14

Attribute grammars in the functional style

S.D. Swierstra and P.R. Azero
Dept. of Computer Science, Utrecht University
P.O.Box 80.089, 3508 TB Utrecht, The Netherlands
tel: +(31)(30) 253 1454, fax: +(31)(30) 251 3791
e-mail: {swierstra,pablo}@cs.ruu.nl

Abstract

For a long time, attribute grammars have formed an isolated programming formalism. We show how we may embed the attribute grammar approach in a modern functional programming language. The advantages of both sides reinforce each other: the former provides compositionality and the latter naming abstraction and higher-orderness. Through a sequence of program transformations we show different aspects of the techniques involved. We conclude with the observation that an attribute grammar oriented algorithm development may be a fruitful one, and may go hand in hand with a more algebraic style of program development.

Keywords

attribute grammars, functional programming, parsing, pretty printing

1 INTRODUCTION

The evolution of programming languages during the past years is affecting the way we build software. Especially when using the new generation of functional programming languages like ML or Haskell, we may use powerful abstraction and typing mechanisms, both based on a sound mathematical foundation. We advocate the use of such languages as system implementation languages.

When constructing a new combinator library one is extending an existing programming language with a new sub-language; the naming, typing and abstraction mechanisms are borrowed from the original language (Fokker 1995, Hudak et al. 1996, Hughes 1995, Swierstra et al. 1996, Azero et al. 1997). It should come as no surprise that the attribute grammar formalism, which traditionally has been used for describing implementations of programming languages, could be a source of inspiration here. However for making use of the ideas of attributes grammars one has always been limited to choosing a specific source formalism and a specific target language. Fortunately, as we

Systems Implementation 2000 R. Nigel Horspool (Ed.)

will show, it is nowadays quite straightforward to use the attribute grammar based way of thinking when programming in the setting of a modern, lazily evaluated functional language: it is the declarative way of thinking in both formalisms which bridges the gap!

Thinking in terms of attribute grammars is a useful approach to 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.

Our approach is demonstrated by a working example: a table formatter. We start in section 2 with its specification, and a functional parser for the language describing tables. In the next step we enrich the parser with attribute grammar style definitions. In section 3 we add attribute computations for the heights of the elements by introducing appropriate algebras. In section 4 we add the computation of the widths of the table elements, and show how to combine height and width computations. Section 5 describes how to compute the formatted table, and in section 6 we present some conclusions.

2 A PARSER FOR TABLES

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

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

An example of accepted input and the associated output is given in figure 1.

```
<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)

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

(b)

Figure 1 Table formatting: (a) input and (b) associated output

Our first version of the table formatter parses the input and returns the abstract syntax tree. In subsequent sections we modify parts of it to com-

pute the actual formatting. The program is written in the programming language Haskell (Hammond et al. 1997), and uses so-called parser combinators (Fokker 1995) – here mostly defined as infix operators: functions which construct parsers out of more elementary parsers, completely analogous to the well-known recursive descent parsing technique. An example of the advantages of embedding a formalism (in our case context-free grammars) in a language which provides powerful abstraction techniques is that this automatically gives us an abstraction mechanism for the embedded language (in our case the context-free grammars). Since we already have a naming mechanism available we do not have to deal separately with the concept of **nonterminal**.

2.1 Parsing with combinators: giving structure

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

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

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

The **sequence** combinator `<+>`, composes two parsers sequentially. The meaning of the combined result is computed by applying the result of the first component to the second. The **choice** combinator `<|>` constructs a new parser which may perform the role of either argument parser. Finally, `succeed`, the parser that returns a parser that always succeeds (recognises the empty string) and returns the argument of `succeed` as its semantic value. Sequence, choice and succeed form, together with `sptoken` which recognises just its argument, the basic constructors of parsers for context free languages.

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

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

Thus, it applies a function, the so called **semantic function**, to the semantic results parsing the structures. 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 (Swierstra et al. 1993, Wadler 1990).

Now let us take a look at the program in figure 2, 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.

```
type Alg_List a b = ( b, a -> b -> b)
type Alg_Table t rs r es e
  = (rs  -> t,Alg_List r rs,es -> r,Alg_List e es,(String -> e,t -> e))

taggedwith :: String -> Parser a -> Parser a
taggedwith s p = (\_ y _ -> y) <$> topen s <+> p <+> tclose s
  where topen  s = sptoken ("<"  ++ s ++ ">")
        tclose s = sptoken ("</" ++ s ++ ">")

format_table :: Alg_Table t rs r es e -> Parser Char t
format_table (sem_table,sem_rows,sem_row,sem_elems
             ,(sem_selem,sem_telem)               )
  = let table = sem_table <$> taggedwith "TABLE"
                  (p_list sem_rows (taggedwith "TR"
                     (sem_row <$> p_list sem_elems (taggedwith "TD"
                        (sem_selem  <$> spstring
                        <|> sem_telem <$> table)))))
    in  table
```

Figure 2 Parsing tables

Its semantics are: recognise the 'open' tag `s`, then (combinator `<+>`) recognise the structure `p`, then (again `<+>`) parse the 'close' tag. The lambda expression in front of the `<$>` takes the three recognised elements and returns the middle one, since this is the only part of the recognised string which bears meaning.

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

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

p_list :: Alg_List a b -> Parser c a -> Parser c b
p_list    alg@(zero, op)  p          =  op <$> p <+> p_list alg p
                                    <|> succeed zero

-- Some useful algebras
init_list = ([], (:))
max_alg   = (0 , max)      -- Compute the max element of the list
sum_alg   = (0 , (+))      -- Compute the sum of the list
```

Figure 3 Parsing tables

The Kleene * from the two first table rules is realised by the combinator `p_list` (see figure 3). The first argument of `p_list` is a tuple of two values: `(zero,op)`, 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 p_list is a parser for p-structures.

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

Simple elements like strings and tokens can be parsed with the combinators spstring and sptoken respectively. All the defined parsers take streams of characters as input. The type of the results depend on what we want to do with the recognised structure. We will deal with this in the next section.

2.2 Simulating structure walks: adding semantics

By providing different definitions for the algebras passed to the p_list-calls and for the sem_antic functions we may describe quite different results. Note that it is the completely polymorphic formulation of our parsing algorithm which allows us to provide such different definitions. Our first set of definitions will deliver a data structure holding the table:

```
type Table = Rows
type Rows  = [ Row ]
type Row   = Elems
type Elems = [ Elem ]
data Elem  = SElem String | TElem Table

table = format_table (id,init_list,id,init_list,(SElem,TElem))
```

The type of the element returned by table is Table. We always look at the type of the first parameter of the table algebra. It is already possible in the previous functions to see the role played by the semantic functions and the list algebras – figure 4(a). The latter apply functions to the collected elements, and the former provide intermediate computations such as transforming data types, collecting intermediate values and computing new values. In the following sections we will focus on the systematic description of these functions.

We only give a polymorphic collection of functions (the algebra for tables) corresponding to such computations.

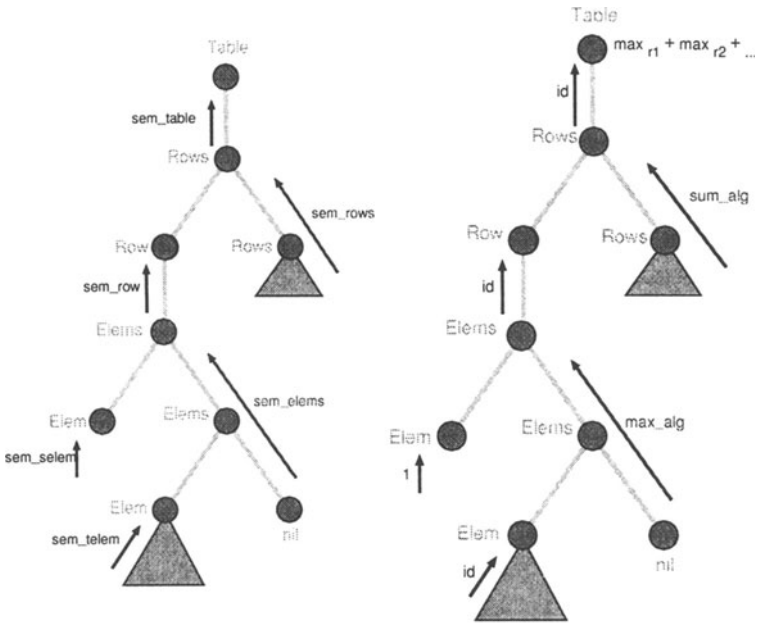

Figure 4 Computations over trees

2.3 Walks, trees: where are they?

In the previous section we have seen how we can use algebras to describe the construction of abstract syntax trees. All we are using these trees for is to compute the meaning of the recognised structure. As when using attribute grammars, we want to express this meaning in a compositional way: the meaning of a structured object is expressed in terms of its substructures. Expressed in a more mathematical style: we have to define a homomorphism from the initial algebra (the abstract syntax trees) to some other algebra (the meaning). Such homomorphisms have become known as catamorphisms (Meijer et al. 1991). 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 programmers and attribute grammar systems.

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.

3 COMPUTING THE HEIGHT

As a first step let us define the functions for computing the height of a table. Figure 4(b) depicts an attribute grammar view of the solution. We have one synthesised attribute `height`. The height of an element is the height of a simple element, `1`, or the height of a nested table. The height of a row is the maximum of the heights of the elements of the row, and the height of a table is the sum of all the heights of the rows. This computational structure is actually what `p_list` is capturing: roll over the elements of the list, taking every element into account, accumulating a result. Thus the list algebra `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 `sem_rows` is `sum_alg`. The complete set of functions is

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

`sem_table` and `sem_row` do not need special transformations, 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 synthesised attributes and, (b) attribute computations are nicely described by algebras.

4 COMPUTING THE WIDTHS

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

```
star :: Alg_List a  b -> Alg_List [a] [b]
```

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

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

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

Figure 5 Computing heights and widths

```
star (e, op) = (repeat e, zipWith op)
```

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

```
infixr <.>          -- infix binary operator, right associative
(<.>) :: Alg_List b a -> Alg_List d c -> Alg_List (b,d) (a,c)
(a,f) <.> (c,g) = ((a,c), \(x,y) (xs,ys) -> (f x xs, g y ys))
```

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

Finally, the result of the computation for a table must be a pair, but we obtain a list of widths from the application of `p_list`. Thus we need a further transformation `id `x` sum`. The **product** combinator `x` only applies its argument functions to the corresponding left and right elements of the pair. The new version of the program is shown in figure 5.

Let us note that: (a) we can have several synthesized attributes by grouping them in tuples, (b) computations for such tuples can be constructed out of computations for the elements (`<.>`, `star`, `split` and `x`), and (c) the operators

on algebras: composition and star, and `split` and product are independent of the problem at hand and could have been taken from a library (at link or run time for example). It is even possible to define an algebra composition combinator for table algebras, but we do not do it here.

5 FORMATTING

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 6. The table layout is constructed by placing these boxes besides and on top of each other.

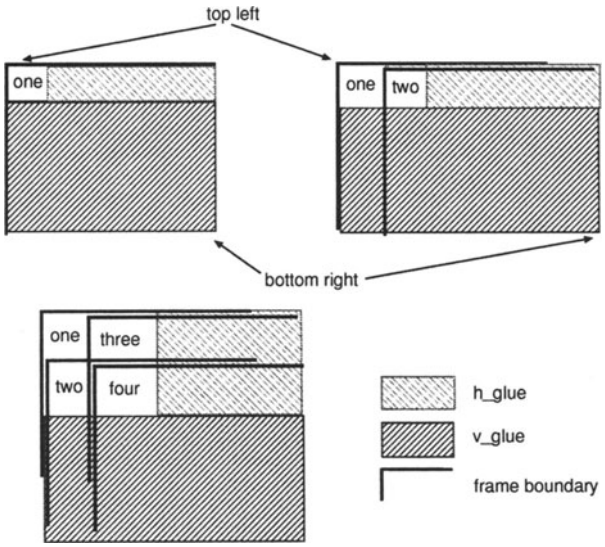

Figure 6 Superposition of boxes

We follow the same line of reasoning as before for synthesising the 'formatted' table. Actually the table will be represented by a (partially parameterised) function which creates the formatted table when its final height and width (parameters) are passed to it. Another way of looking at it is that all nodes in the parse tree accept two further inherited attributes indicating these values, and which are used to compute one further synthesised attribute: the formatted table. The code for the semantic functions and the algebras is shown in figure 7.

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

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

apply2fst = lift ($) fst

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

Figure 7 Computing the formatted table

equipped with a nice top left corner frame – delineating the quarter plane – as you can see in figure 6.

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

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

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` with `apply2fst thid`. Then the grid is closed, with `bot_right` placing the bottom and right lines, and correcting of the actual size of the table. The implementation of box manipulation functions is given in figure 8.

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 has two inherited attributes, the height and a list of widths, and one synthesised attribute, the layout of the element. The inherited attributes are passed down to its chil-

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

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

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

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 '-'
```

Figure 8 Functions for manipulating boxes

dren, 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 synthesised attributes are combined together using the `zipWith (++)` (but in general any `f`). Thus we have some patterns of attribute manipulation: pass down a global value (`fork`), pass down and split a composed value (`decons` if the value is a list and we want to decompose a list into its head and tail), combine inherited attributes (`<||>`) and combine synthesized attributes (`<=>>`), see figure 9(b).

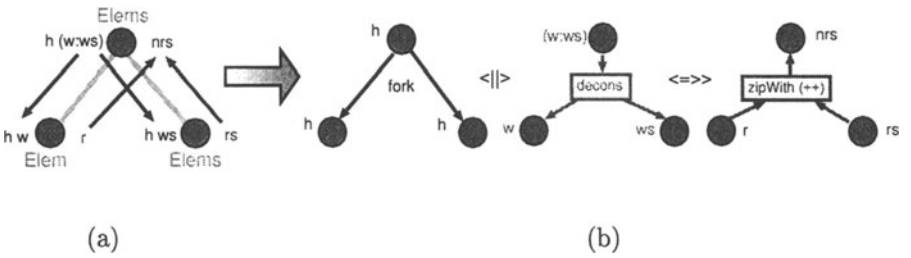

Figure 9 Attribute computation: (a) example (b) combining patterns

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:

```
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
```

Note that there are no inherited attributes as such. We create partially parameterised 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 synthesised (like the height of a row), but once their value have been computed they can be used in a subsequent computation; thus acting as inherited attributes.

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

Furthermore, the program can be generalised rather straightforward to a polytypic function (Jeuring et al. 1996) because the constructors are general. Although not presented here, the algebra composition operator `<.>` can be defined for any arbitrary data type constructor `f`.

6 CONCLUSIONS

The techniques shown in this paper are being presented to students in courses of compiler construction and formal languages. One of the advantages is to present at the same time concepts and their implementations. Thus, the students can immediately experience the ideas in their practical work.

The soundness of the underlying theory enables the compositional approach, and the use of a pure functional language enables straightforward implementation (exemplified in the paper with the algebra composition combinator). In the course of the paper we have shown some advantages of this compositionality. By combining segments of programs in a semantic setting, we are

closer to a compositional implementation of some of the so called design patterns (Gamma et al. 1995). In fact, by implementing catamorphisms (which are almost describable in Haskell) we have almost achieved this goal.

The abstraction mechanisms of the implementation language (in this case Haskell) permit us to be abstract, and extend the language with abstract constructions that immediately become a new additional vocabulary (like the `p_list` combinator or the algebra composition).

These are properties we would expect to be supported in modern system implementation languages.

Acknowledgments

The authors would like to thank Lambert Meertens, Eelco Dijkstra, Roland Backhouse and Daan Heijen for their comments and corrections.

REFERENCES

Azero P. and Swierstra S.D. (1997) Design and implementation of a pretty printing library. (In preparation)

Fokker J. (1995) Functional Parsers, in *Advanced Functional Programming* (eds. J. Jeuring and E. Meijer), LNCS 925. Springer, Berlin.

Gamma E., Helm R., Johnson R. and Vlissides J. (1995) *Design Patterns*, Addison-Wesley, Reading, Mass.

Hammond K. and Peterson J., eds (1997) Haskell 1.4 Report. Available at: `http://haskell.org`

Hudak P. (1996) Haskore Music Tutorial, in *Advanced Functional Programming: Second International School* (eds. J. Launchbury, E. Meijer and T. Sheard), LNCS 1129. Springer, Berlin.

Hughes J. (1995) The design of a pretty-printing library, in *Advanced Functional Programming* (eds. J. Jeuring and E. Meijer), LNCS 925. Springer, Berlin.

Meijer E., Fokkinga M. and Paterson R. (1991) Functional Programming with Bananas, Lenses, Envelopes and Barbed Wire, in *Proceedings of the 5th ACM Conference on Functional Programming Languages and Computer Architecture* (ed. J. Hughes).

Jeuring J. and Jansson P. (1996) Polytypic Programming, in *Advanced Functional Programming: Second International School* (eds. J. Launchbury, E. Meijer and T. Sheard), LNCS 1129. Springer, Berlin.

Swierstra S.D. and de Moor, O. (1993) Virtual Data Structures, in *Formal Program Development* (eds. B. Möller and H. Partsch and S. Schuman), LNCS 755. Springer, Berlin.

Swierstra S.D. and Duponcheel L. (1996) Deterministic, Error Correcting Combinator Parsers, in *Advanced Functional Programming: Second International School* (eds. J. Launchbury, E. Meijer and T. Sheard),

LNCS 1129. Springer, Berlin.
Wadler P. (1990) Deforestation: Transforming Programs to Eliminate Trees. *Theoretical Computer Science*, **73**, 231-48.

7 BIOGRAPHY

S. Doaitse Swierstra holds a master degree in theoretical physics (1976) from the RijksUniversiteit Groningen, and a Ph.D. in Computer Science (1981) from Twente University, both in the Netherlands. Since 1983 he holds a chair in Computer Science at Utrecht University. He is a member of IFIP Working Group 2.1, and chairman of the board of the Dutch Research School IPA. His interests are: programming methodology, compiler construction, functional programming, attribute grammars and formal verification of distributed algorithms.

Pablo R. Azero has a degree in Computer Science from the Universidad Nacional del Sur, Argentina; currently he is a second year PhD student at the Computer Science Department at the Utrecht University. His current research interests are the design and implementation of programming languages, with emphasis on lazy functional languages and software tools.