Designing and Implementing Combinator Languages

S. Doaitse Swierstra[1], Pablo R. Azero Alcocer[1], and João Saraiva[2,1]

[1] Department of Computer Science, Utrecht University, P.O.Box 80.089, 3508 TB Utrecht,The Netherlands, {doaitse,pablo,saraiva}@cs.uu.nl
[2] University of Minho, Braga, Portugal

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; despite this we have been overwhelmed by an everlasting flow of all kinds of special purpose programming languages. Attempts to bridge this gap between a single language and infinitely many caused research into so-called extensible programming languages.

In a certain sense every programming language with a binding construct is extensible. In these lectures we will show that it is the unique combination of *higher order functions*, an *advanced type system* (polymorphism and type classes) and the availability of *lazy evaluation* that 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, we want to spend some time on explaining what actually constitutes a programming language. A proper programming language description contains at least:

- a *concrete context-free grammar*, describing the appearance of the language
- an *abstract context-free grammar*, describing the structure of the language
- *context sensitive conditions* that capture the constraints that 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 directly expressed in a compositional way, or in terms of a fixed-point of a function that 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 program; one of the most common ways of doing so is by giving a *denotational semantics*, which boils down to describing how a function representing the meaning of that program can be derived from the abstract program structure.

Of course one can design a new language by defining all the above components from scratch. 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 and it would be nice if we could borrow this from some existing language.

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 preprocessor. Such extensions are almost all syntactic and do not use any form of global analysis to steer their behavior. An exception to this is the C++ template mechanism, in which the programmer also gets access to the types of the manipulated expressions.
- By incorporating 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 is usually not confluent.
- By giving access to an underlying interpreter, providing reflection. In this way 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, and especially strong typing is easily lost.

Besides these approaches there is a fourth one, that we call *embedding*, described in the next subsection.

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. 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 subsection 1.1:

- for describing the concrete representation (or syntax if you prefer that term) of our extension we 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, that nicely correspond to abstract syntax trees.
- for describing context sensitive restrictions we will use catamorphisms (see the chapter on Generic Programming of these lecture notes and [7]), 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. It is here that the fact

that we can use higher order functions plays a crucial role. For the domains and co-domains of the functions denoting the semantics we may use Haskell types again.

We want to emphasize that this approach has been very fruitful and has already led to several nice combinator libraries[1,10,2,15]. The main advantage of this approach is that when extending a language through the definition of a set of combinators, we get the naming, abstraction and typing mechanism for free, since this was already part of the underlying language.

There are two important aspects of the Haskell tping system that makes this approach even more attractive:

- *polymorphism* allows the language extension to be conservative. I.e. it may be possible to manipulate values of the original programs and at the same time we may guarantee that this is done in a safe way. We will see an example of this when we introduce the parser combinators.
- type classes allow us to link the new constructs to existing types, and to manipulate existing kind of values in a type-safe way without limiting ourselves to a fixed set of predefined types.

As we will see, it is not always attractive to explicitly 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 use in describing programming language extensions should thus 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 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 almost for free [4,6].

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.

1.4 Overview

In section 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 [14,8], and with how to design programs by way of defining 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 are getting more complicated: soon

one needs to be a book-keeping genius to keep track of what you are writing, calculating and combining. In the course of this discussion it will become clear that an approach that solely relies on monads in attacking these problems will not work out as expected.

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

Section 4 forms a large case study in which we attack the pretty printing problem as described in [2]. Hughes defines a set of operators that 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.

Next we will show some of the consequences of our techniques when it is taken in its simplest form, and describe some program transformations, that finally may result in a large set of relatively small strict, pure functions. So even ML-programmers should be happy in the end. Finally we will summarise the approach taken.

2 Compositional Programs

We start by developing a somewhat unconventional 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 such data structures as abstract syntax trees. When computing a property of such a recursive object (i.e. the representation of a program in a new language) we define two kinds of functions: those for describing how to recursively visit the nodes of a tree (the catamorphisms), and those used in forming algebras that describes what to compute at each visited node.

One of the most important steps in this process is deciding what the carrier type of such algebras is to be. Once this step has been taken, these types will be a guideline for further design steps. 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 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 a language in a *semantically compositional* way. We will give three examples of the techniques involved, followed by a conclusion about the strengths and weaknesses of this approach.

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 ([13]). Many have wondered how this

```
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

program achieves its goal, since at first sight it seems that it is impossible to compute anything with this program. We will use this problem, and a sequence of different 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 their associated signature. The *carrier type* of an algebra is that type describing the objects of the algebra. We represent it by a type parameter to the signature type `Tree_Algebra`:

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

The associated evaluation function `cata_Tree` systematically replaces the constructors `Leaf` and `Bin` by their corresponding operations from the algebra `alg` that is passed as an argument.[1]

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 1 an example of an argument with its result is given.

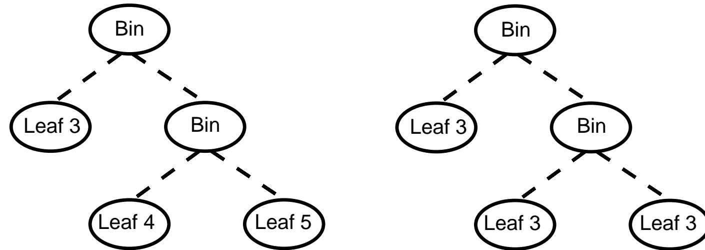


Fig. 1. The function `rep_min`

[1] Note that this function could have been defined using the language PolyP from the second lecture of this volume.

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

Listing 2: rm.sol1

Straightforward Solution The straightforward solution to the Rep_Min problem consists of a function in which `cata_Tree` is called twice: once for computing the minimal leaf value, 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 used as a global variable in the `rep_alg`ebra, that in its turn is an argument to the tree constructing call of `cata_Tree`. In figure 2 we have shown the flow of the data in a recursive call of `cata_Tree`, when computing the minimal value. One of the disadvantages of this solution is that, since

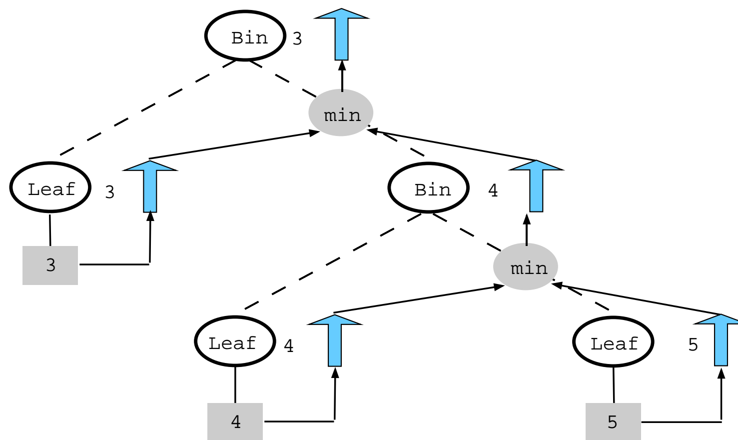


Fig. 2. Computing the minimum value

we call `cata_Tree` twice, in the course of the computation the pattern matching associated with the inspection of the tree nodes is performed twice for each node in the tree. Although this is not a real problem in this solution we will try to construct a solution that calls `cata_Tree` only once. We will do so by transforming the current program in a number of steps.

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

Listing 3: rm.sol2

Lambda Lifting Our first step results in listing 3. In this program the global variable `m` has been removed. The second call of `cata_Tree` now does not construct a `Tree` anymore, but instead *a tree constructing function* of type `Int -> Tree`, that takes the computed minimal value as an argument. Notice how we have emphasized the fact that a function is returned through some superfluous notation: the first lambda in the lambda expressions constituting the algebra `rep_alg` is there because of the signature of the algebra requires so, the second lambda is there because the carrier set of the algebra contains functions of type `Int -> Tree`. This process is done routinely by functional compilers and is known as *lambda-lifting*. In figure 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-arrows to the right correspond to the results 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 (down-arrow), in our case the minimum value, and that returns a `Tree` (up-arrow). The call of `cata_Tree` constructs this final function by using the small functions from the `rep_alg` algebra as building blocks. These small functions can be identified with the small data flow graphs in figure 4.

Tupling Computations In the next formulation of our solution `cata_Tree` is called only once. Note that in the last solution the two calls of `cata_Tree` don't interfere with each other. As a consequence we may perform both computation of the tree constructing function and the minimal value in one traversal, by tupling the results of the computations. The solution is given in listing 4. First a function `tuple_tree` is defined. This function takes two `Tree_Algebra`s as arguments and constructs a third `Tree_Algebra`, that has as its carrier tuples of the carriers of the original algebra's. The resulting computation is shown in figure 5.

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 a single function, that takes as its arguments the cartesian product of all the

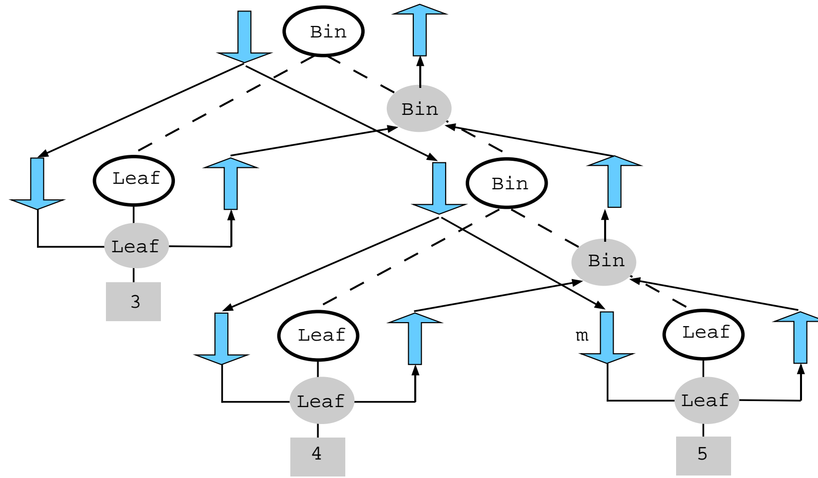


Fig. 3. The flow of information when building the result

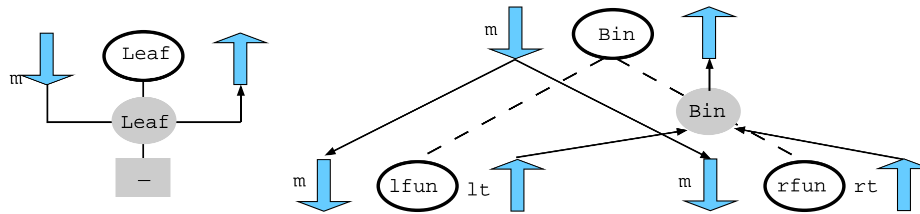


Fig. 4. The building blocks

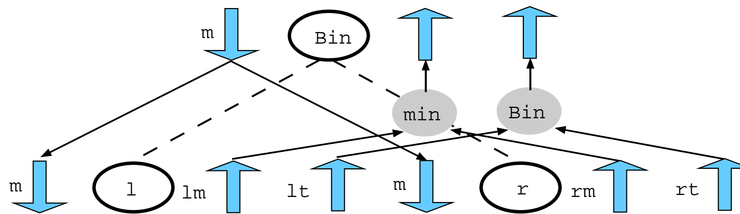


Fig. 5. Tupling the computations

```
infix 9 `tuple_tree`

tuple_tree :: Tree_Algebra a -> Tree_Algebra b -> Tree_Algebra (a,b)
(leaf1, bin1) `tuple_tree` (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_tree` rep_alg)

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

Listing 4: rm.sol3

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)`, that is isomorphic to just `Int`, and the result type becomes `(Int, Tree)`, so as the carrier we get the type `Int -> (Int, Tree)`.[2]

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`, that together achieve the same effect. The new version of our program is given in listing 5.

Notice how we have again introduced extra lambdas in the definition of the functions making up the algebra, in an attempt to make the different rôles of the parameters explicit. 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 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, and especially to those who used to program in LISP or ML, languages that require arguments to be evaluated completely before the call is evaluated (so-called *strict evaluation* in contrast to lazy evaluation). Because of this surprising behavior 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 even defined in terms of itself (as in `ones=1:ones`), although this would not have been a problem.

[2] Notice that the first component of the result does not depend on the `Int`-argument; it is just computed "at the same time" as the `Tree` that does depend on the argument.

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

Finally we give in listing 6 the version of this program in which the function `cata_Tree` has been unfolded, thus obtaining the original solution given in Bird[13].

Recapulating we have systematically transformed a program that inspects each node twice into an equivalent program that inspects each node only once. In doing so we were forced to pass part of the result of a call as an argument to that very same call. Lazy evaluation made this possible.

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

table	→	`<TABLE>`	*rows*	`</TABLE>`
rows	→		*row**	
row	→	`<TR>`	*elems*	`</TR>`
elems	→		*elem**	
elem	→	`<TD>`	*string* | *table*	`</TD>`

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

Fig. 6. Table formatting

A Parser for Tables We start defining the parser for the input language. As we will see the parser actually is a combination of the parsing process and the computation of a catamorphism over the abstract syntax tree constructed by the parsing process.

The parser is written with so-called parser combinators [15] – here mostly defined as infix operators: functions that 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 that provides powerful abstraction techniques is that this automatically gives us an abstraction mechanism for the embedded language (in our case the context-free grammars). Although it is not the main purpose of this paper to treat combinator parsers we have incorporated this part for the sake of completeness, and to show how to link the semantics of a language to recognized program structures.

Parsing with combinators: discovering structure In the first section we have mentioned that, when defining a programming language, we may want to distinguish the concrete syntax from the abstract syntax. In this paper we will assume the availability of a set of parsing combinators, that enables us to construct such a mapping without almost any effort.

Before we describe the structure of the combinator `taggedwith` that will be used to construct a parser for recognizing 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:

```
<*>  :: Eq s => Parser s (a -> b) -> Parser s a -> Parser s b
<|>  :: Eq s => Parser s a        -> Parser s a -> Parser s a
succeed :: a -> Parser s a
sym  :: Eq s => s                             -> Parser s s
--
<$>  :: Eq s => (a -> b)          -> Parser s a -> Parser s b
<*-> :: Eq s => Parser s a        -> Parser s b -> Parser s a
<-*> :: Eq s => Parser s b        -> Parser s a -> Praser s a
<$-> :: Eq s => (a -> b)          -> Parser s c -> Parser s (a -> b)
```

The type `Parser` is rather complicated and has been taken from [15]. Here it suffices to know that a `Parser s a` will recognize a sequence of tokens (`[s]`) and return a value 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 result of the second. The *choice* combinator `<|>` constructs a new parser that may perform the role of either argument parser. The parser combinator `succeed` creates a parser that always succeeds (recognizes the empty string) and returns the argument of `succeed` as its semantic value. The parser combinator `sym` returns a parser that recognizes the terminal symbol represented by its argument. Sequence, choice, succeed and sym form the basic constructors for parsers for context-free languages. In our example we will assume that we have a scanner that maps the input onto a sequence of `Token`s, and that such tokens may be recognized by elementary parsers for keywords (constructed with `pKey`), and for lower case identifiers (`pVarid`).

One of the things to notice here is that the type of the parsers is completely polymorphic in the result of the parsers, and that the definitions of the parser combinators only allow us to combine partial results to the results of complete trees in a type safe manner. Furthermore we have introduced a context `Eq s` to precisely contrain the kind of token sequences we are able to parse.

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 [16,11].

Now let us take a look at the program in 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 call `taggedwith s p` returns only the result recognized by the call of `p` in its body.

```
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 <*-> pSym '>'
        tclose s = pKey ("</")  <*-> pKey s <*-> pSym '>'

format_table :: Alg_Table table rows row elems elem
             -> Parser Token table
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 :: Eq s => 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; sizes are positive
sum_alg   = ((+), 0)  -- Sum all elements
```

Listing 8: List manipulation

The Kleene * in two grammar rules of our Table_Formatting problem are realized by the combinator `pFoldr` (see 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 a parser for `p`-structures. A parser `pFoldr (op,zero) p` recognises a sequence of `p`-structures, and `foldr`s the results using the binary operator `op` to combine results and using `zero` as its unit element; so `pFoldr sum_alg p_Integer` recognises a sequence of integers and returns their sum as a result, provided that `pInteger` recognises a single integer.

Finally we have a look at the function `format_table`. We see that it takes for each nonterminal of the describing grammar an algebra consisting of functions that describe how to construct the semantic value for a production 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 recognizing 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`.

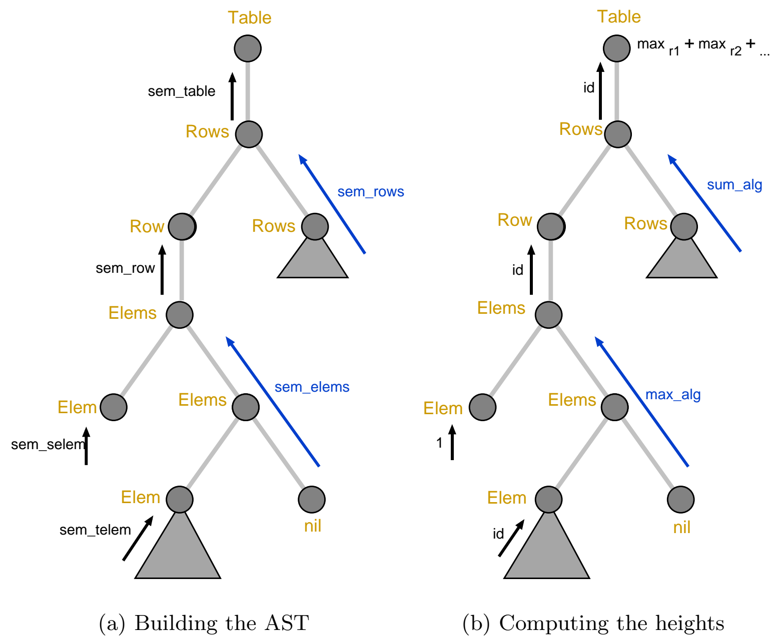
Exercise 1. A more traditional solution to linking the parsing phase with the semantic phase would have been to construct a tree first, that is subsequently mapped onto a final semantic domain using a catamorphism. Define appropriate data types, and the associated catamorphisms. How should the `Alg_Table` show up in your program?

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 compute quite different results. The set of definitions:

```
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))
```

describes the data structure holding the table as the result of the parsing process. The type of the element to be returned by `table` is `Table`. It is already possible in the previous functions to see the role played by the semantic functions and the list algebras – figure 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.

Walks, trees: where are they? In the previous section we have seen how we can define an algebra that describes the computation of the abstract syntax tree



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

Fig. 7. Computations over trees

itself: the *initial algebra*. For defining the semantics of such a tree we now have to define the catamorphism from the initial algebra (the abstract syntax trees) to some other algebra (the meaning). 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. The construction of the abstract syntax tree is fused with the catamophism giving a meaning to this tree, and thus the actual tree never comes into existence.

Computing Heights As a first step to a solution to the Table_Formatting problem we will focus on computing the height of the elements, the rows and the table itself. We will ignore the sizes taken by the dividing lines for a while. Figure 7(b) depicts an attribute grammar view of the solution. 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 the heights of all 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 use `sum_alg`. The complete algebra for computing heights now 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.

Computing the Widths At the table level, the computation of widths deserves a bit of attention. We will not be able to deduce any width of a column until we have recognized the last row of the table. But instead of keeping the widths of all the elements, we maintain a list with the maximum width found for each column. Whenever a complete new row has been recognized, the width of each element has to be compared with the thus far computed width of its corresponding column.

```
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)
5
hw_table = ( id `x` sum, sum_alg `tuple_list` star max_alg
           , id `x` id , max_alg `tuple_list` init_list
           , ( (const 1) `split` length , id `x` id ) )

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

tuple_list :: Alg_List b a -> Alg_List b' a' -> Alg_List (b,b') (a,a')
(f, e) `tuple_list` (f', e')
15  =  (\(x, x') (xs, xs') -> (f x xs, f' x' xs'), (e, e'))
```

Listing 9: Computing heights and widths

For this purpose we introduce the *star* combinator that lifts an algebra to the corresponding algebra on lists:

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

The combinator `star` takes an algebra, and returns an algebra that 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 widths), and deliver them together. With a row we have associated the list of the widths of all its the elements: `init_list`.

Following our algebraic style of programming we define a *tupling* combinator that takes two algebras and returns an algebra that computes a pair of values. In this way it is possible to structure the computations even more. Note that the composition is at the semantic level and not only syntactic.

```
infixr `tuple_list`
tuple_list :: Alg_List b a -> Alg_List b' a'
           -> Alg_List (b,b') (a,a')
(f, e) `tuple_list` (f', e')
  =  (\(x, x') (xs, xs') -> (f x xs, f' x' xs'), (e, e'))
```

Thus we use `max_alg `tuple_list` 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 `tuple_list` 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` applies its argument functions to the corresponding left and right elements of the pair. The new version of the program is shown in listing 9.

Let us note that:

- we can compute several properties of a tree at the same time by tupling them
- computations for such tuples can be constructed out of computations for the elements of the tuples (`tuple_list`, `star`, `split` and `x`)
- 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
- these operators could have been automatically derived using the language PolyP

Exercise 2. Can you provide a tupling operator for table algebras?

Formatting Once we have computed the widths of all columns and the heights of all rows we can start to work on the formatting of the table. The approach will be very similar to the one taken in the Rep_Min problem. Instead of computing the formatted table directly we will compute a function that, once it gets passed the widths of the columns, builds the formatted table. Furthermore the computation of these functions will again be tupled with the computation for the widths and the heights. These table building functions will be constructed out of row-building functions that will construct a formatted row, once they get passed the height of that row and the widths of the columns.

To format the table we do the following: elements are made to be the top-left element of a quarter plane (we call them `Boxes`), extending to the east and the south, see figure 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 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 element, 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.

When the processing of a row has finished we shape the row horizontally. This is possible because the final height of the row is known, and can be passed on to all the boxes. This *surgery* is performed by

```
layout_table
  = ( bot_right . mk_table
    , v_compose `tuple_list` sum_alg `tuple_list` star max_alg
    , \(fmtrow, hwds@(h, wds)) -> (fmtrow h, hwds)
5   , h_compose `tuple_list` max_alg `tuple_list` init_list
    , ( mk_box . ((:[]) `split` (const 1) `split` length)
      , mk_box
      )
    )
```

Listing 10: Computing the formatted table

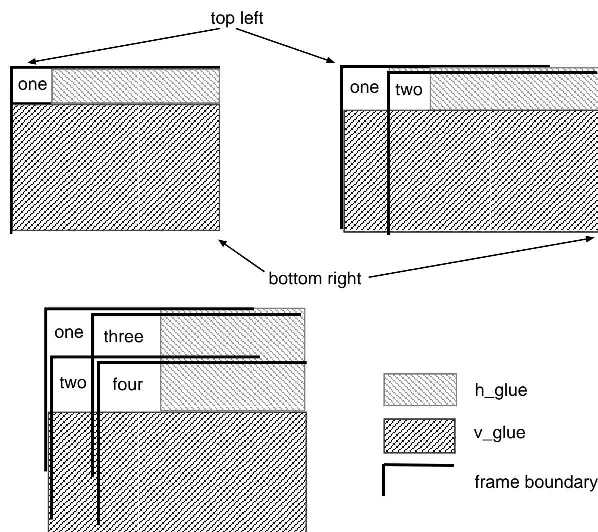


Fig. 8. Superposition of boxes

```
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 = \(fmttable, (h, wds)) -> (fmttable wds, (h, sum wds))
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

`\(fmtrow, hwds@(h, wds)) -> (fmtrow h, hwds)`. We have written this function in a pointwise style, in order to show how the flow of data proceeds.

At the table level, the rows already formatted are `v_composed`. 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 vertically. This is done in `mk_table`. Finally the grid is closed, with `bot_right` placing the bottom and right lines, and correcting the actual size of the table. The implementation of box manipulation functions is given in 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 9(a): an `Elems` cons node takes two arguments, the height and a list of widths, and returns a formatted row, the layout of the element. The arguments 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 the dataflow pattern in the leftmost graph in figure 9(a) is composed of three subgraphs: 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 those subcomputations (`<||>`) and combine this with the upflowing part of the dataflow graph (`<=>>`), see figure 9(b).

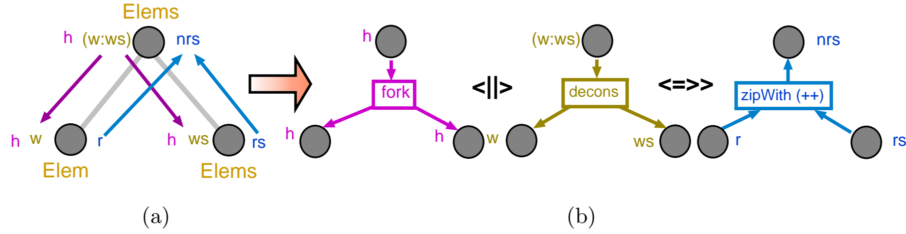


Fig. 9. Attribute computation: (a) example and (b) combining patterns

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.

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

Furthermore, the program can be generalized rather straightforwardly to a polytypic function [3] because the constructors are general. Although not presented here, the tupling operator `tuple_list` 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 of figure 9(a). On the one hand it is nice that we can define such patterns of passing values around by the introduction of extra combinators; on the other hand it is quite cumbersome to have to keep track of which element in the tuples represent what values.

2.3 Defining Catamorphisms

In previous subsections we have defined the catamorphisms needed for the data types involved explicitly. The question arises whether this can be done in a more generic way.

In the chapter on generic programming it has been shown how we can, given a functor, define the initial data type, corresponding to that functor. How to

```
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

define the catamophism corresponding to that functor in Haskell is by now well known: in listing 13 we provide the definitions taken from [5]. Note that if we take this approach we still have to indicate that a specific data type constructor is a functor, by defining the appropriate class instance. In the language PolyP it is no longer necessary to do this last step explicitly since there we can define the function `cata` once and for all as a polytypic function.

Unfortunately these approaches do not readily extend to the case where we have to deal with several mutually recursive data types, like the Table_Formatting problem. Although this is not a problem at all from a theoretical point of view, the actual coding is quite elaborate. In the next subsection we will show how to define two mutually recursive data types and we hope that from that it will be clear how to proceed in the general case.

Mutually Recursive Data Types As the running example we will take the definition of and the code generation for a small block-structured language. The language has two nonterminals: `Expr` and `Decls`. Since our language has two nonterminals, we want to define two mutually recursive data types and thus we start with the definition of a class `Functor_2` for bi-functors (see listing 14). For getting hold of the joint fixpoint of two functors we define the data type constructor `Rec_2`, a straightforward extension of the `Rec` we have seen before. In listing 14 we have used two arguments, but in the most general case we would have n arguments, and by rotating the arguments one step at each next element

```
data Rec f = In (f (Rec  f))

out :: Rec f -> f (Rec f)
out (In o) = o

class Functor f where
  map :: (a -> b) -> (f a -> f b)

cata :: Functor f => (f a -> a) -> Rec f -> a
cata phi = phi . map (cata phi) . out
```

Listing 13: Catamorphisms for a single recursive data type

```
data Rec_2 f1 f2 = In_2 (f1 (Rec_2 f1 f2) (Rec_2 f2 f1))

out_2 :: Rec_2 f -> f (Rec_2 f)
out_2 (In_2 t) = t

class Functor_2 f where
  map_2 :: (a1 -> b1, a2 -> b2) -> (f a1 a2 -> f b1 b2)

cata_2 :: (Functor_2 f, Functor_2 g)
       => ( f e d     -> e, g d e     -> d)
       -> ( Rec_2 f g -> e, Rec_2 g f -> d)
cata_2 phis@(phi1, phi2) = (r1, r2)
  where r1 = phi1 . map_2 (r1, r2) . out_2
        r2 = phi2 . map_2 (r2, r1) . out_2
```

Listing 14: Catamorphisms for two recursive data types

```
data Expr  e d  =  Con_Int   Int
                |  Var       String
                |  If        e e e
                |  Apply     e e
5               |  Where     e d
data Decls d e  =  Dec       d String e
                |  None

type Expr_Algebra  e d = ( Int -> e         -- integer constant
10                       , String -> e      -- variable
                         , e -> e -> e -> e -- conditional expr.
                         , e -> e -> e      -- application
                         , e -> d -> e      -- where clause
                         )
15 type Decls_Algebra d e = ( d -> String -> e  -> d
                         , d
                         )
```

Listing 15: Bi-functors and its associated algebras

in the right and side of `Rec_n` they would all make it once to the first place in the argument list of `Rec_n`.

Finally in listing 14 we define the corresponding mutually recursive bi-catamorphism `cata_2`.

The Block-Structured Language Since our language has two nonterminals we need two recursive data types, and thus define two bi-functors `Expr` and `Decls` in listing 15. Having done this we may define the associated algebras `Expr_Algebra` and `Decls_Algebra`.

Since we have taken a slightly different shape of our algebras in our definition of `cata_2` we also introduce two functions that transform such a tupled representation of an algebra into functions with type `Expr e d -> e` and `Decls d e -> d`:

```
expr_choose :: Expr_Algebra e d -> (Expr e d -> e)
expr_choose (f_Con_Int, f_Var, f_If, f_Apply, f_Where) e
      = case e of
          Con_Int  i       -> f_Con_Int i
          Var      s       -> f_Var s
          If ce te ee      -> f_If    ce te ee
          Apply    e1  e2  -> f_Apply e1 e2
          Where  e d       -> f_Where e d

decls_choose :: Decls_Algebra d e -> (Decls d e -> d)
decl_choose (f_Dec, f_None) d
```

```
instance Functor_2 Expr where
  map_2 (ef, df) e = case e of
                      Con_Int  i      -> Con_Int i
                      Var      s      -> Var s
                      If    ce te ee  -> If (ef ce) (ef te) (ef ee)
                      Apply f  arg    -> Apply (ef f) (ef arg)
                      Where body decl -> Where  (ef body)
                                                (df decl)

instance Functor_2 Decls where
  map_2 (df, ef) d = case d of
                      Dec decls s e -> Dec (df decls) s (ef e)
                      None          -> None
```

Listing 16: Functor_2 instances

```
    = case d of
        Dec d s e -> f_Dec d s e
        None      -> f_None
```

When taking the combined fix-point of these functors the parameter e is going to be replaced by the type of an expression tree, and the parameter d by the list of declarations.

Having done this our next step is to make make the bi-functors Expr and Decls instance of the class Functor2. The definitions are shown in listing 16.

Code Generation Having set up all the above machinery all we have to do is to define the appropriate algebras. We will not go deeply into the kind of code to generate. The following should serve as a good enough description of what we want to achieve:

```
    expression         |             generated code
-----------------------|----------------------------------------
 if 3 then x else y    |   Enter 2, Loadint 4,  Store (1,0),
 where x = 4           |            Loadint 5,  Store (1,1),
       y = 5           |            Loadint 3,  Brfalse  0,
                       |            Load (1,0), Bralways 1,
                       |   Label 0, Load (1,1),
                       |   Label 1,
                       |   Leave 2
```

Notice that we need to introduce an environment that for each declared identifier keeps track of its lexical level and its displacement, and besides that we have to keep track of a label counter for compiling the conditional expressions. We first provide some auxiliary declarations:

```
type Index      =  Int
type Lex_level  =  Int
type Address    =  (Lex_level, Index)
type Env        =  String -> Address
type Lab        =  Int

data Code       =  Loadint  Int      |  Loadbool Bool
                |  Enter    Int      |  Leave    Int
                |  Brfalse  Lab      |  Bralways Lab
                |  Label    Lab
                |  Load     Address  |  Store    Address
                |  Call
                deriving Show
```

Having done all this we can now define the two algebras. As the carrier set for the expressions we take functions mapping the cartesian product of an environment and a lexical level and a label counter onto the generated code and an updated label counter, whereas for the declarations we take functions mapping an environment/level pair and a label counter onto generated code, an integer counting the number of declarations contained in the list processed and an updated label counter. Now, once we have defined two appropriate algebras `sl_expl` and `sl_decls` (see listing 17) we can define our compiler by:

```
compile_expr = code
 where
  (exprf, declf) = cata_2 (expr_choose sl_expr,decl_choose sl_decl)
  (code, _)      = exprf (null_env, 0) 0
```

2.4 Discussion

In this section we have treated three different algorithms: the Rep_Min problem, the Table_Formatting problem, and the definition of a small compiler for expressions. In the Rep_Min problem we have shown how we may construct circular programs. In the Table_Formatting problem we have shown how we may define algebra's in a step wise fashion and define special combinators to construct new algebra's out of other algebra's. Finally we have shown that we may define the concept of a catamorphism once and for all once we know with how many mutually recursive data types we have to deal with.

There is however also a somewhat sobering conclusion: although everything can be done in a very systematic way, it is still a lot of work. We identify the following problems:

1. when moving to higher order domains and composite results we have to keep track of which value is at which parameter position, or at which position in the resulting cartesian products.
2. extending the nonterminals with an extra alternative makes us go through all the data types, algebra's and instance declarations.

```
sl_expr = ( \ i        -> \_       -> \lc -> ([Loadint i   ], lc)
          , \ s        -> \(env,_) -> \lc -> ([Load (env s)], lc)
          , \ ce te ee -> \ el     -> \lc
                       -> let (cc, clc) = ce el (lc + 2)
                              (tc, tlc) = te el clc
                              (ec, elc) = ee el tlc
                          in (    cc
                               ++ [Brfalse lc]
                               ++ tc
                               ++ [Bralways (lc + 1)]
                               ++ [Label lc]
                               ++ ec
                               ++ [Label (lc + 1)]
                             , elc )
          , \ e1  e2   -> \el -> \lc -> let (e1c, lc1) = e1 el lc
                                            (e2c, lc2) = e2 el lc1
                                        in  (e2c++e1c++[Call],lc2)
          , \ e d -> \(env,lev) lc ->
                       let (ec,elc)          = e (denv, lev+1) lc
                           (dc,dnum,denv,dlc) = d (env , lev+1) elc
                       in  ( [Enter dnum]++dc ++ ec++[Leave dnum]
                           , dlc
                           )
          )
sl_decl = (\ d s e -> \el@(env,lev) lc
                      -> let (dc,dnum,denv,dlc) = d el lc
                             (ec, elc)    = e (denv,lev) dlc
                             nenv  = \ss -> if  s == ss
                                            then (lev,dnum)
                                            else denv ss
                         in  ( dc++ec++[Store (nenv s)]
                             , dnum+1
                             , nenv
                             , elc
                             )
          ,           \(env,_)     lc -> ([], 0, env, lc)
          )
```

Listing 17: Algebras for compiling the expression language

3. the construction of the related catamorphisms is cumbersome, especially when we add another related data type. Furthermore the approach taken is unnecessarily complicated since it in principle deals with the case that each data type is reachable from all others in a set of mutually recursive data types. In general this will not be the case.

In the next section we will introduce special syntax in order to cope with these three problems. Unfortunately we will have to leave the semantic compositionality, and replace it with a syntactic one.

3 Attribute Grammars

In the previous section 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 that at some point was 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 Haskell code. The conclusion will be that we can design programs like the ones in the previous section 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 provided by a preprocessor. Our current opinion however is that the advantages of our approach for developing combinator libraries by using a separate attribute grammar formalism allows transformations and ease of formulation that are hard to beat by an approach completely based on semantic composition.

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 section, 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 easily be changed and expanded without having to change the original program texts.

3.1 The Rep_Min Problem

In listing 18 we show the formulation of the Rep_Min problem, using our attribute grammar notation.

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

DATA Root | Root  tree: Tree

-- Computation of the minimum 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

-- 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 the computed minimal value

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

Listing 18: RepMin1.ag

The first two `DATA` declarations introduce the grammar corresponding to the structure of our problem. The `ATTR` declarations specify the inherited and 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, that 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 referred to as `left_res`, whereas an inherited attribute `minval` is referred to as `lhs_minval`, since it is an attribute of the nonterminal at the left hand side of the production. 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 19 is generated.

Exercise 3. 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 20. Here we see that attributes may be declared together with the introduction of a new nonterminal, a new alternative or a new semantic rule. Furthermore many straightforward so-called *copy rules* can easily be inferred by the system. It is the extension of the notation that 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. This approach has the advantage that if we have several attributes following this pattern we do not have to introduce a new monad describing this joint 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 converting the first letter to lower case. 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)
-- catas
sem_Tree (Tree_Leaf i) =  sem_Tree_Leaf  i
sem_Tree (Tree_Bin left right)
 =  sem_Tree_Bin  (sem_Tree left)  (sem_Tree right)
-- funcs
sem_Tree_Leaf i lhs_minval =  ( (Leaf lhs_minval), int )
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` righth_m) )
--------------------- Root ------------------------
data Root = Root_Root Tree
          deriving Show
-- semantic domains
type T_Root =  Tree
-- catas
sem_Root (Root_Root tree) =  sem_Root_Root  (sem_Tree tree)
-- funcs
sem_Root_Root tree = let{ (tree_res, tree_m) = tree tree_m}in tree_res
```

Listing 19: 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 20: 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 21: 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 22: 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 section, by combining algebras we really had semantic compositionality: the algebras could be defined and compiled separately only to be composed at the very last moment.

Since we are 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 strictly 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 that are to be combined by the system.

We start with the grammar in listing 21 that directly corresponds to the type of the abstract syntax trees presented before. In the program in listing 22 we import the previous file (the line `--< TableData.ag`) and introduce for each nonterminal a synthesized attribute containing its minimal height in the formatted table. In listing 23 this version is extended further with the attributes and semantic functions for computing the minimal 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.

```
--< TableHeight.ag

SEM  Table  [ -> mws: Int ]
  |  Table  LHS .mws  = "lmw + 1"
5           LOC .lmw  = "sum rows_mws"
SEM  Rows   [ -> mws: Rowwidths ]
  |  Nil    LHS .mws  = "repeat zero"
  |  Cons   LHS .mws  = "zipWith max row_mws rows_mws"
ATTR Row    [ -> mws: Rowwidths ]
10 ATTR Elems  [ -> mws USE ":" "[]" : Rowwidths ]
SEM  Elem   [ -> mws: Int ]
  |  Str    LHS .mws  = "length string + 1"
  |  Tab    LHS .mws  = "table_mws + 1"
-->type Rowwidths = [Int]
```

Listing 23: TableWidths.ag

In listing 23 we see some other language elements. Lines preceded with `-->` are literally copied into the generated file. In this way additional Haskell functions and type definitions can be passed on to the generated program, thus obviating the need to edit the generated file to contain `import ..` lines. The semantic rule `LOC.lmw = ...` introduces a local attribute, that in the generated semantic function results in the declaration of a local variable `lmw`.

Furthermore it is possible to provide a binary operator and a unit element, together with the introduction of a synthesized attribute (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 the same name are combined using the binary operator, and if no such attributes exist the unit element is taken as its value. We go however a step beyond the kind of polytypism in PolyP since the composition here depends on the name of a part of the result of a child; something that cannot be expressed in the current version of PolyP.

In the next step the downwards distribution of the computed final heights and widths to the individual elements is described, so each element can be formatted according to the actual size it occupies in the formatted table (listing 24). 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 widths flow back into the abstract syntax tree. Finally we add the computation of the final formats, i.e. sequences of lines in listing 25.

In listing 25 we see another extension of the formalism: the `EXT` clause. The effect of this clause here is 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 and thus is interpreted as an external nonterminal. It does not show up as a parameter referring to a child in the generated semantic functions, but nevertheless a call is gen-

```
--< 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 24: TableDistr.ag

```
--< TableDistr.ag

SEM Table [ -> lines : Lines]
  | Table LHS .lines = "bot_right lmw rows_lines"
ATTR Rows [ -> lines USE "++" "[]" : Lines]
ATTR Row  [ -> lines : Lines]
SEM Elems [ -> lines : Lines]
  | Nil   LHS .lines = "repeat []"
  | Cons  LHS .lines = "zipWith (++) top_Left_ls elems_lines"
          LOC .haws  = "head lhs_aws"
SEM Elem  [ -> lines : Lines]
  | Str   LHS .lines = "[string]"

-->type Lines = [String]

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

Listing 25: 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 26: TableFinal.ag

erated. In this way we may incorporate calls to external computations in the generated semantic functions.

We now come to a final convention: if an inherited attribute has been declared and in the rule an attribute with that name would be allowed as a semantic function such semantic functions are generated automatically. So in listing 25 we actually have inserted a call to an external function, passing on some of the available attributes. The final addition of some glue is given in listing 26.

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 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 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 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 parent too (`lhs_def`). If this is the case, that value is taken. If not it is checked whether its left hand side neighbor `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  ((ite_code ce_code te_code ee_code lhs_labels), ee_labels)
```

and we see that the `Labels`-value is nicely 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 that are composed with the `USE` clause.

4 Pretty Printing

In this section we attack the pretty-printing problem as described in [2,12]. 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 10(a). 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.

```
a. IF c THEN t ELSE e FI
b. IF c THEN t
        ELSE e
   FI
c. IF c
   THEN t
   ELSE e
   FI
```

(a) Possible layouts

```
pp_ites c t e
  =   ifc >|<  thent >|< elsee  >|< fi
  >^< ifc >|< (thent >-< elsee) >-< fi
  >^< ifc >-<  thent >-< elsee  >-< fi
  where ifc   = text "if"   >|< c
        thent = text "then" >|< t
        elsee = text "else" >|< e
        fi    = text "fi"
```

(b) Specification

Fig. 10. Pretty-printing an `IF-THEN-ELSE-FI` structure

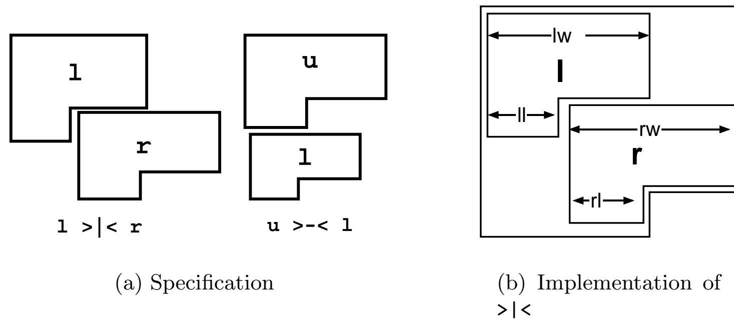
We define a layout to be *optimal* (nicest or prettiest) if it takes the least number of lines, while still not overflowing the right page margin. The examples in figure 10(a) are optimal for page widths 40, 28, and 15 respectively. Taking the layout *b.* with respect to a page width of 35 is thus considered non optimal.

Our approach is based on the relation between the height and the width of a layout: higher when elements cannot be placed next to each other 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 10(a).

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 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.

A possible description of the possible layouts is shown in figure 10(b).

A possible description of the possible layouts is shown in figure 10(b). The function `text` converts strings into layouts, `>|<` places its two arguments beside each other, `>-<` places them above each other and `>^<` combines two possible layouts. In addition to 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 11(a).



(a) Specification (b) Implementation of >|<

Fig. 11. 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 27. We rely on the existence of a set of basic combinators that generate alternative layouts as lists sorted by decreasing width and increasing height, 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 [9].

In the attribute grammar of listing 27 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.

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 nonterminals that stand for external function calls), as you can see in listing 28. [3]

Since we want to be able to work with many different versions of our basic combinators we indicate the system to prefix all generated calls with `pw_` using the directive `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 11(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`.

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 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`. A frame contains two values: one describing the total width available for representing the text in the tree it is associated with and one for describing how much space is at most available to the last line of that text.

Now have a look at the example in figure 12, and assume a page width of 20. At the root node we start with $(20, 20)$, that is the bound 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) (see figure 11(b)).

[3] One might compare this with the effort to convert the program into monadic form in order to use a reader monad.

```
-- 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 27: 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 28: Filtering with page width

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 `c` can be discarded locally, 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.

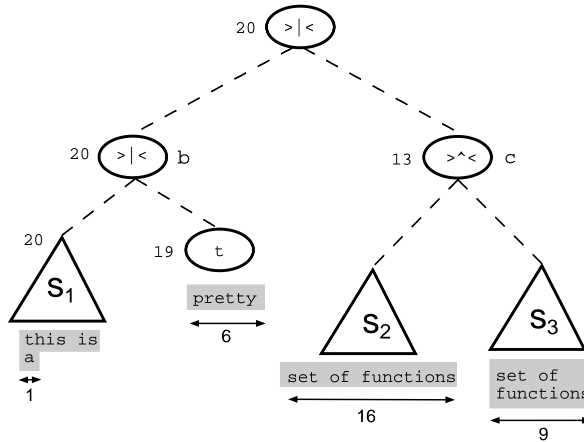


Fig. 12. The frame limit

In listing 29 we show how to compute the minimal space used by a node, that is needed to compute the frames for its fellow nodes. In figure 13(a) we depict the attribute computations involved in the operation `>|<`.

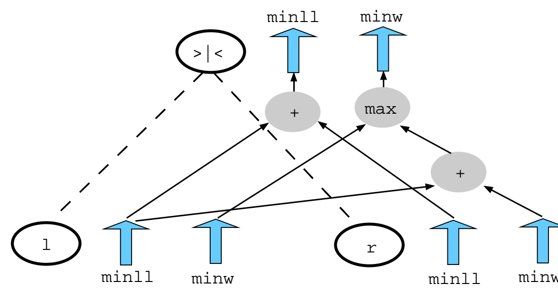
In listing 30 the semantic functions for passing frames downwards are shown, and an illustration of the data flow for the operation `>|<` is shown in figure 13(b). Recall that we do not have to code all data flows, only the relevant computations are made explicit. Copy rules involving passing information around are generated automatically as explained 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 31 we add the synthesis of the formats and an attribute for handling error conditions.

Exercise 4. 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 program `FRPP.ag` so that the computation of heights is included.

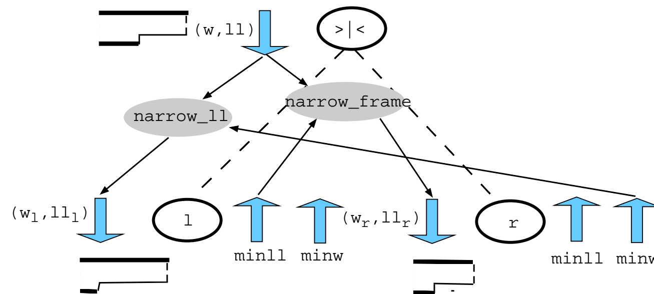
Before starting to read the next section it is useful if you try to solve the following problem. The combinator `hv :: Formats -> Formats -> Formats` has the following behavior:

```
--< 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"
  | Above  LHS.minll = "lower_minll"
  | Choice LHS.minw  = "opta_minw `min` optb_minw"
              .minll = "opta_minll `min` optb_minll"
```

Listing 29: Computing min bounds: `FRPP.ag`



(a) Min limits for `>|<`



(b) Frame limits for `>|<`

Fig. 13. 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,lhs_pw)"

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

Listing 30: 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 31: Error and formats: `FRPP.ag`(end)

```
? 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.

Exercise 5. 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 functions to which secretly a frame is passed. Thus each occurrence of `a` and `b` in the body of `hv` leads to a separate computation. We have thus lost *sharing* as an unfortunate consequence of moving to 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:

par acts as placeholder for a shared expression
apply binds the shared expression to their placetakers as a form of β-reduction

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

```
--< 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 32: Extending the PP to PPC

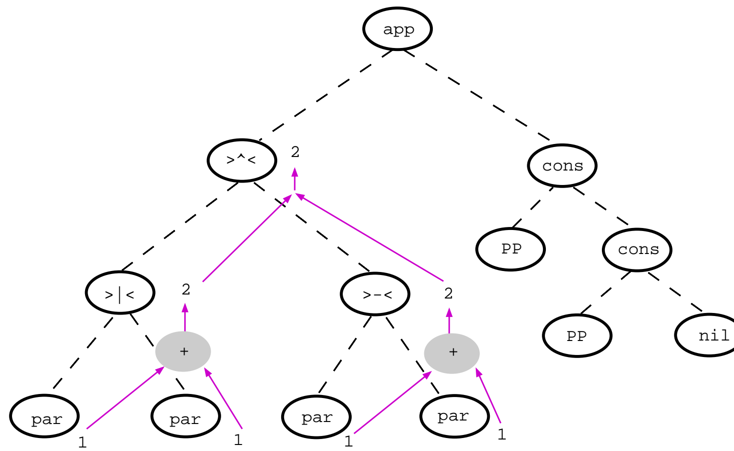
```
hv a b  = (par >|< par >^< par >-< par) ‘app‘ [a, b]
example = hv (text "hello") (text "world!")
```

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

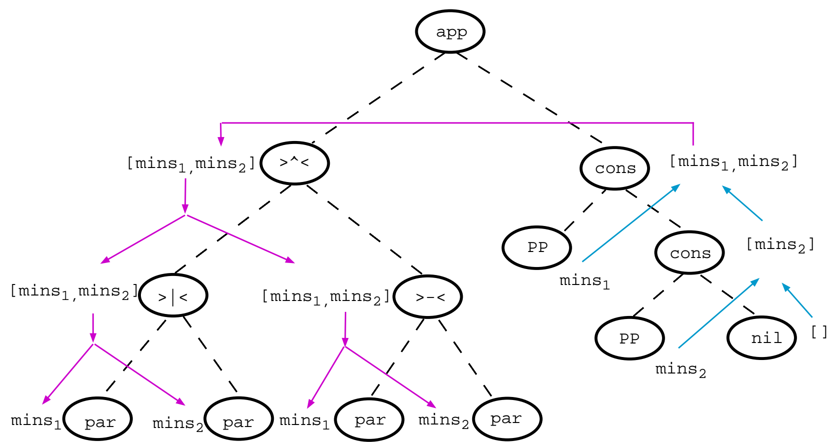
Extending the grammar with Par and Apply We introduce a different non-terminal in the grammar for those “complicated formats” as shown in listing 32. `text` and `empty` nodes are excluded since they can not contain placeholders.

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

- Compute for each `>-<` and `>|<` nodes the number (`numpars`) of `par` occurrences in both its subtrees (figure 14(a) and listing 33)
- Compute the minimal sizes of the arguments (`fillmins`) and distribute this information over the tree, using the `numpars` computed in the first step (figure 14(b) and listing 34)
- Now all sizes of all leaves have become available, we may compute the minimal sizes (`minll` and `minw`) of all nodes (in listing 35), that in their turn may be used to
- Compute the frames for all nodes, that also will provide a frame to all the `par` nodes (in listing 35), which information (`reqs`) can be
- Collected, and compared on the way up (figure 15(a) and listing 33), 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 15(b) and listing 34)
- When these solutions have reached their final destinations the original computation can take place (figure 16 and listing 36).



(a) Collect number of **par**



(b) Collect **fillmins** and distribute them

Fig. 14. 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 33: Collecting placeholders

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 `app` node, i.e. when the shape of the required argument does not match the shape of the actual argument.

4.4 Discussion

In this section we developed combinators for a language whose elements are not taken from a flat domain: instead they depend on values that are not known statically. By going to a higher-order domain not all the properties of first class elements are available (in our case we lost sharing). We lost thus support from the host language, having to reintroduce the mechanism to recover those properties. The presented mechanism is still incomplete, because we can have common subexpressions that we may want to share. Back to the example of figure 10, we now can write:

```
pp_ite c t e
  =       par  >>|<<   par  >>|<<  par   >>|<<  par
    >>^<<  par  >>|<<  (par  >>-<<  par)  >>-<<  par
    >>^<<  par  >>-<<   par  >>-<<  par   >>-<<  par
    >>$<  [ text "if"   >|< c
          , text "then" >|< t
          , text "else" >|< e
          , text "fi"
          ]
```

As we can see, there are still subexpressions that are combinations similar to the previous `hv` structure and we may want to share. A solution of this problem is to extend the combinator `app` to accept placeholders in the list of placetakers. This extension together with a complete mechanism for error manipulation is used in

```
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"
5               .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"
10         upper.fillfmts = "ufs"
                .fillmins = "ufm"
           lower.fillfmts = "lfs"
                .fillmins = "lfm"

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

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

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

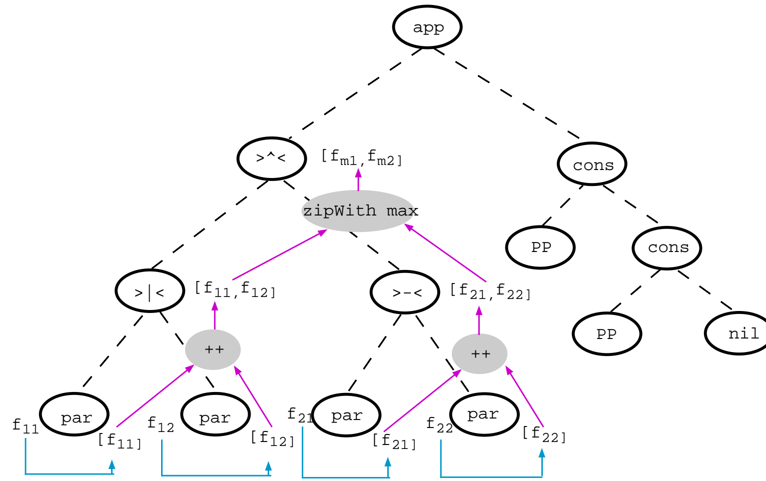
Listing 34: Collecting placetakers and distributing them

```
SEM PPC [ frame: T_Frame <- ]
  | 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"

SEM PPC [ -> minw, minll: Int ]
  | 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"
```

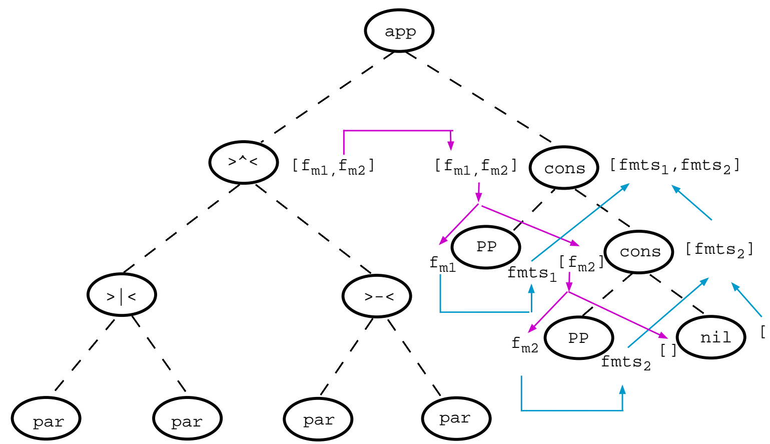
Listing 35: Computing the minimal values

```
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 36: Producing the final formats and error messages

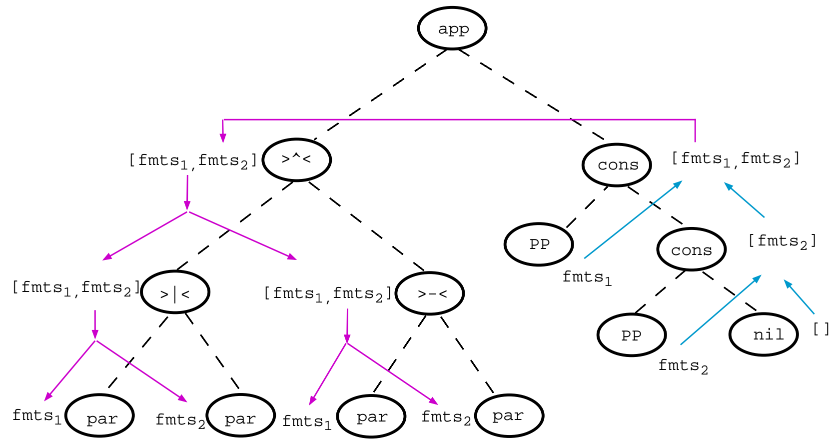


(a) Compute frames at par positions and collect them upwards

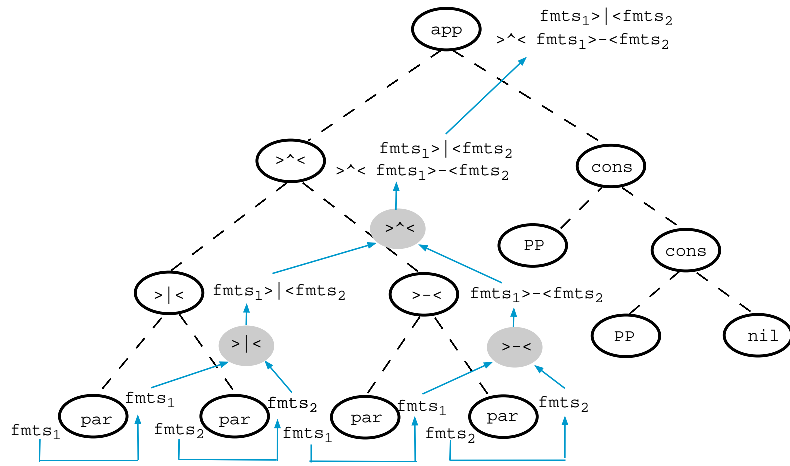


(b) Distribute frames and collect **fillfmts**

Fig. 15. Attribute computations with par and app (cont.)



(a) Distribute formats



(b) Compute final result

Fig. 16. Attribute computations with par and app (final)

the actual version of our pretty printing combinator library.[4] We think it is a convincing example of the need of mechanical support for the implementation of advanced combinator languages. The attribute grammar paradigm has been indispensable in getting the correct implementation.

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)` that has the combined effect. It is this fact that enables 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 makes it quite easy to refer in one computation to results computed in another computation. However one may wonder 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 that, lazily evaluated, manages to deal with dependencies from its results to its arguments, we will now construct a set of smaller functions that do not exhibit such behavior, 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 that 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*, that we have given in figure 5.2.

[4] See the combinator's web site: `http://www.cs.uu.nl/groups/ST/Software`.

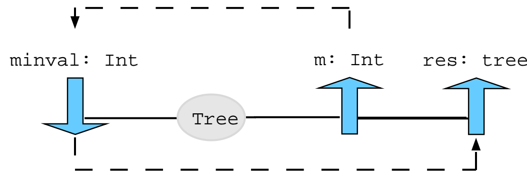


Fig. 17. Flow type of `Tree`

With the dashed arrow we indicate the dependencies occurring in the context in which the tree catamorphism is used. The trick in getting rid of these right-to-left dependencies, that demand lazy evaluation, is to split the function into two functions, as shown in figure 18. If we inspect the dependencies between the

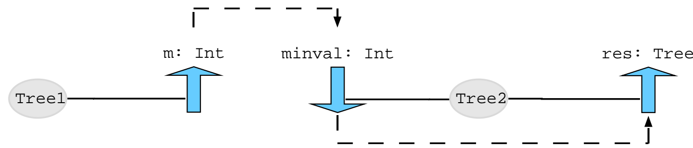


Fig. 18. Flow types of `Tree1` and `Tree2`

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 19. 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 (`fillmins`) at the above and beside nodes. This constitutes a "second pass". Once all the sizes of the `par`s 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 passed down the tree. Now it has become possible to collect the maximal sizes available for the corresponding `par` occurrences, that are collected, compared and returned in the synthesized attribute `reqs`. This will be used in the application 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

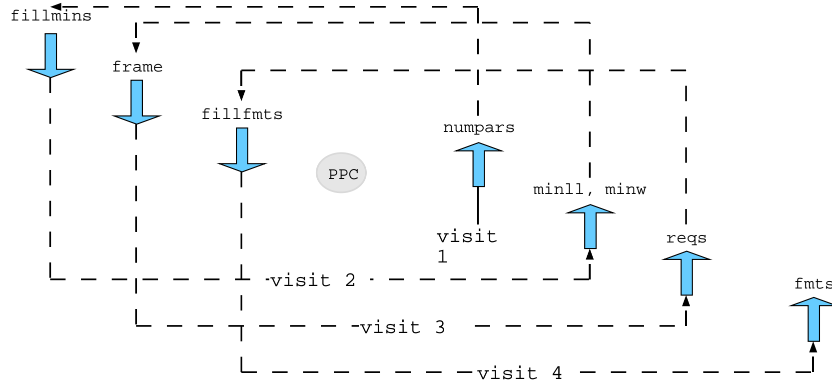


Fig. 19. Flow type of PPC

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[5] for the combinator `>-<`, in the case children may contain `par` nodes, is given in listing 37. 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. These dependencies make the purely algebraic approach cumbersome to use when more and more computations get intertwined.
- functions constructed at earlier visits that encompass the rest of the work to be done at each of the children (if not completed). The subscripts refer 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.

6 Conclusions

In the beginning of this lecture we have argued that one should not take the step to designing a new language too easily, and in one of the later chapters we have introduced a new language ourselves for describing attribute grammars. The question that thus arises immediately is whether it would have been possible to describe this way of programming again by the introduction of a suitable set of combinators. Recently steps have been taken in pursuing this direction by Oege de Moor in Oxford based on the concept of extensible records. Although we think that our notation still has a lot to offer we hope it will be feasible with future versions of the Haskell type system to achieve the ease of formulation provided by our small system. Wouldn't it be nice if we could describe the threading

[5] 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 37: Code generated by LRC for the combinator `>-<`

of an attribute by means of a function manipulating a grammatical structure, and wouldn't it even be nicer if we could have programs analyse themselves and transform themselves into the strict form as shown in the previous section, without having to go through a separate system.

For the time being we hope to have shown that thinking in terms of attribute grammars allows one to design new combinator languages in an incremental way, and to implement them in an efficient way. The pretty printing combinators have been a joy to study and implement and we hope that you agree with us.

Aknowledgements We want to thank all the people who have been working with us in recent years on the problems described. We want to thank especially Johan Jeuring, David Barton and Eelco Visser for providing comments on the paper and Oege de Moor, Ganesh Sittampalam and our students at Utrecht University for using the attribute grammar system.

References

1. 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 152
2. 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 152, 153, 185
3. Jeuring J. and Jansson P. Polytypic programming. In Meijer E. Launchbury J. and Sheard T., editors, *Advanced Functional Programming: Second International School*, number 1129 in Lecture Notes in Computer Science, pages 68–114. Springer-Verlag, Berlin, 1996. pages 170
4. T. Johnsson. Attribute grammars as a functional programming paradigm. In G. Kahn, editor, *Functional Programming Languages and Computer Architecture*, volume 274 of *Lecture Notes in Computer Science*, pages 154–173. Springer-Verlag, September 1987. pages 152
5. M. P. Jones. Functional programming with overloading and higher-order polymorphism. In Jeuring J. and Meijer E., editors, *Advanced Functional Programming*, number 925 in Lecture Notes in Computer Science, pages 97–136. Springer-Verlag, Berlin, 1995. pages 171
6. M. Kuiper and S. D. 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 152
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 151
8. O. de Moor and R. Bird. *Algebra of Programming*. Prentice-Hall, London, 1997. pages 152
9. Azero P. and Swierstra S.D. Optimal pretty-printing combinators. Available at: `http://www.cs.ruu.nl/groups/ST/Software/PP/`, April 1998. pages 188

10. Hudak P. Haskore music tutorial. In Meijer E. Launchbury J. and Sheard T., editors, *Advanced Functional Programming: Second International School*, number 1129 in Lecture Notes in Computer Science, pages 38–67. Springer-Verlag, Berlin, 1996. pages 152
11. Wadler P. Deforestation transforming programs to eliminate trees. *Theoretical Computer Science*, 73:231–48, 1990. pages 161
12. Wadler P. A prettier printer. Available at: `http://cm.bell-labs.com/cm/cs/who/wadler/topics/recent.html`, March 1998. pages 185
13. Bird R. Using circular programs to eliminate multiple travesals of data. *Acta Informatica*, 21:239–50, 1984. pages 153, 159
14. 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 152
15. Swierstra S.D. and Duponcheel L. Deterministic, error correcting combinator parsers. In Meijer E.and Sheard T. Launchbury J., editor, *Advanced Functional Programming: Second International School*, number 1129 in Lecture Notes in Computer Science, pages 184–207. Springer-Verlag, Berlin, 1996. pages 152, 160, 161
16. S.D. Swierstra and O. de Moor. Virtual data structures. In Partsch H. Möller B. and Schuman S., editors, *Formal Program Development*, number 755 in Lecture Notes in Computer Science, pages 355–371. Springer-Verlag, Berlin, 1993. pages 161

Table of Contents

Designing and Implementing Combinator Languages 150
S. Doaitse Swierstra, Pablo R. Azero Alcocer, João Saraiva
1 Introduction 150
1.1 Defining Languages 150
1.2 Extending Languages 151
1.3 Embedding languages 151
1.4 Overview 152
2 Compositional Programs 153
2.1 The Rep_Min problem 153
2.2 Table_Formatting 159
2.3 Defining Catamorphisms 170
2.4 Discussion 175
3 Attribute Grammars 177
3.1 The Rep_Min Problem 177
3.2 The Table_Formatting Problem 181
3.3 Comparison with Monadic Approach 184
4 Pretty Printing 185
4.1 The general Approach 187
4.2 Improving Filtering 188
4.3 Loss of Sharing in Computations 193
4.4 Discussion 196
5 Strictification 201
5.1 Introduction 201
5.2 Pretty Printing Combinators Strictified 201
6 Conclusions 203