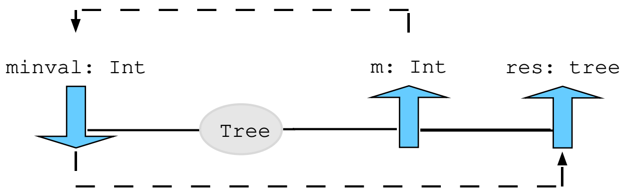


Figure 5.1: Flow type of `Tree`

With the dashed arrow we have indicated the dependencies occurring in the context in which the tree catamorphism is used. The trick in getting rid of this right-to-left dependencies, which necessitate lazy evaluation, is to split the function into two functions, as shown in figure 5.2. If we inspect

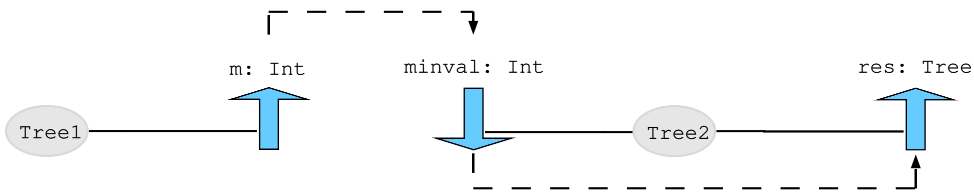


Figure 5.2: Flow types of `Tree1` and `Tree2`

the dependencies between the attributes in the pretty printing libraries (and this can be done automatically) we find the dependencies for the complicated pretty printing type `PPC` as shown in figure 5.3. Initially we may compute the number of `par` occurrences (`numpars`), since this is a purely syntactical issue, and the number does not depend on any other attribute value. Next we can use this number to split the list of minimal sizes the fill-ins will have (`fillsmins`) at the above and beside nodes. This constitutes a "second pass". Once all the sizes of the `pars` have become available we may now return the `minw` and `minll` attributes. They can in their turn be used to adjust the value with which the filtering has to be done (`frame`) when it is being passed down the tree. Now it has become possible to collect the maximal sizes available for the corresponding `par` occurrences, which are now collected, compared and returned in the synthesized attribute `reqs`. This will be used in the application

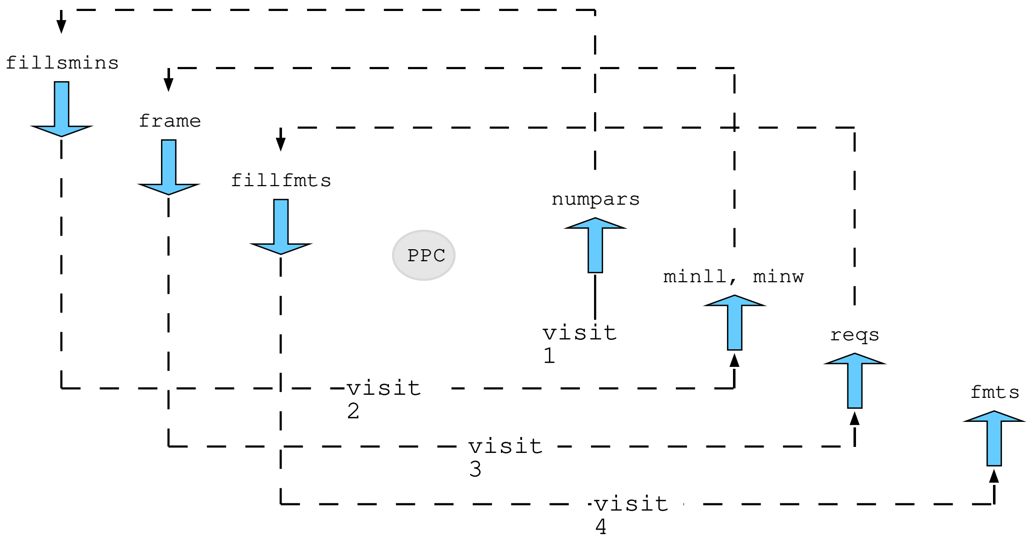


Figure 5.3: Flow type of PPC

node to compute the actual list of formats (`fillfmts`) to be used at the `par`-occurrences, and having available this we may at last construct the sought list of candidate formats for each node in the tree. Although the constructor at each node is only inspected once in this process, thanks to the deforestated approach we have taken, we may say that the tree is "traversed" four times.

The code generated by the attribute grammar system LRC[1] for the combinator `>-<`, in the case children may contain `par` nodes is given in listing listing 32 . Each of the generated functions takes three kinds of arguments:

- values computed in previous visits and still needed in this or one of the later visits. It are such dependencies which make the purely algebraic approach cumbersome to use once more and more computations get intertwined.
- functions constructed at earlier visits which encompass the rest of the work to be done at each of the children (if not completed). The subscript refers to the visit number they stand for.
- inherited attributes that became available since the previous visit and that are enabling further computation in this visit.

[1] http://www.cs.uu.nl/groups/ST/Software

```
lambda_BesideC_1  left_1 right_1
 = ((lambda_BesideC_2  left_numpars right_numpars
     left_2 right_2)
   , numpars)
  where
      (right_2,right_numpars) = right_1
      (left_2 ,left_numpars) = left_1
      numpars             = left_numpars + right_numpars

lambda_BesideC_2  left_numpars right_numpars
                  left_2 right_2
                  fillsmins
 = ((lambda_BesideC_3  left_minll left_numpars right_minw right_numpars
     left_3 right_3)
   , minll , minw)
  where
      left_fillsmins           = take left_numpars fillsmins
      (left_3,left_minll , left_minw) = left_2 left_fillsmins
      right_fillsmins          = drop right_numpars fillsmins
      (right_3, right_minll , right_minw) = right_2 right_fillsmins
      minll  = left_minll + right_minll
      minw   = max left_minw (left_minll + right_minw)

lambda_BesideC_3  left_minll left_numpars right_minw right_numpars
                  left_3 right_3
                  frame
 = ((lambda_BesideC_4  frame left_numpars right_numpars
     left_4 right_4) , reqs)
  where
      left_frame           = narrow_ll right_minw frame
      (left_4,left_reqs)   = left_3 left_frame
      right_frame          = narrow_frame left_minll frame
      (right_4,right_reqs) = right_3 right_frame
      reqs                 = left_reqs + right_reqs

lambda_BesideC_4  frame left_numpars right_numpars
                  left_4 right_4
                  fillsfmts
 = (fmts)
  where
      left_fillsfmts  = take left_numpars fillsfmts
      left_fmts       = left_4 left_fillsfmts
      right_fillsfmts = drop right_numpars fillsfmts
      right_fmts      = right_4 right_fillsfmts
      fmts            = beside_fmts frame left_fmts right_fmts
```

Listing 32: Code generated by LRC for the combinator `>-<`

Bibliography

[1] de Moor O. and Bird R. *Algebra of Programming*. Prentice-Hall, London, 1997. pages 3

[2] Fokker J. Functional parsers. In Jeuring J. and Meijer E., editors, *Advanced Functional Programming*, number 925 in Lecture Notes in Computer Science, pages 1–52. Springer-Verlag, Berlin, 1995. pages 3

[3] Hughes J. The design of a pretty-printing library. In Jeuring J. and Meijer E., editors, *Advanced Functional Programming*, number 925 in Lecture Notes in Computer Science, pages 53–96. Springer-Verlag, Berlin, 1995. pages 3, 4, 33

[4] Jeuring J. and Jansson P. Polytypic programming. In Meijer E.and Sheard T. Launchbury J., editor, *Functional Programming: Second International School*, number 1129 in Lecture Notes in Computer Science, pages xxx–xxx. Springer-Verlag, Berlin, 1996. pages 22

[5] Thomas Johnsson. Attribute grammars as a functional programming paradigm. In G. Kahn, editor, *Functional Programming Languages and Computer Architecture*, volume 274 of *LNCS*, pages 154–173. Springer-Verlag, September 1987. pages 3

[6] Matthijs Kuiper and Doaitse Swierstra. Using attribute grammars to derive efficient functional programs. In *Computing Science in the Netherlands CSN'87*, November 1987. `ftp://ftp.cs.ruu.nl/pub/RUU/CS/techreps/CS-1986/1986-16.ps.gz`. pages 3

[7] Fokkinga M. Meijer E. and Paterson R. Functional programming with bananas, lenses and barbed wire. In Hughes J., editor, *Proceedings of the 1991 ACM Conference on Functional Programming Languages and Computer Architecture*, number 523 in Lecture Notes in Computer Science, pages 124–44. Springer-Verlag, Berlin, 1991. pages 16

[8] Azero P. and Swierstra S.D. Optimal pretty-printing combinators. Available at: `http://www.cs.ruu.nl/groups/ST/Software/PP/`, April 1998. pages 35

[9] Hudak P. Haskore music tutorial. In Meijer E.and Sheard T. Launchbury J., editor, *Functional Programming: Second International School*,

number 1129 in Lecture Notes in Computer Science, pages xxx–xxx. Springer-Verlag, Berlin, 1996. pages 3

[10] Wadler P. Deforestation transforming programs to eliminate trees. *Theoretical Computer Science*, 73:231–48, 1990. pages 13

[11] Wadler P. A prettier printer. Available at: `http://cm.bell-labs.com/cm/cs/who/wadler/topics/recent.html`, March 1998. pages 33

[12] Bird R. Using circular programs to eliminate multiple travesals of data. *Acta Informatica*, 21:239–50, 1984. pages 5, 11

[13] Kieburtz R. and Lewis J. Programming with algebras. In Jeuring J. and Meijer E., editors, *Advanced Functional Programming*, number 925 in Lecture Notes in Computer Science, pages 267–307. Springer-Verlag, Berlin, 1995. pages 3

[14] Swierstra S.D. and de Moor O. Virtual data structures. In Partsch H. Möller B. and Schuman S., editors, *Formal Program Development*, number 755 in Lecture Notes in Computer Science, pages xxx–xxx. Springer-Verlag, Berlin, 1993. pages 13

[15] Swierstra S.D. and Duponcheel L. Deterministic, error correcting combinator parsers. In Meijer E.and Sheard T. Launchbury J., editor, *Functional Programming: Second International School*, number 1129 in Lecture Notes in Computer Science, pages xxx–xxx. Springer-Verlag, Berlin, 1996. pages 3, 12, 13