# On the Incremental Evaluation of Higher-Order Attribute Grammars

**Over de Incrementele Evaluatie van Hogere-Orde Attributengrammatica's**

(met een samenvatting in het Nederlands)

PROEFSCHRIFT

ter verkrijging van de graad van doctor aan de Universiteit Utrecht op gezag van de rector magnificus, prof.dr. G.J. van der Zwaan, ingevolge het besluit van het college voor promoties in het openbaar te verdedigen op dinsdag 30 juni 2015 des middags te 4.15 uur

door

**Jeroen Bransen**

geboren op 14 september 1987 te Utrecht

| | |
|---|---|
| Promotor: | Prof.dr. S. D. Swierstra |
| Copromotor: | Dr. A. Dijkstra |

The work in the thesis has been carried out under the auspices of the research school IPA (Institute for Programming research and Algorithmics) and is part of the research programme Non-Invasive Incremental Evaluation, which is (partly) financed by the Netherlands Organisation for Scientific Research (NWO).

# Preface

The cover of this thesis shows an ant colony that is building a bridge of ants standing on top of each other. The cover image is not only a nice photograph, but also symbolises several of my interests, such as my love for walking through the forest. There are three different elements in the image that symbolise different aspects of the basis of this thesis.

My background in *artificial intelligence* is symbolised by the ant colony itself. In the field of artificial intelligence ant colonies are studied for their interesting behaviour: every single ant uses only very simple rules for deciding in which way to move, yet the combination of many of those ants together leads to complex patterns. For example, the shortest path to a source of food is found and utilised by all ants even though no single ant is capable of finding such shortest path by itself.

The leaves of the tree on which the ants walk symbolise *attribute grammars*. Attribute grammar computations, on which the work in this thesis is based, are computations over tree-shaped structures which are extended with attributes. Such attributes are often visualised as values that are attached to the tree, just like the ants that walk on the tree. Furthermore, ants gather food from the tree in the same way that attribute grammars gather information from the tree structure.

Finally the bridge made out of ants standing on top of each other symbolises the *incremental* construction of results. Whenever a new ant arrives it builds upon the result achieved by the ants that arrived earlier; the bridge is extended with the newly arrived ant to form a larger bridge. Such an incremental way of constructing a bridge is in a way similar to the incremental evaluation of attribute grammars as described in this thesis: instead of starting all over again when more ants are there, the ants simply reuse the previous result of building a bridge.

In addition to the explanation of the cover image, the preface of a thesis is the natural place to thank people. Although I am thankful to many people, I have chosen not to explicitly mention all of them. For Doaitse, my promotor, and Atze, my copromotor, I make an exception.

Before Doaitse retired he was actively involved in my research, often showing up in my office to explain a new idea he came up with that morning while cycling to university. I also enjoyed the many lunches where Doaitse explained things about functional programming, finance, politics and many other subjects to me and the other group members. After he retired the daily contact got lost, but via Skype and face to face meetings now and then he kept supervising me and has provided a lot of valuable feedback on all aspects of my research.

Atze has taught me many aspects of the life as a scientist. To contrast Doaitse, who had many suggestions for research, Atze helped me to focus on the important bits that would lead to concrete results. The weekly meetings with Atze were a good way of getting me on the right track, either by pushing me to work a bit harder on certain subjects, or by slowing me down to avoid spending too much time on wild ideas. Without Atze his supervision this thesis would have probably not been finished by now.

To conclude: Doaitse and Atze, thank you very much for the supervision. I have enjoyed the past four years a lot and you have played an important role in them. To all others who have somehow contributed to this thesis, colleagues, members of the reading committee, family members, friends, students, etc.: thank you very much, I hope that you have enjoyed my enthusiasm and the interaction with me as much as I have enjoyed it. Have fun reading!

# Contents

# 1 Introduction

Computers perform many computations over input data that changes over time. For example, a text editor keeps the table of contents up-to-date while a document is being edited, an e-mail client updates its interface when new e-mails arrive and an IDE shows compilation errors during program construction. For such programs to be useful in practice we expect them to respond promptly to such changes to their input.

For the programmer of such applications this poses a problem, as the computations take more time when the input data grows. The solution is to construct programs that update their results incrementally, such that small changes to the input lead to a short computation time. To obtain such short computation times the program may try to reuse results computed from the previous input.

In general constructing an incremental version of an algorithm is complex and error-prone, and much more complicated than writing a non-incremental version. For instance, in the case of a compiler, constructing a correct and efficient compiler is by itself already a hard task and adapting such compiler to behave incrementally complicates matters even more. This raises the question whether we can (semi)automatically get an incremental version of an otherwise non-incremental program.

In this thesis we therefore take the approach of automatically generating incremental evaluators from a declarative definition. With this approach the programmer can think about the program in terms of a non-incremental version, and get the incremental version "for free".

## 1.1 General overview

The work in this thesis is based on *attribute grammars* [Knuth, 1968] (AGs), a formalism for specifying computations over tree structures in a declarative way. From the perspective of the programmer the usage of attribute grammars for software development can be limiting; attribute grammars provide only a restricted form of constructing programs. From the perspective of the automatic generation of incremental evaluators, however, this restricted form is exactly where we find a handle to attack our problem. We believe that attribute grammars are the right level of abstraction: they are restricted enough for the automatic generation of incremental evaluators, while there are still many programs that can be constructed in an attractive way by expressing them as attribute grammars.

A particular type of such programs are compilers. The input of a compiler is the source code in some language, represented by an abstract syntax tree. The abstract syntax tree is then analysed and transformed into a resulting program expressed in some backend language, probably again represented by a tree structure. Because attribute grammars provide an attractive way of specifying computations on trees they are useful in compiler construction.

The *Utrecht Haskell Compiler* (UHC) [Dijkstra et al., 2009], which is built using attribute grammars, is the main motivation for this work. We want the UHC to update its compilation results incrementally: when some file is compiled, changed and compiled again, we want the second compilation run to take only a short time when the changes are small. Of course this is not always achievable since a small change may have a large effect on the complete compilation result, but in many cases efficiency can be gained by storing intermediate results from the first compilation. The work in this thesis forms the first step in the automatic construction of an incremental version of the UHC.

## 1.2 Code

The functional programming language Haskell [Peyton Jones, 2003] is used for implementing the techniques described in this thesis, and we assume the reader to be familiar with that programming language. As the UHC is a research project rather than an industrial strength compiler, we use the Glasgow Haskell Compiler[1] (GHC) version 7.8.3 with several of its extensions to the Haskell language for compilation. The techniques for incremental attribute grammar evaluation are however not specific to Haskell and could as well be implemented in another (functional) language. For the incremental evaluation of attribute grammars without higher-order attributes, the lazy semantics of Haskell can have a positive effect on the effectiveness of the incremental evaluation, but laziness is not essential. For the support of higher-order attributes we do however make essential use of lazy evaluation.

It is a standard practice to format the code nicely instead of printing it verbatim and in this thesis we follow that convention. For example, `Set.empty` is formatted as $\emptyset$ and `++` is formatted as $+\!\!+$. In some cases multiple different functions may be typeset as the same symbol, for instance `` `elem` ``, `` `Set.member` `` and `` `Map.member` `` are all typeset as $\in$. From the context it should always be clear which function is used.

We have implemented the techniques described in this thesis as a proof of concept for which the source code can be found online[2]. All relevant implementation details are given in this thesis; the repository can be regarded as an extra reference in case implementation details are unclear.

## 1.3 Guestbook example

To illustrate the techniques of incremental evaluation and the difficulties in the manual construction of such incremental evaluators, let us take a look at the following short story. The example introduced in the story is also used in later chapters of this thesis as a running example.

### 1.3.1 Introduction

Imagine a hotel with a guestbook containing two types of entries: the arrival of a guest and the departure of a guest. Furthermore, upon departure guests can also write a small review of the hotel and assign a grade to their stay. Figure 1.1 shows a few example entries of such a guestbook. Note that the guestbook is ordered from new to old; the most recent entry is at the beginning of the list and the oldest entry at the end.

Now the hotel owner is interested in knowing the average grade that clients give to his hotel and therefore he writes a Haskell program to compute this grade

[1] `https://www.haskell.org/ghc/`
[2] `https://github.com/jbransen/uuoagc`

```
LEAVE Bransen - 7.6 - I liked the fast internet connection
LEAVE Dijkstra - 8 - The atmosphere is great for taking pictures!
ARRIVE Dijkstra
LEAVE Swierstra - 6 - Nice hotel, but the beds are too short
ARRIVE Bransen
ARRIVE Swierstra
```

Figure 1.1: Example guestbook entries

```
type Name = String
data Entry = Arrive Name
           | Leave  Name Double String
type Guestbook = [Entry]
average :: [Double] → Double
average nums = sum nums / genericLength nums
avgGrade₁ :: Guestbook → Double
avgGrade₁ gb = average [g | Leave _ g _ ← gb]
```

Figure 1.2: Haskell code for representing the guestbook and computing the average grade

(Figure 1.2). The guestbook consists of a list of entries with the latest entry at the beginning of the list. As expected, the code returns 7.2 as the average grade for the example guestbook, which makes the hotel owner quite happy.

### 1.3.2 Complicating matters

Unfortunately, soon after constructing this wonderful piece of code, the hotel owner realises that there are two problems with this approach: honesty and efficiency. Or more precisely: dishonesty and inefficiency.

**Honesty** First of all not all guests are honest and it happens that guests enter fake reviews under another name. For obvious reasons the hotel owner wants to exclude those reviews from the grade computation, since they do not give a fair indication of the quality of the hotel. To solve this problem he decides to only include reviews

```
avgGrade₂ :: Guestbook → Double
avgGrade₂ = average ∘ trueReviews

trueReviews :: Guestbook → [Double]
trueReviews [ ]                = [ ]
trueReviews (Arrive _    : xs) = trueReviews xs
trueReviews (Leave n g _ : xs) = if   n ∈ signedIn xs
                                 then g : trueReviews xs
                                 else     trueReviews xs

signedIn :: Guestbook → Set Name
signedIn [ ]               = ∅
signedIn (Arrive n   : xs) = n ‘Set.insert‘ signedIn xs
signedIn (Leave n _ _ : xs) = n ‘Set.delete‘ signedIn xs
```

Figure 1.3: Computing the average grade for all honest guests

from guests that have actually arrived, and to accept only the first review of a guest after arrival in case of multiple reviews from the same guest. His first coding attempt is given in Figure 1.3.

The *signedIn* function computes for a given guestbook which guests are currently signed in. Keep in mind that the latest entry is at the beginning of the list. In order to retrieve all reviews from honest guests (*trueReviews*), we check for each guests which leaves whether this guest was indeed checked in at that time. Note that we do not disallow multiple arrival entries from the same guest.

**Efficiency problem 1 (lack of sharing)** Although this code gives the required answer, there is a problem: it has become less efficient! To be precise, the running time changed from $O(n)$ to $O(n^2)$ where $n$ is the number of entries in the guestbook[3]. The reason for this is that for each *Leave* entry in the guestbook, the functions *signedIn* and *trueReviews* are both called on the tail of the guestbook and thus the complete trailing list is traversed. Another way of viewing this is that in *trueReviews* there are calls to two recursive functions instead of just one, and because the recursive call to *trueReviews* can have two recursive calls again, the complexity has increased from linear to quadratic.

[3]We assume constant time set operations here, which in practice is not the case. However, the logarithmic factors introduced by the set operations do not influence the main message of this story so we ignore them in this example

```
avgGrade₃ :: Guestbook → Double
avgGrade₃ = average ∘ snd ∘ tupled where
  tupled :: Guestbook → (Set Name, [Double])
  tupled []              = (∅, [])
  tupled (Arrive n   : xs) = let (signedIn, trueReviews) = tupled xs
                             in  (n ‘Set.insert‘ signedIn
                                 , trueReviews)
  tupled (Leave n g _ : xs) = let (signedIn, trueReviews) = tupled xs
                             in  (n ‘Set.delete‘ signedIn
                                 , if    n ∈ signedIn
                                   then g : trueReviews
                                   else     trueReviews)
```

Figure 1.4: Tupled version of computing the average grade for all honest guests

This problem can be solved by *tupling* of the functions *trueReviews* and *signedIn*. The idea of the tupling technique is that we compute the results of both functions in one go, as result of which we only have one recursive call. In Figure 1.4 we show the resulting code for which the running time is in $O(n)$ again.

**Efficiency problem 2 (big data)** A larger problem is that execution takes more and more time as the guestbook grows, even though each call takes time linear in the length of the list argument. The hotel owner wants to recompute the average grade whenever a new review is added, but as time passes there are so many guestbook entries (the hotel has become really popular) that the code takes minutes to run, for each new single entry that is added!

### 1.3.3 Incremental computation

In order to solve such problems the hotel owner wants to revert to *incremental computation*. When a function works on data that changes over time, as is the case in our example, it is often possible to create a version of the function that caches some intermediate values as a result of which the function can efficiently be (re)computed whenever only a small part of its argument has changed.

In Figure 1.5 we present an incremental version of the guestbook code. The intermediate state contains both the set of signed in members at a given point in time

```
type State₄ = (Set Name, [Double])
avgGradeI₄ :: State₄ → Double
avgGradeI₄ = average ∘ snd
avgGradeU₄ :: Entry → State₄ → State₄
avgGradeU₄ (Arrive n) (signedIn, trueReviews)
  = (n ‘Set.insert‘ signedIn, trueReviews)
avgGradeU₄ (Leave n g _) (signedIn, trueReviews)
  = (n ‘Set.delete‘ signedIn
    , if   n ∈ signedIn
      then g : trueReviews
      else     trueReviews)
avgGrade₄ :: Guestbook → Double
avgGrade₄ = avgGradeI₄ ∘ foldr avgGradeU₄ (∅, [ ])
```

Figure 1.5: Incremental version of the tupled guestbook code

together with the list of true reviews found so far. The function $avgGradeU_4$ is used for updating: given a new guestbook entry it updates the state. With $avgGradeI_4$ the current average can be extracted from the state. For convenience we have also added an $avgGrade_4$ function with the same type as before which simply adds all entries from the guestbook one by one and returns the final result.

This transformation is a step closer to the optimal solution, but one important problem remains: the addition of a single entry still takes $O(n)$ time! The remaining problem is that even though a list of grades from honest guests is constructed efficiently, the computation of the average grade still requires a traversal of the whole list.

Figure 1.6 shows the final version of the code which takes constant time for each additional entry. To achieve this result the hotel owner applied the same tupling and incrementalisation trick to the computation of the average by caching the length and sum of the list. The state for computing the average contains the length of the list and the sum and can be updated with the $averageU_5$ function. In the $avgGradeU_5$ the *AvgState* is incrementally updated with a call to $averageU_5$, such that adding a single entry takes only constant time.

The hotel owner is happy that this solution works well and proud that he managed to come up with this solution. There are however a few drawbacks of this approach.

```
type StateAvg = (Double, Double)

averageI_5 :: StateAvg → Double
averageI_5 (sum, len) = sum / len

averageU_5 :: Double → StateAvg → StateAvg
averageU_5 x (sum, len) = (sum + x, len + 1)

type State_5 = (Set Name, StateAvg)

avgGradeI_5 :: State_5 → Double
avgGradeI_5 = averageI_5 ∘ snd

avgGradeU_5 :: Entry → State_5 → State_5
avgGradeU_5 (Arrive n) (signedIn, avgState)
  = (n ‘Set.insert‘ signedIn, avgState)
avgGradeU_5 (Leave n g _) (signedIn, avgState)
  = (n ‘Set.delete‘ signedIn
    , if   n ∈ signedIn
      then averageU_5 g avgState
      else            avgState)

avgGrade_5 :: Guestbook → Double
avgGrade_5 = avgGradeI_5 ∘ foldr avgGradeU_5 (∅, (0, 0))
```

Figure 1.6: Efficient incremental version of the tupled guestbook code

- The code has become less readable and not quite as nice as the code from Figure 1.3, which computes exactly the same result but less efficiently. The two functions have been merged and therefore the design criterion of 'separation of concerns' has been broken.

- The code only supports a very limited set of incremental changes: the efficient update only works when a new element is added to the beginning of the list. In practice it can however also happen that an earlier entry needs to be changed. For example, when a name was misspelled this code needs to process the whole guestbook again, even though theoretically only the last few entries need to be processed.

- The functions that we have defined only have a single argument which is the guestbook: in practice functions often have other parameters containing contextual information, which highly complicates the incremental computation.

### 1.3.4 The big picture

In the previous sections we have illustrated with a small example (1) why incremental computation is useful, and (2) that manually writing functions that behave incrementally is a tedious job. It should not come as a surprise that this effect is even larger for real-world applications, with many functions that need to be interleaved and merged in order to get the desired incremental behaviour. Also, when going from list-like structures like the guestbook to tree-like structures like the source code of a programming language, the types of changes are more complex.

In this thesis we solve this problem as follows: we ask the end-users, such as the hotel owner, to write their code as an attribute grammar, and let the attribute grammar compiler automatically generate code that efficiently handles changes to the input data. In the rest of this thesis this approach is explained in more detail, but the essential insight is that attribute grammars have the right expressivity: their expressiveness is restricted enough to allow the attribute grammar compiler to generate efficient incremental code, but at the same time expressive enough to allow many programs to be written as an attribute grammar in an attractive way.

Concretely, in this thesis we describe techniques that support the following workflow.

- The user writes his code as an attribute grammar, which is a formalism that is suitable for writing compilers. The attribute grammar definitions are declarative in nature and enable aspect oriented programming. For our example the attribute grammar definition is similar to the code given in Figure 1.3.

- The compiler generates code that can be used to efficiently compute the attributes (similar to function results) for a given abstract syntax tree efficiently, with automatic tupling as in Figure 1.4. This automatic tupling is a direct result of using attribute grammars and is not related to the use of the incremental evaluation techniques described in this thesis.

- The attribute grammar compiler also generates functions for efficiently updating the computation after a change to the input. These functions take a description of the change to the abstract syntax tree and a state resulting from the previous evaluation of the attributes. In a way this resembles the code in Figure 1.6 but it is more general. For example, the deletion of the first entry only takes constant time, and the insertion of an entry somewhere later in the list only takes time linear in the length of the list up to the position at which the entry is inserted.

In this thesis we describe how the above can be achieved using so-called higher-order attribute grammars. A higher-order attribute is an attribute value which itself is a tree structure over which attributes can be computed. In our example the list of grades from honest guests is such a value; we first build up this list in the *trueReviews* function and then compute the average over this list in the *average* function.

In compiler construction higher-order attributes can also be used to model different stages in a pipeline architecture like a compiler. Each phase in the compiler pipeline takes as input an abstract syntax tree, and returns a new (internal) representation of the program which is also an abstract syntax tree. For example, most Haskell compilers contain a separate pass in which type inferencing takes place, taking as input an untyped program and returning the program with explicit types for each subtree. Higher-order attribute are used to take the result of one phase and instantiate this result as the abstract syntax tree to be used as input for the next phase.

## 1.4 Definitions

An *evaluator* is a function that computes the result for the computation represented by its argument. An attribute grammar is a declarative specification of a computation and is therefore not directly executable. Instead, an attribute grammar compiler generates from this declarative specification a runtime evaluator that computes the result for a given input.

We call a function $f^I$ *incremental* when its result is efficiently computed given the result of the previous call and a change to the previous input. Formally, given a function $f$, an input $x$ and a changed input $x \oplus \delta x$ we say that $f^I$ is the incremental version of $f$ when $f\ (x \oplus \delta x) = f^I\ (f\ (x), \delta x)$ and $f^I$ computes this result more efficiently. In our setting $f^I$ does not necessarily depend on the result $f\ (x)$ and the change $\delta x$ only, but may use any additional data that was computed in earlier calls to $f$ or $f^I$, such as the internal state of the example.

## 1.5 C$^\sharp$ example

A second and more real-world example used in this thesis is a small C$^\sharp$ compiler[4] that supports a subset of C$^\sharp$, expressed as an attribute grammar. Supported programs consist of a single class containing only static functions, the primitive types **int**, **char** and **bool**, and the standard control structures **if**, **for** and **while**. No library functions are implemented except from printing a value to the screen (but only for **int**, **char** or **bool** types, because *String*s are not supported). In Figure 1.7 we show an example C$^\sharp$ program that can be compiled with our compiler.

The input of the compiler is a program written in this subset of C$^\sharp$, and the output of the compiler is code for the *Simple Stack Machine* (SSM)[5]. The SSM is a stack machine emulator that allows easy debugging with stepping, breakpoints and memory overview. The SSM language is still readable by humans but the operations are close to real world machine models.

The result of compiling the *fac* function from Figure 1.7 to SSM is shown in Figure 1.8 with annotations to explain the code. The precise meaning of all commands is not important here, but the figure illustrates that the SSM language is low level. Note that names are used for function calls, but that in other places like `BRA` (Branch Always) and `BRT` (Branch if True) relative addresses are used.

### 1.5.1 Architecture

Discussing the full implementation of our C$^\sharp$ compiler is out of the scope of this thesis, and we only show its general architecture here. In Chapter 2 we show some more implementation details when introducing attribute grammars. The full implementation be found in the code repository.

The compiler consists of two phases: in the first phase the declared variables for each function are collected; in the second phase these are used for constructing the resulting SSM code. In the second phase an environment is constructed containing the names of the parameters and local variables and their relative addresses on the

[4]This compiler is the result of an exercise for the Languages and Compilers course

[5]`http://www.staff.science.uu.nl/~dijks106/SSM/`

```
class Hello
{
   void Main ()
   {
      print (fac (square (2)));
   }
   int square (int x)
   {
      return x ∗ x;
   }
   int fac (int x)
   {
      int r = 1;
      for (int t = 1; t ⩽ x; t = t + 1)
         r = r ∗ t;
      return r;
   }
}
```

Figure 1.7: Example C$^\sharp$ program that is supported by our compiler

stack, which are used in the rest of the code generation whenever variables are used.

In the SSM execution, at the beginning of each function execution the values for the parameters are already on the stack, and for the local variables extra space is reserved on the stack using the `LINK` instruction. The `LINK` instruction lets the *mark pointer*, a special register, point to the current top of the stack, such that the location of the parameters and local variables on the stack can be computed relative to the mark pointer. Before returning from a function call the `UNLINK` instruction is used to free the space for local variables from the stack and restore the mark pointer to its previous value, which was also stored on the stack by `LINK`.

With this setup the rest of the code generation is fairly straightforward. For a block of statements the SSM instructions are simply concatenated, for function calls special instructions take care of storing the current execution location, for expressions we have instructions for each standard operator and finally for implementing loops we have a collection of jump instructions available. The jump instructions such as `BRA` take as argument relative addresses which are computed from the size of the generated code.

## 1.6 Thesis overview

The work in this thesis is based on several papers that have been published elsewhere. These papers roughly correspond to the following sections.

- [Bransen et al., 2012]: Section 3.3
- [Bransen et al., 2015b]: Section 3.4
- [Bransen and Magalhães, 2013]: Chapter 4
- [Bransen et al., 2014b]: Chapter 5
- [Bransen et al., 2015a]: Chapter 6
- [Bransen et al., 2014a]: Chapter 7

I hereby would like to thank the co-authors of these papers, who all contributed to the final result of this thesis. In particular I like to mention José Pedro Magalhães who wrote most of the text of the [Bransen and Magalhães, 2013] paper on which Chapter 4 is almost entirely based, Thomas van Binsbergen and Koen Claessen who have both contributed to Section 3.4, and Arie Middelkoop who has contributed to Section 3.3.

```
fac:
        LINK 2    ;; Create stack space for variables r and t
        LDC 1     ;; Load 1 on stack
        LDS 0     ;; Duplicate top of stack
        LDLA 1    ;; Load address of r
        STA 0     ;; Store 1 in r
        AJS -1    ;; Pop from stack
        LDC 1     ;; Load 1 on stack
        LDS 0     ;; Duplicate top of stack
        LDLA 2    ;; Load address of t
        STA 0     ;; Store 1 in t
        AJS -1    ;; Pop from stack
        BRA 26    ;; Jump 26 bytes (to jump target 1)
        LDL 1     ;; Load value of r (jump target 2)
        LDL 2     ;; Load value of t
        MUL       ;; Multiply r * t
        LDS 0     ;; Duplicate top of stack
        LDLA 1    ;; Load address of r
        STA 0     ;; Store r * t in r
        AJS -1    ;; Pop from stack
        LDL 2     ;; Load value of t
        LDC 1     ;; Load 1 on stack
        ADD       ;; Add t + 1
        LDS 0     ;; Duplicate top of stack
        LDLA 2    ;; Load address of t
        STA 0     ;; Store t + 1 in t
        AJS -1    ;; Pop from stack
        LDL 2     ;; Load value of t (jump target 1)
        LDL -2    ;; Load value of x
        LE        ;; Compare t <= x
        BRT -33   ;; Jump -33 bytes if true (to jump target 2)
        LDL 1     ;; Load value of r
        STR R3    ;; Store r in R3
        UNLINK    ;; Clean up stack space of r and t
        RET       ;; Return to caller
```

Figure 1.8: Example SSM code for the *fac* function with manual annotations

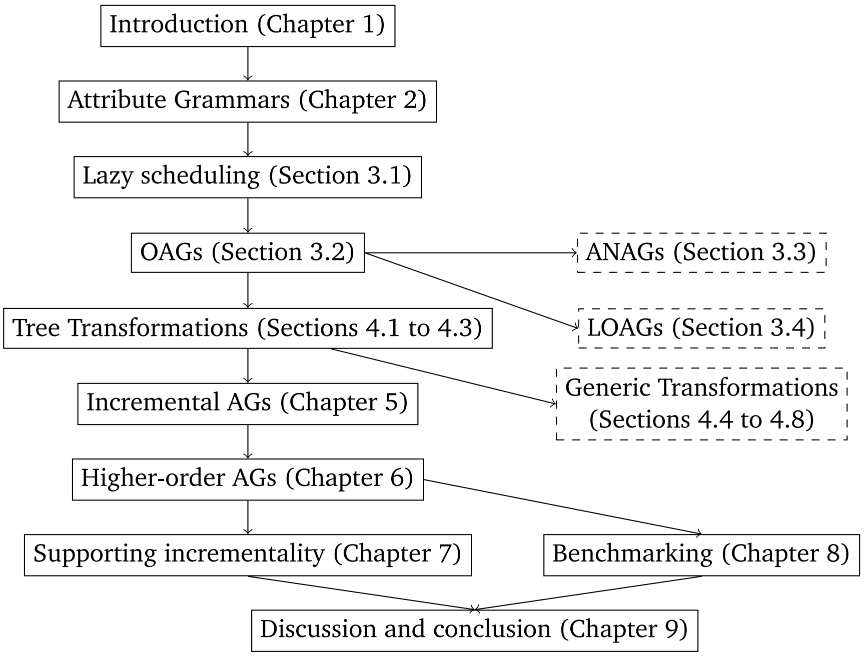
Introduction (Chapter 1)
Attribute Grammars (Chapter 2)
Lazy scheduling (Section 3.1)
OAGs (Section 3.2)
ANAGs (Section 3.3)
LOAGs (Section 3.4)
Tree Transformations (Sections 4.1 to 4.3)
Generic Transformations (Sections 4.4 to 4.8)
Incremental AGs (Chapter 5)
Higher-order AGs (Chapter 6)
Supporting incrementality (Chapter 7)
Benchmarking (Chapter 8)
Discussion and conclusion (Chapter 9)

Figure 1.9: Graph visualising the dependencies between the chapters in this thesis

### 1.6.1 Dependency graph

In Figure 1.9 we show an overview of the chapters in this thesis and their dependencies. Although all sections in this thesis of course contain useful information, for understanding it is not necessary to read all of them in the order as they appear in this thesis. In particular the sections in dashed boxes can be skipped by readers who are not interested in the details dealt with there.

The thesis starts with this introduction, followed by an introduction to attribute grammars in Chapter 2. In Chapter 3 different scheduling algorithms are explained, of which the OAG algorithm is the most important one for the main story line. The representation of changes to abstract syntax trees is discussed in Chapter 4; the first three sections of that chapter illustrate the representation using the two running examples of this thesis and are preliminary information for the later chapters, while the rest of the scheduling chapter is about representing these transformations generically using generic programming techniques. In Chapter 5 and Chapter 6 the core technique of this thesis is explained. Chapter 7 we describe in what way attribute grammars need to be constructed to support the effectiveness of our technique, and in Chapter 8 we show some problems related to benchmarking together with the benchmarking results for the two running examples. Finally, in Chapter 9 we wrap up with a discussion of related work and conclusion.

# 2
# Attribute Grammars

In this chapter we introduce attribute grammars (AGs) as they are used within the rest of this thesis with our two running examples. We define the essential AG terminology and provide the reader with some intuition about AGs.

AGs describe the *decoration* of trees with attributes with the purpose of performing computations over a tree. Such attributes are attached to nodes and their values are accessible to either parent or child nodes to be used in the computation of other attributes. For example, in a type checking phase of a compiler one typical attribute holds a table mapping variables that are in scope to types and another attribute contains the resulting type of the expression. The first would pass information from a node to its children (inherited) while the second would pass information from a node to its parent (synthesized).

According to the original definition of [Knuth, 1968], an attribute grammar consists of three parts: a context-free grammar describing the collection of trees we are about to decorate, for each node type a collection of attributes to be associated with it and a collection of semantic rules describing how to compute the value of attributes in terms of other locally accessible attributes. Over the years many variations on this simple formalism have been proposed, such as higher-order attributes [Vogt et al., 1989], reference attributes [Hedin, 2000], door attribute grammars [Hedin, 1994], and many classes of attribute grammars have been identified, usually based on properties enabling some specific evaluation strategies.

The work in this thesis has been performed in the context of the *Utrecht University Attribute Grammar Compiler* (UUAGC) [Swierstra et al., 1998], which uses a slightly different representation. Instead of a context-free grammar describing the concrete syntax of the language, an abstract syntax tree forms the starting point of evaluation. The following sections introduce the different components of AGs step by step. We introduce both the underlying concepts and the UUAGC syntax which is also used in the rest of this thesis.

## 2.1 Abstract Syntax Tree

The Abstract Syntax Tree (AST from now on) forms the basis of the AG computation and is defined as a family of algebraic data types describing the tree structure for which attributes are to be computed. These algebraic data types are similar to Haskell data types, consisting of a set of constructors each having a sequence of types. Because AGs were once introduced as an extension to context-free grammars, we still use the terms *nonterminal* and *production*. In the context of the AST a nonterminal refers to a data type, whereas a production refers to a constructor of this data type. Furthermore, other Haskell types are referred to as *terminals*.

The children of each production are named such that we can refer to these children in the semantic rules later on. Each child has its own type which can be either a nonterminal or a terminal; terminal children can have any Haskell type and are not used to define attributes over as their values themselves can be used in other attribute value computations.

### 2.1.1 Guestbook example

In Figure 2.1 we show the AST definition of the AG implementation of our guestbook example. The AG consists of a single nonterminal named *Guestbook* with three productions: *Empty*, *Arrive* and *Leave*. The *Empty* production represents an empty guestbook (or empty list in the Haskell version) whereas the *Arrive* and *Leave* productions represent the two possible types of guestbook entries. Both have a nonterminal child named *tl* containing the rest of the entries and a terminal child *name* which is the name of the corresponding guest. *Leave* has two more terminal children: *grade* and *review*.

Note that the representation is slightly different from the Haskell version: instead of a data type *Entry* and a type for lists we only use only a single data type.

```
data Guestbook
  | Empty
  | Arrive name   :: Name
           tl     :: Guestbook
  | Leave  name   :: Name
           grade  :: Double
           review :: String
           tl     :: Guestbook
```

Figure 2.1: Attribute grammar representation of the guestbook, which is isomorphic to the original definition

This data type is isomorphic to the Haskell version; we feel that this representation is slightly more intuitive for explaining attribute grammars.

### 2.1.2 C$^\sharp$ example

The AST of our C$^\sharp$ compiler from Chapter 1 is described by the family of data types given in Figure 2.2. The start nonterminal is *Class* which has a single alternative containing a name and a list of members, each being either a variable or a method. Methods consist of a return type, a name, a list of parameters and a body. The body usually consists of a (possibly compound) statement.

**Special list syntax** The UUAGC provides special syntax to deal with lists. With the definition **type** $T = [U]$ a nonterminal $T$ is introduced which is a list of elements of type $U$, which again can be either a nonterminal or a Haskell type. This is completely equivalent to having defined the following data type.

```
data T
  | Cons hd :: U
         tl :: T
  | Nil
```

We can define attributes over the *Cons* and *Nil* constructor as usual. In the underlying representation however the built-in list type of Haskell is used.

```
data Class       | ClassC      name   :: String    members :: MemberL
type MemberL = [Member]
data Member      | MemberD     decl   :: Decl
                 | MemberM     rtype  :: Type      name    :: String
                               params :: DeclL     body    :: Stat
type StatL       = [Stat]
data Stat        | StatDecl    decl   :: Decl
                 | StatExpr    expr   :: Expr
                 | StatIf      cond   :: Expr
                               true   :: Stat      false   :: Stat
                 | StatWhile   cond   :: Expr      body    :: Stat
                 | StatFor     init   :: Decl      cond    :: Expr
                               incr   :: Expr      body    :: Stat
                 | StatReturn  expr   :: Expr
                 | StatBlock   stats  :: StatL
data Const       | ConstInt    val    :: Int
                 | ConstBool   val    :: Bool
                 | ConstChar   val    :: Char
type ExprL       = [Expr]
data Expr        | ExprConst   const  :: Const
                 | ExprVar     name   :: String
                 | ExprOper    op     :: String
                               left   :: Expr      right   :: Expr
                 | ExprFun     name   :: String    params  :: ExprL
type DeclL       = [Decl]
data Decl        | DeclC       vtype  :: Type      name    :: String
data Type        | TypeVoid
                 | TypePrim    ptype  :: String
                 | TypeObj     otype  :: String
                 | TypeArray   itype  :: Type
```

Figure 2.2: The data type definitions of the C$^\sharp$ compiler.

## 2.2 Synthesized attributes

*Synthesized* attributes contain information that is computed in a bottom-up way, from the children to the root. This direction is the same as the result of a recursive function, and synthesized attributes are therefore used for storing the final result of an AG computation. They may however also be used to compute intermediate values which are to be passed back down the tree again, even at internal nodes.

In this section we introduce the syntax for defining and using synthesized attributes. For this we need two components: the *attribute definition* and the *semantic rules*.

The attribute definition introduces an attribute by giving it a name, a type and a location. Since many nonterminals may have similar attributes serving a similar purpose, we can provide a collection of nonterminals for which we want to have an attribute with this name. No confusion can arise, since when we use an attribute name it can always be inferred from the context with which node the attribute is associated. The name must be unique for each nonterminal for which it is defined and is used, in combination with the reference to a node, in the semantic rules to refer to this attribute. Its type can be either a nonterminal or a Haskell type.

The semantic rules describe for each production of the corresponding nonterminals how the values of the attributes are to be computed. Each rule contains an arbitrary well-typed Haskell expression, possibly containing references to other attributes that are in scope, which are synthesized attributes of the children, inherited attributes (explained in the next section) and values of terminal children.

### 2.2.1 Guestbook example

Figure 2.3 shows how the *signedIn* and *trueReviews* function from the example may be expressed using AG syntax. The **attr** keyword introduces an attribute for the nonterminal *Guestbook*. In this case, the attribute *signedIn* has a Haskell type *Set Name*; the curly braces are used to escape to Haskell. In the UUAGC syntax the curly braces are not always strictly necessary but can be used to make a clear distinction between terminals (from the Haskell world) and nonterminals (from the AG world).

The semantic rules are defined with the **sem** keyword, followed by the nonterminal name and for each production a list of rules. Rules defining a synthesized attribute are of the form **lhs**.$a = e$, where $a$ is the attribute name and $e$ the expression. The abbreviation **lhs** stands for the *left-hand side*, referring to the left-hand side of a production rule in a context-free grammar. In other words, **lhs** refers to the parent node. To define the value of the synthesized attribute we define the value for an attribute that is accessible by the parent node.

```
attr Guestbook
  syn signedIn :: {Set Name}
sem Guestbook
  | Empty  lhs.signedIn = ∅
  | Arrive lhs.signedIn = @name ‘Set.insert‘ @tl.signedIn
  | Leave  lhs.signedIn = @name ‘Set.delete‘ @tl.signedIn
attr Guestbook
  syn trueReviews :: {[Double]}
sem Guestbook
  | Empty  lhs.trueReviews = [ ]
  | Arrive lhs.trueReviews = @tl.trueReviews
  | Leave  lhs.trueReviews = if   @name ∈ @tl.signedIn
                             then @grade : @tl.trueReviews
                             else          @tl.trueReviews
```

Figure 2.3: Attribute grammar version of the *signedIn* and *trueReviews* functions

The *@c* and *@c.a* occurrences in the right-hand side of the expression refer to other values in scope: *c* is the name of a (terminal) child and *a* the name of an attribute associated with child *c*. In the former case we refer to the subtree as a (Haskell) value to be used, and in the second form to an attribute with name *a* associated with child *c*.

**Computing the average grade** Another part of the guestbook code computing synthesized information is the function *average*. While we could just call the Haskell *average* function in some semantic rule, we prefer to express it as an AG computation. Lists have a tree structure and can therefore be fitted into an attribute grammar based formalism.

In Figure 2.4 we show how to compute the average of a list of floating point numbers. The *length* and *sum* attributes implement the corresponding Haskell functions, while *average* is the attribute that combines them.

**Separation of concerns** At this point we want to remark that the compositional nature of AGs allows for aspect oriented programming. The three attributes defined in Figure 2.4 can also be separated into different **attr** and **sem** blocks, and may even be defined in separate files. The attribute grammar compiler combines all defini-

```
type DL = [Double]
attr DL
  syn length  :: {Double}
  syn sum     :: {Double}
  syn average :: {Double}
sem DL
  | Nil  lhs.length  = 0
         lhs.sum     = 0
         lhs.average = 0
  | Cons loc.length  = @tl.length + 1
         loc.sum     = @tl.sum + @hd
         lhs.average = @loc.sum / @loc.length
```

Figure 2.4: Attribute grammar version of computing the average of a list of numbers

tions into a single evaluator, which allows the programmer to separate concerns as he likes.

**Local attributes** In the rules associated with the *Cons* constructor the **loc** keyword appears. It refers to *local attributes*, which are attribute values that can only be used in the production for which they are defined. In this case the values of **loc**.*length* and **loc**.*sum* are used in the rule for computing **lhs**.*average*. Local attributes resemble Haskell **let** bindings. In this case their use is necessary because it is not possible to refer to defined synthesized attributes such as **lhs**.*length*.

**Copy rules** A careful reader may have spotted a potential mistake in Figure 2.4: the *Cons* constructor has no definitions for **lhs**.*length* and **lhs**.*sum*! Fortunately, those definitions are automatically generated by the UUAGC, thanks to the so-called *copy rules* feature. When a semantic rule for a certain attribute is absent, the UUAGC generates a rule which gets the missing value from an attribute with the same name. In case of a synthesized attribute this can be, in order of priority, a local attribute, a synthesized attribute of a child or even its own inherited attribute. Similarly, for an inherited attribute of a child, the value of the synthesized attribute with the same name of its closest left sibling providing such an attribute or the value of the inherited attribute of the parent is copied when no explicit rule is given.

The use of copy rules might seem a bit strange at first glance, but they are useful and intuitive to use in practice. Often there are attributes that only need to change in specific locations and otherwise are just to be passed on. For example, an environment containing all bound variables is changed only in places where new variables are introduced, but can be copied unchanged at all other types of nodes. Situations in which this arises closely resembles the use of the *Reader*, *Writer* and *State* monads in Haskell.

### 2.2.2 C$^\sharp$ example

The C$^\sharp$ compiler contains many different synthesized attributes for which we do not show the code here. Figure 2.5 gives part of the code for defining *code* and *declVars*.

The *code* attribute is introduced for all nonterminals and at the root contains the final result of the compiler, which is a list of SSM instructions. We only show two of its definitions. The exact instructions generated are not important for the understanding of this thesis, but one can see that for each method we generate a label, generate instructions for allocating room on the stack for local variables, insert the code of the body and finally take care of cleaning up the stack and return from the function.

In the *declVars* attribute we collect the names of all declared variables. We use a list because the index in the list is used later to compute the location of the variables in memory.

**Use rules** Both the *code* and the *declVars* declaration use the keyword **use**. This keyword is used in combination with the copy rules to automatically generate rules in case some are missing. The first argument to **use** is an operator which combines two values of the children, and the second argument is the value to be used when there are no children with this attribute. These two operations together form a *monoid*.

For the *declVars* attribute this is actually the complete definition. Only at the *DeclC* constructor we introduce a new variable, and at all other places the results are simply combined.

## 2.3 Inherited attributes

Dual to synthesized attributes we have *inherited* attributes, which are used to pass information from a node to its children. Inherited attributes resemble the argu-

```
attr Class MemberL Member StatL Stat ExprL Expr DeclL Decl Const
  syn code use {++} {[ ]} :: {[Instr]}
sem Class
  | ClassC     lhs.code = [Bsr "Main", HALT] ++ @members.code
sem Member
  | MemberM lhs.code = [LABEL @name, LINK (length @loc.locs)]
                        ++ @body.code
                        ++ [UNLINK, RET]
attr DeclL Decl StatL Stat
  syn declVars use {++} {[ ]} :: {[String]}
sem Decl
  | DeclC lhs.declVars = [ @name ]
```

Figure 2.5: Synthesized attributes *code* and *declVars*

ments of a Haskell function and provide some contextual information. For instance, an environment containing bound variables or a value containing the option flags as passed to the compiler.

Apart from the direction the inherited attributes are similar to synthesized attributes. They are again introduced in an **attr** block with the keyword **inh** instead of **syn**. How their value is to be computed is defined by semantic rules too. In this case a node should define the value for all inherited attributes of its children.

Our guestbook example does not make use of inherited attributes, since all information is computed in a bottom-up way. This bottom-up behaviour allows for easier construction of an incremental version, but later in this thesis we show how to support inherited attributes in incremental computations and how they can have a negative influence on incremental evaluation (Section 7.2).

### 2.3.1 C♯ example

Figure 2.6 contains the definition of the inherited attribute *env*, a mapping from variable names to their location on the stack relative to the mark pointer. The parameters are located below the mark pointer as they have been pushed to the stack before the function call occurred, whereas local variables are located above the mark pointer.

```
attr StatL Stat ExprL Expr Decl
  inh env :: {Map String Int}
sem Member
  | MemberM body.env    = Map.fromList ( @loc.params ++ @loc.locs)
            loc.params = zip (reverse @params.declVars) [-2,-3 ..]
            loc.locs   = zip @body.declVars [1 ..]
```

Figure 2.6: The inherited attribute *env*

The *env* attribute contains a Haskell *Map* and is associated with statements, expressions and declarations, but only in the *MemberM* constructor a semantic rule is given. Note that even though *Member* has no inherited attributes, the *body* attribute of nonterminal *Stat* does have an inherited attribute *env* for which a value needs to be defined. For all other productions this value is automatically copied by the copy rules.

Another thing to notice is that the inherited attribute *env* depends on the synthesized attribute *declVars*. This construct can therefore be thought of as a two-pass AG: in the first pass the declared variables are collected into an environment, and in the second pass this environment is used to generate the code. To determine such an evaluation order is however left to the AG compiler and there is no need for the user to specify one.

For synthesized attributes there is no such thing as the **use** keyword; each child has only a single parent and therefore there is nothing to be combined. The initial value of an inherited attribute can be set in a semantic rule, which is often done at the top level node (starting nonterminal).

## 2.4 Chained attributes

A *chained attribute*, sometimes called *threaded attribute*, is a special name for a pair of an inherited and a synthesized attribute that share their name and type. Such an attribute pair is defined with the **chn** keyword in an **attr** block. When there are no explicit semantic rules given, the copy rule mechanism copies the attributes as before. The inherited value of a chained attribute is copied to the inherited attribute of the first child of a node with that attribute, the synthesized value of the first child to the inherited attribute of the second child with that attribute, and so on, and finally the synthesized value of the last child with that attribute to the synthesized attribute of the current node.

One common use of chained attributes is to provide a facility for the generation of fresh names. As we do not have something like a global mutable state, it is not trivial to generate fresh names in different locations. However, global state can be simulated by chaining a single attribute value through the tree which can be read and updated in each node of the tree (compare this with the use of a *State* monad in Haskell). This works especially well for generating fresh names as the order in which the names are generated does not matter, only their uniqueness.

Chained attributes are not often used in this thesis, but in Section 7.3 we show that they can destroy efficient incremental behaviour. We will show some common techniques for circumventing the problems thus arising.

## 2.5 Higher-order attributes

Attribute grammars as introduced in the previous section with inherited and synthesized attributes are useful as such, but the addition of *higher-order attributes* [Vogt et al., 1989] is what makes them especially suitable for the implementation of compilers. Higher-order attributes may be used to model different compiler phases and intermediate data structures in a natural way.

A higher-order attribute is an attribute value which itself is an AST over which attribute values can be computed. In the UUAGC this is implemented by letting the user define *higher-order children*, which are constructed at run-time and can depend on attribute values. Such a higher-order child may of course also contain higher-order children again and attributes of the higher order child may depend on attributes of its parent and vice versa. We call AGs with higher-order children *higher-order attribute grammars* (HOAGs).

The expressive power of HOAGs comes with a price however: achieving incremental evaluation of HOAGs is harder than incremental evaluation of AGs without higher-order attributes. This extra difficulty is the focus of this thesis.

### 2.5.1 Guestbook example

To complete the guestbook example we need to compute the average of the list of true grades stored in *trueReviews*. Not surprisingly we can express this using a higher-order child as shown in Figure 2.7. In this code we also introduce a *Top* nonterminal and production, which are used to wrap the attributes at the top level, a commonly used pattern.

```
data Top | Top gb :: Guestbook
attr Top
  syn average :: {Double}
sem Top
  | Top inst.revs     :: DL
        inst.revs     = @gb.trueReviews
        lhs.average = @revs.average
```

Figure 2.7: Top level wrapper for the guestbook in which a higher-order child is instantiated

The **inst**.$c :: N$ syntax instantiates a child $c$ of nonterminal type $N$. Note that here no primitive Haskell type can be used, as we can only compute attributes over nonterminals. The **inst**.$c = e$ is a semantic rule that defines the value of the higher-order child $c$ using expression $e$. As with other semantic rules, $e$ can be an arbitrary Haskell expression in which attribute values in scope can be referred to.

In our example the higher-order child consists of the list of true grades. In the rest of the semantic rules the child *revs* can now be used as if it were a regular child node. As the type *DL* has no inherited attributes there are no extra definitions for inherited attributes, and only the *average* synthesized attribute is used and copied to the synthesized attribute of the top level node. Note that the last line could have been left out as it would be automatically generated by the copy rule mechanism; we have included it here for clarity.

### 2.5.2 C$^\sharp$ example

In the C$^\sharp$ example higher-order children are used to specify the different compiler phases. Specifically, the AST contains two loop constructs: **while** and **for**, for which the generated code is quite similar. To avoid code duplication we only define the code generation for the **while**, and we desugar the **for** into a **while** statement. Figure 2.8 shows the implementation of the desugaring step. The child *block* has type *Stat* and its value is an AST describing the way in which the **for** can be desugared. In particular, the *init* statement is first executed, followed by a **while** with the *cond* as condition, and the body of the while contains the *body* of the **for** followed by the *incr* statement. Finally the *declVars* and *code* synthesized attributes of the *StatFor* are copied from the results for *block*.

```
attr Expr ExprL Stat StatL Decl Type Const
  syn copy :: self
sem Stat
  | StatFor inst.block    :: Stat
            inst.block    = StatBlock [
                                StatDecl @init.copy,
                                StatWhile @cond.copy (StatBlock [
                                  @body.copy,
                                  StatExpr @incr.copy
                                ])
                              ]
            lhs.code      = @block.code
            lhs.declVars  = @block.declVars
```

Figure 2.8: Using a higher-order child to desugar a **for** into a **while** statement

**Self rules** This example also illustrates another feature of the UUAGC: the *self rules*. The identifier *self* is a special name resolving to the name of the nonterminal for which this attribute is defined. For example, for *Expr* the *copy* attribute has type *Expr*. Furthermore, rules are generated that construct a tree which is an exact copy of the AST. This may sound a bit counter intuitive at first, because *@init* and *@init.copy* are the same tree. However, for implementational reasons it is not possible to refer to nonterminal children directly and thus using *@init* is not permitted. We therefore use the *self* rules to create an attribute we can refer to.

## 2.6 Minimal higher-order attribute grammars

A concept that we have introduced in [Bransen et al., 2014a] is that of so-called *minimal higher-order attribute grammars* (MHOAGs). The difference with HOAGs is that all types must be nonterminals and the right-hand side of the semantic rules must consist of a constructor (production) with an attribute reference for each of its children. In other words, in MHOAGs we do not rely on any backend language such as Haskell for defining the semantic rules, but define a minimal language for attribute grammar computations.

MHOAGs are not suitable for practical usage but have the property that they are fully inspectable, which implies that the attribute grammar compiler has complete knowledge of the program flow. In regular HOAGs however we can have arbitrary Haskell expressions for which the attribute grammar compiler only knows which attributes are used, but not in what way. MHOAGs are Turing complete even though they do not depend on Haskell or any other language. Although we do not use MHOAGs for the main goal of this thesis, we do mention them as they may be useful as intermediate representation in an attribute grammar compiler, or even as a representation for an attribute grammar specific virtual machine.

# 3 Scheduling

An AG compiler, such as UUAGC, compiles the AG code into a runtime evaluator. In the pipeline of the UUAGC the AG code is compiled into Haskell which is then compiled by GHC into an executable. The semantic rules, which can be arbitrary Haskell expressions, are directly copied into the Haskell code generated by the UUAGC, with the attribute references contained in them replaced by generated Haskell variable names.

The runtime evaluation of the attributes can be performed in several different ways. Because attributes may depend on other attributes, an evaluation order must be found such that each semantic rule is only scheduled for evaluation when all attributes that the rule refers to have been computed. The process of finding such an evaluation order is called *scheduling* and we distinguish between two different types of scheduling: *dynamic* and *static*.

**Dynamic scheduling** Dynamic scheduling is performed at runtime when a concrete AST is given for which the attributes are to be computed. The generation of such an evaluator is straightforward by relying on Haskell's *lazy evaluation* as we will describe in Section 3.1. The drawback of this approach is however that no static guarantees are given: when the attribute grammar definition is cyclic, the runtime evaluator may enter an infinite loop. Furthermore, runtime scheduling is relatively expensive, especially when generating incremental evaluators.

**Static scheduling** We talk about static scheduling when an evaluation order is defined at compile time. This is much harder for the AG compiler since no concrete AST is given; only the grammar is known. The static scheduling problem is to find a scheduling strategy based on the grammar such that for each AST the schedule obtained from following this strategy is valid, where valid means that the attributes can be computed in such an order that all dependencies of an attribute are computed before the attribute itself.

**AG classes** Based on the algorithm used for static scheduling different classes of AGs can be identified. A well-known class is the class of *Ordered Attribute Grammars* (OAGs) [Kastens, 1980] which is discussed in Section 3.2. The analysis to obtain a static scheduling strategy for this class takes time polynomial in the size of the grammars, but as we show there are many practical AGs that fall outside this class and cannot be scheduled.

The algorithm by [Kennedy and Warren, 1976] which is discussed in Section 3.3 works for all *Absolutely Noncircular Attribute Grammars* (ANAGs), which is the largest class of AGs for which a (partially) static scheduling algorithm is known. The algorithm is however a combination between static and dynamic scheduling; there is a static guarantee that no loop will be encountered at runtime, but certain decisions on the precise evaluation order are postponed until runtime.

The sweet spot in the AG classes as used in this thesis are the *Linearly Ordered Attribute Grammars* (LOAGs) [Engelfriet and Filè, 1982], which we discuss in Section 3.4. The class of LOAGs is the largest class for which a completely static schedule can be constructed. Hence we have the following class hierarchy.

$$\text{OAGs} \subset \text{LOAGs} \subset \text{ANAGs}$$

Note that the name of the OAGs is a bit counter intuitive in this respect, because the class of LOAGs is strictly bigger.

One important problem of the LOAG scheduling is that the corresponding analysis is computationally hard: statically finding a schedule for a LOAG is NP-complete and therefore no polynomial algorithms are known. Fortunately there exist algorithms that work well in practice, which we show in Section 3.4. The class of LOAGs is the starting point for the rest of the thesis in which we investigate the incremental evaluation of AGs. The runtime evaluator which is eventually generated for LOAGs has the same structure as the runtime evaluator for OAGs, and the rest of this thesis can therefore be read with the information from Section 3.1 and Section 3.2 only.

## 3.1 Lazy evaluation

The simplest purely functional implementation of AGs [Saraiva, 1999] in terms of scheduling (or rather the lack thereof) makes use of *folds*, or *catamorphisms*. A fold defines a general way of recursing over a data type using a given algebra. The algebra defines for each constructor how to compute the result given the results of the children.

The carrier type of the algebra defines which information is computed over the data type. For our translation of AGs relying on lazy evaluation we choose the carrier type for each nonterminal to be a function from a tuple consisting of all the inherited attributes of that nonterminal to a tuple consisting of all the synthesized attributes of that nonterminal. As most interesting AGs are many-sorted algebras, we speak about data types and result types instead.

### 3.1.1 Guestbook example

We show a part of the translation of the guestbook code in Figure 3.1. The type *TGuestbook* is the carrier type for the algebra, which in this case is just a tuple of the two synthesized attributes. Because this example does not have inherited attributes we could have given it the isomorphic type $() \rightarrow (Set\ Name, [Double])$.

The function *semGuestbook*, which we call a *nonterminal semantic function*, is the recursive function taking care of visiting the children of the node at hand. It uses the algebra which contains for the constructors *Empty*, *Arrive* and *Leave*, the corresponding functions *semGuestbookEmpty*, *semGuestbookArrive* and *semGuestbookLeave* to which we refer as the *production semantic functions*. The (production) semantic function for the *Arrive* production is shown, which takes the name and the result of the *tl* child as argument and returns the result for this *Arrive* node by applying the semantic rules as they were defined by the AG. The functions *semGuestbookEmpty* and *semGuestbookLeave* are similar.

In the top level wrapper the list of true grades is instantiated as a higher order child. Figure 3.2 shows the translation in which the nonterminal semantic function *semDL* is called to compute the attribute values of the valid grades.

### 3.1.2 C$^\sharp$ example

In the C$^\sharp$ example most nonterminals are equipped with both inherited and synthesized attributes. As with the AG explanation we do not show the full code and only highlight some of the most important aspects.

The type of the evaluator for the *Stat* nonterminal is:

```
type TGuestbook = (Set Name, [Double])
semGuestbook :: Guestbook → TGuestbook
semGuestbook Empty = semGuestbookEmpty
semGuestbook (Arrive _name _tl) =
  semGuestbookArrive _name (semGuestbook _tl)
semGuestbook (Leave _name _grade _review _tl) =
  semGuestbookLeave _name _grade _review (semGuestbook _tl)
semGuestbookArrive :: Name → TGuestbook → TGuestbook
semGuestbookArrive name_ tl_ =
  let _lhsOsignedIn     :: Set Name
      _lhsOsignedIn     = name_ `Set.insert` _tlIsignedIn
      _trueReviews      = _tlItrueReviews
      _lhsOtrueReviews  :: [Double]
      _lhsOtrueReviews  = _trueReviews
      _tlIsignedIn      :: Set Name
      _tlItrueReviews   :: [Double]
      (_tlIsignedIn, _tlItrueReviews) = tl_
  in  (_lhsOsignedIn, _lhsOtrueReviews)
```

Figure 3.1: Part of the lazy runtime evaluator of the guestbook AG

```
semTopTop :: TGuestbook → TTop
semTopTop gb_ =
  let _gbIsignedIn      :: Set Name
      _gbItrueReviews :: [Double]
      (_gbIsignedIn, _gbItrueReviews) = gb_
      revs_val_           :: DL
      revs_val_           = _gbItrueReviews
      _lhsOaverage      :: Double
      _lhsOaverage      = _revsIaverage
      _revsIaverage     :: Double
      _revsIlength       :: Double
      _revsIsum           :: Double
      (_revsIaverage, _revsIlength, _revsIsum) = semDL revs_val_
  in  (_lhsOaverage)
```

Figure 3.2: Instantiation of the higher-order child *revs* in the lazy runtime evaluator of the guestbook AG

```
type TStat = Map String Int → ([Instr], [String])
```

Remember that *Stat* has a single inherited attribute *env* and two synthesized attributes *code* and *declVars*, which are all present in the type. In order to get access to the inherited attribute the production semantic functions of the nonterminal *Stat* now get an extra argument:

```
sem_Stat_StatDecl :: TDecl → TStat
sem_Stat_StatDecl decl_ _lhsIenv =
  let... in...
```

When a production has a child of type *Stat*, like the *MemberM* production, the inherited attribute *env* is passed as an argument. In Figure 3.3 we show that the argument *_bodyOenv* is passed to *body_* to produce the synthesized attributes of *body*.

Apart from inherited attributes, this example shows an important aspect of the lazy evaluator: a *circular* program! If we take a closer look at the code in Figure 3.3, we see that the argument *_bodyOenv* depends on *_locs*, which depends on *_bodyIdeclVars*. In other words: the argument *_bodyOenv* of the function call depends on the result *_bodyIdeclVars* of that very same function call.

```
semMemberMemberM :: TType → String → TDeclL → TStat
                              → TMember
semMemberMemberM rtype_ name_ params_ body_ =
  let _paramsOenv      :: Map String Int
      _paramsOenv      = Map.empty
      _paramsIcode     :: Code
      _paramsIdeclVars :: [String]
      (_paramsIcode, _paramsIdeclVars) = params_ _paramsOenv
      _loc_params      = zip (reverse _paramsIdeclVars) [−2, −3 ..]
      _loc_locs        = zip _bodyIdeclVars [1 ..]
      _bodyOenv        :: Map String Int
      _bodyOenv        = Map.fromList (_loc_params ++ _loc_locs)
      _bodyIcode       :: Code
      _bodyIdeclVars   :: [String]
      (_bodyIcode, _bodyIdeclVars) = body_ _bodyOenv
      _lhsOcode        :: Code
      _lhsOcode        = [LABEL name_, LINK (length _loc_locs)]
                           ++ _bodyIcode
                           ++ [UNLINK, RET]
  in (_lhsOcode)
```

Figure 3.3: The lazy translation of the semantic function for the *MemberM* production, which is circular

In a strict language such a definition is impossible, but in a lazy language like Haskell such circular definitions do not pose any problem, provided of course that the resulting definition makes sense. As shown in [Bird, 1984] this can even lead to more efficient code than a non-circular version of the code, because the constructor for each node is only used once in a pattern match during evaluation of the tree. In this case our definition is well-founded since the computation of *declVars* does not depend on the argument *env* and therefore *declVars* can be computed without evaluating *env*. For *code* the value of *env* is necessary, but this does not pose a problem either since *env* does not depend (directly nor indirectly) on *code*.

Lazy evaluation can thus be used to dynamically obtain an evaluation order to compute the attributes of the tree: values are only computed when strictly necessary, and if there are no truly cyclic definitions this works. As stated before, the problem with this approach is obviously that erroneous definitions, which encounter a loop at runtime, are not detected statically, i.e. before evaluation starts. Furthermore, the incremental evaluation as discussed later in this thesis can not easily be combined with circular programs and once programs start to loop it may be very hard to find out why.

## 3.2 Ordered AGs

In contrast to relying on lazy evaluation in order to find a schedule for the evaluation of the attributes, we may analyse the grammar and try to statically find a deterministic order in which we can walk over the tree while evaluation attributes, in such a way that whenever we evaluate an attribute it is guaranteed that all the attributes referred to in its definition have been evaluated already, thus avoiding a lot of dynamic tests.

The first static scheduling algorithm that we discuss is the *Ordered Attribute Grammar* algorithm [Kastens, 1980]. The OAG algorithm is a polynomial time algorithm that (by definition[1]) can schedule all attribute grammars in the OAG class. The OAG class does however not contain all AGs for which an order can be statically found. We want to remark again that the name OAG is therefore confusing, but we use it for consistency.

The OAG algorithm analyses the grammar and generates a runtime evaluator with a fixed evaluation order. To construct this fixed order the algorithm analyses the dependencies between the attributes to construct a *visit interface* for each nonterminal. A visit interface is a list of visits with a set of inherited and a set

[1]The class of OAGs is defined as all AGs that can be scheduled by Kastens' algorithm

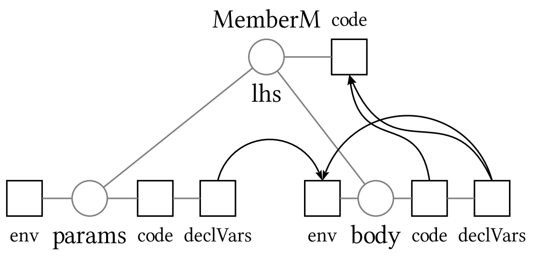

Figure 3.4: $PDG_{MemberM}$ for the C$^\sharp$ AG. Note that the children *rtype* and *name* are not shown here.

of synthesized attributes assigned to each visit, defining in what order the corresponding attributes are computed. Every attribute is assigned to exactly one visit. In each visit the values for the inherited attributes of that visit have become available, while the synthesized attributes of that visit must be computed. For each visit all inherited attributes upto and including that visit are available to compute the synthesized attributes of that visit. The runtime evaluator for the attribute grammar can be generated from these visit interfaces in a straightforward way, and the key part of the algorithm is thus the construction of a valid visit interface for each nonterminal.

### 3.2.1 Dependency graphs

The scheduling algorithms analyse the grammar to find an evaluation order such that for all ASTs generated by that grammar the evaluation order satisfies all *direct dependencies*. These direct dependencies are defined by the semantic rules because each attribute on the left-hand side is defined in terms of the attributes in the right-hand side of the rule. The attributes in the right-hand side of the rule therefore need to be computed before evaluating that semantic rule.

All scheduling algorithms which try to find a static evaluation order start from a so-called *dependency graph*. A dependency graph represents the dependencies between attributes; we use it both in the implementation of the algorithm and in the visualisation of the algorithm. The direct dependency graphs consist only of dependencies directly induced by the semantic rules, while the dependency graphs may also contain dependencies introduced in other ways, for instance by the scheduling. The dependency graph for a production $P$ is called $PDG_P$.

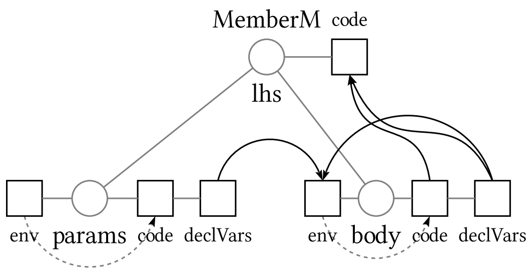

Figure 3.5: Induced $PDG_{MemberM}$

In Figure 3.4 we show part of the direct $PDG_{MemberM}$ for the C$^\sharp$ example. The circles represent the nonterminals in a production, with the topmost node being the parent node and the bottom nodes the children. Attached to the children are the boxes representing the attributes, for which we use the convention of drawing inherited attributes to the left of the node and the synthesized attributes to the right of a node.

The arrows denote the flow of information. For example, we have an arrow *params.declVars* $\rightarrow$ *body.env* because the *params.declVars* attribute is used in computing the *body.env* attribute. Note that here again the presentation may be slightly misleading at first: we speak about dependency graphs, but the arrow $a \rightarrow b$ means that $b$ depends on $a$, and not the other way around! We do however stick to this convention as it is standard in AG literature and intuitive from an AG perspective. The true dependency graph could easily be obtained by reversing all arrows.

## 3.2.2 Induced dependencies

After constructing a dependency graph for each production, the OAG algorithm constructs an *induced dependency graph* for each production. The induced dependencies are dependencies introduced by dependencies in the child or parent nodes of a production. For example, in our *MemberM* production the child *body* is of type *Stat* and any production of the *Stat* nonterminal could appear as a child of *MemberM* in a concrete AST. One of the productions of *Stat* is *StatReturn*, which (again indirectly) uses *env* to compute its synthesized attribute *code* inducing the dependency *body.env* $\rightarrow$ *body.code* which is added to the graph. In Figure 3.5 we show the full induced $PDG_{MemberM}$, where the dashed lines represent the induced dependencies.

The construction of the induced dependency graphs is not straightforward. The addition of a dependency to one production may lead to a new path from one attribute to another at its parent or one of its children, which leads to the addition of further induced dependencies in other productions. An iterative process is therefore required to compute the fixed point of all induced dependency graphs. Because there is only a finite number of valid dependencies this process obviously terminates.

By definition, if a cycle is found in any of these induced dependency graphs the attribute grammar does not belong the OAG class. On the other hand the fact that the induced dependency graphs are acyclic does not guarantee yet that the grammar is in the OAG class, as we show in Section 3.2.6.

### 3.2.3 Nonterminal dependency graph

The *nonterminal dependency graph* for a nonterminal $N$ called $NDG_N$ is a pessimistic approximation of all dependencies for that nonterminal. $NDG_N$ contains a node for each attribute of $N$, and an edge $a \rightarrow b$ if for any $p$ there exists a path $a \rightarrow b$ in the induced $PDG_p$ with $a$ and $b$ being attributes of the same node of type $N$. Note that the production $p$ does not need to be a production of $N$, because children of type $N$ can also appear in productions of other nonterminals.

An important remark here is that the OAG algorithm constructs a *global order* on the attributes of a nonterminal; every occurrence of that nonterminal uses the same dependencies, irrespective of the specific AST used for its children, its context in the tree, or the production being used at this location. As a static schedule is a schedule in which we do not have to take a further decisions during evaluation; we take a worst case approach and assume that all dependencies of all productions occur at the same place at the same time.

### 3.2.4 Visit interfaces

From the nonterminal dependency graphs the *visit interfaces* are constructed. A visit interface defines for a nonterminal in what order the attributes of that nonterminal are to be computed. Specifically, a visit interface consists of one or more visits, where each visit consists of a (possibly empty) set of inherited attributes and a (nonempty) set of synthesized attributes. Every attribute of that nonterminal belongs to exactly one visit.

A requirement for the visit interfaces is that they should comply with the dependencies from the nonterminal dependency graphs. Let $visit\ (a)$ be the visit number in which attribute $a$ is scheduled, where the visits are numbered in increasing order. The lowest visit number is the first visit. For any dependency (direct or indirect)

$i \rightarrow s$ for inherited attribute $i$ and synthesized attribute $s$, it should be the case that $\mathit{visit}\ (i) \leqslant \mathit{visit}\ (s)$. In other words, the inherited attribute $i$ must be available before or in the same visit in which $s$ is computed. For a dependency $s \rightarrow i$ it must be the case that $\mathit{visit}\ (s) < \mathit{visit}\ (i)$, which means that $s$ needs to be computed in an earlier visit than $i$. As the direct dependencies are always between different types of attributes, we do not need to restrict $i \rightarrow i$ or $s \rightarrow s$.

**Construction** To construct the visit interface for nonterminal $N$ from $NDG_N$ the OAG algorithm proceeds as follows.

- Repeat for $v = 0$, $v = 1$, etc. as long as $NDG_N$ is non-empty:
  - For each inherited attribute $i$ such that there is no $s \rightarrow i$ in $NDG_N$:
    - Put $i$ in visit $v$.
  - For each synthesized attribute $s$ such that for each $i \rightarrow s$ in $NDG_N$ $i$ is in visit $v$:
    - Put $s$ in visit $v$.
  - For each attribute $a$ in visit $v$:
    - Remove $a$ and all edges connected to $a$ from $NDG_N$.

The result of this algorithm is that each attribute is scheduled as early as possible and that the resulting schedule has the least number of visits. In Section 3.2.6 we show that it can also be the case that the algorithm fails to find a schedule because of the extra dependencies that are introduced by this algorithm, which are not reflected in the description above.

For the *Stat* nonterminal of our C$^\sharp$ example, which is also the type of the *body* child of *MemberM*, this algorithm constructs two visits: visit 0 computes the synthesized attribute *declVars*, and visit 1 gets the inherited attribute *env* and computes the synthesized attribute *code*. This order satisfies all dependencies as shown in Figure 3.5.

### 3.2.5 Code generation

In the code generated from the visit interfaces, the type of the evaluator for each nonterminal encodes the visit interface of that nonterminal in the following way. Instead of a single evaluator type, we construct a type for each visit in the visit interface, which takes as arguments the inherited attributes of that visit and returns a tuple containing the synthesized attributes of that tuple. For all but the last visit of a nonterminal, one more thing is returned: the evaluator for the next visit.

For our C$^\sharp$ example this results in the following types:

$$\begin{aligned}&\textbf{type } \mathit{TStat} \;\; = ([\mathit{String}], \mathit{TStat}_1)\\ &\textbf{type } \mathit{TStat}_1 = \mathit{Map\ String\ Int} \rightarrow [\mathit{Instr}]\end{aligned}$$

As the type of the full evaluator for *Stat* is the type of the first visit, we use the name *TStat* instead of $\mathit{TStat}_0$. As we see the first visit also returns something of type $\mathit{TStat}_1$, which is the evaluator for the second visit of the *Stat* nonterminal.

The code for the *MemberM* production is shown in Figure 3.6. The first difference to the lazy version (Figure 3.3) is that we have nested the **let**... **in**... to make the evaluation order explicit in the code. Although this is not strictly necessary, making the evaluation order explicit may help the compiler in generating more efficient code. Furthermore, the computation of the attributes of *params_* and *body_* is now split into multiple calls, one for each visit. The circular definition has now been unfolded by first computing *_bodyIdeclVars*, then constructing *_bodyOenv* after which *_bodyIcode* can be computed. Because of the explicit evaluation order the code can also be written in a monadic style.

In Figure 3.7 we show the generated code for the *StatReturn* production, which has two visits. In the first visit *_lhsOdeclVars* is computed, which for **return** is an empty list, since no variables are declared. The evaluator for the subsequent visit is then defined in a let binding such that this evaluator can be returned together with *_lhsOdeclVars*. In the second visit the code of the child is computed based on the current environment. Note that in this way subsequent visits can refer to attributes that were declared in an earlier visit, which happens often in AGs with multiple visits (but not in this example).

### 3.2.6 Counter example

As indicated there exist AGs for which the algorithm by Kastens fails to find an evaluation order even though a linear order exists. While [Kastens, 1980] states that such AGs do not often show up in practice, we have encountered this situation quite often, especially when the number of attributes grows. In this section we show a very simple example AG for which Kastens' algorithm fails.

The AST of our counter example is a binary tree with integer values in the leaves. We define two attributes: one for giving a unique label to each leaf in the tree and one for computing a list of all values in the leaves in pre-order. These computations may seem unrelated and artificial, but such combinations of computations often occur in compiler construction.

```
semMemberMemberM :: TType → String → TDeclL → TStat
                          → TMember
semMemberMemberM rtype_ name_ params_ body_ =
  let _paramsIdeclVars                :: [String]
      (_paramsIdeclVars, params_2) = params_ in
  let _paramsOenv :: Map String Int
      _paramsOenv = Map.empty in
  let _paramsIcode :: Code
      _paramsIcode = params_2 _paramsOenv in
  let _loc_params   = zip (reverse _paramsIdeclVars) [−2, −3 ..] in
  let _bodyIdeclVars            :: [String]
      (_bodyIdeclVars, body_2) = body_ in
  let _loc_locs      = zip _bodyIdeclVars [1 ..] in
  let _bodyOenv      :: Map String Int
      _bodyOenv      = Map.fromList (_loc_params ++ _loc_locs) in
  let _bodyIcode     :: Code
      _bodyIcode     = body_2 _bodyOenv in
  let _lhsOcode      :: Code
      _lhsOcode      = [LABEL name_, LINK (length _loc_locs)]
                       ++ _bodyIcode
                       ++ [UNLINK, RET]
  in (_lhsOcode)
```

Figure 3.6: Code generated for the OAG translation of the *MemberM* production

```
semStatStatReturn :: TExpr → TStat
semStatStatReturn expr_ =
  let _lhsOdeclVars :: [String]
      _lhsOdeclVars = [] in
  let semStatStatReturn_1 :: TStat_1
      semStatStatReturn_1 _lhsIenv =
        let _exprOenv  :: Map String Int
            _exprOenv  = _lhsIenv in
        let _exprIcode :: [Instr]
            _exprIcode = expr_ _exprOenv in
        let _lhsOcode  :: [Instr]
            _lhsOcode  = _exprIcode ++ [STR R3, UNLINK, RET]
        in (_lhsOcode)
  in (_lhsOdeclVars, semStatStatReturn_1)
```

Figure 3.7: Code generated for the OAG translation of the *StatReturn* production

```
data Tree | Leaf val :: {Int}
          | Bin  l, r :: Tree

attr Tree
  chn label :: {Label}

sem Tree  | Leaf lhs.label = nextLabel @loc.label
                 loc.label =  @lhs.label
          | Bin  l.label   =  @lhs.label
                 r.label   =  @l.label
                 lhs.label =  @r.label

attr Tree
  chn vals :: {[Int]}

sem Tree  | Leaf lhs.vals = @val : @lhs.vals
          | Bin  l.vals   = @r.vals
                 r.vals   = @lhs.vals
                 lhs.vals = @l.vals
```

Figure 3.8: Simple AG for which a linear evaluation order exists which is not found by the OAG algorithm

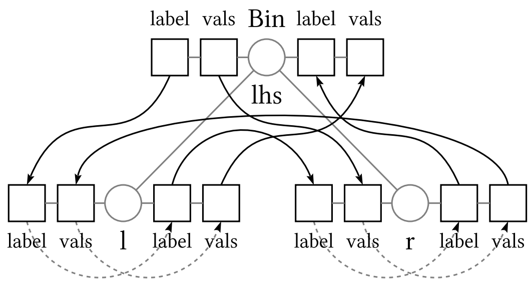

Figure 3.9: Induced $PDG_{Bin}$

In Figure 3.8 we show the full code. In order to compute a unique label for each leaf we introduce some type *Label* and a function *nextLabel* :: *Label* → *Label* that gives the next available (unique) label based on the current one. The *label* attribute is a chained attribute holding the next label value to be handed out, and is copied through the tree in a standard pre-order traversal. At each *Leaf* the current label is stored such that other rules may use **loc**.*label*, and the next label is passed on to be used in the labelling of subsequent leaves. Note that due to the copy rules most of the semantic rules might have been left out; we have included them for clarity.

Collecting the values in the leaves is also done with a chained attribute. We do compute only synthesized information so one might expect a single synthesized attribute to suffice. Since list concatenation is expensive a synthesized attribute will in general not produce the list of leave values in linear time. A better solution is to introduce another chained attribute in which we collect the *Leaf* values; note that this attribute is chained in the opposite order: the inherited attribute of the root is passed to the right child first, in the tree the leaves of that tree are prepended, and the updated list is then passed to the left child, in order to be extended with the nodes from that subtree. Finally the result computed from the left subtree becomes the result of the tree at hand. The initial empty list is given at the top level, which is not shown in the code.

**Induced dependency graph** We show the induced $PDG_{Bin}$ in Figure 3.9. It is not hard to see that no information flows between the two computations. The OAG scheduling algorithm therefore constructs a single visit with inherited attributes *label* and *vals* and synthesized attributes *label* and *vals*.

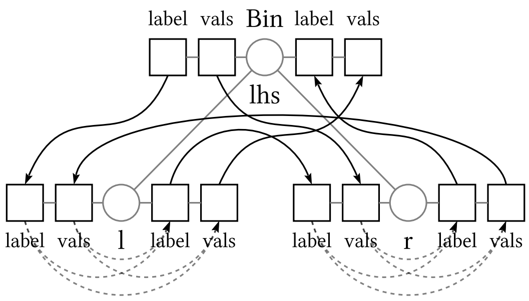

Figure 3.10: Extended $PDG_{Bin}$

**Extended dependencies** The result of combining multiple attributes into a single visit is that extra dependencies are induced. For each visit it must be the case that the inherited attributes of that visit can be computed before the synthesized attributes of that visit. To complete the induced $PDG_N$ into the *extended dependency graph* these extra edges are added between each pair of inherited and synthesized attributes of the same visit.

For our example there is only one visit such that the extra edges are *label* → *vals* and *vals* → *label*, both arrows from an inherited to an (albeit unrelated) synthesized attribute. In Figure 3.10 we show this extended $PDG_{Bin}$, which unfortunately contains a cycle: *l.vals* → *l.label* → *r.label* → *r.vals* → *l.vals*. It is therefore not possible to generate code that computes these attributes without relying on lazy evaluation, and this AG is therefore not *compatible*, meaning it is not in the class of OAGs.

### 3.2.7 Augmenting dependencies

One solution to the problem that such AGs cannot be scheduled with the OAG algorithm, is to introduce *augmenting* or fake dependencies. Such dependencies are not present in the semantic rules, but may be added to the source code separately with the sole purpose of guiding the scheduling process. The AGs that can be scheduled with the addition of a collection of augmenting dependencies form the class of *arranged orderly attribute grammars* [Kastens, 1980], which is exactly the same as the class LOAG, which contains all AGs for which a static linear order exists.

In our example there are multiple augmenting dependencies that may be added. Intuitively, the resulting schedule should consist of two visits, one computing the *label* inherited and synthesized attribute and the other computing the *vals* attributes.

Indeed, adding an additional dependency between the synthesized attribute *label* and the inherited attribute *vals* solves the problem.

The problem with such augmenting dependencies is that there may be exponentially many of them. Finding a set of augmenting dependencies to construct a valid schedule is an NP-hard combinatorial problem [Engelfriet and Filè, 1982]. In some cases, like the small example above, these can be found manually, but in large projects like the UHC this is a tedious job and therefore it is not feasible to do this manually. We furthermore note that usually a large attribute grammar is developed in an incremental way, and continuously adapting the set of augmenting dependencies to the new extended grammar distracts from the main programming task. In the next sections we therefore present some improved scheduling algorithms.

## 3.3 Absolutely Noncircular AGs

The algorithm from [Kennedy and Warren, 1976], from now on referred to as K&W, is the second scheduling algorithm we discuss. This algorithm can find a schedule for attribute grammars in the class of *absolutely noncircular attribute grammars*, which is the largest class of AGs for which a scheduling algorithm is known. The property of absolutely noncircularity [Knuth, 1968] is a pessimistic approximation of true noncircularity [Knuth, 1971]; there exist AGs that do not belong to the class of ANAGs for which none of the ASTs contains a cycle in its attribute dependencies.

The most important difference with the OAG algorithm is that certain decisions on the exact evaluation order are delayed until runtime, which makes the scheduling partially dynamic. Instead of a visit interface the K&W algorithm constructs a *visit graph* for each nonterminal, representing a set of different visit interfaces that can be used at runtime based on the context. These visit graphs are constructed from *input-output graphs*, which are induced dependency graphs in which the context is not taken into account. In Section 3.3.1 we give a definition of input-output graphs, and in Section 3.3.2 we give the definition and the construction of a visit graph. In Section 3.3.3 we describe the generation of a runtime evaluator from such a visit graph, which we illustrate with the *label* and *vals* example of Section 3.2.6. For the [Bransen et al., 2012] paper we have created an efficient implementation in Haskell, of which we discuss the most important aspects in Section 3.3.4.

### 3.3.1 Input-output graph

An input-output graph resembles an induced dependency graph as used in the OAG algorithm, and it is used in a similar way. The important difference is that in an input-output graph the dependencies for the parent node (the context) are not taken into account. Hence the name input-output graph: only dependencies from inherited (input) to synthesized (output) attributes are induced.

The input-output graph for a production $p$ we call $PIO_p$ and the input-output graph for a nonterminal $N$ we call $NIO_N$. The $NIO_N$ is the union over all dependencies from the productions of $N$; for each production $p$ of $N$, if there is a path from inherited attribute $i$ to synthesized attribute $s$ of the *lhs* of $p$ in $PIO_p$, there is an edge $i \rightarrow s$ in $NIO_N$. The $PIO_p$ consists of the $PDG_p$ together with all dependencies for its children; for each child $c$ in $p$ of type $M$, the dependencies from $NIO_M$ are present in $PIO_p$.

As with the induced dependency graphs, the process of constructing all input-output graphs is a fixed point computation. Again, as the number of possible dependencies is finite the algorithm terminates. When none of the production input-output graphs contain a cycle, the AG is absolutely noncircular and the K&W algorithm finds a valid schedule.

### 3.3.2 Visit graph

The OAG algorithm defines a global order on the attributes of a nonterminal in terms of a visit interface that works in every context. As we have shown in Section 3.2.6 such a global order may introduce extra dependencies leading to cycles. To avoid that problem the K&W algorithm constructs a visit graph, which represents the different visit interfaces for all possible contexts of a nonterminal. The choice of the actual visit interface is taken at runtime based on the context, and the algorithm guarantees that for every possible AST the runtime evaluator is able to choose a visit interface such that no cycle is encountered at runtime.

The visit graph for a nonterminal $N$ is a directed acyclic graph with a special starting vertex. Every possible path from the starting vertex to a leaf vertex represents a visit interface, which in this context we call *visit sequence*. The vertices in the graph represent the possible states of a nonterminal, where a state is the set of attributes that have already been evaluated. Every edge in the graph corresponds to a visit to a node, hence each edge is labelled with a set of inherited and a set of synthesized attributes. For every visit sequence it must be the case that the sets of inherited attributes along the path are all disjoint, and similarly for the synthesized attributes.

With every edge we also associate an *execution plan* for each production of the corresponding nonterminal. This execution plan is a list of instructions for computing the values of the synthesized attributes of that visit. These instructions are

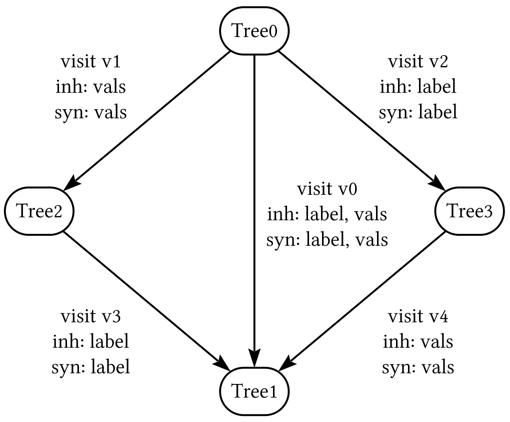

Figure 3.11: The visit graph of the example

either the computation of an inherited attribute of a child, the invocation of the visit of a child, or the computation of a synthesized attribute.

The key algorithm of this section is an adapted version of the K&W algorithm for the construction of the visit graph including the execution plans. This algorithm performs stable and predictable inference of the evaluation order which determines visits in a demand driven way with per visit the smallest set of inherited attributes that are needed to produce the demand set of synthesized attributes, and per visit the largest set of synthesized attributes that can be derived from the inherited attributes. If the AG is absolutely noncircular, thus none of the input-output graphs contains a cycle, this algorithm returns a complete visit graph. At run time this leads to a predictable evaluation order, but it is partially dynamic because the actual visits performed for each node depend on the collection of new inherited attributes that have become available for the parent node.

**Example** Figure 3.11 shows the visit graph for our example, containing three visit sequences. The execution plans for the productions are not made explicit in this figure. Visit $v_0$ is the main visit, which is also invoked at the top level, in which both inherited attributes are available at the start of the visit. For the *Leaf* we can compute both synthesized attributes directly, but for the *Bin* production we need to perform child visits. As we have already seen it is impossible to do this in a single visit. For the left child visit $v_1$ is done first and then visit $v_3$, and for the right child visit $v_2$ is done first and then visit $v_4$. Hence, for scheduling visit $v_0$ of the *Bin* production extra visits are added to the visit graph. Execution plans can be created for all visits using only these visit sequences.

**Construction** The algorithm works by gradually building up the visit graphs from the $NIO_p$'s. Every visit graph has exactly one vertex without incoming edges which is the *starting vertex*.

To construct the initial visit graph for a nonterminal $N$ we create a starting vertex and one pending edge plus the corresponding target vertex, containing all inherited and synthesized attributes defined on $N$. In our example $Tree_0$ is the starting vertex and $v_0$ the initial pending edge.

The main loop of the K&W algorithm constructs the visit graphs for all nonterminals simultaneously. Pending edges are handled one by one, thereby possibly adding new pending edges to the visit graph of other nonterminals. For each pending edge of the visit graph for some $N$ an execution plan is constructed for each production $p$ of $N$. Once a pending edge is handled it is marked final and the algorithm terminates when there are no more pending edges. As the set of possible edges is finite the algorithm always terminates.

For each visit graph we keep for every vertex in that graph a list with a state for each of the children of the productions. Such a state contains the set of attributes that have already been computed for a child $c$ and is represented by a reference to a vertex in the visit graph of the nonterminal $N$ of which $c$ is an instance. Initially every child state is the starting vertex of the visit graph for $N$.

For the construction of an execution plan for a production $p$ we use marks on the $PIO_p$. Marked vertices represent the attributes that have already been evaluated and thus are available. To handle a pending edge we mark the vertices corresponding to the inherited attributes assigned to the pending edge. The goal is then to mark all synthesized attributes assigned to this pending edge. A vertex can be marked only if all its dependencies are already marked. For synthesized attributes of children a child visit needs to be performed, and we maximise the number of synthesized attributes that we compute in the child visit. This child visit is where potential new pending edges are added. The vertices in $PIO_p$ are recursively marked until all synthesized attributes are marked. If the production input-output graphs do not contain cycles this is always possible.

Apart from the synthesized attributes of the children that are strictly needed for computing the desired synthesized attributes of the current visit, we also add synthesized child attributes that depend only on the inherited child attributes that are already evaluated. In other words, we add synthesized child attributes that can already be computed without introducing extra child visits. By eagerly adding such synthesized attributes we avoid constructing many almost similar visits and thus limit the growth of the visit graph resulting from the K&W approach.

### 3.3.3 Runtime evaluator

The runtime evaluator coming from the K&W algorithm can be thought of as taking the visit graph and the AST as parameters, and evaluating the attributes according to the execution plans in the visit graph. However, as the visit graph is computed statically at compile time we can already partially apply the evaluator to the visit graph and directly generate an evaluator in which the visit graph is implicitly encoded.

The type of the evaluator resulting from applying the semantic function to the AST is similar to that of the OAG code; given the values of some inherited attributes the values of some synthesized attributes and the function for the next visit are returned. In this case however an extra parameter needs to be given to specify *which* visit needs to be performed, as there can be multiple possible visits. This extra parameter is given by the parent node, for which the execution plan specifies which child visits need to be performed. The extra parameter is thus where the scheduling is dynamic, as at runtime extra pattern matches need to be performed to find the right visits.

Figure 3.12 shows some important parts of the generated code for the example. As the K&W algorithm is not the one finally used in this thesis, we do not go into much detail here and leave out many implementation details. We do however highlight some of the key elements of the generated runtime evaluator.

For every node in the visit graph there is a data type similar to $\mathit{TTree\_s}_0$ storing the possible visits for a node in that state. The corresponding type $\mathit{KTree\_s}_0$ has a constructor for each possible visit and is a generalized algebraic data type [Cheney and Hinze, 2003, Xi et al., 2003] such that the type of the corresponding visit is propagated at the type level. When invoking a visit, which is illustrated in $v_0$, the $\mathit{KTree\_v}_1$ parameter fixes the type of the return, which is this case is a function from *Label* to a tuple consisting of a *Label* and a new state for the $l$ child. Finally the $k_0$ function is an example of where the dynamic scheduling takes place; based on the parameter one of the visit functions is selected and because of the GADT pattern matching these visit functions can have different types.

We do not go into further details but like to point out again that the generated code is purely functional and strongly typed. The type checker proves that the code has the define-before-use property and that no attribute depends indirectly on itself.

```
type $TTree = TTree\_s_0$
data $TTree\_s_0 = CTree\_s_0$ {
  $inv\_Tree\_s_0 :: \forall\ t.\ KTree\_s_0\ t \to t$
  }
data $KTree\_s_0\ k$ where
  $KTree\_v_0 :: KTree\_s_0\ TTree\_v_0$
  $KTree\_v_1 :: KTree\_s_0\ TTree\_v_1$
  $KTree\_v_2 :: KTree\_s_0\ TTree\_v_2$
type $TTree\_v_0 = (Label, [Int]) \to (Label, [Int])$
type $TTree\_v_1 = Label$             $\to (Label, TTree\_s_2)$
$sem\_Tree\_Bin :: T\_Tree \to T\_Tree \to T\_Tree$
$sem\_Tree\_Bin\ l\_\ r\_ = st_0$ where
  $st_0 =$ let $k_0 :: KTree\_s_0\ t \to t$
              $k_0\ KTree\_v_0 = v_0$
              $k_0\ KTree\_v_1 = v_1$
              $k_0\ KTree\_v_2 = v_2$
              $v_0 :: TTree\_v_0$
              $v_0 = \ldots$
                  let $(\_lIlabel, \_l_2) = inv\_Tree\_s_0\ l\_\ KTree\_v_1\ \_lOlabel$ in
                  $\ldots$
              $\ldots$
           in $CTree\_s_0\ k_0$
  $\ldots$
```

Figure 3.12: Small part of the generated code for the K&W algorithm for the example

The generated code can be seen as the evidence of the proof that the AG is absolutely noncircular. For further details we would like to refer to [Middelkoop, 2012].

### 3.3.4 Functional implementation

For the UUAGC we have implemented the K&W algorithm efficiently in Haskell and here we give some implementation details. As with the code generation only highlight the important aspects but do not go into detail because the K&W algorithm is not used in the rest of this thesis, and it is only explained for educational reasons.

**Dependency graphs** The first ingredient of the scheduling is how we represent dependency graphs efficiently in Haskell (Figure 3.13). An important property of the dependency graphs is that no new vertices are added to the graph during the execution of the algorithm which means that upon construction of the initial graph the list of all vertices is known. We use this property to assign a unique number to each vertex at construction time and we use these numbers as *Array* indices thus giving constant lookup time.

The algorithm from [Knuth, 1968], which we call Knuth-1, is an algorithm that statically determines whether an AG is *absolutely noncircular*. In our implementation the Knuth-1 algorithm is used both as a check whether the AG can be scheduled with the K&W algorithm and as a way to construct the $PIO_p$ for every production $p$.

In the Knuth-1 algorithm there are several operations on the input-output graph for which an efficient implementation is important: *graphEdges*, the enumeration of all edges in the graph, *graphInsert*, the insertion of a new edge into the graph, and *graphContainsEdge*, a check whether two vertices are connected. Also, it is important to know which vertices become connected as a result of the insertion of a new edge.

To accomplish this we maintain the invariant that the graph is always transitively closed. Keeping such an invariant may lead to more efficient update operations [La Poutré and van Leeuwen, 1988]. We store the graph as a set of successors and a set of predecessors for each vertex. Upon insertion of a new edge we transitively close the graph in the obvious way. The return value of *graphInsert* is the list of newly added edges, excluding the one in the argument.

We use the ST Monad [Launchbury and Peyton Jones, 1994] for performing efficient in-memory updates in a functional setting. Using the *STRef* in the successors and predecessors *Array*, the sets are updated in place without the need to update the *Array* structure itself.

```
type Vertex  = ...                 -- External vertex type
type Edge    = (Vertex, Vertex)    -- External edge type
type IVertex = Int                 -- Internal representation of a vertex
type IEdge   = (IVertex, IVertex)  -- Internal representation of an edge

data DepGraph s =                  -- Representation of the graph
  DepGraph {vertexIMap   :: Map   Vertex  IVertex
           ,vertexOMap   :: Array IVertex Vertex
           ,successors   :: Array IVertex (STRef s (Set IVertex))
           ,predecessors :: Array IVertex (STRef s (Set IVertex))}

graphConstruct     :: [Vertex]     → [Edge] → ST s (DepGraph s)
graphInsert        :: DepGraph s → Edge     → ST s [Edge]
graphContainsEdge  :: DepGraph s → Edge     → ST s Bool
graphSuccessors    :: DepGraph s → Vertex   → ST s [Vertex]
graphPredecessors  :: DepGraph s → Vertex   → ST s [Vertex]
graphVertices      :: DepGraph s →            ST s [Vertex]
graphEdges         :: DepGraph s →            ST s [Edge]
```

Figure 3.13: Dependency graph representation

**Input-output graphs** As explained, the K&W algorithm constructs a $PIO_p$ for every production $p$ by constructing a $NIO_N$ for every nonterminal $N$, which represents the union over all possible dependencies for a tree rooted by $N$. We implement the construction of these input-output graphs as a work-list algorithm that alternates between adding dependencies coming from an $NIO_N$ to a $PIO_p$ and vice versa. The work-list contains the pending dependency edges: edges that have been added to one of the graphs and may have to be added to other graphs. Initially the list of pending edges consists of all initial edges (direct dependencies) of $PIO_p$ for all productions $p$. The main function is implemented as:

```
knuth₁ :: [NontM s] → ST s ()
knuth₁ nonts = do
  nes ← forM nonts $ λnont → do
    pend ← mapM graphEdges (productions nont)
    return (pend, nont)
  knuth₁′ nes   -- run worklist algorithm on initial graph
```

The type *NontM* represents a nonterminal in the AG containing its productions, attributes and semantic rules. *NontM* also contains the input-output graphs to which extra edges are added by the Knuth-1 algorithm.

To add pending edges coming from some $PIO_p$ to a $NIO_N$ the following helper function is used.

```
addProdNont :: ([[Edge]], NontM s) → ST s [Edge]
```

The argument of this function is a pair consisting of a list of edges that needs to be added and the corresponding nonterminal. The return value is the list of all edges that are newly added due to taking transitivity into account. These new edges are then taken as the new list of pending edges which are to be added to the corresponding $PIO_p$'s:

```
addNontProd :: ([Edge], NontM s) → ST s [[Edge]]
```

Again, this function takes a pair of a list of new edges and the corresponding nonterminal, and returns for each nonterminal a list of edges that were added to its production input-output graphs due to transitivity. The new edges must be taken as new pending list.

The helper function $knuth_1'$ recursively alternates between adding edges to the nonterminal input-output graphs and adding edges to the production input-output graphs, and it terminates when the list of pending edges is exhausted.

```
runVG              :: VG s a     → ST s a
insertInitialNode  :: NontM s → VG s VGNode
createPending      :: VGNode → [Identifier] → [Identifier]
                                           → VG s VGEdge
selectPending      :: VG s VGEdge
getInherited       :: VGEdge  → VG s [Identifier]
getSynthesized     :: VGEdge  → VG s [Identifier]
markFinal          :: VGEdge  → VG s ()
getProductions     :: VGEdge  → VG s [VGProd]
onMarkedDG         :: (ProdDepGraphM s → ST s a) → VGProd
                                                  → VG s a
isDGVertexFinal    :: VGProd  → Vertex → VG s Bool
setDGVerticesFinal :: VGProd  → [Vertex] → VG s ()
getChildState      :: VGProd  → Identifier → VG s VGNode
addChildVisit      :: VGProd  → Identifier   → VGEdge
                                           → VG s VisitStep
addVisitStep       :: VGProd  → VisitStep → VG s ()
repeatM            :: VG s ()   → VG s ()
```

Figure 3.14: Functions available in *VG* monad

```
knuth'_1 :: [([[Edge]],NontM s)] → ST s ()
knuth'_1 nonts = do
  edges ← mapM addProdNont nonts
  let nontedges = concat edges
  when (¬ (null nontedges)) $ do
    perprod ← mapM (λ(_,x) → addNontProd (nontedges,x)) nonts
    newlist ←  zipWithM (λ(_,nt) me → return (me,nt))
                    nonts perprod
    when (any (¬ ∘ null) perprod) $ knuth'_1 newlist
```

**Visit graph representation** In order to maintain the visit graph we use a monad, *VG*, built on top of the *ST* monad. In this way the representation of the visit graph is separated from the actual algorithm. This greatly improves the readability of the code.

The *VG* monad is defined as follows:

```
type VG s a = ErrorT String (StateT (VGState s) (ST s)) a
```

The inner monad is the *ST* monad with threaded state *s*. On top of this there is a *State* monad with state *VGState s*, which contains the visit graph representation. The topmost monad is the *Error* monad which is used for capturing failure and error messages. This is also used in the implementation of the function *repeatM* :: *VG s* () → *VG s* () that repeats the execution of the argument until *mzero* (failure) is encountered. The *VGState s* is used for storing the visit graph and all necessary related data. Figure 3.14 shows type signatures of the functions that are available in the *VG* monad.

**Visit graph construction** At the start of the algorithm we create a starting vertex for each nonterminal and the corresponding pending edge.

```
kennedyWarren :: [NontM s] → VG s [Maybe VGEdge]
kennedyWarren nonts = do
  initvs ← forM nonts $ λnont → do
    nd      ← insertInitialNode nont
    initv   ← createPending nd (inh nont) (syn nont)
    return initv
  ...
```

The main loop for handling pending edges is implemented using *repeatM* as follows:

```
repeatM $ do
  pend  ← selectPending
  prods ← getProductions pend
  inhs  ← getInherited   pend
  syns  ← getSynthesized pend
  forM prods $ λprod → do
    ...
  markFinal pend
return initvs
```

When there are no more pending edges the *selectPending* function will result in *mzero*, thereby breaking the *repeatM* loop and the algorithm terminates. The final visit graph and the corresponding execution plans can now be retrieved from the internal representation.

The marking of attributes is implemented as recursive function that assigns a number to every vertex in a depth-first way. For every attribute the number is the maximum of all its predecessors, and for a synthesized child attribute it is the maximum plus one, because one extra child visit needs to be performed. The *foldChildVisits* helper function implements this behaviour.

We mark the inherited attributes as final and then call the *foldChildVisits*:

```
setDepGraphVerticesFinal prod (map createLhsInh inhs)
(vis, i) ← foldM (foldChildVisits prod) ([ ], 0) (map createLhsSyn syns)
setDepGraphVerticesFinal prod (map fst vis)
...
```

The return value *vis* has type [(*Vertex*, *Int*)] and indicates the vertices that correspond to rules or attributes that need to be evaluated at this stage, together with the corresponding child visit number.

To generate the final execution plans (implemented in terms of *addVisitStep*) we group the $vis_2$ (*vis* combined with the extra synthesized child attributes) by visit number. For every visit we first evaluate all corresponding rules and then add the desired child visits.

```
forM (groupSortBy (comparing snd) vis₂) $ λvisit → do
  let (chatr, rules) = partition isChildAttr $ map fst visit
    -- Rules have been added to the list in reverse order
  forM (reverse rules) $ λrule → do
    addVisitStep prod (Sem rule)
    -- Group by child
  forM (groupSortBy (comparing getChildName) chatr) $ λchildvs → do
    let cinhs  = map getName $ filter isChildInh childvs
    let csyns  = map getName $ filter isChildSyn childvs
    let cname = getChildName $ head childvs
    curstate ← getChildState prod cname
    target ← createPending curstate (fromList cinhs) (fromList csyns)
    step    ← addChildVisit prod cname target
    addVisitStep prod step
```

For the full implementation we refer the reader to the source code of the UUAGC[2].

### 3.3.5 Discussion

We have formulated our version of the K&W algorithm in a rather different way than the original formulation of [Kennedy and Warren, 1976]. One important dif-

[2]`http://hackage.haskell.org/package/uuagc`

ference involves the marking of the input-output graph vertices. Furthermore, in the original formulation the child visits are performed based on availability of inherited attributes, and all child visits that can be done are done. Our approach works demand-driven, in the sense that we only do child visits that are strictly needed for the computation of the requested synthesized attributes. This optimisation thus removes unnecessary child visits and limits the growth of the visit graph.

An important drawback of the K&W algorithm its running time grows exponentially because the visit graphs can have a size which is exponential in the number of attributes. Our experience has shown that for practical AGs the visit graphs stay small because usually the AG programmer has some evaluation order in mind.

In the rest of this we do not use the K&W algorithm because the runtime evaluator is dynamic. When doing incremental computation the visits are memoized, but changes higher up in the AST can lead to a different context in which other visits need to be performed, even though the values of the attributes stay the same. We therefore continue in the next section with a static scheduling algorithm that leads to a complete linear order for all practical AGs we have encountered.

## 3.4 Linearly Ordered AGs

As explained in the introduction we believe that the LOAG class is the sweet spot of the class hierarchy; all AGs that we have ever programmed fall in that class, while it is still possible to find a schedule statically. The scheduling problem for LOAGs is however NP-complete [Engelfriet and Filè, 1982], which in the earlier days of AG scheduling was used as an argument to not use LOAGs.

In our work on scheduling we did however find that despite the NP-hardness of the problem, we can do the scheduling efficiently in practice. We have been able to construct AGs that can not be efficiently scheduled, but these are highly artificial and we argue that these AGs are never written in practice. Intuitively, for the AGs that fall outside of the OAG class like our *label* and *vals* example, there are many valid schedules. This means that any (heuristic) search strategy for such schedules quickly encounters a valid solution if any exist.

We have implemented two different algorithms for LOAG scheduling: a backtracking algorithm and an algorithm using SAT-solvers. We shortly explain the backtracking algorithm in Section 3.4.1, but we leave out the details as the SAT-based algorithm is what we finally adopt, as explained in Section 3.4.2 and further. Note that the result of both of these approaches is a visit interface for each nonterminal, such that the runtime evaluators that are generated are the same as those of the OAG algorithm. The only difference is in the construction of the visit interfaces.

### 3.4.1 Backtracking algorithm

The backtracking algorithm is extension of the OAG algorithm and is described in more detail in [Van Binsbergen et al., 2015]. The algorithm proceeds by running the OAG algorithm and adding augmenting dependencies when cycles are encountered in the extended dependency graph. The augmenting dependencies are chosen from all edges in the cycle that are not in the induced dependency graph, such that one edge in the cycle is reversed and this edge does not lead to a cycle in the induced dependency graph directly. As a result of this other cycles can be induced, and the algorithm therefore recursively adds augmenting edges until a schedule is found.

Picking the (correct) augmenting edges is a combinatorial problem, however, and the algorithm may make wrong choices and end up in a situation where no possible augmenting edges can be added anymore. That is where the backtracking comes in; the algorithm backtracks one of the choices and picks another edge to resolve that cycle. In the worst case the algorithm makes the wrong choices for every edge, leading to exponential running time.

Experiments on the UHC and other AG projects however resulted in an interesting observation: on all real-world AGs the algorithm never needs a backtracking step and all its initial choices lead to a valid schedule. The algorithm therefore works well in practice and has running time similar to the OAG algorithm, but does not need any manual augmenting dependencies to be added to the source code!

### 3.4.2 SAT algorithm

The solution to the LOAG scheduling problem we finally adopt is the translation to the Boolean satisfiability problem (SAT), which is the standard NP-complete problem [Cook, 1971]. Even though the worst case runtime of all known SAT-solving algorithms is exponential in the input size, many SAT-solvers work very well in practice [Claessen et al., 2009]. By translating into the SAT problem we can therefore use an efficient existing SAT-solver to solve our problem and even benefit from future improvements in the SAT-community. In our implementation we use MiniSat[3] [Eén and Sörensson, 2004].

To encode this problem in SAT we represent each edge in the dependency graphs as a Boolean variable, with its value indicating the direction of the edge. For the direct dependencies the value is already set, but for the rest of the variables the SAT-solver may choose the direction of the edge. Ensuring cycle-freeness requires us to encode transitivity with SAT-constraints, which in the straight-forward solution

[3]`http://minisat.se`

leads to a number of extra constraints cubic in the number of variables. To avoid that problem we make our graphs *chordal* [Dirac, 1961]. In a chordal graph every cycle of size $> 3$ contains an edge between two non-adjacent nodes in the cycle. In other words, if there exists a cycle in the graph, there must also exist a cycle of at most three nodes. This allows us to encode cycle freeness much more efficiently by only disallowing cycles of length three. Chordality has been used previously to encode equality logic in SAT using undirected graphs [Bryant and Velev, 2002]; to our knowledge this is the first application of chordality to express cycle-freeness of directed graphs.

Apart from the fact that this translation into the SAT problem helps in efficiently (in practice) solving the scheduling problem, there also is another benefit: it is now possible to encode extra constraints on the resulting schedule. We show two scheduling optimizations that are interesting from an attribute grammar point of view, for which the optimal schedule can be found efficiently by expressing the optimisation in the SAT problem.

### 3.4.3 Translation into SAT

To represent the scheduling problem as a Boolean formula we introduce a variable for each edge, indicating the direction of the edge. The direct dependencies coming from the source code are constants, but for the rest of the edges the SAT-solver can decide on the direction. However, the encoding has been chosen in such way that a valid assignment of the variables corresponds to a valid schedule.

Our algorithm has the following steps:

1. Construct a $NDG_N$ for each nonterminal $N$ and add an edge between all pairs of inherited and synthesized attributes
2. Construct a $PDG_p$ for each production $p$ and add an edge for every direct dependency
3. Make all graphs chordal
4. Introduce a SAT variable for each edge in any of the graphs including the chords added in step 3. Variables of edges between attributes of the same nonterminal must be shared between nonterminal dependency graphs and production dependency graphs
5. Set the value of all variables corresponding to direct dependencies
6. Exclude all cycles of length three by adding constraints

7. Optionally add extra constraints for optimizations

The first two steps have been explained in the previous sections. Step 3 is explained below, step 4 is trivial, and step 5 and 6 follow from the explanation below. Finally, step 7 is explained in Section 3.4.5.

### Chordal graphs

A chordal graph is an undirected graph in which each cycle of length $> 3$ contains a *chord*. A chord is an edge between two nodes in the cycle that are not adjacent. As a consequence each cycle of length $> 3$ can be split up into two smaller cycles, which implies that if a chordal graph contains a cycle it must also contain a cycle of size three. Chordal graphs are therefore sometimes also referred to as *triangulated graphs*.

In our case the graphs are directed, but we can still apply the same trick! Imagine a graph with a cycle of length four: $a \to b \to c \to d \to a$. Because the graph is chordal there must also be an edge between $a$ and $c$ (or $b$ and $d$, but then the following argument is similar). Regardless of the direction of the arrow between $a$ and $c$ there always exists a cycle of length three as well: either $a \to b \to c \to a$ or $a \to c \to d \to a$. Hence, if we make our graph chordal by adding edges which may have an arbitrary direction and explicitly exclude all cycles of length three, we ensure that no cycles can exist at all.

### Chordal graph construction

There are several algorithms for making a graph chordal. We use an algorithm based on the following alternative definition of a chordal graph:

**Definition 1.** *An undirected graph is* chordal *if and only if it has a perfect elimination order. A perfect elimination order is an ordering $v_1, \dots, v_n$ of the vertices of $G$ such that in the graph $G[v_1, \dots, v_i]$, $\forall i (1 \leqslant i \leqslant n)$, the vertex $v_i$ is simplicial. A vertex $v$ is called simplicial in a graph $G$ if the neighbourhood of $v$ forms a connected component in $G$. The graph $G[v_1, \dots, v_i]$ is the induced subgraph of $G$ containing only the vertices $v_i, \dots, v_i$ and the edges between these vertices.*

From this definition we can construct the following algorithm for making a graph chordal:

1. While the graph still contains vertices:

    (a) Select a vertex $v$ from the graph

(b) For every pair $(a, b)$ of unconnected vertices in the neighbourhood of $v$:
    i. Add the edge $(a \longleftrightarrow b)$ to the graph

(c) Remove $v$, and all edges connected to $v$, from the graph.

One important open question in this algorithm is the order in which the vertices should be chosen. In Section 3.4.4 we show the results for several heuristics that we have implemented and tried on the UHC. We would like to remark that regardless of the heuristic used, this approach always leads to much smaller SAT problems than encoding transitivity in the SAT problem for ruling out cycles.

#### Finding the schedule

When the constructed Boolean formula is given to the SAT-solver, the result is either that the formula is not satisfiable, meaning that no schedule has been found, of satisfiable, meaning that there is a schedule. It is not hard to see that the formula is satisfiable if and only if there exists a valid schedule for the given attribute grammar definition, and in this thesis we therefore give no formal proof of this claim.

In the case where the formula is satisfiable, we obviously want to find the result. From the SAT solver we can ask for the truth value of each variable in the solution. When our algorithm keeps the connection between edges and variables we can then complete our directed graph and trivially find the complete order for all attributes from that. The constraints guarantee that this graph contains no cycles.

#### Shared edges

One important implementation detail is that of shared edges. As explained, the nonterminal dependency graphs and the production dependency graphs share the edges that define the order of the attributes. Because each edge is represented by a variable in the SAT problem we can simply encode this by assigning the same variable to the shared edges.

However, as we also make both graphs chordal, the edges added to make the graphs chordal can also be shared. This is exactly what our implementation does, such that the SAT problem is kept as small as possible. The implementation is therefore slightly more complicated than explained in the previous sections.

### 3.4.4 Chordal graph heuristics

As explained in Section 3.4.3 we need to find an order in which to handle the vertices such that the resulting SAT problem is as small as possible. In Table 3.1 we

| Order | #Clauses | #Vars | Ratio |
|---|---|---|---|
| $(\lvert\mathscr{D}\rvert,\lvert\mathscr{S}\rvert,\lvert\mathscr{C}\rvert)$ | 21,307,812 | 374,792 | 57.85 |
| $(\lvert\mathscr{D}\rvert,\lvert\mathscr{C}\rvert,\lvert\mathscr{S}\rvert)$ | 8,301,557 | 220,690 | 37.62 |
| $(\lvert\mathscr{S}\rvert,\lvert\mathscr{D}\rvert,\lvert\mathscr{C}\rvert)$ | 12,477,519 | 287,151 | 43.45 |
| $(\lvert\mathscr{S}\rvert,\lvert\mathscr{C}\rvert,\lvert\mathscr{D}\rvert)$ | 8,910,379 | 241,853 | 36.84 |
| $(\lvert\mathscr{C}\rvert,\lvert\mathscr{D}\rvert,\lvert\mathscr{S}\rvert)$ | 3,004,705 | 137,277 | 21.89 |
| $(\lvert\mathscr{C}\rvert,\lvert\mathscr{S}\rvert,\lvert\mathscr{D}\rvert)$ | 3,359,910 | 156,795 | 21.43 |
| $(\lvert\mathscr{D}\rvert+\lvert\mathscr{S}\rvert,\lvert\mathscr{C}\rvert)$ | 12,424,635 | 386,323 | 32.16 |
| $(\lvert\mathscr{D}\rvert,\lvert\mathscr{S}\rvert+\lvert\mathscr{C}\rvert)$ | 8,244,600 | 219,869 | 37.50 |
| $(\lvert\mathscr{D}\rvert+\lvert\mathscr{C}\rvert,\lvert\mathscr{S}\rvert)$ | 2,930,922 | 135,654 | 21.61 |
| $(\lvert\mathscr{S}\rvert,\lvert\mathscr{D}\rvert+\lvert\mathscr{C}\rvert)$ | 8,574,307 | 236,348 | 36.28 |
| $(\lvert\mathscr{S}\rvert+\lvert\mathscr{C}\rvert,\lvert\mathscr{D}\rvert)$ | 3,480,866 | 157,089 | 22.16 |
| $(\lvert\mathscr{C}\rvert,\lvert\mathscr{D}\rvert+\lvert\mathscr{S}\rvert)$ | 3,392,930 | 157,568 | 21.53 |
| $(\lvert\mathscr{C}\rvert+\lvert\mathscr{D}\rvert+\lvert\mathscr{S}\rvert)$ | 3,424,001 | 148,724 | 23.02 |
| $(3*\lvert\mathscr{S}\rvert*(\lvert\mathscr{D}\rvert+\lvert\mathscr{C}\rvert)+(\lvert\mathscr{D}\rvert*\lvert\mathscr{C}\rvert)^2)$ | $\underline{2,679,772}$ | $\underline{127,768}$ | $\underline{20.97}$ |

Table 3.1: Table showing the number of clauses and variables required for solving the MainAG of the UHC, selecting the next vertex in the elimination order based on different ways to compare neighbourhoods.

show the results of different heuristics for the MainAG of the UHC. In this table we use three different sets: $\mathscr{D}$ is the set of direct dependencies (step 2, Section 3.4.3), $\mathscr{C}$ is the set of edges that are added to make the graph chordal (step 3, Section 3.4.3) and $\mathscr{S}$ is the set of edge between all inherited and synthesized pairs (step 1, Section 3.4.3). For each of the sets we take only the edges in the neighbourhood of $v$ for comparison.

## 3.4.5 Optimisations

Expressing the scheduling problem as a SAT problem and using an existing SAT-solver can improve the running time of the scheduling, but that is not the only advantage. In the SAT problem one can easily add extra constraints to further influence the resulting schedule. In this section we show two of such optimisations that are useful from an attribute grammar perspective. These optimisations have not been implemented in the release version of the UUAGC, but we have run preliminary experiments to verify that they work as expected.

### Interacting with the solver

Instead of directly expressing all constraints in the initial SAT problem, we use a different trick for implementing the two optimisations: interacting with the solver. After the initial scheduling problem has been solved, we can ask for the truth value of all variables to construct the schedule. MiniSat also keeps some state in memory that allows us to add extra constraints to the problem and ask for a new solution. In this way we can start with an initial solution and interact with the solver until some optimum has been reached.

### Minimising visits

The result of the static scheduling is a runtime evaluator that computes the values of the attributes for a given abstract syntax tree. The total order for each nonterminal defines in what order attributes should be computed, but in the implementation of the evaluator we make use of a slightly bigger unit of computation: a visit.

Because invoking a visit at runtime may have a certain overhead, we would like the number of visits to be as small as possible. In other words, in the total order on the attributes we would like to minimise the number of places where a synthesized attribute is followed by an inherited attribute, because that is the location where a new visit needs to be performed.

It is theoretically impossible to minimise the total number of visits performed for the full abstract syntax tree, because at compile-time we do not have a concrete abstract syntax tree at hand and only know about the grammar. We therefore try to minimise the maximum number of visits for any nonterminal, which is the number of alternating pairs of inherited and synthesized attributes in the total order.

In our algorithm, we use efficient counting constraints, expressed in the SAT solver using sorting networks [Codish and Zazon-Ivry, 2010]. This enables us to count the number of true literals in a given set of literals and express constraints about this number. A standard procedure for finding a solution for which the minimal number of literals in such a set is true can be implemented on top of a SAT-solver using a simple loop.

We use the following algorithm for minimising the maximal number of visits:

1. Construct the initial SAT problem and solve
2. Construct the set of all production rules $P$
3. Construct counting networks that count the number of visits $V(p)$ for all production rules $p$ in $P$

4. Count the number of visits for each production rule in the current solution; let $M$ be the maximum value
5. Repeat while $M > 0$:
    (a) Add constraints that express that for all productions $p$ in $P$: $V(p) \leqslant M$
    (b) Construct a counting network that counts how many production rules $p$ have $V(p) = M$
    (c) Compute a solution for which this number is minimised using the loop described above
    (d) Remove all $p$ in $P$ for which now $V(p) = M$ from the set $P$
    (e) Compute the new maximum value $M$ of $V(p)$ for all $p$ left in $P$

The above algorithm features a complicated combination of counting networks; one network for each production rule and one network for each corresponding output of these networks. Still, the procedure finds optimal solutions quickly in practice, in times that are negligible and not practically measurable compared to the time of generating the initial SAT-problem. The number of iterations for the minimisation loops has never been more than 5 in any of our problems.

The algorithm is guaranteed to find the global optimum. For our largest example, the solution found had a total of 130 visits, which was 29 visits less in total than the previously known optimum, found using backtracking heuristics.

One could criticise the usefulness of this particular optimisation for attribute grammars. Indeed, details on how one should optimise the number of visits depend on the kind of trees we are going to run the compiled code on. Our point is that we can easily express variants of optimisations. For example, we can also minimise the sum of all visits using a similar (but simpler) procedure to the one above. Again, the running time of that procedure is short.

### Eager attributes

Another optimisation is the ability to define *eager attributes*. Eager attributes are attributes that should be computed as soon as possible, and they must be annotated by the attribute grammar programmer as such. We would like our scheduling algorithm then to schedule them as early as possible in the total order.

As an example, in a typical compiler there is an attribute containing the errors that occur in compilation. When running the compiler one is typically first interested in knowing if there are any errors; if so they must be printed to the screen and the compiler can stop its compilation. If there are no errors, then all other work that is not strictly necessary for the generation of errors can be done to complete the compilation.

In order to schedule a given attribute as early as possible, we are going to partition all attributes contained in the grammar into two sets $E$ (for early) and $L$ (for late). The idea is that $E$ contains all attributes that may be needed to be computed before the eager attribute (i.e. there exist production rules which require this), and $L$ contains all attributes that we can definitely compute after knowing the eager attribute (i.e. no production rule requires any attribute in $L$ for computing the eager attribute). We want to find a schedule for which the size of $E$ is minimal.

To compute this, we introduce a SAT variable $E(a)$ for every attribute $a$, that expresses whether or not $a$ is in $E$ or not. We set $E(a)$ to be true for the initial eager attribute $a$. We go over the graphs for the nonterminals and production rules and generate constraints that express that whenever $a$ points to $b$ and we have $E(b)$, then we also need $E(a)$.

Finally, we use a counting network for all literals $E(a)$ and ask for a solution that minimises the number of literals in $E$.

We have run this algorithm on ever output attribute of the top-level nonterminal of all our examples. For our largest grammar, the hardest output to compute took 1 second. So, while a harder optimisation than the previous one, it is doable in practice.

### 3.4.6 Code generation

The code generation for this algorithm is the same as for the OAG algorithm (Section 3.2.5), as the result of the LOAG algorithm is also a global order on the attributes of each nonterminal. For the *label* and *vals* example this results in two visits that can also be constructed by adding augmenting dependencies to the OAG algorithm.

## 3.5 Runtime comparison

In order to give an indication of the differences in runtime for the scheduling algorithms we have run them on several AG projects of which we could obtain the source code. The Utrecht Haskell Compiler (UHC) [Dijkstra et al., 2009] is the first case of which the source code consists of several attribute grammar definitions together with Haskell code. The biggest attribute grammar in the UHC, called *MainAG*, consists of 30 nonterminals, 134 productions, 1332 attributes (44.4 per nonterminal) and 9766 dependencies and is the biggest attribute grammar we know of.

| Algorithm | OAG | K&W | LOAG-bt | LOAG-SAT |
|---|---|---|---|---|
| UHC MainAG | - | 33s | 13s | 9s |
| Asil Test | - | 1.8s | 4.4s | 3.4s |
| Asil ByteCode | - | 0.6s | 29.4s | 2.8s |
| Asil PrettyTree | - | 390ms | 536ms | 585ms |
| Asil InsertLabels | - | 314ms | 440ms | 452ms |
| UUAGC CodeGeneration | - | 348ms | 580ms | 382ms |
| Pigeonhole principle | - | 107ms | 1970ms | 191ms |
| Helium TS_Analyse | 190ms | 226ms | 235ms | 278ms |

Table 3.2: Comparison of the four scheduling algorithms

The other test cases we have used for testing are the UUAGC itself, the *Asil* [Middelkoop et al., 2012] tool which is a byte code instrumenter, the *Helium* compiler [Heeren et al., 2003] which is a Haskell compiler specifically intended for students learning Haskell, and an encoding of the Pigeonhole principle. The Pigeonhole principle is a SAT formula which is unsolvable, but we removed one clause such that there exists exactly one valid schedule, resulting in an artificial attribute grammar that is hard to schedule.

In Table 3.2 we show the compilation times for the examples for the four different algorithms. All times include parsing of the attribute grammar description and code generation. In case of the SAT approach adding chords takes most time, while the SAT-solver takes less than a second to find a solution in all cases.

## 3.6 Conclusion

In this chapter we have shown several classes of AGs with respect to finding an evaluation order statically. We have illustrated the difficulties in scheduling and shown several algorithms that statically find a schedule for the AGs in those classes. Furthermore, we have illustrated how we generate runtime evaluators for some of these classes, which are extended in later chapters to support incrementality. The LOAG algorithm from Section 3.4 using SAT-solvers is what we finally use in the remainder of this thesis, and to simplify matters we assume that all AGs fall in the LOAG class. This is not technically true, but as we have not encountered AGs outside of this class our methods are widely applicable.

# 4 Tree Transformations

The incremental evaluation of AGs as described in this thesis handles changes to the input incrementally. However, in order for such an approach to work it is necessary to know exactly in which way the input may change. We tackle this problem of representing transformations on values in this chapter. We look at the problem from a datatype-generic perspective [Gibbons, 2007], such that our description of transformations is strongly-typed and applicable to a large class of data types.

Note that we are not exclusively interested in computing a *difference* between two terms, as [Lempsink et al., 2009] did. Instead, we focus on the more general notion of encoding *transformations* as they happen (as captured, for example, by a graphical user interface such as a structure editor) and representing these transformations in a way that minimises the duplication of data. Avoiding duplication of data is not only useful for the efficient representation of changes, but essential for the effectiveness of the incremental evaluation of AGs as explained later in this thesis. The code from this chapter is available on Hackage[1].

This chapter is organised as follows. We first illustrate the need for better representations of transformations in Section 4.1. In Section 4.2 we illustrate our solution with the guestbook example, and we extend our solution to a more complex solu-

[1] `http://hackage.haskell.org/package/transformations`

tion for the C♯ case in Section 4.3. For readers who are interested in the main story line of this thesis, the above preliminary sections contain sufficient information for understanding the subsequent chapters of this thesis.

We continue with a generic programming approach to the problem by introducing generic programming concepts in Section 4.4. In Section 4.5 we introduce zippers and paths and we describe the generic encoding of the guestbook example in Section 4.6. The generic programming approach is continued by extending it to families of data types like the C♯ example in Section 4.7. Finally, in Section 4.8 we wrap up with some discussion and conclusion.

## 4.1 Transformation operations

In this section we show a number of transformations that we want to express. We use the following data type for representing example transformations on expressions.

```
data Expr = Var    String
          | Const  Int
          | Neg    Expr
          | Add    Expr Expr
```

An expression is either a named variable, an integer constant, the negation of an expression, or the addition of two expressions. The following are sample expressions.

```
expr1, expr2, expr3 :: Expr
expr1 = Add (Const 1) (Var "a")
expr2 = Add (Const 1) (Neg (Var "a"))
expr3 = Add (Var "a") (Const 1)
```

We use these sample expressions to illustrate the type of changes we like to represent.

**Insertion** An insertion can be seen as a transformation that extends a value with one or more extra constructors. Consider the following transformation:

$Var$ "a" $\rightsquigarrow$ $Neg$ ($Var$ "a")

The arrow "$\rightsquigarrow$" is used to indicate a transformation, with the old term on the left and the new term on the right. The right-hand side of this transformation can be seen as arising from completing the expression *Neg* _ with reusing the original expression *Var* `"a"`, where the underscore indicates a hole in an expression. Our example $expr_1$ can be transformed into $expr_2$ using such an insertion, which happens in the context of the second subtree of the full expression.

**Deletion** A deletion removes part of an expression. For example, $expr_1$ can be seen as arising from the deletion of the *Neg* constructor in $expr_2$. In reality, however, we see deletion as a form of replacement: $expr_1$ arises by replacing the *Neg x* expression by *x* in $expr_2$. We do not consider the deletion of a full subtree as a valid transformation because, in general, it results in ill-typed expressions. Instead, with deletion we means the replacement of a subtree by a smaller subtree.

**Swap** Insertion and deletion are edit operations considered by general tree difference algorithms, such as that of [Lempsink et al., 2009]. Consider now the transformation from $expr_1$ to $expr_3$, which has the following shape:

$$Add\ a\ b \rightsquigarrow Add\ b\ a$$

It is possible to encode this transformation with insertion and deletion operations alone. However, such an approach has two drawbacks. First, it is verbose, requiring both a deletion and an insertion. Moreover, it does not adequately encode the fact that the subexpressions *a* and *b* remain unchanged through the transformation and do re-appear in the result. This problem is particularly relevant when the subexpressions being swapped are large, and the attributes that have already been computed do not change due to the swap.

**Rotation** A rotation transformation involves rearranging the nesting structure of a tree. A common example is reassociating binary operators:

$$Add\ a\ (Add\ b\ c) \rightsquigarrow Add\ (Add\ a\ b)\ c$$

In rotations, we want to keep track of the fact that some subexpressions (in our example, *a*, *b*, and *c*) remain unchanged and are just rearranged in their ancestors. Although swap also falls under this definition of rotations, we mention it separately because it is used as an example throughout this chapter.

**Duplication** Duplication is a transformation typically arising from a copy-paste operation in an editor. For example:

$$\textit{Add}\ a\ (\textit{Const}\ 0) \rightsquigarrow \textit{Add}\ a\ a$$

In this transformation, the subexpression $a$ has been duplicated. This is not the same as just inserting $a$, as we want to remember that the inserted subexpression is not new, but just a copy of something already existing. The representation of duplication is particularly interesting in the context of incremental evaluation, where values computed over $a$ may be preserved after a copy-paste operation due to the information of the two subtrees being identical.

### 4.1.1 Localisation

A concept that is relevant to all types of transformation is that of *localisation*. Consider the following transformation:

$$\begin{aligned} &\textit{Neg}\ (\textit{Neg}\ (\textit{Neg}\ (\textit{Neg}\ (\textit{Neg}\ (\textit{Neg}\ (\textit{Neg}\ (\textit{Const}\ 1))))))) \\ \rightsquigarrow\ &\textit{Neg}\ (\textit{Neg}\ (\textit{Neg}\ (\textit{Neg}\ (\textit{Neg}\ (\textit{Neg}\ (\textit{Neg}\ (\textit{Const}\ 2))))))) \end{aligned}$$

A compact representation of this transformation should not mention the entire spine of *Neg* constructors. Instead, changes must be represented in a local way focusing on a part of the tree only, which in this case is only the *Int* value.

### 4.1.2 Diff is not enough

We argue here that the existing solutions are not sufficient for expressing the desired sharing between the source and target of a transformation. The "standard" way of tracking changes between values is to use a diff algorithm. The diff algorithm by [Lempsink et al., 2009] is a type-safe, datatype-generic diff that may be used to determine changes in terms: given a transformation $t_1 \rightsquigarrow t_2$, *diff* $t_1$ $t_2$ returns an *edit script* describing how to transform $t_1$ into $t_2$. An associated *patch* operation can be used to apply an edit script to a term, obtaining a transformed term.

However, standard edit scripts only contain copy, insert, and delete operations. While these suffice to describe every transformation, the resulting edit description is often not faithful to the actual change that occurred. This is easily seen in a swap transformation, which, in an edit script, is represented by deletion and insertion. As an example, the edit script resulting from computing the difference between *Add* (*Var* `"a"`) (*Var* `"b"`) and *Add* (*Var* `"b"`) (*Var* `"a"`) is the following.

*Cpy Add* \$ *Cpy Var* \$ *Ins* `"b"` \$ *Ins Var* \$
*Cpy* `"a"` \$ *Del Var* \$ *Del* `"b"` \$ *End*

To apply the edit script a pre-order traversal of the source expression is performed while the target expression is built in a pre-order way as well. We start with the *Add* constructor which can be copied, as well as the *Var* constructor in the left child. Now, the source expression contains the value `"a"` while the target has value `"b"`, and thus `"b"` is inserted. This completes the left child of the target, so for the right child the *Var* constructor is inserted, after which the `"a"` of the source can be copied. Now the *Var* and `"b"` values of the source that are not used are deleted and the edit is complete.

There are multiple edit scripts that transform the source expression into this target, but none of them keeps track of the fact that the inserted expressions are not “new”, losing adequate sharing between transformations. We could extend existing diff algorithms with a swapping operation, but this is not enough to capture rotation, or duplication. Trying to add new edit operations to capture each different transformation we can think of is tiresome, and we have no guarantee that we covered all possible transformations. As such, we instead try to take a more general approach in describing transformations, being as abstract as possible as to what type of transformations are allowed, but making sure that sharing of subexpressions is made explicit, with minimal duplication of information.

## 4.2 Guestbook example

Before we explain the generic approach to this problem we illustrate the representation with the guestbook example. Remember that the guestbook is essentially a list with a constructor for an empty guestbook (*Empty*), and two different constructors for an entry (*Arrive* and *Leave*). As the guestbook has a list-like structure the type of transformations is somewhat simpler than for most tree-structures, which makes the representation easier to explain.

### 4.2.1 Paths

The first part of the representation enables us to represent locations in the tree. For this we use their path from the root node with a data type named *Path* as follows.

```
data Path = End
          | Arrive_tl Path
          | Leave_tl  Path
```

The *End* constructor indicates the end of the path, while the other two constructors indicate that we are at an entry (*Arrive* or *Leave*) and take a step into the remainder of the list, for which another part of the path is given. In essence this data type encodes the index of the entry in the list to which the path points.

In this case the last two constructors might be combined into a single constructor. As a path points to a node in a given tree, it is always known whether there is an *Arrive* or *Leave* constructor at that position. In the generic setting explained later, however, it is necessary that these are different and we therefore use this definition.

### 4.2.2 Guestbook values with references

We represent a change to the tree by a path describing the location of the root of the change, called the subtree, and a new tree that is to be inserted in that location. Instead of always replacing the subtree with a full new tree, we allow the new tree to contain references to locations in the original tree, thus making it possible to reuse multiple parts of the original tree. In this way an insertion of an entry *Arrive* "a" into the guestbook can be modelled by a path *p* and a *Arrive* "a" (*Ref p*), indicating that the value at location *p* should be replaced by the *Arrive* constructor with a child "a" and as its tail the value that was originally located at *p*.

The representation of such trees with references for our guestbook is the following.

```
data GuestbookR = EmptyR
                | ArriveR Name GuestbookR
                | LeaveR  Name Double String GuestbookR
                | Ref     Path
```

The data type for trees with references is an extended version of the data type it expresses paths into, with a *Ref* constructor added, which holds a *reference* to a part of the tree that is to be reused.

At this point it is important to notice that paths are absolute and therefore relative to the root of the tree being edited. At first glance this may seem to lead to extra overhead because a local change somewhere deep in the tree may thus include multiple paths that share a long common prefix. In order to compactify the representation of different paths with the same prefix we need to introduce some sort of variable bindings, expressing the shared prefix, and pass around an environment in all places where the paths are used, which highly complicates matters. For the incremental attribute grammar evaluation machinery information about shared prefixes does not lead to more efficient evaluation, and we thus only solve the problem of representing changes as precisely as possible here.

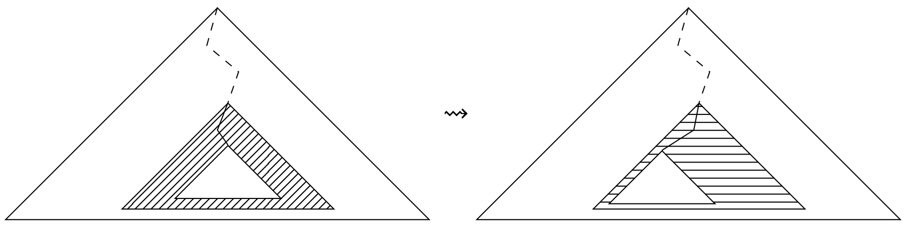

Figure 4.1: Illustration of a tree transformation with the changed part of the tree being shaded.

### 4.2.3 Trees with references

In Figure 4.1 we illustrate insertion, deletion and replacements of nodes. The outermost triangle represents the full tree in which a transformation operation is performed. The innermost triangle represents a subtree that will be reused: we call this part $R$. In the left and right picture $R$ represent the same subtree. The shaded triangle represents the part of the tree that is changed. As the shaded triangle in the left picture does not necessarily correspond to the shaded part in the right picture, we call the left one $S_1$ and the right one $S_2$.

With this representation an insertion can be implemented by choosing $S_1$ to be $R$. This means that in the left picture there is no shaded part and the full subtree $S_1$ is reused. In $S_2$ the shaded part is the part which is inserted, of which $R$ is a child. In a similar way deletion can be modelled by choosing $S_2$ to be $R$. In that case there is no shaded part in the right picture and therefore all nodes in the shaded part in the left picture are removed and the full changed subtree is replaced by the smaller subtree $R$.

In general there can be multiple references; in other words there can be multiple inner triangles in the picture. For instance, when representing a swap operation two references are used. Furthermore, the references do not necessarily need to point into the shaded part of the tree; a subtree from anywhere in the tree can be reused.

### 4.2.4 Full change

Using the paths and replacement values, we can represent a change by tupling those as follows.

$$\textbf{type}\ \mathit{Change} = (\mathit{Path}, \mathit{GuestbookR})$$

The first element of the pair indicates the location at which the change occurs. The second element is the tree that is to be inserted at that location, which can contain references which need to be replaced by the parts from the original tree they refer to.

Finally, a transformation can consist of multiple *Change*s, and thus a full transformation is represented as follows.

```
type Transformation = [Change]
```

Let us illustrate the use of these data types for the guestbook with two examples.

**Insertion** To insert an arrival entry at the beginning of the guestbook, like was done in the introduction, we insert this element at the empty path (root of the tree) as follows.

```
insert :: Transformation
insert = [(End,ArriveR "Magalhães" (Ref End))]
```

The inserted tree is the arrive constructor with as its tail a reference to the tree at the path *End*, which is the root of the original tree. In other words, the full original tree is the tail of the new tree, and therefore the result of this transformation is that a new element has been inserted.

**Deletion** In a similar way we can represent a deletion from the guestbook. In order to delete the fourth entry[2] from the example guestbook (Figure 1.1) from the introduction, we need to insert a tree which consists only of a reference to the entry after the deleted one.

```
delete :: Transformation
delete = [(Leave_tl (Leave_tl (Arrive_tl End))
         ,Ref (Leave_tl (Leave_tl (Arrive_tl (Leave_tl End)))))]
```

## 4.3 C♯ example

The representation of the guestbook is not only simple because of the specific list-like structure, but also because it only contains a single data type. All types in

[2]Which is the "LEAVE Swierstra" entry, as the entries are ordered from newest to oldest

```
data Stat | StatDecl   decl  :: Decl
          | StatExpr   expr  :: Expr
          | StatIf     cond  :: Expr
                       true  :: Stat false :: Stat
data Expr | ExprConst const :: Const
          | ExprOper  op    :: String
                      left  :: Expr right :: Expr
```

Figure 4.2: Some of the data type definitions of the C♯ compiler shown again.

the representation are therefore the same. In general however this is not the case, and the generic approach has work for families of mutually recursive data types like the C♯ data types (Figure 2.2). In this section we explain how to implement this technique in a type-safe way illustrated with some of the data types of the C♯ compiler implementation.

### 4.3.1 Paths

For the paths we need to carry type information on the source and target of the path as we would like to represent paths through the different types of nodes in the AST within a single *Path* data type. We therefore change the path to a *Generalized Algebraic Data Type* [Cheney and Hinze, 2003, Xi et al., 2003], such that *Path f t* represents a path in a tree of type *f* (from) pointing to a node of type *t* (to). For a full path the *f* parameter equals the type of the root of the AST.

The constructors of this path data type are the *End* constructor for the empty path and a constructor for each nonterminal child for each production. For example, for the *CSharp* types from Figure 4.2 this results in the following path data type.

```
data Path f t where
  End            ::                   Path f    f
  StatDecl_decl  :: Path Decl  t → Path Stat t
  StatExpr_expr  :: Path Expr  t → Path Stat t
  StatIf_cond    :: Path Expr  t → Path Stat t
  StatIf_true    :: Path Stat  t → Path Stat t
  StatIf_false   :: Path Stat  t → Path Stat t
  ExprConst_const :: Path Const t → Path Expr t
  ExprOper_left  :: Path Expr  t → Path Expr t
```

```
ExprOper_right :: Path Expr  t → Path Expr t
...
```

For each of the constructors the argument is a path with *f* set to the type of the corresponding child, and it is polymorphic in *t*. As a return type the parameter *f* is the type of the constructor itself, and *t* is the same as the *t* of its child because that is the type of the node the path points to. For the *End* constructor these two parameters must be equal.

Note that the type parameter *f* does not only give extra information on the type level, but is also prevents some "wrong" types to be constructed. For example, the path *StatIf_cond* (*StatIf_cond End*) is not type correct because the *cond* child of the *StatIf* constructor has type *Expr*, and a *StatIf* can therefore never appear as first child of *StatIf*.

However, it is still possible to construct paths that are invalid, for instance by using the *StatIf_true* at a place where there is a *StatDecl* constructor. In order to prevent such paths to be constructed we need dependent types, because the exact set of valid paths for a given tree depends on that tree. As this is not easy to capture at the type level in Haskell we allow such possible types to be representable and rely on the generation algorithm not to construct incorrect paths.

### 4.3.2 Trees with references

The representation of the trees with references for multiple data types is similar to the single data type case. For each nonterminal we have a corresponding type of tree with references, and for each production *C* of that nonterminal we have a constructor *CR*. Furthermore, we have a *Ref* constructor for representing a *reference* containing a path.

Because the paths have two type parameters in the representation for multiple data types, we need to set the value of these type parameters in the *Ref* constructor where the path is an argument. The type of the second parameter *t* is set to the nonterminal for which this data type is generated, as the path should point to a node of that same type in order for it to be used in that position. The *f* parameter should be the type of the top level node of the tree, which we add as a type level argument *top* to the reference type to be instantiated at the invocation of the evaluation machinery. For the *Stat* nonterminal the data type is therefore the following.

```
data StatR top
   = Stat_Ref       (Path top Stat)
   | StatStatDeclR (DeclR top)
```

```
    | StatStatExprR (ExprR top)
    | StatStatIfR   (ExprR top) (StatR top) (StatR top)
      ...
```

The other constructors and other types are generated in the same way from their original counterparts by postfixing the names with *R*, adding a *Ref* constructor and propagating the *top* type.

### 4.3.3 Full change

Using the paths and trees with references, we can represent a change by a pair of those values as before. However, as the type of the replacement depends on the type of the node that the path points to, we can not directly specify this as a pair. Instead, we use a *type family* (or type level function) that maps the type *a* (for example *Stat*) to the corresponding type of trees with references (for example *StatR*).

```
type family   ReplType a    top :: *
type instance ReplType Stat top = StatR top
type instance ReplType Expr top = ExprR top
...
```

Using this type we can represent changes, which are also parametrized over the type of the top level node. We use a GADT to hide the type of the node at which the change takes place.

```
data Change top where
  Change :: Path top t → ReplType t top → Change top
```

Finally, a full transformation is again a list of such changes, parametrized over the type of the top level node.

```
type Transformation top = [Change top]
```

Note that different elements in this list can insert nodes of different types.

### 4.3.4 List support

The representation for changes we have described in the previous sections works for all regular data types including common types such as lists. In a typical attribute grammar there are many different lists, and generating paths and trees with references for each of them works as expected. This does however lead to redundant code as the generated data types for these lists with references are isomorphic as well as paths over them. A similar problem appears with other common container types likes *Maybe*.

To avoid redundancy we adopt UUAGC's special support for lists and extend it to trees with references. We add the following two generic constructors to the *Path* data type.

```
data Path f t where
  PathL_hd :: (Path a     t) → Path [ a ] t
  PathL_tl  :: (Path [ a ] t) → Path [ a ] t
    ...
```

The general representation of lists with references, with the parameter *a* instantiated to the type of the elements of the list and *ar* to the type of the trees with references of the elements of the list, is the following.

```
data ListR a ar top
   = List_Ref    (Path top [ a ])
   | ListConsR ar (ListR a ar top)
   | ListNilR
```

Note that the pattern used here for the type parameter *a* can be used for other data types with type parameters as well. For concrete lists we define a convenient type alias.

```
type StatLR top = ListR Stat StatR top
```

Although this type alias is not strictly necessary, it makes some code simpler as for each type we now have a corresponding type of trees with references.

This concludes the first part of this chapter, in which the representation of tree transformations as used in the rest of this thesis is explained. The given approach is based on extra data types being generated for representing paths and trees with references. The rest of this chapter describes an approach based on generic programming techniques for representing such data types generically for all families of mutually-recursive data types.

## 4.4 Generic programming for regular functors

To tackle the problem of representing transformations generically, we first introduce the generic programming library that we use for our solution. As we will see in the coming sections, our solution revolves around annotating recursive positions in data types. As such, a library with an explicit encoding of recursion (for example with a fixed-point view [Holdermans et al., 2006] on data) suits us best. We can either pick `regular` [Van Noort et al., 2008], a library which supports only regular data types, or `multirec` [Rodriguez Yakushev et al., 2009], a generalisation of `regular` that supports mutually-recursive families of data types. For presentation purposes, we use `regular`, as it is easier to understand our solutions in the single data type case. We have also written an implementation using `multirec`, which we describe in Section 4.7.

This section provides only a brief introduction to `regular` for educational reasons. For more details, the reader is referred to [Van Noort et al., 2008].

## 4.4.1 Representation

Data types are encoded in `regular` using the following five *representation types*:

```
data U         r = U
data I         r = I r
data K a       r = K a
data (f :+: g) r = L (f r) | R (g r)
data (f :×: g) r = f r :×: g r
```

Unit, encoded by *U*, is used for constructors without arguments. Recursive positions, encoded by *I*, denote occurrences of the data type being defined. Constants, encoded by *K*, are used for all other constructor arguments. Sums, encoded by :+:, are used to denote choice between constructors, while products, encoded by :×:, are used for constructors with multiple arguments. The `regular` library also contains representation types for dealing with data type meta-information such as constructor and selector names, but we elide those from our presentation as they are not necessary.

As an example, the *Guestbook* data type is encoded in `regular` as follows:

```
type GB_PF =   U
         :+: (K Name :×: I)
         :+: (K Name :×: K Double :×: K String :×: I)
```

Note that $GB_{PF}$ (of kind $* \to *$) encodes the *pattern functor* of *Guestbook*, also known as its *open* version. To obtain *Guestbook*, we need to “close” $GB_{PF}$, replacing the recursive positions under *I* with $GB_{PF}$ again. This can be done using a type-level fixed-point operator:

```
data μf = In (f (μf))
```

Now, $\mu\, GB_{PF}$ is a data type that is isomorphic to *Guestbook*.

## 4.4.2 Functoriality of the representation types

The `regular` library encodes data types as *functors*; the recursive positions are abstracted into a parameter *r*. As such, we can provide *Functor* instances for the representation types. These are unsurprising, with the action being transported across sums and products, ignored in units and constants, and applied at the recursive positions:

```
instance Functor U where
  fmap _ U = U
instance Functor (K a) where
  fmap _ (K x) = K x
instance Functor I where
  fmap f (I r) = I (f r)
instance (Functor f, Functor g) ⇒ Functor (f :+: g) where
  fmap f (L x) = L (fmap f x)
  fmap f (R x) = R (fmap f x)
instance (Functor f, Functor g) ⇒ Functor (f :×: g) where
  fmap f (x :×: y) = fmap f x :×: fmap f y
```

This functoriality is used to define catamorphisms over the representation types.

## 4.4.3 Embedding user-defined types

To provide a convenient interface for generic functions, `regular` uses a type class to aggregate generic representations of user data types. This class defines how to represent each data type and how to convert to and from its representation:

```
class Regular a where
  type PF a :: * → *
  from :: a → PF a a
  to   :: PF a a → a
```

The type family *PF* encodes the pattern functor of the data type being represented. The conversion functions *from* and *to* do not operate on “fully generic” representations of type $\mu\,(PF\ a)$. Instead, *from* and *to* operate on representations that are generic on the top level, and all recursive positions contain values of the original data type *a*. This choice enables inlining optimizations and thus leads to more efficient code [Magalhães, 2013].

We can now complete our encoding of *Guestbook* in `regular`:

```
type instance PF Guestbook = GB_PF
instance Regular Guestbook where
  from  Empty           = L U
  from (Arrive n t)     = R (L (K n :×: I t))
  from (Leave  n d s t) = R (R (K n :×: K d :×: K s :×: I t))
  to (L U)                                  = Empty
  to (R (L (K n :×: I t)))                  = Arrive n t
  to (R (R (K n :×: K d :×: K s :×: I t))) = Leave n d s t
```

Instances of the *Regular* class are tedious to write by hand; fortunately, the `regular` library includes Template Haskell code to automatically generate these instances for user data types.

### 4.4.4 Generic functions

We can now define generic functions by giving a case for each representation type. We use a type class for this purpose, followed by five instances. As an example we show a generic function that lists all the immediate children of a given term in Figure 4.3.

The function *gchildren* operates on generic representations. We also define the function *children* which operates directly on user data types, by first converting them to generic representations:

```
children :: (Regular a, Children (PF a)) ⇒ a → [a]
children = gchildren ∘ from
```

The *to* function is not used here because *from* leaves the recursive positions unchanged and the children are thus already of type *a*.

## 4.5 Zippers and paths

The zipper is a data structure used to represent traversals in a term, which we use to represent paths. It is a type-indexed data type [Hinze et al., 2002]: every algebraic data type induces a zipper, generically. We provide a brief introduction to zippers

```
class Children f where
  gchildren :: f r → [r]
instance Children U where
  gchildren _ = [ ]
instance Children I where
  gchildren (I x) = [x]
instance Children (K a) where
  gchildren _ = [ ]
instance (Children f, Children g)
             ⇒ Children (f :+: g) where
  gchildren (L x) = gchildren x
  gchildren (R x) = gchildren x
instance (Children f, Children g)
             ⇒ Children (f :×: g) where
  gchildren (x :×: y) = gchildren x ++ gchildren y
```

Figure 4.3: Generic function for retrieving all children

in this section because they form a key part of our solution. A detailed description, however, is out of the scope of this thesis; [Rodriguez Yakushev et al., 2009], for example, describe a zipper for families of data types.

For now we focus on a zipper for regular functors. The zipper encodes a position of focus on a value, together with the surrounding context. These two elements are stored in the *Loc* data type:

```
data Loc a where
  Loc :: (Regular a) ⇒ a → [Ctx (PF a) a] → Loc a
```

A location is the point currently in focus in the zipper (of type *a*), and the path to the focal point. This path is stored as a stack of one-hole *contexts*. The context is given by the derivative of the pattern functor representing the data type [McBride, 2001]. This type-indexed data type is encoded in `regular` as a data family, indexed over the five representation types:

```
data family Ctx (f :: ∗ → ∗) :: ∗ → ∗
data instance Ctx U         r
data instance Ctx (K a)     r
data instance Ctx I         r = CId
data instance Ctx (f :+: g) r = CL  (Ctx f r)
                              | CR  (Ctx g r)
data instance Ctx (f :×: g) r = C₁  (Ctx f r) (g r)
                              | C₂  (f r)      (Ctx g r)
```

Units and constants contain no recursive positions and as such have an empty context. The *CId* constructor signals a recursive position. Sums can either have a context on the left (*CL*) or on the right (*CR*). For products, we can choose to traverse the first argument, keeping the second argument intact ($C_1$), or to do the opposite ($C_2$).

Zipper operations, such as navigation functions, can be defined over the type of contexts. However, we are only interested in the type of contexts to encode paths in data structures and ignore other useful applications like structure navigation. A context instantiated with units for the recursive positions (*r* set to ()) effectively encodes a direction of navigation on a data structure to one of its children [Gibbons, 2013]. Paths on a data type are then a list of such directions on the corresponding pattern functor:

```
type Dir  f = Ctx f ()
type Path a = [Dir (PF a)]
```

## 4.6 Generic representation of transformations

Given the representation for our examples and the encoding of data types in the `regular` library as well as the zippers, this section shows how to represent transformations generically.

### 4.6.1 Representation

The first part of the representation of transformations is the notion of paths in a tree. We represent paths using a zipper context as explained in Section 4.5. To represent trees with references we extend the pattern functor of a type $a$ to allow for references at recursive positions:

```
data WithRef a b = InR (PF a b)
                 | Ref (Path a)
```

This representation resembles the meta-variable extension for generic rewriting of [Van Noort et al., 2008], the difference being that we extend with *Path* instead of a meta-variable. The type $\mu\,(\mathit{WithRef}\ a)$ is isomorphic to the type $a$ extended with a *Ref* constructor and thus represents a full tree possibly containing multiple references.

A transformation is then a list of localised insertions of trees with references:

```
type Transformation a = [(Path a, μ (WithRef a))]
```

The *Path* describes the location of the insertion. Note that this *Path* describes a path in the intermediate state of the tree, after insertions earlier in the list have been applied. The *Path*s in the *Ref*s, however, describe a path in the original tree.

The actual representation of the insertion of the *Arrive* constructor as shown in Section 4.2.4 is as follows:

```
addArrive :: Transformation Guestbook
addArrive =
  [([ ], In (InR (R (L (K "Magalhães" :×: I (In (Ref [ ])))))))]
```

As we have shown in Section 4.4.3, the value *Arrive n t* is represented as *R* (*L* (*K n* :×: *I t*)). In the *addArrive* transformation these constructors are therefore used to build a value representing *Arrive* with a reference.

### 4.6.2 Applying transformations

To apply a transformation, the original tree is taken as a starting value, and the localised insertions are performed one by one to produce a resulting tree:

$$
\begin{aligned}
&\mathit{apply} :: \mathit{Editable}\ a \Rightarrow \mathit{Transformation}\ a \to a \to \mathit{Maybe}\ a\\
&\mathit{apply}\ e\ t = \mathit{foldM}\ (\lambda a\ (p,c) \to \mathit{mapP}\ (\mathit{flip}\ \mathit{lookupRefs}\ c)\ p\ a)\ t\ e
\end{aligned}
$$

The inserted value is constructed using function *lookupRefs*, which will be discussed shortly. The function *mapP* takes care of inserting the value at the correct position, as indicated by its path argument. The class constraint *Editable* is used as an alias for all necessary instances (for example *Regular*).

**Resolving references** We resolve references from a tree by replacing them with values that we look up from another tree:

$$
\begin{aligned}
&\mathit{lookupRefs} :: \mathit{Editable}\ a \Rightarrow a \to \mu\,(\mathit{WithRef}\ a) \to \mathit{Maybe}\ a\\
&\mathit{lookupRefs}\ r\ (\mathit{In}\ (\mathit{InR}\ a)) = \mathit{fmap}\ \mathit{to}\ (\mathit{fmapM}\ (\mathit{lookupRefs}\ r)\ a)\\
&\mathit{lookupRefs}\ r\ (\mathit{In}\ (\mathit{Ref}\ p)) \phantom{)} = \mathit{extract}\ p\ r
\end{aligned}
$$

This function simply recurses over the tree and uses *extract* to find the part of the tree that is to be reused.

**Extracting children** The *extract* function takes a path and the original tree and returns the subtree at that location. It uses the generic function *gextract*, which extracts a child given a *Dir*ection:

$$
\begin{aligned}
&\mathit{extract} :: (\mathit{Editable}\ a, \mathit{Monad}\ m) \Rightarrow \mathit{Path}\ a \to a \to m\ a\\
&\mathit{extract}\ [\,] \qquad\quad = \mathbf{return}\\
&\mathit{extract}\ (p : ps) = \mathit{gextract}\ (\mathit{extract}\ ps)\ p \circ \mathit{from}\\
&\mathbf{class}\ \mathit{Extract}\ f\ \mathbf{where}\\
&\quad \mathit{gextract} :: \mathit{Monad}\ m \Rightarrow (a \to m\ a) \to \mathit{Dir}\ f \to f\ a \to m\ a
\end{aligned}
$$

The instances of *Extract* are unsurprising and are therefore not shown here.

**Indexed mapping** To update the tree in *apply* we use a map function that restricts its application to a specific part of the tree:

$$
\begin{aligned}
&\mathit{mapP} :: (\mathit{MapP}\ (\mathit{PF}\ a), \mathit{Monad}\ m, \mathit{Regular}\ a) \Rightarrow\\
&\qquad\quad (a \to m\ a) \to \mathit{Path}\ a \to a \to m\ a
\end{aligned}
$$

```
mapP f []       = f
mapP f (p : ps) = liftM to ∘ gmapP (mapP f ps) p ∘ from

class MapP f where
  gmapP :: Monad m ⇒ (b → m b) → Dir f → f b → m (f b)
```

The instances of *MapP* pattern match simultaneously on the path and on the tree such that the given function is only applied at the recursive position to which the path points. Note that the recursion is solved in *mapP* such that *gmapP* takes a *Dir* as argument which describes a single "step" of the full path. The instance for the :×: is the following, the other instances are similar.

```
instance (MapP f, MapP g) ⇒ MapP (f :×: g) where
  gmapP f (C₁ p _) (x :×: y) = liftM₂ (:×:) (gmapP f p x) (return y)
  gmapP f (C₂ _ p) (x :×: y) = liftM₂ (:×:) (return x) (gmapP f p y)
```

## 4.6.3 Generic diff

We can now automatically generate a transformation from one tree into another (a *diff* operation). This $diff :: a \rightarrow a \rightarrow Transformation\ a$ should obey the following law:

$$\forall\ a, b.\ apply\ (diff\ a\ b)\ a \equiv Just\ b$$

For any given $a$ and $b$ there are many different ways to transform $a$ into $b$. For example, $b$ can be inserted directly at the top level, completely replacing $a$; this is a valid transformation, albeit unsatisfactory since all sharing is lost.

In this section we describe a *diff* function that uses maximal sharing, meaning that only values that are not present in $a$ are inserted into $b$. The algorithm recursively builds up a set of insertions that transform $a$ into $b$. As the *diff* function is relatively large, we present it in a step-wise way, "uncovering" parts of its definition as we describe each subcomponent.

Note that the algorithm we describe is not necessarily the best possible in terms of efficiency or usability. The main goal of this section is to illustrate how such an algorithm can be constructed, and to provide an example of how to use our representation of transformations.

**Overview** The algorithm works in a top-down way by traversing the origin and target trees from the root towards the children. At each node, the best set of insertions is chosen based on whether the current node matches the target tree, whether parts of the original tree can be reused, and based on the insertions for the children. We now describe each subcomponent of the algorithm.

**Existing children** To maximise sharing, existing parts of the tree should be used whenever possible. The following function gathers all subtrees together with their corresponding locations in the tree:

```
childPaths :: (Regular a, Children (PF a)) ⇒ a → [(a, Path a)]
childPaths a = (a, [ ]) : [(r, n : p) | (c, n) ← children (from a)
                                      , (r, p) ← childPaths c]
```

In the *diff* function we gather all these subtrees with paths in a list for the original tree:

```
diff :: ∀ a. Editable a ⇒ a → a → Transformation a
diff a b = ... where
  cps :: [(a, Path)]
  cps = childPaths a
```

**Base cases** The recursive function that constructs the insertions is called *build*. It takes three parameters: a *Bool* indicating whether the current tree has been inserted, the current tree $a'$, and the target tree $b'$. The base cases are implemented as follows:

```
diff a b = build False a b where
  ...
  build :: Bool → a → a → Transformation a
  build False a' b' | a' ≡ b' = [ ]
  build ins   a' b'           = case lookup b' cps of
                                  Just p  → [([ ], In (Ref p))]
                                  Nothing → ...
```

The trivial base case is when $a'$ and $b'$ are equal, and $a'$ has not just been inserted. In case $a'$ has been inserted, which can happen when the parent of $a'$ did not exist in the original tree, we continue the search for reuse.

The second base case is when $b'$ is present in the list of subtrees of $a$; in that case, we simply build a *Ref* containing the path to that subtree.

**Shallow equality** In our quest for reuse, we need to be able to check whether two trees are equal at least in their first constructor. For this we use *shallow equality*:

```
class SEq f where
  shallowEq :: f a → f a → Bool
```

The instances of this class are standard, except for the case of the recursive position where we always return *True*.

In case the roots of two trees are equal, they can be left unchanged and we can continue trying to unify their children. This is implemented in the *construct* function:

```
build ins a′ b′ = ... where
  construct :: Bool → a → Maybe (Transformation a)
  construct ins′ c =
    if shallowEq (from c) (from b′)
    then Just ∘ concat ∘ updateChildPaths $
           zipWith (build ins′) (children c) (children b′)
    else Nothing
```

This function returns a transformation containing the edits for the children, based on some current tree *c*. The function *updateChildPaths* extends the *Path*s for all edits with the current child indices.

**Reusing parts of the original tree** In case no subtree of the original tree can be directly reused as a replacement for the full subtree that is being constructed, we try to reuse only the top part of an existing subtree. Using the *construct* function, we recursively create a list of insertions that transforms this existing subtree into the target subtree:

```
build ins a′ b′ = ... where
  ...
  reuses :: Maybe (Transformation a)
  reuses = foldl best Nothing [ addRef p (construct False x)
                              | (x, p) ← childPaths ]
    where addRef p = fmap (([ ], In (Ref p)) :)
```

Since there might be several valid possibilities, we use a function *best* to pick the “best” transformation. The definition of what the best transformation is varies from application to application; in our implementation, we have chosen to return the transformation with the fewest insertions.

**Insertion** When no existing parts of the tree can be reused, we are forced to insert. This insertion is again a tree with references, and we thus recursively continue constructing insertions that reuse existing parts:

```
build ins a' b' = ... where
  ...
  insert :: Transformation a
  insert = ([ ], r') : e' where
    Just r  = construct True b'
    (r', e') = partialApply (withRef b') r
```

Function *withRef* lifts a regular tree to a tree with references (never introducing the *Ref* constructor). As insertion can never fail we do not return a *Maybe* here.

Initially, an insertion inserts the full target subtree. However, in order to maximise sharing, we recursively try to replace parts of this target subtree by parts coming from the original tree, using references. To make the inserted value as small as possible, we directly apply these insertions to the inserted tree using *partialApply*, thereby replacing parts of the inserted tree by references.

**Completing the diff** Finally, we combine the previous definitions to construct the return value for the diff. The preferred return value is the case where a value is reused, and only if no values can be reused is the insertion returned:

```
diff a b = build False a b where
  ...
  build ins a' b' =
    case lookup b' cps of
        ...
      Nothing → maybe insert id uses where
        uses :: Maybe (Transformation a)
        uses = if ins then reuses <|> construct ins a'
                      else  reuses `best` construct ins a'
```

We use the Swierstra dike [Löh and Magalhães, 2013] operator <|> as a left-biased choice for *Maybe* values.

**Efficiency** The diff algorithm as presented in this section has an exponential running time, which is not very useful in practice. However, because the arguments to *build* are always subtrees of *a* or *b* *memoization* can be used to store the results of *build* for repeated calls. If $a$ and $b$ both have at most $n$ nodes (and thus $n$ subtrees), then the running time of the algorithm with memoization becomes $\mathrm{O}(n^3)$. We have implemented the memoized variant of *diff*: it can be found in the companion Hackage package.

### 4.6.4 Improving the interface

A problem with the generic representation is that it is not convenient for manual use. The representation contains many constructors, and especially when the data type that is represented has many constructors, the number of $L$ and $R$ parameters is significant. In order to solve this problem we can make use of *pattern synonyms*, which give the possibility to name a more complex pattern, by creating an alias for it. These patterns can be automatically be generated by Template Haskell just as other instances, making the whole internal representation opaque to the user.

## 4.7 Family of data types

We have presented our approach using the `regular` library for generic programming. However, this imposes the significant restriction that we can only represent single data types like the guestbook examples. We have also developed a solution using the `multirec` library for generic programming, which allows us to support families of mutually recursive data types like the C♯ example. In this section we describe some of the modifications required for representing transformations over families of data types.

### 4.7.1 Representation

In `multirec`, the basic unit of generic representation is the *family*. We represent families as a type variable $f :: * \to *$. Families are *indexed*, and each index is one data type in the family. We represent indices using the type variable $\iota :: *$.

For example, for the C♯ data types the family is defined as follows.

```
data AST :: * → * where
  IClass  :: AST Class
  IMember :: AST Member
  IStat   :: AST Stat
    ...
```

Each constructor of this data type, for instance *IClass*, is the index in the family *AST*. These indices are used to establish the connection between the term level and the type level.

The representation types in the `multirec` library are similar to the ones in `regular`, with some extra parameters. Their definition is as follows.

```
data U         (r :: ∗ → ∗) ι = U
data I κ       (r :: ∗ → ∗) ι = I  {unI  :: r κ}
data K a       (r :: ∗ → ∗) ι = K {unK :: a}
data (f :+: g) (r :: ∗ → ∗) ι = L (f r ι) | R (g r ι)
data (f :×: g) (r :: ∗ → ∗) ι = f r ι :×: g r ι
```

In the actual `multirec` library there are a few more representation types that we do not show here. For example, there is special support for container types like lists.

There are several differences to the `regular` case. All types are parametrized over the type index $\iota$ of the data type they represent. Furthermore, the parameter $r$ is applied at the recursive position and is used for representing the resulting type of recursive functions, which may depend on the type of the children.

The *I* constructor has an additional parameter $\kappa$, which is the type index of the child. Note that this is essential as the child can have a different type than the parent node, which means that $\iota$ and $\kappa$ can be different. For convenience later the *I* and *K* constructors are now written using record syntax for easy unwrapping.

An additional representation type is the following.

```
data (f ▷ ι) (r :: ∗ → ∗) κ where
  Tag :: f r ι → (f ▷ ι) r ι
```

The *Tag* constructor fixes the type index to indicate to which type a constructor belongs to.

As with the `regular` case, there is a type family *PF* that encodes the pattern functor for the data type being represented. The implementation is similar to that of the `regular` case and we only show a small part of the pattern functor for the C♯ compiler here.

```
type CSharp_PF = (     I Decl
                   :+: I Expr
                   :+: (I Expr :×: I Stat :×: I Stat)
                   ...
                 ) ▷ Stat
                :+:
```

$$
\begin{array}{ll}
( & \quad I\ \mathit{Const} \\
\quad :+: & K\ \mathit{String} \\
\quad :+: & (K\ \mathit{String}\ :\times:\ I\ \mathit{Expr}\ :\times:\ I\ \mathit{Expr}) \\
\quad \ldots & \\
) \rhd \mathit{Expr} & \\
\ldots &
\end{array}
$$

The first three cases are for the *StatDecl*, *StatExpr* and *StatIf* constructors, and the other three cases for the *ExprConst*, *ExprVar* and *ExprOper* constructors. The cases for all other constructors are similar.

Note that because of the extra type parameters the kind of the pattern functor is the following.

$$\textbf{type family}\ \mathit{PF}\ (\phi :: * \to *) :: (* \to *) \to * \to *$$

The argument $\phi$ is the type family (*AST* in our case), then the $* \to *$ parameter is the parameter $r$, and finally the last parameter is $\iota$, the type index. We thus instantiate this as follows.

$$\textbf{type instance}\ \mathit{PF}\ \mathit{AST} = \mathit{CSharp}_{\mathit{PF}}$$

Finally an instance of the following type class is used to convert from and to this generic representation.

$$
\begin{array}{l}
\textbf{class}\ \mathit{Fam}\ \phi\ \textbf{where} \\
\quad \mathit{from} :: \phi\ \iota \to \iota \qquad\qquad \to \mathit{PF}\ \phi\ I_0\ \iota \\
\quad \mathit{to} \quad :: \phi\ \iota \to \mathit{PF}\ \phi\ I_0\ \iota \to \iota
\end{array}
$$

The type $I_0$ is a simple type level identity wrapper. We do not show the instance *Fam AST* here as it is similar to the `regular` case, with the addition of the first parameter which guides the types. As with the `regular` library, there is a Template Haskell implementation that generates the pattern functor and the necessary instances.

## 4.7.2 Generic functions

Figure 4.4 shows the implementation of the *shallowEq* function for a family of mutually recursive data types. The $r$ parameter is not used here as we never use the elements at the recursive positions.

At first glance it may appear as if the $\phi\ \iota$ parameter is superfluous, as it is never used in a pattern match. It is however needed to fix the type in the recursive calls to *shallowEq*.

```
class SEq φ (f :: (∗ → ∗) → ∗ → ∗) where
  shallowEq :: φ ι → f r ι → f r ι → Bool
instance SEq φ (I κ) where
  shallowEq _ _ _ = True
instance SEq φ U where
  shallowEq _ _ _ = True
instance Eq a ⇒ SEq φ (K a) where
  shallowEq p (K a) (K b) = a ≡ b
instance (SEq φ f, SEq φ g) ⇒ SEq φ (f :+: g) where
  shallowEq p (L a) (L b) = shallowEq p a b
  shallowEq p (R a) (R b) = shallowEq p a b
  shallowEq _ _       _       = False
instance (SEq φ f, SEq φ g) ⇒ SEq φ (f :×: g) where
  shallowEq p (a :×: b) (c :×: d) = shallowEq p a c ∧ shallowEq p b d
instance SEq φ f ⇒ SEq φ (f ⊳ ι) where
  shallowEq p (Tag a) (Tag b) = shallowEq p a b
```

Figure 4.4: Generic function for checking shallow equality

### 4.7.3 Zippers and paths

The zipper for `multirec` is a straightforward extension of the zipper for `regular`. The data types in the *Ctx* data family take two extra parameters, following the pattern of the representation types. In the `regular` implementation the *Loc* contained a list of such *Ctx* values, each describing a single "step" in the path. In the `multirec` case these values need to be indexed over their type using the following representation.

$$\begin{array}{l}\textbf{data}\ \mathit{Ctxs} :: (* \to *) \to * \to (* \to *) \to * \to * \,\textbf{where}\\ \quad \mathit{Empty} :: \mathit{Ctxs}\ \phi\ a\ r\ a\\ \quad \mathit{Push}\ \ :: \phi\ a \to \mathit{Ctx}\ (\mathit{PF}\ \phi)\ a\ r\ \iota \to \mathit{Ctxs}\ \phi\ \ b\ r\ a \to \mathit{Ctxs}\ \phi\ b\ r\ \iota\end{array}$$

Note that this essentially encodes a list, but with a type index as extra piece of information for each element in the list.

Finally, the *Loc* type representing the value currently in focus together with the list containing the contexts is represented as follows.

$$\begin{array}{l}\textbf{data}\ \mathit{Loc} :: (* \to *) \to (* \to *) \to * \to * \,\textbf{where}\\ \quad \mathit{Loc} :: (\mathit{Fam}\ \phi, \mathit{Zipper}\ \phi\ (\mathit{PF}\ \phi)) \Rightarrow\\ \qquad\quad \phi\ \iota \to r\ \iota \to \mathit{Ctxs}\ \phi\ \iota\ r\ a \to \mathit{Loc}\ \phi\ r\ a\end{array}$$

### 4.7.4 Generic representation of transformations

To encode trees with references generically in `multirec` we use a data type similar to that in the `regular` case.

$$\begin{array}{l}\textbf{data}\ \mathit{WithRef}\ \phi\ \mathit{top}\ a = \mathit{InR}\ (\mathit{PF}\ \phi\ (\mathit{WithRef}\ \phi\ \mathit{top})\ a)\\ \qquad\qquad\qquad\qquad\qquad\ \mid \mathit{Ref}\ (\mathit{Path}\ \phi\ a\ \mathit{top})\end{array}$$

Apart from the extra type parameters, there is another difference here. Instead of using the $\mu\cdot$ operator to fix the recursive positions in later stage, we directly fix the recursion in the *InR* constructor by using *WithRef* for the recursive argument. The reason for this change is that the order of the type parameters matters for defining $\mu\cdot$, and using it here introduces unnecessary difficulties.

Finally a full transformation is again a list of localised insertions consisting of the location and the replacement value. Again we need a type index and we therefore define yet another data type for representing a single insert.

$$\textbf{type}\ \mathit{Transformation}\ \phi\ \mathit{top} = [\mathit{Insert}\ \phi\ \mathit{top}\ \mathit{top}]$$

$$\begin{aligned}&\textbf{data}\ \mathit{Insert}\ \phi\ \mathit{top}\ \iota\ \textbf{where}\\ &\quad \mathit{Insert} :: \phi\ t \to \mathit{Path}\ \phi\ t\ \iota \to \mathit{WithRef}\ \phi\ \mathit{top}\ t \to \mathit{Insert}\ \phi\ \mathit{top}\ \iota\end{aligned}$$

This concludes the representation of tree transformations in `multirec`, on which the rest of the work in this thesis can be based. The type indices ensure that the representation is strongly typed. It is however, as explained earlier, still possible to represent invalid paths but our *diff* and other library functions never do so.

**Apply and diff** We do not discuss the implementations of the *apply*, *diff* and related functions as their implementation is in essence similar to those of the `regular` case. However, the type indices that need to be passed around make the code much harder to read and obfuscate the function types with many polymorphic parameters.

Let us now look at the types of *apply* and *diff*. The $\iota$ argument is simply the type of the top level node. For intermediate nodes the type indices are stored in the *Insert* constructor.

$$\begin{aligned}&\mathit{apply} :: \forall\ \phi\ \iota.\ \mathit{Editable}\ \phi \Rightarrow\\ &\qquad \phi\ \iota \to \iota \to \mathit{Transformation}\ \phi\ \iota \to \mathit{Maybe}\ \iota\\ &\mathit{diff}\ \ \ :: \forall\ \phi\ \iota.\ \mathit{Editable}\ \phi \Rightarrow\\ &\qquad \phi\ \iota \to \iota \to \iota \to \mathit{Transformation}\ \phi\ \iota\end{aligned}$$

As with the `regular` code we have the following property.

$$\forall\ p\ a\ b.\ \mathit{apply}\ p\ a\ (\mathit{diff}\ p\ a\ b) \equiv \mathit{Just}\ b$$

In other words, for every two values $a$ and $b$ of the same type in the family, of which $p$ is the index, we have that if we compute the *diff* between them and apply this transformation to $a$, we get back $b$. Furthermore, our code ensures that the transformation resulting from *diff* uses as much sharing as possible.

**Child lookup and memoization** One particular implementation difficulty where work is involved for the `multirec` case, is the list of children that could be reused and the implementation of the memoization. Both contain a table with parts of the subtree. However, in the `multirec` case different subtrees can have a different type, and therefore it is not possible to directly construct a list of all such values.

The solution is to maintain a separate list for each of the types in the family. We use type classes to create a heterogeneous list storing those subtrees. In order to do a lookup the type classes guide the usage of the right list. Another possible solution is to use a list of *Dynamics* in which the type is part of the lookup key; such an approach is simpler but less efficient as runtime type comparisons are necessary.

## 4.8 Discussion and conclusion

In this chapter we have highlighted the importance of a good representation of transformations. We have shown some representations of transformations for the running examples and have given implementations for computations using this representation. We now review related work and discuss some shortcomings of our approaches, together with possible directions for future work.

### 4.8.1 Related work

The most closely related work to ours is that of [Lempsink et al., 2009]. They describe how to define a generic, type-safe diff algorithm that operates on families of data types. Their notion of "transformation" is encoded by an edit script, which contains insertion, deletion, and copy operations only. They also define an associated *patch* function that transforms a value according to an edit script. However, as we mentioned previously, our work goes beyond the notion of diff.

The ATerm library [Van den Brand and Klint, 2007] provides a representation for the creation and exchange of tree-like data structures in an untyped setting. The implementation is based on maximal subterm sharing by representing terms as a directed acyclic graph.

The technique of extending pattern functors for supporting additional functionality is commonplace. We have used zippers in this work; other applications include selections of subexpressions [Van Steenbergen et al., 2010] and generic storage [Visser and Löh, 2010].

### 4.8.2 Shortcomings

While our solution provides a good basis for an efficient representation of transformations, there are some potential limitations and shortcomings.

**Type safety** Our approach is type-safe in the sense that, when our *diff* and *apply* functions are used as intended, they do not go wrong at runtime. Potential sources of failure, such as navigating to non-existent positions by calling *apply* with the wrong arguments, lead to runtime failures in the *Maybe* monad.

However, we could aim higher and try to check validity of transformations already at compile-time. This would require a significantly more complicated approach and certainly some form of dependent types: the types of the navigation functions in the zipper depend on the type of value they are applied to. This would require a generalisation of the derivative concept; even the dissection of [McBride, 2001] does not express this concept. Aiming for more type-safety would probably be an interesting adventure in a dependently-typed approach to representing transformations. Using really generic zippers [Kiselyov, 2011], which are derived once and for all for all data types, may also be a solution.

**Error handling** Currently, we handle failure by returning *Nothing*. While this is preferable to runtime failure, it is not very informative. An easy way to improve the usability of our transformations would be to provide more useful feedback in case of failure, such as a *String* detailing what went wrong and where.

### 4.8.3 Future work

The performance of the *diff* function could be improved, as abstract syntax trees can contain many nodes in practical applications. As mentioned in Section 4.6.3, its complexity, with memoization, is $O(n^3)$. Cubic behaviour might still be unacceptable in practical scenarios, but lowering this bound would require trading preservation of reuse for speed. It remains to see where the balance between these two factors lays.

Another way to improve the performance is to optimise the handling of transformations with many paths sharing some common prefix. The representation could be extended to share such common prefixes so as to better support localised insertions.

# 5 Incremental AG evaluation

In this chapter we describe how to write attribute evaluators that can efficiently respond to changes in the AST. Although this chapter forms an important part of the final solution, the techniques described here do not work for higher order attributes. We postpone the discussion for higher order attributes to Chapter 6.

The basic idea of our incremental evaluation technique is to cache the visit functions: we store the previous input and output of a visit, and whenever the inputs are unchanged the previous output is returned without recomputation. Because visits can invoke visits of the children, this can lead to superlinear speedups. Because the AST can change, we also need to keep track of changes to child nodes for deciding when to recompute, complicating matters a bit.

## 5.1 Overview

We start this chapter by giving an informal overview of the representation and implementation of our incremental attribute grammar evaluation machinery. An important feature of our implementation is that it is purely functional; there are no side effects or global state. In our informal explanation we do however use the concept of global state, as we believe that it is more intuitive to explain our techniques in such a way. Furthermore, we have used the concept of a global state in the definition of incrementality.

The evaluator for a given AST is represented by a data type which has the same tree structure as the AST, extended with additional information. This extra information is stored within closures that can be invoked for each node. For example, every node contains a function for each of its visits which, given the inherited attributes for that visit, computes the synthesized attributes of that visit, possibly by invoking visits of its children.

After a visit for a certain node has been evaluated, the inherited attributes passed to and synthesized attributes returned by this visit are stored. Whenever the visit is invoked again with the same inherited attributes, which can happen because of changes elsewhere in the AST, the stored synthesized attributes are directly returned and therefore no duplicate computations occur. This is a form of memoization with a cache size of one, because only the immediate previous parameters and result are stored. This memoization is where the speedup comes from, since this may result in a large part of the AST not being revisited after a (small) change.

Another function serves to propagate a change to the AST, in such way that the resulting evaluator tree is updated in the same way as the AST changes. However, this updating leaves unchanged parts of the AST in their original state in order to retain the incremental behaviour.

There are two design principles used to store this information. The first is that we use closures in which the previous inputs and outputs are stored to represent the memoized visit functions. These values are therefore not inspectable from the outside, but can only be used by the invoked closure.

The second design principle is that these functions are stored in a way similar to the state monad, except that each node has its own state. We simulate this by giving the current state as an argument to each function and return the (possibly updated) state from each of these functions. For example, when invoking a child visit the current state is updated with the new state of the children, such that future invocations of functions of those children can be handled more efficiently. Here the state is a collection of closures for the visit functions of that node together with some other data as explained later.

## 5.2 Representation

Before describing the functional implementation of the incremental evaluation machinery, we introduce the data types and function signatures that are used for representing attribute grammar evaluators at runtime. For convenience we use Haskell's

$$
\begin{array}{lll}
\multicolumn{3}{l}{\mathbf{data}\ \mathit{TGuestbook\ top} = \mathit{TGuestbookEmpty}\ \{\ldots\} \mid \mathit{TGuestbookArrive}\ \{} \\
\quad \mathit{tguestbook\_v_0} & :: & \mathit{TGuestbook\ top} \to \\
 & & ((\mathit{Set\ Name}, \mathit{DL}), \mathit{TGuestbook\ top}), \\
\quad \mathit{tguestbook\_v_0\_dirty} & :: & \mathit{Bool}, \\
\quad \mathit{tguestbook\_lookup} & :: \forall\, t. & \mathit{TGuestbook\ top} \to \\
 & & \mathit{Path\ Guestbook\ t} \to \\
 & & \mathit{SemType\ t\ top}, \\
\quad \mathit{tguestbook\_change} & :: \forall\, r. & \mathit{TGuestbook\ top} \to \\
 & & (\forall\, t.\ \mathit{Path\ top\ t} \to \mathit{SemType\ t\ top}) \to \\
 & & \mathit{Path\ Guestbook\ r} \to \\
 & & \mathit{ReplType\ r\ top} \to \\
 & & \mathit{TGuestbook\ top}, \\
\quad \mathit{tguestbook\_tl} & :: & \mathit{TGuestbook\ top} \\
\multicolumn{3}{l}{\quad \} \mid \mathit{TGuestbookLeave}\ \{\ldots\}}
\end{array}
$$

Figure 5.1: Representation for the *Arrive* production of the *Guestbook* nonterminal

*record syntax*, which allows us to give a name to each child of a constructor. A Haskell compiler usually generates a function for each name which can be used to retrieve the corresponding child from the constructor.

As an example, the representation of the evaluator for the AST of the guestbook example is shown in Figure 5.1. The type *TGuestbook* represents the internal state of an evaluator for the attribute grammar version of *Guestbook* from Figure 2.1 and contains a constructor for each of the productions. We have only shown the case for the *Arrive* production here but the others are similar, except for the fields for the nonterminal children, which is *tl* in this case. The first four functions and types are the same for all productions of the *Guestbook* nonterminal.

To give an overview we mention each field shortly first, and we give a more detailed explanation below. The data type has a type level parameter *top*, which is propagated to all evaluators. This parameter is instantiated at the time of the top level invocation of the attribute grammar machinery to the type of the top level node of the AST. The parameter is used in the representation of the trees with references, as references are always paths starting at the top of the AST, which can have a type different from other nodes. For this same reason the type families *SemType* and *ReplType* are used to construct the right types. In this particular guestbook example the *top* parameter and type families can be avoided as the AST consists of the single type *Guestbook* only, but we have included them here for explanatory reasons to show how the machinery works for a family of mutually recursive data types such as the C$^\sharp$ example.

The record constructor of the evaluator state for an arbitrary production *P* of nonterminal *N* contains fields for the visit functions, flags indicating which visits needs to be recomputed, a function for retrieving the evaluator of a subtree, a function for propagating a change into a subtree, and a field for each of the nonterminal children of the production *P*.

**Visit functions** For each visit *X* of the nonterminal *N* we have a visit function *vX*, which corresponds to visit *X* and computes the synthesized results of that visit. The function takes the current state of the evaluator and the inherited attributes of that visit and returns the synthesized attributes of that visit and the updated state.

For the *Guestbook* nonterminal our only visit has no inherited attributes so the $v_0$ function takes only a single parameter.

**Dirty flag** For each visit of *N* we have a *dirty* flag indicating that a visit should be re-evaluated. When the dirty flag is set to *True* this means that the state used by this visit changed since the last evaluation and that the visit may return a different result when invoked. This state change can both occur in the production itself as well as (deep down) in its children. A *False* dirty flag indicates that the visit is guaranteed to return the same results when passed the same inherited attributes as last time. When the values of the inherited attributes have changed due to changes elsewhere in the tree, the visit needs to be recomputed regardless of the *dirty* flag.

**Lookup function** The *lookup* function is used to retrieve the evaluator for the node at the location given by its parameter. The first parameter of this *lookup* function is again the current state of the evaluator, and its second parameter is the *Path* describing for which node the evaluator should be returned. As the type of the result depends on the *Path*, this function is universally quantified over the target type *t* of the *Path*. The return value of *lookup* is the evaluator for the node of type *t* referred to by the path.

Looking up an evaluator for a certain node is used when the AST is changed and a reference is inserted, where the AST referred to originates from a different location than the location where it is inserted. For the inserted reference we want to use the existing evaluator of that node and we therefore retrieve the evaluator for a given path with the *lookup* function.

**Change function** For the implementation of the actual transformation of the AST, the *change* function is used. The first parameter of the *change* function is again the current state of the evaluator. The third and fourth parameters contain a *Path* for the location of the change and a replacement value for that location, and the function returns the new state of the evaluator for the current node such that the change has been applied. The second parameter is the lookup function for the top level node, such that an evaluator can be retrieved for a *Path* relative to the top of the AST.

**Children** Finally, the constructor contains a field for the state of the evaluator of each child. These are used in all previous functions: to compute visits the visits of the children may need to be invoked, the dirty flags depend on the dirty flag of the child visits, the lookup may need to be propagated to children and changes may need to be propagated to children.

The children are the only fields that differ for different productions of the same nonterminal. The children are only used "internally" by the previously described functions and are never directly retrieved by other nodes like the parent node.

### 5.2.1 Nonterminal and evaluator types

In order to make a connection between the type of a nonterminal and its corresponding evaluator type, we use another type family. The type *SemType* maps the nonterminal types to the corresponding evaluator types in the following way.

```
type family   SemType a          :: * → *
type instance SemType DL         = TDL
type instance SemType Guestbook  = TGuestbook
type instance SemType Top        = TTop
```

One place where the *SemType* family is used is in the *change* function, which is polymorphic in the target type of the change.

## 5.3 Functional implementation

In this section we describe our purely functional implementation of the incremental evaluation machinery. We illustrate the implementation with the *Arrive* constructor of the example. The production semantic function is implemented as follows, with *lookup*, *change* and $v_0$ bound in the **where**-clause:

```
semGuestbookArrive :: Name → TGuestbook top → TGuestbook top
semGuestbookArrive _name _tl =
  TGuestbookArrive {
    tguestbook_lookup   = lookup,
    tguestbook_change   = change,
    tguestbook_v0       = v0,
    tguestbook_v0_dirty = True,
    tguestbook_tl       = _tl
  } where
    ...
```

The actual visit code is implemented in the code below as part of the **where**-clause above, in a function called $realv_0$. In each visit it takes the state of the children and the inherited attributes if applicable and returns the synthesized attributes and the new state of the children. These functions are then wrapped in other functions to support incremental evaluation in case nothing has changed.

```
realv0 :: TGuestbook top → ((Set Name, DL), TGuestbook top)
realv0 tl0 = ((_lhsOsignedIn, _lhsOtrueReviews), tl1) where
  ((_tlIsignedIn, _tlItrueReviews), tl1) = (tguestbook_v0 tl0) tl0
  _lhsOsignedIn      = _name ‘Set.insert‘ _tlIsignedIn
  _lhsOtrueReviews = _tlItrueReviews
```

For the computation of the *signedIn* and *trueReviews* values of the child *tl* the visit $v_0$ of the child is invoked, with the current state of the child as argument (in addition to the first occurrence of $tl_0$ used to retrieve the visit function from that current state) and returning the new state of the child. Finally, together with the *signedIn* and *trueReviews* values for the current node the new state of the child is returned.

The wrapping code for a visit first performs the visit as usual by calling the $realv_0$ function. After that the visit function in the evaluator is replaced by a memoizing version that directly returns the synthesized attributes in case nothing has changed. This memoizing version stores in its closure the values of the current inherited and synthesized attributes such that in future calls these can be used for the memoization.

```
v0 :: TGuestbook top → ((Set Name, DL), TGuestbook top)
v0 cur = ((_lhsOsignedIn, _lhsOtrueReviews), res) where
  ((_lhsOsignedIn, _lhsOtrueReviews), tl) = realv0 (tguestbook_tl cur)
  res = update $ cur {
                    tguestbook_v0         = memv0,
                        tguestbook_v0_dirty = False,
                        tguestbook_tl       = tl
                      }
memv0 :: TGuestbook top → ((Set Name, DL), TGuestbook top)
memv0 cur′ = if    ¬ (tguestbook_v0_dirty cur′)
             then ((_lhsOsignedIn, _lhsOtrueReviews), cur′)
             else  v0 cur′
```

In this case there are no inherited attributes so no equality checks on the inherited attributes need to be performed. When inherited attributes are present the condition of the **if** in *memv*$_0$ also includes an equality check between the old and new values of the inherited attributes.

The *update* function is a helper function that is used to update the *dirty* flags after evaluation has completed, either in visits of the current node or in its children. The static dependency graph generated by the scheduling algorithm is used to generate this function, such that the *dirty* flag of a visit is only updated when one of its dependencies has changed.

```
update :: TGuestbook top → TGuestbook top
update cur = cur {
  tguestbook_v0_dirty = tguestbook_v0_dirty cur
                      ∨ tguestbook_v0_dirty (tguestbook_tl cur)
}
```

To get the evaluator residing at a given *Path* the *lookup* function is used. This function is implemented by propagating the request to the given *Path* and then returning the evaluator. Note that on the type level the target type *t* of the path is already present, and at the end of the path we can return the current evaluator since the *End* constructor is the witness to the fact that *t*∼*Guestbook* in this case. Actually, as the Guestbook example consists of a single data type these type arguments could have been avoided, but we keep them for a complete explanation that holds for a family of mutually-recursive data types too.

```
lookup :: ∀ t. TGuestbook top → Path Guestbook t → SemType t top
lookup cur End                    = cur
lookup cur (GuestbookArrive_tl ps) =
  tguestbook_lookup (tguestbook_tl cur) (tguestbook_tl cur) ps
```

The *change* function is used to propagate a change to the evaluator. When the current evaluator is changed we replace the full evaluator with the new one, and otherwise we propagate the change to the corresponding child. After propagating the type we update the *dirty* flags.

```
change :: ∀ r. TGuestbook top → (∀ t. Path top t → SemType t top) →
               Path Guestbook r → ReplType r top → TGuestbook top
change cur lu End                            repl = semGuestbookR lu repl
change cur lu (GuestbookArrive_tl ps) repl = update_tl ps $
   cur {
      tguestbook_tl = tguestbook_change (tguestbook_tl cur)
                           (tguestbook_tl cur) lu ps repl
   }
```

The updating of the dirty flags is slightly less trivial; one may think that we need to invalidate all visits in which the child *e* is used because somewhere in that subtree something has definitely changed. However, it may be the case that no information from that changed node is ever used. Therefore, we adopt the following strategy: whenever the direct child of a node is replaced all visits in which that child is used are invalidated, and otherwise we use the *update* function to propagate changes. This is implemented as follows.

```
update_tl :: Path f t → TGuestbook top → TGuestbook top
update_tl End cur = cur {tguestbook_v0_dirty = True}
update_tl _   cur = update cur
```

Finally, for the changed child the new evaluators have to be constructed or reused. We implement this in the following function, which resembles the *semGuestbook* function, except for the fact that it takes a lookup function as first argument to retrieve the evaluators for the reused nodes.

```
semGuestbookR :: (∀ t. Path top t → SemType t top) →
                     GuestbookR top → TGuestbook top
semGuestbookR lu (Guestbook_Ref p)   = lu p
semGuestbookR lu (GuestbookEmptyR) = semGuestbookEmpty
semGuestbookR lu (GuestbookArriveR name tl) =
   semGuestbookArrive name (semGuestbookR lu tl)
semGuestbookR lu (GuestbookLeaveR name grade review tl) =
   semGuestbookLeave name grade review (semGuestbookR lu tl)
```

### 5.3.1 Example invocation

To illustrate the usage of our evaluation machinery we show with some concrete values how the functions are invoked and what the results of those calls are. We use the example guestbook from the introduction which is represented as follows.

```
example :: Guestbook
example =
  Leave "Bransen" 7.6
    "I liked the fast internet connection" $
  Leave "Dijkstra" 8
    "The atmosphere is great for taking pictures!" $
  Arrive "Dijkstra" $
  Leave "Swierstra" 6
    "Nice hotel, but the beds are too short" $
  Arrive "Bransen" $
  Arrive "Swierstra" $
  Empty
```

Note that the $ operator is function application but associating to the right which is used to avoid many parentheses.

To perform the initial evaluation of the attributes the semantic wrapper function needs to be invoked. This returns the evaluator for which the top level visit can be invoked to retrieve the result and the new state of the evaluator.

```
st1           = semTop (Top example)
(grade, st2) = (ttop_v0 st1) st1
```

The value of *grade* is 7.2 as expected.

Now let us delete the *Leave* "Swierstra" entry from the guestbook. We represent that change by a path and a replacement value. The path needs to point to the fourth entry and can thus be represented as follows.

```
path = Top_gb (Leave_tl (Leave_tl (Arrive_tl End)))
```

The replacement needs to be of type *GuestbookR* and contains a reference to the rest of the guestbook. We represent this by using a path to the entry after the deleted entry as follows.

```
repl = Guestbook_Ref
         (Top_gb (Leave_tl (Leave_tl (Arrive_tl (Leave_tl End)))))
```

To push this change to our evaluator we call the *change* function, which takes as an argument the *lookup* function of the top level node.

$$st_3 = (ttop\_change\ st_2)\ st_2\ ((ttop\_lookup\ st_2)\ st_2)\ path\ repl$$

Finally, we can retrieve the result by calling the top level visit function.

$$(grade_2, st_4) = (ttop\_v_0\ st_3)\ st_3$$

The result is that $grade_2$ now has the value 7.8, and for computing this value the evaluation machinery only traversed the AST up to the deleted element. However, the overall complexity of this evaluation is still linear in the number of entries in the full guestbook, due to the use of the higher order attribute. This problem is explained and solved in the next chapter.

### 5.3.2 Intra-visit attributes

One difficulty that does not occur in our running example is that of so called *intra-visit* attributes. In linearly ordered attribute grammars the computation of the synthesized attributes may depend on inherited attributes of that visit or earlier visits. However, with the implementation that we propose the inherited attributes of previous visits are not in scope and need to be explicitly passed to the visit in which such an attribute is used.

For the standard non-incremental evaluation the UUAGC uses a *visit-tree* approach [Saraiva et al., 2000]. With a visit-tree approach it is statically computed which attributes from the earlier visit are used in subsequent visits, and these are passed as extra arguments to the second visit. For the second visit the attributes used in later visits are passed on to the third visit, and so on.

In our implementation we can do better. Since we are already explicitly encoding the current state of a node, the attributes used in later stages can be stored inside this record. Whenever such an attribute is updated due to the recomputation of the visit in which the attribute was declared, the visits in which the attribute is used are invalidated by setting their corresponding *dirty* flag to *True*. The result of this is that only visits that really use the intra-visit attribute are recomputed and no intermediate visits that only pass such value on to a subsequent visit.

## 5.4 Conclusion

As indicated in the introduction of this chapter the described technique does not support higher order attributes. For example, for the guestbook computation the result of the first visit is incrementally computed and the list of true grades can be therefore quickly be updated after a small change. However, for the computation of the average this full new list is traversed to compute the sum and length for the average, similar to the code in Figure 1.5. In other words, the connection between the constructed list of grades and the decorated list of grades is lost, and after a change the list needs to be fully redecorated. We solve that problem in the next chapter.

# 6
# Higher-order attributes

In the previous chapter we have described a runtime evaluator for the incremental evaluation of attribute grammars based on change propagation and memoisation. However, when higher-order attributes [Vogt et al., 1989] are used the increase in efficiency of that evaluator is lost. In this chapter we therefore extend the incremental attribute grammar evaluator such that the expected speedups also apply when using higher-order attributes. This chapter describes the main contribution of this thesis.

As expected, our new runtime evaluator that can efficiently handle changes to higher-order attribute grammars as described in this chapter does not come for free. In order for our technique to apply an attribute grammar should satisfy certain restrictions, which we describe later in this chapter. Unfortunately this means that for many attribute grammars this technique is not directly applicable, but that the attribute grammar must be changed to satisfy these restrictions. In Chapter 7 we discuss in what way exactly an attribute grammar can be changed.

In Section 6.1 we describe the problem with higher-order attributes in more detail. In Section 6.2 and Section 6.3 we show how to alter the runtime evaluator such that the incremental speedups are retained for higher-order attributes. Finally in Section 6.4 we describe the restrictions that apply and potential shortcomings of our new approach.

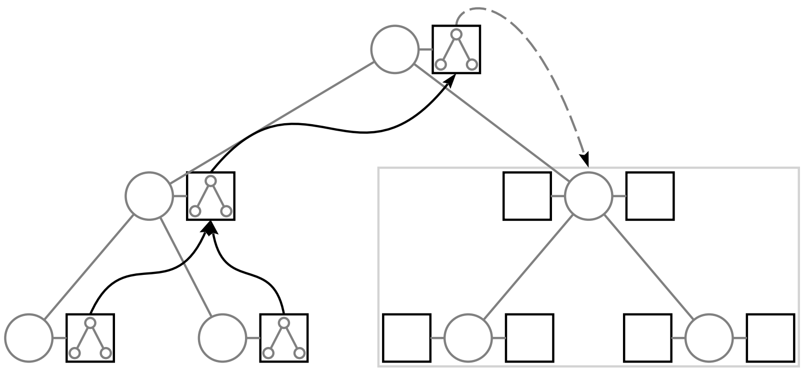

Figure 6.1: Instantiation of a higher-order child

## 6.1 Problem

The problem with higher-order attributes is similar to the first incremental version of the guestbook as described in the introduction in Section 1.3.3. When the code is changed to an incremental version, the list of true grades is built up efficiently. However, nothing is gained because the full list of grades still needs to be traversed in order to compute the average!

Figure 6.1 is a visualisation of the instantiation of a higher-order child. The root node has a single regular child, which is the subtree on the left. In a synthesized attribute of this subtree a value is computed which itself is also an AST. At the root node this AST is instantiated as a higher-order child, which is the subtree in the grey rectangle. The instantiation is implemented by calling the nonterminal semantic function corresponding to the type of the higher-order child, and after instantiation attributes are computed for the higher-order child in the same way as for regular children; inherited attributes of the higher-order child need to be defined by the node at which the child is instantiated, and synthesized attributes of the higher-order child are available to that node.

The problem with an incremental change to a higher-order child is illustrated in Figure 6.2. The change representation is visualised as $\delta$, and the changed parts are coloured grey. In this picture only one node of the left subtree has changed, as a result of which the corresponding synthesized attribute has changed. At the root node, where the higher-order child is instantiated, the only available information

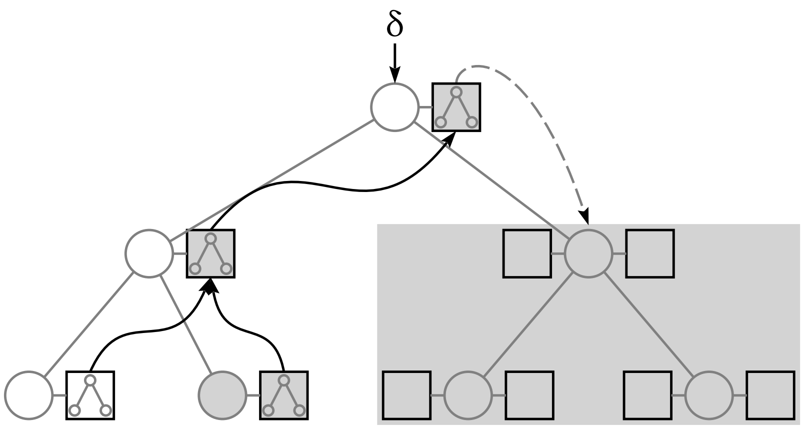

Figure 6.2: Illustration of the problem with a change to a higher-order child, where the fine-grained changes are lost and the whole higher-order child is replaced

is that the value for the higher-order child has changed. The higher-order child is therefore instantiated again and completely redecorated; the internal state of that higher-order child is lost and the fine-grained change propagation as we had in place for the initial AST is lost.

## 6.2 Solution

To solve this problem we can use the same technique for higher-order children as we use for the initial AST: after a change to a higher-order child, use a *diff* algorithm to compute the difference between the old and the new child, and propagate the change representation coming from the *diff* into the higher-order child in the same way as to the initial AST. Computing the *diff* is expensive, however, and we therefore do not expect any efficiency gains from such an approach. Furthermore, in our setting we can do better: we have information about the construction of the higher-order child, in particular we already know where differences occur in the attributes used for the construction of the higher-order child.

Instead of using a *diff* algorithm to find the changes, we track, for each attribute

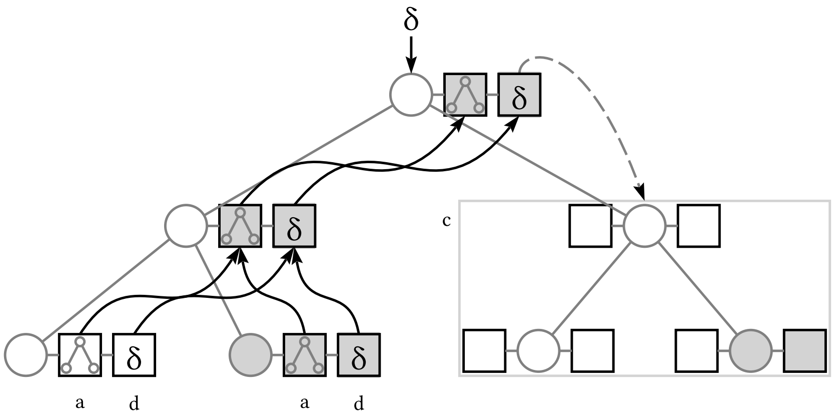

Figure 6.3: Illustration of change propagation to the higher-order child *c* by tracking the changes to *a* in *d*

used in the construction of a higher-order child, in what way its value changed in comparison with the previous evaluation, as visualised in Figure 6.3. We represent a change to such attributes in the same way as a change to the initial AST, by means of a replacement value. The first time that a higher-order child is instantiated it is decorated as usual, but after changes to the AST the replacement values are used to propagate the incremental changes to the higher-order child.

Our approach is similar to that of [Cai et al., 2014], who discuss the automatic generation of incremental evaluators for higher-order languages by statically constructing derivatives of functions. To obtain incremental speedups they assume the existence of certain user-defined change structures that specify for all base-types how changes can be constructed and represented. The change tracking for our incremental evaluation of higher-order attributes resembles the construction of derivatives for higher-order functions. However, in our case the regular incremental attribute grammar forms the basic change structure, and therefore there is no need for the user to specify any extra information to obtain incremental speedups.

**Solution overview** Concretely, we extend our approach from the previous chapter in the following way. First we perform an analysis to find all attributes that are used to construct a higher-order child, called *higher-order attributes*. In Figure 6.3 the higher-order attributes are depicted as synthesized attributes containing a small tree and are named *a*.

For these higher-order attributes we track the changes relative to the last time their value was computed. These changes are represented by replacement values which we synthesize as an extra attribute; we call this the *derived change*. In Figure 6.3 the derived change of *a* is depicted by the synthesized attribute *d* containing a $\delta$.

At the place where a higher-order child is instantiated, the derived change is propagated to the higher-order child in the same way as changes to the initial AST are propagated. In Figure 6.3, the higher-order child *c* can by itself be regarded as a first-order AST to which the change *d* is propagated; whether *d* comes from a *diff* algorithm or from another source does not matter for the propagation of changes to *c*.

Note that we distinguish between two different yet similar parts of the incremental computation: changes to the initial AST, and changes to higher-order children. The changes to these two parts are propagated equivalently; in both parts a given change representation describes in what way the AST is to be changed, and the attributes are updated correspondingly, which is implemented efficiently with memoized visit functions as described in the previous chapter. The difference, however, lies in how the change representation is obtained; for the initial AST the change representation is obtained from an external source such as a *diff* algorithm or a structure editor, while the change representation for a higher-order child is derived from the construction of the higher-order attribute from which the child is instantiated.

Higher-order children can of course also contain higher-order children themselves, and those higher-order children can again contain other higher-order children, and so forth. In that case only the change representation for the initial AST is obtained from an external source, and for all other higher-order children the change is derived from their construction. To keep this explainable we only describe our technique for an initial AST with a single higher-order child, but the technique applies to more complex AGs as well.

**Localised changes** Another small difference between the change representation for the initial AST and the derived change for a higher-order attribute, is that the former consists of a pair of a path and a tree with references, while the latter consists only of a tree with references. The path which is part of the change representation for the initial AST gives the location of the change; a change to a higher-order child however always happens relative to the root of that child because of the way in which it is constructed.

This difference has no effect on the incremental evaluation; when a change to the initial AST conceptually introduces a local change to a higher-order child, the derived change only contains the nodes on the path of the higher-order child towards the location where the change happens, and all other children of those nodes are references to existing parts of the higher-order child. This representation is less compact than the pair, but has exactly the same result in terms of visits that are recomputed.

## 6.3 Implementation

We now discuss the implementation for the support of higher-order children using the example, in the same way as we did in the previous chapter. In Section 6.3.1 we show how the paths used in the derived changes are constructed, in Section 6.3.2 we show the construction of the derived changes, and in Section 6.3.3 we show how the derived changes are propagated to the higher-order child.

### 6.3.1 Paths

The derived changes contain references to the original value of the corresponding higher-order attribute in case nothing has changed. Such references must contain a path relative to the root of the higher-order child in which the derived change is used. However, a higher-order attribute may be instantiated in multiple places, and such a path is therefore dependent on the specific instantiation of a higher-order child. We thus construct the derived changes using relative paths, which later are completed to become absolute paths for a concrete higher-order child.

Conceptually, the resulting derived change is a function from a path to a replacement value. The path argument gives the location in the higher-order child at which this replacement value is used, and based on that path the derived change can be constructed containing references relative to the root of that higher-order child. The same path may however also be used in future calls to the same visit function, when the higher-order attribute is unchanged and a reference can be returned. The path argument is therefore an extra argument to the evaluator, instead of only an argument to the derived change itself.

Concretely, for each higher-order attribute we add an extra path argument to the visit function in which that attribute is computed. This path is then extended in every visit function call, to reflect the (relative) location in the higher-order child in which it is used.

**Path as a difference list** Because paths are extended at the end, we do not directly pass the path itself, but instead we use a construction similar to a *difference list*. The path is represented as a function from *Path* to *Path*, which given the rest of the path returns the complete path.

In our case the paths carry on the type level information about the types of their origin and target node. When a (partial) path is passed as an argument to a child, the type of the target of the final path is not yet known, and the function is therefore polymorphic in the target type of the path. For example, the type of a partial path starting at a node of type $X$ and currently pointing at a node of type $Y$ is the following.

$$\forall\, t.\, Path\ Y\ t \to Path\ X\ t$$

In other words, given the rest of the path from the node of type $Y$ towards the target node of type $t$, this function constructs a path from the starting node of type $X$ towards the target node of type $t$.

**Example** For the guestbook example the type of the visit function is now the following.

$$\begin{aligned} tguestbook\_v_0 ::\ & TGuestbook\ top \\ & \to (\forall\, t.\, Path\ DL\ t \to Path\ DL\ t) \\ & \to ((Set\ Name, DL, DLR\ DL), TGuestbook\ top) \end{aligned}$$

The second argument of this visit is the path, represented as a difference list, indicating in which location in the higher-order child the list of true grades is used. The *DLR DL* which has been added to the result is the derived change and is discussed in the next section.

One might be inclined to think that it is possible to move the path argument inside the pair, as the derived change is the only value that depends on it, as follows.

$$\begin{aligned} tguestbook\_v_0 ::\ & TGuestbook\ top \\ & \to ((Set\ Name, DL \\ & \quad\ , (\forall\, t.\, Path\ DL\ t \to Path\ DL\ t) \to DLR\ DL \\ & \quad\ ), TGuestbook\ top) \end{aligned}$$

This is not possible because it is not the case that only the derived change depends on it: the next value of the derived change, which is returned after a change to this subtree of the initial AST, also may depend on it. As the (memoized) visit function is stored in the internal state, the *TGuestbook top* also indirectly depends on the path argument.

```
v₀ :: TGuestbook top
      → (∀ t. Path DL t → Path DL t)
      → ((Set Name, DL, DLR DL), TGuestbook top)
v₀ cur pDL_trueReviews = ... where
  ... =  realv₀ (tguestbook_tl cur) pDL_trueReviews
  memv₀ cur′ pDL_trueReviews′ =
    if    ¬ (tguestbook_v₀_dirty cur′)
    then ((_lhsOsignedIn, _lhsOtrueReviews
           , List_Ref (pDL_trueReviews End)), cur′)
    else  v₀ cur′ pDL_trueReviews′
realv₀ tl₀ pDL_trueReviews = ... where
  pDL_trueReviews_tl    = if   _name ∈ _tlIsignedIn
                          then pDL_trueReviews ∘ PathL_tl
                          else pDL_trueReviews
  ... = tguestbook_v₀ tl₀ tl₀ pDL_trueReviews_tl
  _lhsOsignedIn         = _name ‘Set.delete‘ _tlIsignedIn
  _lhsOtrueReviews      = if   _name ∈ _tlIsignedIn
                          then _grade : _tlItrueReviews
                          else _tlItrueReviews
  _lhsOtrueReviewsRDL   = if   _name ∈ _tlIsignedIn
                          then ListConsR _grade _tlItrueReviewsRDL
                          else _tlItrueReviewsRDL
```

Figure 6.4: Relevant part of the *semGuestbookLeave* function to illustrate how the derived change is computed

## 6.3.2 Derived change construction

We use the guestbook example again to illustrate in what way the paths are propagated and how the derived changes are computed. The relevant code, in this case for the *Leave* production, is shown in Figure 6.4. The **if-then-else** construction that appears three times can be combined into one with tupling; we use the current version for clarity.

The first thing to notice is the propagation and usage of the path. In the $realv_0$ function the path *pDL_trueReviews* is extended with the *PathL_tl* constructor in case a grade is added to the corresponding list. This results in the desired behaviour: when the list of true grades is passed unaltered to the parent, the path that is passed to the child should also be unaltered. However, when an element is inserted at the head of the list, the path should point one element further in the list. In that case the *PathL_tl* constructor is appended to the path by means of function composition, which can be used because of the difference list structure.

Note that this definition is cyclic, as the expression for the **if** uses the attribute *_tlIsignedIn*, which comes from the child. However, as the path is never used to construct the synthesized attributes but only in future evaluations of the same visit this construction works out. Here it is essential to have lazy evaluation, as the cyclic definition only works in a lazy setting.

Furthermore, in the $memv_0$ definition the path is used to construct the derived change in case nothing has changed. The *End* constructor is passed as an argument to change the difference list into an actual path, and then the path is wrapped in the *List_Ref* constructor to point to the existing list at the given position in the corresponding higher-order attribute. However, if the visit needs to be performed again the new value of the path is used.

Also, in the $realv_0$ function a derived change called *trueReviewsRDL* is built in case the visit needs to be performed again. This derived change has the same structure as the value it represents (*trueReviews* in this case) but the constructors are changed to the ones for the change representation, such that its children can contain references. Note that in this case the : (cons) of the list is replaced by the *ListConsR* constructor.

## 6.3.3 Change propagation

At the place where the higher-order child is instantiated, which in the case of the guestbook is at the top level node, extra work needs to be done to use and propagate the derived change to the higher-order child. We implement this in the following way.

In the internal state for the top level node we store the state of the higher-order child in a *Maybe*. Initially its value is set to *Nothing* as the child has not been instantiated yet. After the first evaluation, for which the higher-order child is instantiated based on the corresponding higher-order attribute, the internal state is saved as a *Just* in this field, such that this state can be used in the future to propagate the derived changes into.

Figure 6.5 shows the relevant code for the *Top* production. The first thing to notice is that the third argument passed to $tguestbook\_v_0$, which is the path used in the change, is the identity function. The identity function here represents the empty path and is used because of the difference list structure.

```
realv0 :: (Maybe (TDL DL), TGuestbook top)
          → (Double, TDL DL, TGuestbook top)
realv0 (_revs, gb0) = (_lhsOaverage, revs1, gb1) where
  ((_gbIsignedIn, _gbItrueReviews, _gbItrueReviewsRDL), gb1)
    = (tguestbook_v0 gb0) gb0 id
  _inst_revs    = _gbItrueReviews
  _inst_revsRDL = _gbItrueReviewsRDL
  revs0 = case _revs of
    Nothing → semDL _inst_revs
    Just v  → (tdl_change v) v ((tdl_lookup v) v) End _inst_revsRDL
  ((_revsIaverage, _revsIlength, _revsIsum), revs1) = (tdl_v0 revs0) revs0
  _lhsOaverage = _revsIaverage
```

Figure 6.5: Visit code for the top level node

The most important part of the code is the usage of the *_revs* argument, which is the state of the higher-order child. When decorating the child there are two options: when its value is *Nothing* and the child has never been decorated and the semantic function is called on the higher-order attribute as before. When there is a *Just* value the derived changes (in *_inst_revsRDL*) are propagated to the child before performing the visit. Finally, note that the resulting state of the child, which is called $revs_1$, is returned and saved by the wrapper code in a *Just* to be used in subsequent rounds of evaluation.

## 6.4 Restrictions and drawbacks

As mentioned before our approach has some restrictions and drawbacks that we discuss in this section.

### 6.4.1 Inspectability

An important limitation of our technique is that it requires knowledge of the construction of the higher-order attributes. In particular, when a higher-order attribute $a$ is used to construct a new value of a higher-order attribute, a path needs to be constructed to indicate in which part of the new AST $a$ ends up. Because our expressions can be arbitrary Haskell, we can write attribute grammars for which we can not (automatically) find out this information.

For our technique to work we therefore require the construction of higher-order attributes to be of a restricted form in which only constructors, attribute references, and constants are used. In particular, pattern matching is not allowed as it would highly complicate dependency analysis.

In practice there are of course cases where more complicated code is written to construct the higher-order child. Another possibility is to mix observable and non-observable construction of higher-order attributes and only get incremental behaviour when enough attributes can be observed. A concrete example of this is a type inference algorithm, where the AST describes expressions of the untyped lambda calculus and the result is a term in the typed lambda calculus. In that case, the spine of the result, which is the expression itself, can be constructed in an observable way for which incremental behaviour is retained, but the types themselves are considered *black boxes* for which all information is lost after they are reconstructed.

### 6.4.2 Overhead

One common problem of incremental evaluation is that there is always overhead involved. In our case the initial evaluation is slower because of the extra state that is built up next to the actual evaluation, and in some cases the overhead can actually be larger than the actual computation. It is therefore not desirable to apply this technique to all attribute grammars.

A possible solution for this problem is to rely on Haskell's lazy evaluation mechanism to perform the extra work to achieve incremental speedups at the moment that the incremental step happens, resulting in practically no extra runtime in the first evaluation. In earlier work we found promising results in that direction, but in order to support higher-order attributes we need to do more work and evaluate some part of the state in a slightly stricter way.

### 6.4.3 Memory consumption

The price that is paid for the decrease of runtime in incremental evaluation usually is the increase of memory consumption. As expected our incremental evaluation machinery consumes more memory than the non-incremental version, but the memory consumption increases only by a constant amount per node in the AST. Because memory is relatively cheap and the order of magnitude of the total memory consumption for the application does not change with our approach, we think that this increase is not a problem.

### 6.4.4 Equality and granularity

In order to decide whether or not a visit needs to be performed after an incremental update, we check for equality of inherited attributes and propagate the dirty flags for every node. These equality checks may however take more time than the actual attribute computation; our method is too *fine-grained*.

For our technique to be usable in practice we therefore intend to make it more coarse-grained and run these checks only at certain nodes, for example only on nodes where some expensive computation is run. Although this leads to more actual attribute evaluation, it can be much faster in practice. To decide on which nodes to perform the computation we could apply a technique similar to that of [Söderberg and Hedin, 2011], who describe the incremental evaluation of reference attribute grammars based on caching in an imperative setting. To improve the caching behaviour they use a selective caching mechanism based on profiling, thereby optimising the caching to specific use cases.

### 6.4.5 Cache size

The memoization technique we use to store the inherited and synthesized attributes only stores the immediate previous values. This technique can therefore be regarded as using a cache of size one per node. As a result of this, when the AST is changed and then changed again to some previous state, the synthesized attributes need to be computed again.

## 6.5 Conclusion

We have extended the evaluation machinery with the support for higher-order attributes. There is some overhead due to extra information tracking that needs to be performed in order to efficiently handle changes to higher-order children, but the technique can lead to a large decrease in the number of attribute values that need to be computed after a change. The technique does not come for free however, as it must be possible to test for equality on all inherited attributes and the attributes which are used for the construction of higher-order children must be inspectable.

# 7 Supporting incrementality

As explained in the previous chapters our incremental evaluation machinery imposes restrictions on the attribute grammars. We require that we can check for equality of inherited attributes and that the higher-order attributes are constructed in an inspectable way. These restrictions make the technique not widely applicable.

Also specific styles of using attribute grammars will have a negative influence on the effectiveness of our approach to incremental evaluation, such as the use of chained attributes that correspond to the use of state monads. There are several improvements, either manually or (semi-)automatically, that can be made to an attribute grammar to overcome the limitations and improve the effectiveness.

This chapter consists of two parts. In the first part we discuss such improvements and describe how they can be applied manually. As the improvements are not exclusively necessary for the incremental evaluation but can also improve efficiency in general, the first part of this chapter can also be seen as a guideline on how to structure an attribute grammar. The second part of this chapter describes how such improvements can be automatically performed by the AG compiler.

Concretely, in the first part of this chapter we discuss the following improvements:

- *Unique value generation*: in Section 7.1 we discuss why generating unique values using chained attributes has a negative influence on the effectiveness of the incremental evaluation. We explain a different way in which unique values can be generated such that the effectiveness of the incremental evaluation is retained.

- *Projection of inherited attributes*: in Section 7.2 we discuss a related issue with inherited attributes. For certain types of inherited attributes such as environments, in most subtrees only a part of the value is used in the computation of other attributes. By projecting out the relevant part of the inherited attribute we ensure that its value stays the same for many subtrees after a change to the AST, thereby avoiding recomputations.

- *General insights*: in Section 7.3 we generalise the insights from the previous two improvements to explain in what way an AG has to be written to support the incremental evaluation machinery.

- *Equality*: the first hard requirement on the attribute grammars is the need to be able to check for equality on inherited attributes, which we discuss in Section 7.4. One problem is that these equality checks may be impossible or expensive, and we show how they may be avoided.

- *Inspectability*: the restriction that every higher-order attribute must be inspectable is discussed in Section 7.5. The semantic rules for the construction of higher-order attributes cannot consist of arbitrary Haskell expressions but must consist of constructors and attribute references only. For the two running examples of this thesis this restriction was not problematic, but in general it is often desired to use more complex expressions for the construction.

The second part of this chapter starts with Section 7.6 where we discuss in what way the AG compiler can analyse the grammar to automatically perform projections on inherited attributes as discussed in Section 7.2. It is future work to implement such an analysis and Section 7.6 therefore consists of ideas only. In Section 7.7 we describe how arbitrary Haskell programs can be translated to AGs; this technique can be used to solve the problem of the inspectability of higher-order attributes as explained in Section 7.5. Furthermore, such a translation can also be used to achieve automatic tupling and to get incremental evaluation for arbitrary Haskell programs for free, as we discuss in the future work in Section 9.2. Finally, in Section 7.8 we conclude this chapter.

## 7.1 Unique value generation

In many program analyses and transformations there is a need for fresh variables. The standard way of implementing fresh variable generation in attribute grammars is to use a chained attribute *counter* of type *Integer*, which holds the value that is used to uniquely label the next fresh variable. At each use of the *counter* it is updated and passed on (as are all chained attributes).

This design pattern can be directly expressed in AG notation and no special back end support is needed. With incremental evaluation, however, this method of generating fresh variables has a negative effect on the effectiveness of the incremental evaluator, because the chained behaviour propagates changes through the whole AST. Chained behaviour in general has a negative effect on incremental evaluation; a chained attribute is a combination between an inherited and synthesized attribute, and the value of the inherited attributes is used to decide whether a visit should be computed again after a change. Therefore, whenever the value of a chained attribute changes all visits later in the AST using that chained attribute are computed again. Especially in the case where we use it to generate unique numbers the fact that a value has changed with respect to a previous evaluation is not particularly relevant; as long the numbers handed out are unique we are fine.

For example, imagine a type inference algorithm for the lambda calculus with a Hindley-Milner style type system [Hindley, 1969, Milner, 1978], implemented with attribute grammars. Now suppose that an expression $e$ is extended with a let binding at top level, and thus is converted to $\textbf{let}\, x = \ldots\ \ \textbf{in}\, e$ where $x$ does not appear in $e$. It is clear that the type of $e$ has not changed due to this edit action and should not be computed again. However, when the fresh type variables are first generated for the binding of $x$, the chained attribute for the variable generation that is passed to $e$ is different from the first evaluation, so all fresh type variables used in $e$ change! As a result of this not only are all types inferred again, but there is also some overhead involved which results in the incremental evaluation being slower even in such a simple case.

The general problem with fresh variable generation as a chained attribute is that it is overspecified; more restrictions than necessary are imposed. Instead of defining that we need a unique variable at each node, we have defined a strict order in which the variables are to be generated. This results in extra dependencies that in case of incremental evaluation leads to redundant recomputations.

The solution we propose is to abstract over fresh variable generation at attribute grammar level and use a different mechanism for generating fresh variables. One way of handling this is by producing a fresh variable generator for each node in the parsing stage, using the path from the root to the node as seed.

```
newtype Unique a = Unique {runUnique :: Integer → (a, Integer)}
instance Monad Unique where
  f ≫= g = Unique $ λn →
    let (a, n′) = runUnique f n
    in runUnique (g a) n′
  return a = Unique $ λn → (a, n)
getUnique :: Unique Integer
getUnique = Unique (λn → (n, n + 1))
```

Figure 7.1: Implementation of the *Unique* monad for fresh value generation

We propose a different approach: we generate fresh variables in a more imperative way using some form of global state. In a pure setting this can be achieved by running the evaluation in a monadic environment [Wadler, 1990]. We use a monad *Unique* with a function *getUnique* :: *Unique Integer* for fresh variable generation, for which a possible implementation is shown in Figure 7.1. The *Unique* monad can also be implemented using the Haskell *State* monad.

To use this monad the visit functions have to be written in monadic style and the types of the functions need to be altered correspondingly. The type of the evaluator for the *Arrive* production is shown in Figure 7.2. Because the state of the unique number is now not implemented anymore as an attribute, its value does not influence the decision whether a visit needs to be recomputed. Hence, a visit is only recomputed for other reasons and in that case also new unique numbers are generated. When the visit is not recomputed no new numbers are generated and all attributes in which the unique numbers are used stay unchanged.

As a result of this change, the outcome of an incremental evaluation is not necessarily equal to the usual evaluation anymore; they are however equivalent under alpha renaming, which is fine for practical cases where fresh variables are used. In the particular case of type inference a normalisation step could be added which does alpha renaming to some normal form for observational equality.

Unfortunately, there is another problem with this way of generating fresh variables: when a subtree is duplicated due to a transformation and no recomputation happens in the resulting subtrees, their fresh variables are shared. In usual applications this is undesirable and could lead to wrong results. This technique does therefore not work when duplication is allowed in the transformations, but improves the incremental behaviour otherwise.

```
TGuestbookArrive {
  tguestbook_v₀         ::      TGuestbook top →
                                (∀ t. Path DL t → Path DL t) →
                                Unique ((Set Name, DL, DLR DL)
                                        , TGuestbook top),
  tguestbook_v₀_dirty   ::      Bool,
  tguestbook_lookup     :: ∀ t. TGuestbook top →
                                Path Guestbook t →
                                SemType t top,
  tguestbook_change     :: ∀ r. TGuestbook top →
                                (∀ t. Path top t → SemType t top) →
                                Path Guestbook r →
                                ReplType r top →
                                Unique (TGuestbook top),
  tguestbook_tl         ::      TGuestbook top
}
```

Figure 7.2: Representation for the *Arrive* production using the *Unique* monad

## 7.2 Projection of inherited attributes

Our incremental attribute grammar evaluation machinery uses equality of inherited attributes to decide when to recompute. However, changes in inherited attributes do not necessarily imply changes in synthesized attributes, because it can be the case that the computations for the synthesized attributes only use an unchanged part of the inherited attribute.

For example, the *env* attribute in our C$^\sharp$ compiler contains a mapping from variables (both local variables and parameters) to their (relative) location on the stack. In the SSM code generation for a function, each variable that occurs is lookup up in this environment to find its location. When a new local variable declaration is added to the function body, the environment changes and the SSM code for the complete function body is recomputed. This recomputation could be avoided, however, because the location of all other variables is unchanged and the new variable is never looked up.

To accomplish this we may decide to pass only a projection of an attribute, containing the information that interests us here, instead of the complete attribute. Such projections are more likely to remain unchanged. The only variables that are ever looked up *env* are the variables that appear in the function body. Therefore, when a variable is added to *env* which is not used in the body, this does not change the result of the SSM code generation. It does however have a negative influence on the incremental behaviour, since the inherited attributes have changed and recomputation happens.

Concretely, the solution is to switch to a two-visit approach. In the first visit a set of used variables is collected in a synthesized attribute. In the second visit the rest of the attributes are computed based on the inherited *env* similar to the standard approach. Because the used variables are already available for all children at the beginning of the second visit, they can be used to perform a projection on *env*. At every node, only that part of *env* containing information about the variables that are actually used in each child is passed to that child. As a consequence the inherited attribute changes only when the location of one or more of the used variables has changed, in which case different SSM code is indeed to be generated.

## 7.3 General suggestion

As we have discussed in Section 7.1 using a chained attribute for generating fresh variables is bad for the performance of our incremental evaluation machinery. For fresh variable generation we have used a special implementation to ameliorate this loss of performance, but the same reasoning holds for other chained attributes from which we draw a more general conclusion: chained attributes are bad for incremental performance.

Furthermore, Section 7.2 describes how a projection of an inherited attribute can improve the performance. The observation from that section is that changes to inherited attributes result in recomputations, and that doing more work in an earlier phase (gathering the used variables and projecting) can avoid recomputations later.

To support the incremental evaluation machinery we suggest a general pattern: computations must be done as early as possible, by which we mean early in the control flow. In attribute grammar terms this means that operations on inherited attributes should happen as close to the root of the tree as possible, while operations on synthesized attributes should happen as close to the leaves of the tree as possible.

The motivation behind this idea is that in this way we keep the number of values that change after an incremental update to the AST as small as possible. When some computation relies on a value that is changed, it has to be recomputed at some point. Therefore, when that recomputation happens earlier, recomputation of other values that are "in between" may be avoided.

## 7.4 Equality

Our approach uses equality checks on inherited attributes to decide whether the value of the inherited attributes has changed. Such equality checks can however be expensive, in particular more expensive than the recomputation of the attributes in the corresponding visit.

For the correctness of the incremental evaluation machinery it is not essential that the equality check is precise. In particular, the pessimistic approach of always returning *False* still leads to correct results, albeit not so efficiently computed. In some cases it may be a viable approach to avoid equality checks on *some* inherited attributes for which the checks are expensive. This will lead to more attribute evaluation, but can still be cheaper than checking equality and thus will lead to increased efficiency. The decision on where to apply such a trick is hard though and one may be forced to fall back on using profiling techniques.

Another possible way in which the efficiency of the equality checks may be improved is by using *hash-consing*. With hash-consing the runtime machinery ensures that for each value there is a unique representation in memory. Consequently, equal values are shared in memory and equality checks for any type are reduced to checking pointer equality. Unfortunately, implementing hash-consing efficiently is hard, especially for a lazy language such as Haskell, and we have therefore not tried to use this technique. [Saraiva et al., 2000] did however find promising results with such an approach.

## 7.5 Inspectability

A major restriction for the higher-order attributes is that we need to have inspectable construction of those attributes. For simple examples such as the guestbook this is not problematic, but the higher-order child for the C$^\sharp$ compiler from Figure 2.8 already breaks this restriction. The instantiation of the higher-order child was done as follows.

```
inst.block :: Stat
inst.block = StatBlock [
                 StatDecl @init.copy,
                 StatWhile @cond.copy (StatBlock [
                    @body.copy,
                    StatExpr @incr.copy
                  ])
               ]
```

At first glance it may look like we use only constructors and attribute references, but in this expression we also use the special Haskell notation [... ,... ] to construct lists. Automatically analysing this expression to find out in which location in the higher-order child certain attributes are used is currently not possible.

Adding support for this special list syntax solves the problem for this example, but not the general inspectability problem. There are many other Haskell functions that can make the expressions shorter or easier to understand, and supporting all of them would require a full Haskell compiler.

For now we put the burden on the attribute grammar programmer and require that all higher-order attribute rules are inspectable. The C$^\sharp$ example above thus needs to be written in an inspectable way as follows.

```
inst.block :: Stat
inst.block = StatBlock (
                 (:)  (StatDecl @init.copy)
                 ((:) (StatWhile @cond.copy (StatBlock (
                    (:)  @body.copy
                    ((:) (StatExpr @incr.copy) [ ]))))
                 [ ]))
```

For more advanced compilers like the UHC it is however much harder to work around this restriction manually, and in Section 7.7 we therefore propose an automatic translation to solve this problem.

## 7.6 Automatic projection

As discussed in Section 7.2 the effectiveness of our approach is dependent on the changes to the AST and to the inherited attributes, and projection of inherited attributes can help to avoid redundant computations. We have made some suggestions to attribute grammar programmers about how to use projections to improve the effectiveness of the incremental evaluation. It is however to be preferred that the attribute grammar compiler performs such projections automatically, possibly based on manual annotations.

The projection of inherited attributes usually follows the same pattern, and the correctness depends on certain algebraic properties. For example, in the case of an environment in which the type of a variable is looked up we have the following property.

$$v \not\equiv w \rightarrow \mathit{lookup}\ v\ (\mathit{insert}\ w\ t\ \mathit{env}) \equiv \mathit{lookup}\ v\ \mathit{env}$$

Here $v$ and $w$ are variables, *env* is an environment, $t$ is some type and *lookup* is the lookup function. This property specifies that the insertion of another variable into the environment does not influence the result of the lookup. Using this property we can project out for each subtree that part of the environment containing the variables being actually looked up in that subtree, as we can be sure that the other variables do not influence the result.

We believe that such algebraic properties, either manually specified or coming from a library with properties for common types, can be used to do projections automatically. For instance, for every inherited attribute of type *Map a b* the pattern for projections is similar: a synthesized attribute of type *Set a* contains all used keys, which are inserted in every node where a lookup is performed. In every place where the *Map* is passed to a child node, only the keys from the *Set* for that child are passed, as a result of which the inherited attribute only changes when the value for a key that is actually used has changed.

We have not implemented this technique for automatically performing projections, and the idea posed in this section is therefore future work.

## 7.7 Haskell to AG translation

In this section we propose an automatic approach to overcome the inspectability problem. We do so by translating all Haskell expressions to attribute grammars such that the full program is represented as a minimal higher-order attribute grammar. As a result of this we do not only get inspectable construction of higher-order attributes, but we may also gain efficiency on the rest of the program due to tupling which is done automatically in attribute grammars.

In the following sections we introduce a functional language (similar to Haskell) which is the source of the translation, an example, and the translation to minimal higher-order attribute grammars. We describe the approach by giving a translation from a complete functional program to attribute grammars. This translation can also be used for single attribute expressions to automatically obtain inspectable construction of higher-order attributes.

### 7.7.1 A simple functional language

Instead of taking full Haskell as a source for the translation we define a language that we call $\lambda$: the simply typed lambda calculus extended with algebraic data types, recursive let and primitive types and functions. The syntax of $\lambda$ is defined in Figure 7.3, where the $\overline{g}$ construct in the figure denotes zero or more occurrences of some $g$.

**Algebraic data types** Next to primitive types such as *Int*, we use algebraic data types as base types. At top level we have a list of data type definitions, each introducing a new type with a set of constructors. Each constructor has a name and field, which we call children, each with their own type. The data type definitions can be (mutually) recursive and the types of the children therefore are allowed to refer to any of the data types that are declared at top level.

The expression language allows construction of data types via data type constructors. Furthermore, case distinction is used to pattern match on the values. Each case alternative matches on a constructor and binds the values of the children to the given variables in the body of that alternative.

**Recursive let** The recursive let defines a set of mutually recursive functions. As with abstractions, all defined variables must have an explicit type annotation.

| | |
|---|---|
| $T, C, P, x, f$ | -- Type, constructor, primitive type, variable<br>-- and function names respectively |
| $\lambda ::= \overline{d}\ e$ | -- Lambda term |
| $d ::= \textbf{data}\ T\ \overline{c}$ | -- Data type definition |
| $c ::= \mid C\ \overline{\tau}$ | -- Constructor with children |
| $e ::= x$ | -- Variable |
| $\mid\ \lambda x : \tau . e$ | -- Lambda abstraction |
| $\mid\ e\ e$ | -- Application |
| $\mid\ C$ | -- Constructor |
| $\mid\ \textbf{case}\ e\ \textbf{of}\ \overline{a}$ | -- Case distinction |
| $\mid\ \textbf{let}\ \overline{t}\ \textbf{in}\ e$ | -- Recursive let |
| $\mid\ \underline{f : \overline{\tau \rightarrow}\ \tau}$ | -- Primitive function |
| $\tau ::= T$ | -- Type of data type |
| $\mid\ \tau \rightarrow \tau$ | -- Function type |
| $\mid\ \widehat{P}$ | -- Primitive type |
| $t ::= x\ :\ \tau$ | |
| $x = e$ | -- Let definition |
| $a ::= C\ \overline{x} \rightarrow e$ | -- Case alternative |

Figure 7.3: Functional language $\lambda$

$$
\begin{array}{l}
\textbf{data}\ \mathit{Tree}\qquad\ \mid \mathit{Leaf}\ \widehat{\mathit{Int}} \mid \mathit{Bin}\ \mathit{Tree}\ \mathit{Tree}\\
\textbf{data}\ \mathit{Ordering} \mid \mathit{EQ} \mid \mathit{GT} \mid \mathit{LT}\\[4pt]
\textbf{let}\ \mathit{depth}\ :\ \mathit{Tree} \rightarrow \widehat{\mathit{Int}}\\
\quad \mathit{depth} = \lambda t : \mathit{Tree}.\ \textbf{case}\ t\ \textbf{of}\\
\qquad\qquad\qquad \mathit{Leaf}\ n \rightarrow \underline{0}\\
\qquad\qquad\qquad \mathit{Bin}\ l\ r \rightarrow \underline{\mathit{succ}}\ (\underline{\mathit{max}}\ (\mathit{depth}\ l)\ (\mathit{depth}\ r))\\
\quad \mathit{dleaves}\ :\ \mathit{Tree} \rightarrow \widehat{[\mathit{Int}]}\\
\quad \mathit{dleaves} = \lambda t : \mathit{Tree}.\ \textbf{case}\ t\ \textbf{of}\\
\qquad\qquad\qquad\quad \mathit{Leaf}\ n \rightarrow \underline{\mathit{singleton}}\ n\\
\qquad\qquad\qquad\quad \mathit{Bin}\ l\ r \rightarrow \textbf{case}\ \underline{\mathit{compare}}\ (\mathit{depth}\ l)\ (\mathit{depth}\ r)\ \textbf{of}\\
\qquad\qquad\qquad\qquad \mathit{EQ} \rightarrow \mathit{dleaves}\ l\ \underline{+\!\!+}\ \mathit{dleaves}\ r\\
\qquad\qquad\qquad\qquad \mathit{GT} \rightarrow \mathit{dleaves}\ l\\
\qquad\qquad\qquad\qquad \mathit{LT}\ \ \rightarrow \mathit{dleaves}\ r\\
\textbf{in}\ \ \mathit{dleaves}
\end{array}
$$

Figure 7.4: Deepest leaves in a binary tree (straightforward version)

**Primitive functions** Allowing primitive functions is not strictly necessary, but to make the example concise and to be able to leave out irrelevant details primitive functions and types are supported. Type annotations are also needed for primitive functions, but in our example we leave out these type annotations because we believe the reader can easily infer them from the name of the function. Furthermore, we restrict ourselves to first-order functions only; the $\tau$'s in the primitive function calls may only be $T$ or $\widehat{P}$.

### 7.7.2 Example

To illustrate the translation we use an example that is written in $\lambda$. The code is implemented in the way that we feel is the most intuitive and elegant. There exist however more efficient but less elegant implementations. In this section we show both the elegant and the (derived) efficient implementation and in later sections we show how the translation to attribute grammars can result in the automatic construction of the efficient version.

In the example we construct a list containing the deepest leaves from a binary tree. The deepest leaves are all leaves that have a maximum distance to the root of the tree. A straightforward implementation is presented in Figure 7.4.

Two algebraic data types are defined, *Tree* for binary trees and *Ordering* for the result of comparing two integers. The constructor *Leaf* has a single child of type $\widehat{Int}$ which is a *primitive type* which cannot be inspected but only be passed as an argument to a primitive function or as a return value.

The program itself consists of a let definition in which two functions are defined. The function *depth* computes the depth of the binary tree in a straightforward way. The underlined functions are *primitive functions* and are thus left abstract. The functions do what their names suggest; $\underline{max}$ computes the maximum of two integers, $\underline{succ}$ the successor of an integer, $\underline{singleton}$ constructs a singleton list from its argument and $\underline{+\!\!+}$ concatenates two lists.

For the actual translation typing information is needed for every part of the program, including the primitive functions. We therefore assume that the types are known for all primitive functions, but we leave them implicit to make the example more concise.

The *dleaves* function computes the list of deepest leaves of the tree, thereby using the *depth* function to compute the depth of its children in the binary case. If the depths are not equal then the deepest leaves of the deepest subtree are returned, otherwise the two lists of deepest leaves are combined into the result list. In the leaf case the value is returned as a singleton list.

One problem with this definition is in the calls to *depth* in *dleaves*. For every call to *dleaves*, which in the worst case is once for each node $x$, the full subtree of $x$ is traversed by *depth*. This makes the complexity of the algorithm quadratic[1], while a linear solution is possible. With the translation to AGs a linear solution is obtained automatically, because of the tupling of *depth* and *dleaves* which is done automatically in AGs [Kuiper and Swierstra, 1987].

**Efficient version** Before showing the resulting MHAG code, we first show the result of translating the example to an AG and back in Figure 7.5. Note that the code shown here is not the direct output of translating the MHAG code back to $\lambda$, but has been manually prettified to make clear what is going on. The variable and function names have been changed to more meaningful names and some parts have been inlined and $\beta$-reduced to make the code shorter.

The type *TreeSyn* is the carrier type of the algebra and consists of the results for the two recursive functions; *depth* and *dleaves* are now simple projection functions to get the corresponding value. The *semTree* function is a standard nonterminal

[1]For the sake of the example, we assume that $\underline{+\!\!+}$ takes constant time.

```
data Tree     | Leaf $\widehat{Int}$ | Bin Tree Tree
data Ordering | EQ | GT | LT
data TreeSyn  | TreeSyn $\widehat{[Int]}$ $\widehat{Int}$

let depth   : TreeSyn → $\widehat{Int}$
    depth   = λts : TreeSyn. case ts of TreeSyn l d → d
    dleaves : TreeSyn → $\widehat{[Int]}$
    dleaves = λts : TreeSyn. case ts of TreeSyn l d → l
    semTree : Tree → TreeSyn
    semTree = λd : Tree.
                 case d of Bin l r  → semBin (semTree l) (semTree r)
                           Leaf n   → semLeaf n
    semLeaf : $\widehat{Int}$ → TreeSyn
    semLeaf = λn : $\widehat{Int}$. TreeSyn ($\underline{singleton}$ n) $\underline{0}$
    semBin  : TreeSyn → TreeSyn → TreeSyn
    semBin  = λl : TreeSyn. λr : TreeSyn.
              let de   : $\widehat{Int}$
                  de   = $\underline{succ}$ ($\underline{max}$ (depth l) (depth r))
                  le   : $\widehat{[Int]}$
                  le   = semOrd $ho_1$ (dleaves l) (dleaves r)
                  $ho_1$  : Ordering
                  $ho_1$  = $\underline{compare}$ (depth l) (depth r)
              in  TreeSyn le de
    semOrd  : Ordering → $\widehat{[Int]}$ → $\widehat{[Int]}$ → $\widehat{[Int]}$
    semOrd  = λc : Ordering. λdll : $\widehat{[Int]}$. λdlr : $\widehat{[Int]}$.
                 case c of EQ → dll $\underline{+\!\!+}$ dlr
                           GT → dll
                           LT → dlr
in  λt : Tree. dleaves (semTree t)
```

Figure 7.5: Deepest leaves in a binary tree (efficient version)

semantic function, and *semLeaf* and *semBin* the production semantic functions for the corresponding productions.

The case for *Leaf* is the trivial tupling of the two results. The case for *Bin* is slightly more complex, yet similar to the original version. The value *de*, for the *depth* case, is identical to the original formulation. However, it is important to notice that its definition is no longer recursive; the calls to *depth* are simple projection functions retrieving the corresponding value from the tuples. The same holds for the computation of the deepest leaves.

For the result of the call to $\underline{\mathit{compare}}$ there is an extra indirection compared to the original formulation. This indirection can be removed by inlining the *semOrd* function which thus makes the *dleaves* case also similar to the original formulation. The current presentation is however closer to the attribute grammar version of this code, in which a higher-order child is instantiated, so we use that for didactic reasons.

### 7.7.3 Example translation

In Figure 7.6 we show the minimal higher-order attribute grammar code that is the result of translating the example. The first two data types are present in the original as well as in the final $\lambda$ code. The $\mathit{Ordering}'$ and $\mathit{Ordering}''$ data types are intermediate data types that are generated as the result of the translation process for storing intermediate values.

The *Tree* data type has, as expected, two synthesized attributes: *depth* and *dleaves*. The *Ordering* data type has a synthesized attribute *cmpres* which contains a value of type $\mathit{Ordering}'$. This $\mathit{Ordering}'$ has an inherited attribute *arg* (the function argument) and a synthesized attribute of type $\mathit{Ordering}''$, which again takes an inherited attribute *arg* and finally returns a synthesized attribute *res* of type $\widehat{[\mathit{Int}]}$. This pattern can be thought of as a function taking two $\widehat{[\mathit{Int}]}$ arguments and returning an $\widehat{[\mathit{Int}]}$. In the resulting $\lambda$ code as presented in Figure 7.5 the semantic functions for *Ordering*, $\mathit{Ordering}'$ and $\mathit{Ordering}''$ are combined and called *semOrd*.

This indirection is a result of the way the translation scheme is formulated; it can however easily be avoided. We have decided to keep the example results close to the actual translation scheme as it is presented later to help the reader in understanding the described approach.

The semantic rules for the *Leaf* case are straightforward and directly follow the structure of the original functions. In the *Bin* case, the computation of *depth* is also a direct mapping from the original. The small difference is that in MHOAGs (Section 2.6) the right-hand side of an expression can only be exactly one primitive function call, which means that in order to compose functions the intermediate

$$
\begin{array}{lllll}
\mathbf{data}\ \mathit{Tree} & \multicolumn{4}{l}{\mid \mathit{Leaf}\ n :: \widehat{\mathit{Int}}\ \mid \mathit{Bin}\ l :: \mathit{Tree}\ r :: \mathit{Tree}} \\
\mathbf{data}\ \mathit{Ordering} & \multicolumn{4}{l}{\mid \mathit{EQ}\ \mid \mathit{GT}\ \mid \mathit{LT}} \\
\mathbf{data}\ \mathit{Ordering}' & \multicolumn{4}{l}{\mid \mathit{EQ}'\ \mid \mathit{GT}'\ \mid \mathit{LT}'} \\
\mathbf{data}\ \mathit{Ordering}'' & \multicolumn{4}{l}{\mid \mathit{EQ}''\ \mathit{dl} :: \widehat{[\mathit{Int}]}\ \mid \mathit{GT}''\ \mathit{dl} :: \widehat{[\mathit{Int}]}\ \mid \mathit{LT}''\ \mathit{dl} :: \widehat{[\mathit{Int}]}} \\
\mathbf{attr}\ \mathit{Tree} & \multicolumn{4}{l}{\mathbf{syn}\ \mathit{depth} :: \widehat{\mathit{Int}}\ \ \mathbf{syn}\ \mathit{dleaves} :: \widehat{[\mathit{Int}]}} \\
\mathbf{attr}\ \mathit{Ordering} & \multicolumn{4}{l}{\mathbf{syn}\ \mathit{cmpres} :: \mathit{Ordering}'} \\
\mathbf{attr}\ \mathit{Ordering}' & \multicolumn{4}{l}{\mathbf{inh}\ \mathit{arg} :: \widehat{[\mathit{Int}]}\ \ \mathbf{syn}\ \mathit{res} :: \mathit{Ordering}''} \\
\mathbf{attr}\ \mathit{Ordering}'' & \multicolumn{4}{l}{\mathbf{inh}\ \mathit{arg} :: \widehat{[\mathit{Int}]}\ \ \mathbf{syn}\ \mathit{res} :: \widehat{[\mathit{Int}]}} \\
\mathbf{sem}\ \mathit{Tree} & \mid \mathit{Bin} & \mathbf{lhs}.\mathit{dleaves} & = & @\mathit{ho}_3.\mathit{res} \\
 & & \mathbf{inst}.\mathit{ho}_1 & :: & \mathit{Ordering} \\
 & & \mathbf{inst}.\mathit{ho}_1 & = & \underline{\mathit{compare}}\ @l.\mathit{depth}\ @r.\mathit{depth} \\
 & & \mathbf{inst}.\mathit{ho}_2 & :: & \mathit{Ordering}' \\
 & & \mathbf{inst}.\mathit{ho}_2 & = & @\mathit{ho}_1.\mathit{cmpres} \\
 & & \mathit{ho}_2.\mathit{arg} & = & @l.\mathit{dleaves} \\
 & & \mathbf{inst}.\mathit{ho}_3 & :: & \mathit{Ordering}'' \\
 & & \mathbf{inst}.\mathit{ho}_3 & = & @\mathit{ho}_2.\mathit{res} \\
 & & \mathit{ho}_3.\mathit{arg} & = & @r.\mathit{dleaves} \\
 & & \mathbf{lhs}.\mathit{depth} & = & \underline{\mathit{succ}}\ @\mathbf{loc}.a_1 \\
 & & \mathbf{loc}.a_1 & :: & \widehat{\mathit{Int}} \\
 & & \mathbf{loc}.a_1 & = & \underline{\mathit{max}}\ @l.\mathit{depth}\ @r.\mathit{depth} \\
 & \mid \mathit{Leaf} & \mathbf{lhs}.\mathit{dleaves} & = & \underline{\mathit{singleton}}\ @n.\mathit{self} \\
 & & \mathbf{lhs}.\mathit{depth} & = & \underline{0} \\
\mathbf{sem}\ \mathit{Ordering} & \mid \mathit{EQ} & \mathbf{lhs}.\mathit{res} & = & \mathit{EQ}' \\
 & \mid \mathit{GT} & \mathbf{lhs}.\mathit{res} & = & \mathit{GT}' \\
 & \mid \mathit{LT} & \mathbf{lhs}.\mathit{res} & = & \mathit{LT}' \\
\mathbf{sem}\ \mathit{Ordering}' & \mid \mathit{EQ}' & \mathbf{lhs}.\mathit{res} & = & \mathit{EQ}''\ @\mathbf{lhs}.\mathit{arg} \\
 & \mid \mathit{GT}' & \mathbf{lhs}.\mathit{res} & = & \mathit{GT}''\ @\mathbf{lhs}.\mathit{arg} \\
 & \mid \mathit{LT}' & \mathbf{lhs}.\mathit{res} & = & \mathit{LT}''\ @\mathbf{lhs}.\mathit{arg} \\
\mathbf{sem}\ \mathit{Ordering}'' & \mid \mathit{EQ}'' & \mathbf{lhs}.\mathit{res} & = & @\mathit{dl}.\mathit{self}\ \underline{+\!\!+}\ @\mathbf{lhs}.\mathit{arg} \\
 & \mid \mathit{GT}'' & \mathbf{lhs}.\mathit{res} & = & @\mathit{dl}.\mathit{self} \\
 & \mid \mathit{LT}'' & \mathbf{lhs}.\mathit{res} & = & @\mathbf{lhs}.\mathit{arg}
\end{array}
$$

Figure 7.6: Minimal higher-order attribute grammar code corresponding to deepest leaves example

value has to be given a name explicitly (*loc*.$a_1$ in this case).

For the computation of *dleaves* the result of the $\underline{compare}$ function is instantiated as a higher order child $ho_1$. The synthesized attribute *cmpres* represents the function that, given the deepest leaves of the left and the right child, returns the deepest leaves. This value is instantiated as a higher order child $ho_2$ and the deepest leaves of the left child are passed as an argument. The result is again instantiated as a higher order child $ho_3$ and the deepest leaves of the right child are passed as the second argument. The synthesized *dleaves* value is finally the result of $ho_3$.

## 7.7.4 Translation

Before going into details we give an intuitive description of the construction used to translate $\lambda$ expressions to minimal higher-order attribute grammars. We assume that in all cases the initial $\lambda$ expression is well-typed. We illustrate the overview with the minimal higher-order attribute grammar code for the example in Figure 7.6.

Functions are represented by trees which have one inherited attribute for the argument and one synthesized attribute for the result. In the example, $Ordering'$ and $Ordering''$ are such trees. Note that *depth* and *dleaves* are special functions that are handled in a different way which is explained in Section 7.7.6.

We introduce a new nonterminal for every function type, and a new constructor is added to the corresponding nonterminal for each lambda abstraction. In the example $EQ'$, $GT'$, $LT'$, $EQ''$, $GT''$ and $LT''$ are such constructors. Note that the lambda abstractions corresponding to these constructors are not visible in the original $\lambda$ code but are introduced in the step described in Section 7.7.8.

For each free variable in the body of the abstraction we add a child to the corresponding constructor to store the value of the corresponding variable. In the example *dl* is such a child, which is introduced for all constructors of $Ordering''$.

For each case distinction a synthesized attribute is added to the corresponding nonterminal. In the example *depth*, *dleaves* and *cmpres* are such synthesized attributes.

A function is represented by a production of a nonterminal for functions of that type. To simulate function application using attribute grammars, the function is instantiated as higher-order child and the argument is passed as an inherited attribute. The result of the function application is returned as a synthesized attribute. In the example the higher order children $ho_2$ and $ho_3$ are used for simulating function application.

For lambda abstractions the corresponding constructor is returned, which is illustrated in the semantic rules of *Ordering*. For case distinction, the argument is instantiated as a higher order child and the result is the corresponding synthesized argument of this child. In the example this is the case for $ho_1$ and its synthesized attribute *cmpres*.

### 7.7.5 Top level declarations

We define top level declarations as the set of variables that are bound by a recursive let not inside an abstraction or case distinction. In other words, a top level declaration is a let bound variable such that all its free variables are also top level declarations. In our example, the top level declarations are *depth* and *dleaves*.

We let top level declarations be available everywhere and thus depend on the MHOAG semantics to *tie the knot*. Although this is not an essential step in the translation process, this can avoid explosion of the size of the resulting MHOAG because otherwise all top level declarations are passed around explicitly at every function call.

### 7.7.6 Recognising folds

We define *folds* to be expressions of the form

$$\lambda t : \tau .\ \mathbf{case}\ t\ \mathbf{of}\ as$$

such that the set of free variables of the alternatives *as* is empty. Intuitively, such a computation can be written in MHOAGs using a synthesized attribute and thus can be efficiently used to achieve better sharing behaviour.

The success of the overall approach depends on the number of functions that can be efficiently represented in MHOAGs. It is therefore important that the functions in the input are written as folds or can automatically be transformed into folds.

Automatically transforming the input into this form is a separate problem that we do not solve here. However, with some simple rules like the following one can already achieve good results. When $u \not\equiv t$ the $\lambda u : \tau$ can be moved inside the alternatives:

$$\begin{array}{lcl}
\lambda u : \tau .\ \mathbf{case}\ t\ \mathbf{of} & & \mathbf{case}\ t\ \mathbf{of} \\
\quad p_1 \rightarrow e_1 & \rightsquigarrow & \quad p_1 \rightarrow \lambda u : \tau .\ e_1 \\
\quad p_n \rightarrow e_n & & \quad p_n \rightarrow \lambda u : \tau .\ e_n
\end{array}$$

### 7.7.7 Identifying recursive calls

A call of the form $f\ a$ where $a$ is a variable and $f$ is a fold should have special treatment. This is where the better sharing behaviour is obtained. Intuitively, usually a function is represented by some tree structure in the MHOAG code. For evaluating the application the function is instantiated as a higher order child and its argument is passed as an inherited attribute.

In the case that the function is a fold, its argument is a tree itself and thus a copy of this tree needs to be passed as inherited attribute. However, this copy of the tree is then immediately instantiated as a higher order child to retrieve the synthesized attribute for the case distinction. But when the argument is a child of the current node, the synthesized attribute for the case distinction is already available. Furthermore, when multiple computations use the same synthesized attribute the result is now shared. One place where is happens is when the same function is called multiple times with the same argument.

### 7.7.8 Lambda lifting in case alternatives

Case expressions are translated by instantiating the expression as a higher order child and adding the semantics of the alternatives to the data type using a synthesized attribute. The result is that the semantic functions of the alternatives are in a different context and thus the variables that are bound by the environment of the case are not in scope anymore.

The solution for this is to perform lambda lifting of the alternatives. First all free variables of the alternatives together are gathered. Then a lambda abstraction is introduced in each alternative for each free variable. Finally, the variables that contain the values in the context of the case are applied to the result of the case expression.

As an example, consider the following expression, where $v_1 : \tau_1$ and $v_2 : \tau_2$ are bound in the context of the case:

$$\begin{aligned}&\mathbf{case}\ e\ \mathbf{of}\ C_1 \ldots \ \to \ldots\ v_1 \ldots\\ &\qquad\qquad C_2 \ldots \ \to \ldots\ v_2 \ldots v_1 \ldots\end{aligned}$$

This example is translated to the following, operationally equivalent, code:

$$\begin{aligned}&(\mathbf{case}\ e\ \mathbf{of}\ C_1 \ldots \ \to \lambda v_1 : \tau_1.\ \lambda v_2 : \tau_2.\ \ldots\ v_1 \ldots\\ &\qquad\qquad C_2 \ldots \ \to \lambda v_1 : \tau_1.\ \lambda v_2 : \tau_2.\ \ldots\ v_2 \ldots v_1 \ldots)\ v_1\ v_2\end{aligned}$$

For recursive calls of the form $f\ a$ as identified in Section 7.7.7 where $a$ is a free variable in the alternative, a specialised rule is defined. Instead of abstracting over the free variable itself, we abstract over $f\ a$. This specialised rule will make sure that whenever a folding function is called on a substructure, this fold will not be performed again. If we would not have this specialised rule it could be the case that a copy of the substructure needs to be passed to one of the case alternatives after which it is instantiated again as a higher-order child.

### 7.7.9 Translation rules

In Figure 7.7 we show the translation rules for the translation we have described. The translation scheme is of the form $\Gamma; \Delta \vdash_N^P e \rightsquigarrow \varphi; \psi$ and consists of seven elements; $\Gamma$ is the environment in which $\lambda$ variables are mapped to attribute references, $\Delta$ is an application context containing an ordered list of attribute references that still need to be applied, $P$ and $N$ are the current production and nonterminal respectively, $e$ is the $\lambda$ expression that is being translated, $\varphi$ is the resulting attribute reference and finally $\psi$ is the resulting MHOAG.

Some details in the rules have been omitted. In the translation process a nonterminal is introduced for every function type in the $\lambda$ expression. The function *ntp* is used in the translation rules to get the nonterminal name for the given expression. However, as the expression can contain variables that are bound outside this sub expression, this must also depend on some environment containing all bound variables and types. As this environment needs to be passed around in all rules but is not an essential part of the translation itself, we have not made it explicit.

In the generated MHOAG code type signatures for the **inst** and **loc** cases are left out. Again, the type information can in all cases be inferred, but is left out to keep translation rules simpler. Furthermore, only the MHOAG code for generating semantic terms is shown, and we leave the generation of the data types corresponding to the $\lambda$ data types and the attribute declarations implicit.

**Variables** When the application context is empty, for variables the rule VAR.E is used, which performs a lookup of the variable in the environment.

In case of a nonempty application context, the rule VAR.A is used. In this case the variable represents a function and can thus be instantiated as a higher order child. The argument of the function, which is the first value from the application context, is passed as an inherited attribute *arg*. The result of the application is returned in the synthesized attribute *res*.

**Lambda abstraction** For a lambda abstraction and a nonempty application context the rule LAM.A is used to bind the abstracted variable to the variable reference

$$\dfrac{}{\Gamma;\epsilon \vdash^{P}_{N} x \rightsquigarrow \Gamma\,(x);\epsilon}\ \textsc{var.e}$$

$$\dfrac{[x \mapsto y]\ \Gamma;\Delta \vdash^{P}_{N} e \rightsquigarrow \varphi;\psi}{\Gamma;y\ \Delta \vdash^{P}_{N} \lambda x:\tau.e \rightsquigarrow \varphi;\psi}\ \textsc{lam.a}$$

$$\dfrac{\begin{array}{c}\Gamma;\epsilon \vdash^{P}_{N} e_2 \rightsquigarrow \varphi';\psi' \\ \Gamma;\varphi'\ \Delta \vdash^{P}_{N} e_1 \rightsquigarrow \varphi;\psi\end{array}}{\Gamma;\Delta \vdash^{P}_{N} e_1\ e_2 \rightsquigarrow \varphi;\psi\ \psi'}\ \textsc{app}$$

$$\dfrac{\begin{array}{c}\alpha,\beta \text{ fresh} \\ \psi'' = \boxed{\ \mathbf{data}\,N' \mid P_\alpha\,x_1 \dots x_n} \\ \Gamma';\epsilon \vdash^{P_\alpha}_{N'} e \rightsquigarrow \varphi';\psi' \\ x_1 \dots x_n = fv\,(e) \\ \Gamma' = [x \mapsto @\mathbf{loc}.\alpha \dots x_n \mapsto @x_n.\mathbf{self}] \\ N' = ntp\,(e) \\ \psi = \boxed{\begin{array}{l}\mathbf{sem}\,N \mid P \\ \quad \mathbf{loc}.\beta = P_\alpha\,\Gamma\,(x_1) \dots \Gamma\,(x_n)\end{array}}\end{array}}{\Gamma;\epsilon \vdash^{P}_{N} \lambda x:\tau.e \rightsquigarrow \varphi;\psi\ \psi'\ \psi''}\ \textsc{lam.e}$$

$$\dfrac{\begin{array}{c}\alpha \text{ fresh} \\ \psi = \boxed{\begin{array}{l}\mathbf{sem}\,N \mid P \\ \quad \mathbf{inst}.\alpha = \varphi' \\ \quad \alpha.arg = y\end{array}} \\ \Gamma;\Delta \vdash^{P}_{N} x \rightsquigarrow \varphi';\psi'\end{array}}{\Gamma;y\ \Delta \vdash^{P}_{N} x \rightsquigarrow @\alpha.res;\psi\ \psi'}\ \textsc{var.a}$$

$$\dfrac{\begin{array}{c}\alpha,\beta \text{ fresh} \\ \psi = \boxed{\begin{array}{l}\mathbf{sem}\,N \mid P \\ \quad \mathbf{inst}.\alpha = \varphi' \\ \quad \alpha.arg = y\end{array}} \\ \Gamma;\Delta \vdash^{P}_{N} \mathbf{case}\ e\ \mathbf{of}\ a \rightsquigarrow \varphi';\psi'\end{array}}{\Gamma;y\ \Delta \vdash^{P}_{N} \mathbf{case}\ e\ \mathbf{of}\ a \rightsquigarrow @\alpha.res;\psi\ \psi'}\ \textsc{case.a}$$

$$\dfrac{\begin{array}{c}\alpha,\beta \text{ fresh} \\ (P_1 a_1^1 \dots a_1^k \rightarrow e_1) \dots (P_n\ a_n^1 \dots a_n^m \rightarrow e_n) = as \\ \Gamma_i';\epsilon \vdash^{P_i}_{N'} e_i \rightsquigarrow \varphi_i;\psi_i \\ \Gamma_i' = [a_i^1 \mapsto @ch_1.\mathbf{self}, \dots, a_i^k \mapsto @ch_k.\mathbf{self}]\ \Gamma \\ \Gamma;\epsilon \vdash^{P}_{N} e \rightsquigarrow \varphi';\psi' \\ N' = ntp\,(e) \\ \psi = \boxed{\begin{array}{l}\mathbf{sem}\,N \mid P \\ \quad \mathbf{inst}.\alpha = \varphi'\end{array}} \quad \psi'' = \boxed{\begin{array}{l}\mathbf{sem}\,N' \mid P_i \\ \quad \mathbf{lhs}.\beta = \varphi_i\end{array}}\end{array}}{\Gamma;\epsilon \vdash^{P}_{N} \mathbf{case}\ e\ \mathbf{of}\ as \rightsquigarrow @\alpha.\beta;\psi\ \psi'\ \psi''\ \psi_1 \dots \psi_n}\ \textsc{case.e}$$

$$\dfrac{\begin{array}{c}\alpha \text{ fresh} \\ \psi = \boxed{\begin{array}{l}\mathbf{sem}\,N \mid P \\ \quad \mathbf{loc}.\alpha = C\,y_1 \dots y_n\end{array}}\end{array}}{\Gamma;y_1 \dots y_n \vdash^{P}_{N} C \rightsquigarrow @\mathbf{loc}.\alpha;\psi}\ \textsc{con}$$

$$\dfrac{\begin{array}{c}\alpha \text{ fresh} \\ \psi = \boxed{\begin{array}{l}\mathbf{sem}\,N \mid P \\ \quad \mathbf{loc}.\alpha = \underline{f}\,y_1 \dots y_n\end{array}}\end{array}}{\Gamma;y_1 \dots y_n \vdash^{P}_{N} \underline{f} \rightsquigarrow @\mathbf{loc}.\alpha;\psi}\ \textsc{prim}$$

$$\dfrac{\begin{array}{c}f \text{ is a fold} \\ \beta = fsa\,(f) \\ \Gamma;\epsilon \vdash^{P}_{N} a \rightsquigarrow @\alpha.\mathbf{self};\psi\end{array}}{\Gamma;\epsilon \vdash^{P}_{N} f\ a \rightsquigarrow @\alpha.\beta;\psi}\ \textsc{fold}$$

$$\dfrac{\begin{array}{c}(f_1 = e_1) \dots (f_n = e_n) = fs \\ \Gamma';\Delta \vdash^{P}_{N} e \rightsquigarrow \varphi;\psi \\ \Gamma' = [f_1 \mapsto \varphi_1, \dots, f_n \mapsto \varphi_n]\ \Gamma \\ \Gamma';\epsilon \vdash^{P}_{N} e_i \rightsquigarrow \varphi_i;\psi_i\end{array}}{\Gamma;\Delta \vdash^{P}_{N} \mathbf{let}\,fs\ \mathbf{in}\ e \rightsquigarrow \varphi;\psi\ \psi_1 \dots \psi_n}\ \textsc{let}$$

Figure 7.7: Translation from $\lambda$ to a minimal higher-order attribute grammar

from the application context. This is essentially $\beta$-reduction, but it is important to note that sharing is retained.

For an empty application context, the rule LAM.E constructs a new production $P_\alpha$ representing the abstraction. The function *fv* returns all free variables of the expression. The set of free variables returned by *fv* never contains *top level declarations* as defined in Section 7.7.5. For this an environment containing all top level declarations is available, which again we leave implicit.

For each of the free variables a child is introduced to store the value of the variable, and the environment for the translation of the rest of the expression contains exactly these variables. The return value is the instantiation of the new production with the corresponding values of the variables as children.

**Application** The rule APP translates the arguments of the application and adds the resulting attribute reference to the application context. In the translation of $e_1$ this might result in instantiating $e_1$ as a higher order child and pass the argument value as an inherited attribute.

**Recursive let** For translating a set of recursive let definitions, the rule LET translates each definition and adds all resulting attribute references to the environment which is passed to all definitions as well as the body of the let. The resulting attribute reference of the body is used as the result of the translation of the let.

**Case distinction** For case distinction there are two rules. The rule CASE.A is used in case of a nonempty application context. This rule is similar to the rule VAR.A and passes the attributes from the application context one by one as an inherited attribute to the result of the translation of the case itself.

The rule CASE.E is the most complicated one. A case distinction is implemented by introducing a new synthesized attribute $\beta$ for the result. The result of the expression itself is instantiated as a higher order child $\alpha$ and the semantics of the alternatives are added to the corresponding productions.

**Constructor** For a constructor the rule CON instantiates this constructor with the application context. The type correctness of the $\lambda$ expression guarantees that the number of arguments in the application context is exactly the number of children for this constructor.

**Primitive functions** Primitive function calls are translated by the rule PRIM to MHOAG primitive function calls. When the input $\lambda$ expression is type correct, we know that the number of arguments that are applied to the function are exactly the number of arguments that the primitive function takes. Hence, we take the full application context as the arguments for the primitive function.

**Special rules for folds** This set of rules is an embedding of $\lambda$ expressions into MHOAGs. However, to obtain better sharing behaviour the rule FOLD is a specialised rule for the recursive positions as identified in Section 7.7.7. As folds are implemented by case distinctions and case distinctions are translated to a synthesized attribute, the result of a fold call is the value of the corresponding synthesized attribute. The function *fsa* is a lookup function for finding the name of the synthesized attribute corresponding to the fold.

## 7.8 Conclusion

In this chapter we described several ways in which attribute grammars can be structured to improve the incremental behaviour. The unique value generation and inspectability can be handled automatically, but have not been implemented in the UUAGC yet. For the projection of inherited attributes automatic analyses may be possible, but manually transforming attribute grammars to that shape is the best possibility so far. The same holds for avoiding the usage of chained attributes, which is best taken into account by the attribute grammar writer while developing the attribute grammars.

One important remaining question is how much overhead is introduced with the automatic translation from Section 7.7. The automatic translation of functions with multiple parameters introduces many layers of higher-order children, for which our incremental evaluation machinery has to track dependencies. It is therefore not clear if the translation described in this chapter combined with our incremental evaluation technique leads to the desired speedups.

# 8
# Benchmarking

In this chapter we illustrate the effectiveness of the described technique for the incremental evaluation of higher-order attribute grammars. We start this chapter by giving several problems one may encounter during benchmarking which may influence the results either positively or negatively. We then present the results of the time benchmarks and end with memory benchmarks.

The first problem is finding the right input data, which we discuss in Section 8.1. Finding the right input data for benchmarks is not trivial, and we discuss three properties of the data that we believe to be important for reliable benchmarks, and two ways of obtaining such input data. The second problem we discuss is measuring execution time with the interplay of lazy evaluation (Section 8.2), and the last problem is overhead (Section 8.3). We continue this chapter with solutions to these problems and present the benchmarking results, both time measurement and a more abstract measure, for our two running examples in Section 8.4.

Another property of interest is the memory consumption of our incremental machinery compared to the non-incremental code. As extra information is stored to avoid redundant computations it is expected that the memory consumption is higher. We have claimed that, based on the implementation, the memory usage should only increase by a constant factor, which we verify with a simple memory benchmark in Section 8.5.

## 8.1 Input data

The choice of input data for benchmarking is important as it can have a large impact on the results. For example, when benchmarking sorting algorithms by giving only sorted lists as input, most algorithms are expected to finish in short time with a low memory usage. From a comparison point of view that is uninformative however, as the algorithms usually differ much more on unsorted inputs. To compare the algorithms one should therefore provide all kinds of permutations of the lists, for example also including duplicate elements.

We identify three properties of the input data that we consider to be important for constructing benchmarks: correctness (Section 8.1.1), appropriateness (Section 8.1.2), and distribution (Section 8.1.3) of the data. From our experience with benchmarking our incremental evaluation machinery we found that satisfying these three properties is important in order to prevent drawing wrong conclusions, and that satisfying these properties seems to be sufficient for creating reliable benchmarks. Furthermore, we describe how we can obtain input data using existing data (Section 8.1.4) and using data generation (Section 8.1.5), and the types of problems arising from obtaining data in such ways.

### 8.1.1 Correctness

The input data should be *correct* with respect to the input specification of the program to be tested, by which we mean that it should lead to a positive execution result. This property may seem obvious, but in the context of attribute grammars we found that it is important to make sure that this property holds.

A problem with the AST definitions as a data type is that only syntactic structure is enforced. However, programs often put more semantic restrictions on the input data that are not enforced by the AST structure. For example, benchmarking a program that expects sorted lists as input by generating random lists will not result in a fair benchmark as random lists are likely to be unsorted. For unsorted lists such programs usually give an error quickly, resulting in a short yet uninformative runtime. It is obvious that it is easier to generate an arbitrary program that conforms to the context-free grammar of a language, than to generate a well-typed program.

One solution to this problem is to use more precise types such as dependent types [Bove and Dybjer, 2009]. With dependent types it is possible to restrict the AST to precisely capture semantic restrictions like the sortedness of a list. The use of attribute grammars in combination with dependent types has been explored in [Middelkoop et al., 2011] but is outside the scope of this thesis. Here we therefore rely on the programmer creating the benchmark to take the correctness of the input into account.

### 8.1.2 Appropriateness

The second property that input data should satisfy is that is should be *appropriate*: to get benchmark results that reflect actual use of the program the input data needs to be similar to that of actual use. There are often many valid changes that are not expected to occur in actual use, and for informative results we do not want to include those changes in the benchmarks.

For example, in our case of an incremental compiler the changes to the input are the changes that the programmer makes to the source code. In practice we expect such changes to be either local, for example one function definition that has changed, or a global refactoring like the renaming of a single function.

### 8.1.3 Distribution

Related to the appropriateness property is the *distribution* of the input data. For most programs not every correct and appropriate input is equally likely to appear in practice, which needs to be taken into account in benchmarking in order to get results similar to practical use of the program.

For example, for the guestbook example one expects that inconsistencies such as a guest leaving while he was not signed in are rare in practice. Such a change is correct an appropriate because it does happen in practice, but for a good distribution we expect such change to occur only seldom. Furthermore, we expect the hotel to have maximum number of guests that can be signed in at the same time. This means that a random list of *Arrive* and *Leave* entries is not representative for the actual usage of the guestbook, even though the data is correct and the inconsistencies are appropriate in a certain sense.

### 8.1.4 Existing data

One way of obtaining appropriate input data is to take an existing data set. Such a data set can be obtained from actual usage of the program itself, or from programs that have similar inputs. For example, in the case of the UHC the input data consists of Haskell programs with changes to the source code, of which large codebases exist on websites such as `https://github.com` which hosts code repositories for the version control system Git.

Existing data usually satisfies the first two properties. When the data is taken from the actual usage of the program, the data is correct and appropriate by definition. Whether the data has a proper distribution depends on the way the data was obtained. For example, when the data is taken from a code repository the changes are expected to be larger between subsequent commits than the changes between subsequent compilation runs. Hence, to test an incremental compiler the data needs to be gathered in a different way than from the commit history.

### 8.1.5 Data generation

Another way in which input data can be obtained is by generating it with the aid of some randomised data generation tools. The main benefit of data generation is that the user has full control over the generated data and more data can easily be obtained when desired. The problem, however, is that the user needs to make sure that the properties are satisfied, which is not an easy task as we will illustrate.

As a concrete example let us consider a type inferencer for the simply typed lambda calculus. To generate correct input data for such type inferencer we need to generate closed well-typed lambda expressions. The generation of closed lambda expressions, where all variables are bound, is not hard, but many of such expressions are ill-typed and thus not correct input for the type inferencer. A possible solution is to use the type inferencer as a checker for the data generation by filtering out the ill-typed expressions, and use the remaining expressions for time and memory benchmarks.

Unfortunately, such a set of expressions does not meet the appropriateness and distribution properties. The problem is that the fraction of closed lambda expressions being well-typed is small, and that this fraction becomes even smaller when the expressions grow. As a result of this, the filtering of well-typed expressions among randomly generated ones is likely to result in only small and simple closed lambda expressions. Hence, the distribution property is not met, because we expect larger expressions in practical usage of the type inferencer. And even when somewhat larger expressions are generated, it is likely that they are not appropriate as one would expect human written expressions to have more complex forms while the randomly generated well-typed terms are likely to be of some simple form, for example where each bound variable is used at most once.

A solution to this problem is to use the approach by [Claessen et al., 2014], using a property-based data generation tool that generates random instances of data types satisfying the property which are uniformly distributed over for each size. With such an approach it is possible to generate data of a specific size that is correct, appropriate and well distributed. Unfortunately, at the time of writing this thesis the code corresponding to the paper was not yet available and hence we were not able to use such an approach here.

One remark on the previous paragraphs is that for benchmarking a type inferencer it is actually not the case that ill-typed expressions are useless: in practice the type inferencer is confronted with ill-typed expressions and it is therefore useful to include those in a benchmark. However, we do think that mixing these with well-typed expressions skews the results, and we suggest to run separate benchmarks in order to test the behaviour of the type inferencer when given ill-typed expressions.

## 8.2 Lazy evaluation

The lazy evaluation semantics of Haskell can be of great use, for example for constructing circular programs, but at the same time laziness can be problematic for measuring execution time. We discuss three problems with time measurement of Haskell programs.

Evaluation only happens when strictly necessary, which means that for the purpose of time measurement we need to force the evaluation of the result of the computation. In Haskell this can be done using the function *deepseq* from the package `deepseq`[1] which evaluates its argument to normal form. If *seq* or a similar function is used the argument is only evaluated to weak head normal form, meaning that only the outermost constructor is forced. In our case of the C$^\sharp$ compiler this would mean that only the first : of the list of instructions is forced, instead of the full list of instructions as happens with *deepseq*. The evaluation to weak head normal form usually takes only a fraction of the time of the full computation, which means that not forcing the result properly can lead to completely wrong benchmark results.

Another problem is that Haskell shares computation results to avoid redundant computations. In normal applications such behaviour can be beneficial, but with benchmarking it is often desired to explicitly perform the same computation multiple times in order to get more accurate timings. However, when the benchmark function is invoked as *benchmark* ($f\ a$) and evaluates its argument multiple times, the call $f\ a$ is only computed once. To avoid this problem the benchmarking library `criterion`, which we introduce in Section 8.3, takes both $f$ and $a$ as separate parameters and performs the function application multiple times to avoid sharing of the result.

A problem specifically arising with the benchmarking of incremental computation is that we want separate measurements of different parts of the computation.

[1] `http://hackage.haskell.org/package/deepseq`

However, these different parts are not completely independent as the result of an incremental change can only be computed from the state that is produced by the initial computation. Because of lazy evaluation the state is only computed the first time it is used, and separate measurements of the initial run and the incremental change can therefore lead to wrong result in which the time for evaluating the state is measured only once instead of in every benchmark run.

The solution we use is to perform two benchmarks: one benchmark for the initial computation, and one benchmark for the initial computation *followed by* an incremental change. The time spent on the incremental change (and because of the lazy semantics the time spent on actually building up the state from the initial computation) can then be computed from the difference between these two measurements.

## 8.3 Overhead

The intended use of the benchmarking is to compare the runtimes of our incremental evaluation machinery to the non-incremental evaluation machinery in order to evaluate the effectiveness of the incremental approach. To run reliable benchmarks extra boilerplate code is used, which can introduce runtime overhead and thus have a negative influence on the result. We discuss two sources of overhead and how we avoid problems with these sources of overhead.

### 8.3.1 Diff overhead

Throughout this thesis we have assumed that changes to the AST are given by an external process, for instance a structure editor. In order to run reliable benchmarks we need a static data set, which means that the changes can not come directly from a structure editor. Instead, we either obtain data from a data source or generate it, and then use this data to perform time measurements.

To obtain changes from static data we can use a *diff* algorithm such as the one described in Section 4.6.3. For example, in the case of the C$^\sharp$ example such *diff* function can return the changes between the ASTs of two files. The attribute grammar evaluation machinery can then be given the AST of the first file and the change, such that the resulting SSM code for the second file can be computed incrementally.

Using a *diff* algorithm does however not come for free: it uses time and memory to compute the difference. When the time used for computing the diff is only a small fraction of the time used for the attribute evaluation there is no problem, but when the *diff* is expensive and dominates the total runtime the benchmark results are skewed. As the *diff* we have implemented is of the latter kind, we do not include it in the time measurements at all. Instead, before running the benchmark we compute the difference and force its evaluation, such that in the attribute evaluation the changes to the AST are already computed.

### 8.3.2 Benchmarking overhead

To measure the runtime of different programs in a reliable way, each of them is run many times on the same input and the average runtime is used for comparison. Furthermore, statistics on the distribution of runtimes such as standard deviation are computed to ensure that no unpredictable side effects occur. In our experience this has proven to be useful for the discovery of the problems discussed in Section 8.2.

The boilerplate code used to run benchmarks in such way can introduce overhead. We use the `criterion`[2] package for running the benchmarks, which performs all of the time measurement and statistics computation, and avoids the problem of including benchmarking overhead in time measurements. Furthermore, it ensures that benchmarks results are evaluated to normal form and mitigates the problems with undesired sharing.

## 8.4 Results

In this section we show the results of the time benchmarks for our two examples. As indicated we use the `criterion` package for running the benchmarks, and we use the `barchart`[3] package for visualising the results. We show for each of the two benchmarks in what way the data was constructed, what the runtimes of the different approaches are as measured with `criterion`, and a table with the number of evaluation steps.

An *evaluation step* is a measurement unit that we define in order to estimate the potential effectiveness of our incremental evaluation machinery. The problem with time measurement is that all runtime evaluation is measured at once, which does not give a good indication of where the time is spent. Instead, our evaluation steps specify three different types of operations that occur at runtime: the evaluation of a semantic rule, the equality check on an inherited attribute, or an attribute grammar step. Attribute grammar steps are all other operations that the attribute grammar evaluation machinery performs, for example the invocation of a visit or the propagation of a change. The purpose of the evaluation steps is to give a different view

[2]`http://hackage.haskell.org/package/criterion`
[3]`http://hackage.haskell.org/package/barchart`

on the type of operations occurring at runtime, and we do not know the relation between the different operations and the actual runtime.

### 8.4.1 Guestbook example

The guestbook example has been constructed for the use in this thesis and therefore there is no existing data. Although there probably exist similar examples for which data can be found, it can be hard to convert the data to the correct format and get the expected distribution. We have therefore generated the data ourselves.

**Data generation** The first challenge in generating an arbitrary guestbook is to generate names. While it is not important for our application that the names are existing names, we do want that duplicate names appear sometimes. We have therefore chosen to use a list of the 1 000 most common surnames in the United States[4] and generate names by picking an arbitrary element from that list.

We have generated a list of 50 000 guestbook entries starting with the oldest entry. For each entry we generate an *Arrive* entry with frequency 15, a *Leave* entry of a guest that is signed in with a frequency of 10, and a *Leave* entry of an arbitrary guest with a frequency of 1. We have set the maximum number of guests in the hotel to 50 and the *Arrive* entry is only generated when the total number of signed in guests is smaller than 50. Furthermore, the *Leave* entry can only be generated when there are some guests signed in. This method of generating returns a guestbook with 2277 invalid reviews out of the 25461 *Leave* entries in total.

**Timings** In Figure 8.1 we show the time benchmark result for the guestbook example. We have run four different benchmarks: initial, delete, insert leave and insert arrive. The *initial* benchmark is the computation of the grade for the initial guestbook with 50 000 elements. The other three are changes made to the guestbook, which is either deleting or inserting an element at the beginning of the list, in an arbitrary position between the tenth and twentieth element. Each of those changes is relative to the initial guestbook.

We have run three different approaches on these benchmarks.

*GuestbookAG_Lazy* is the lazy translation of the attribute grammar definition of the guestbook and has no support for incremental changes. In order to perform an incremental step the initial guestbook is first altered and then the attribute grammar

[4]`http://www.census.gov/topics/population/genealogy/data/2000_surnames.html`

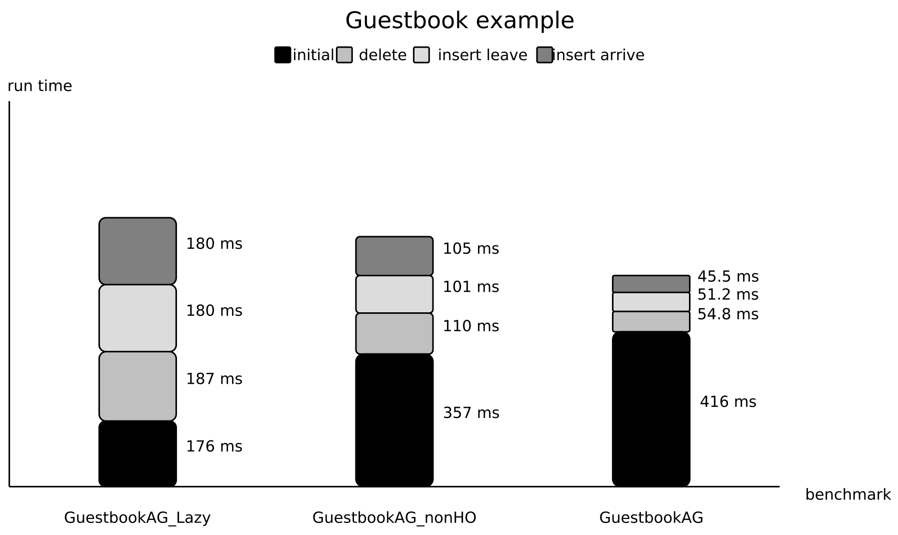

Figure 8.1: Benchmark results of the guestbook example

evaluator is called again with the new guestbook. As expected, all four runs take roughly the same time.

*GuestbookAG_nonHO* is the approach from Chapter 5 which performs incremental evaluation, but does not track changes to higher order children. Hence, in this case the construction of the list of true grades is incremental but the list is always completely traversed in order to compute the average. As we see, the initial run takes about twice as much time as the initial run of the lazy evaluation, which is the overhead introduced by the extra bookkeeping.

Finally, *GuestbookAG* is the incremental evaluator resulting from the technique described in Chapter 6 which is the final result of this thesis. Again, the initial run takes much more time than the lazy version because of the overhead. The computation of the results for the changes are however much faster and after three changes the incremental version is faster than the non-incremental version.

What is not visible is the figure is that the incremental code is actually several orders of magnitude faster. All changes are relative to the initial run and to measure these times we have performed the initial run together with one of the changes and subtracted the runtime of the initial run. However, because of lazy evaluation the state that is build up in the initial run is not actually evaluated in the initial

| | Semantic rules | AG steps | Equality checks |
|---|---|---|---|
| initial | 243169 | 146588 | 0 |
| delete | 38 | 146749 | 0 |
| insert leave | 42 | 146745 | 0 |
| insert arrive | 30 | 146687 | 0 |
| insert + delete | 60 | 146956 | 0 |

Table 8.1: Evaluation steps for the guestbook example

run. After a change the code still needs to evaluate that state, and thus needs to traverse a data structure with a size linear in the length of the input guestbook. The incremental change is therefore a linear computation. However, this is only the case for the first incremental change, and subsequent changes are much faster. For example, a delete after an insert takes only 40 μs, which is a 4500x speedup compared to the lazy variant. Note that such large speedups can only occur in this example when the change happens to be at the beginning of the list.

**Evaluation steps** Apart from the time measurements, we have gathered statistics on the evaluation steps in a separate benchmark run (Table 8.1). The first four benchmarks are the same as the time benchmarks, the last line is an example of two subsequent changes. For each of the changes the number of attribute grammar steps taken is in the same order of magnitude as the initial run, due to the fact that because of lazy evaluation the internal state needs to be evaluated. However, only a small number of semantic rules is evaluated, which is where the speedup comes from. Note that no equality checks occur because there are no inherited attributes in the guestbook example.

Note that for the two subsequent changes the numbers are combined, which means that for the delete after an insert only $146956-146745 = 211$ additional attribute grammar steps are taken and 30 semantic rules are evaluated. When future changes happen at the beginning of the list similar numbers are expected, leading to large speedups.

## 8.4.2 C♯ example

We have used the C♯ compiler throughout this thesis as an example that is closer to the real-world use of attribute grammars for compiler construction than the guestbook example. For the benchmarks we therefore want to use real-world data, which

```
int numdivisors (int n) {
   int c;
   int j = 2;
   int ret = 1;
   while (j ∗ j ⩽ n) {
      if (n % j ≡ 0) {
         c = 0;
         while (n % j ≡ 0) {
            c = c + 1;
            n = n / j;
         }
         ret = ret ∗ (c + 1);
      }
      j = j + 1;
   }
   if (n > 1) ret = ret ∗ 2;
   return ret;
}
```

Figure 8.2: C$^\sharp$ function for computing number of divisors

in this case are C$^\sharp$ programs. Unfortunately, our toy compiler only supports a limited subset of C$^\sharp$, for example the support for *String*s is missing, and thus only a very limited set of C$^\sharp$ programs can be used for benchmarking. We have not been able to find any non-trivial C$^\sharp$ programs that are supported by our compiler, and we have used data generation instead.

**Data construction** As described in Section 8.1.5 the automatic generation of programs is not an easy task. Writing generators for C$^\sharp$ programs that generate valid programs that are similar to those appearing in real-world use of C$^\sharp$ is a hard task. For illustration purposes of the techniques from this these we have therefore chosen to write several C$^\sharp$ files by hand.

We started by constructing a moderately large file (260 lines) containing several arithmetic functions. The largest of those functions is *numdivisors* which computes the number of divisors of its **int** parameter (Figure 8.2). From this file we have then created three new files that each differ from the initial file in a specific way. The first

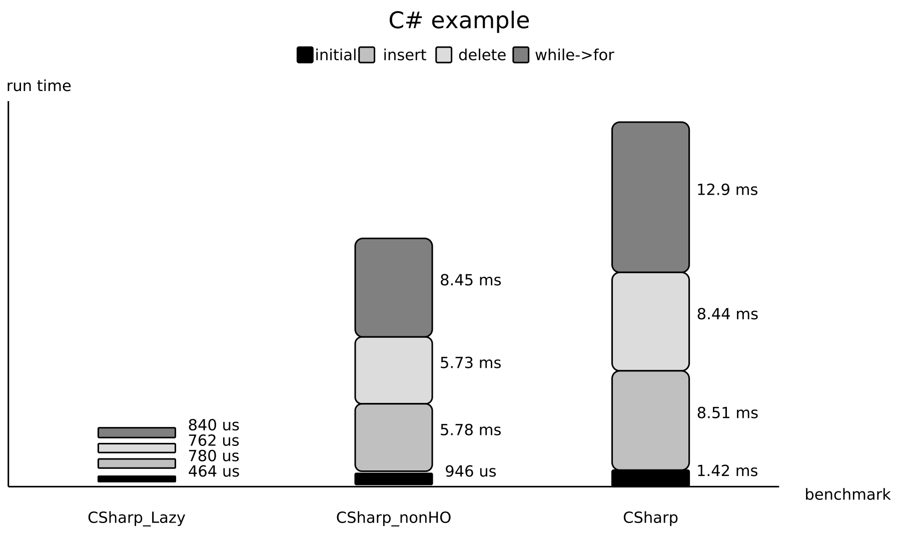

Figure 8.3: Benchmark results of the C$^\sharp$ example

change is the addition of a new function to the file, the second is the deletion of a function from the file, and the last is the refactoring of changing all **while** constructs to **for** constructs. The initial file contains a total of 16 **while** loops, and the result of computing the *diff* between the version with **while** and with **for** loops contains 77 insertions.

**Timings** The results of the time benchmarks are shown in Figure 8.3. The three different approaches are similar to those of the guestbook example, and the initial run with the three different types of changes have been explained in the previous paragraph. From this figure our incremental evaluation machinery does not look promising: the runtime has increased up to a factor 15.

**Evaluation steps** In Table 8.2 we show the evaluation steps for these benchmarks, which are more promising than the time measurements. For the insert and delete the number of evaluated semantic rules and the number of attribute grammar steps have decreased, while only a few equality checks need to be performed.

For the change from **while** to **for** the number of semantic rules that are evaluated is actually lower than necessary; in a normal run on the file with all **while**s replaced

| | Semantic rules | AG steps | Equality checks |
|---|---|---|---|
| initial | 1919 | 6933 | 0 |
| insert | 46 | 3220 | 5 |
| delete | 52 | 3094 | 4 |
| while→for | 2308 | 15556 | 148 |

Table 8.2: Evaluation steps for the C$^\sharp$ example

by **for** there are 3511 semantic rules evaluated and 8509 attribute grammar steps taken. The number of evaluated semantic rules has thus decreased by 35%, but the price that is paid is that many more attribute grammar steps are taken and many equality checks are performed.

From the benchmark results for our two example we can not draw reliable conclusions, but the results suggest that the equality checks are too expensive. In order to get actual speedups it is therefore essential to have more efficient equality checks, for example using a technique such as hash-consing.

## 8.5 Memory consumption

The main goal of this thesis is to minimise the time needed for recomputing the attributes after a change to the AST. To achieve that goal visits are memoized and thus more memory is used. We have claimed that the memory usage only increases by a constant factor when using our techniques for incremental evaluation, which we evaluate in this section.

In order to check the memory usage over time we create a memory benchmark based on the guestbook used for time measurements with 50 000 entries. During the executing of the memory benchmark we alternate between inserting and deleting an entry in the guestbook, such that the total size of the guestbook stays constant.

In Figure 8.4 we show the heap profile of the benchmark. There is a peak in memory usage after the initial run, where about 60 megabytes of data is used. This peak is due to lazy evaluation, which results in the internal state not being evaluated until it is used when the first incremental change happens. After that change has happened the memory usage stays constant at about 35 megabytes. In comparison, the lazy translation of the evaluator uses about 25 megabytes of memory at its peak, which means that the memory usages is indeed within a constant factor and stays constant over time when the size of the AST stays constant.

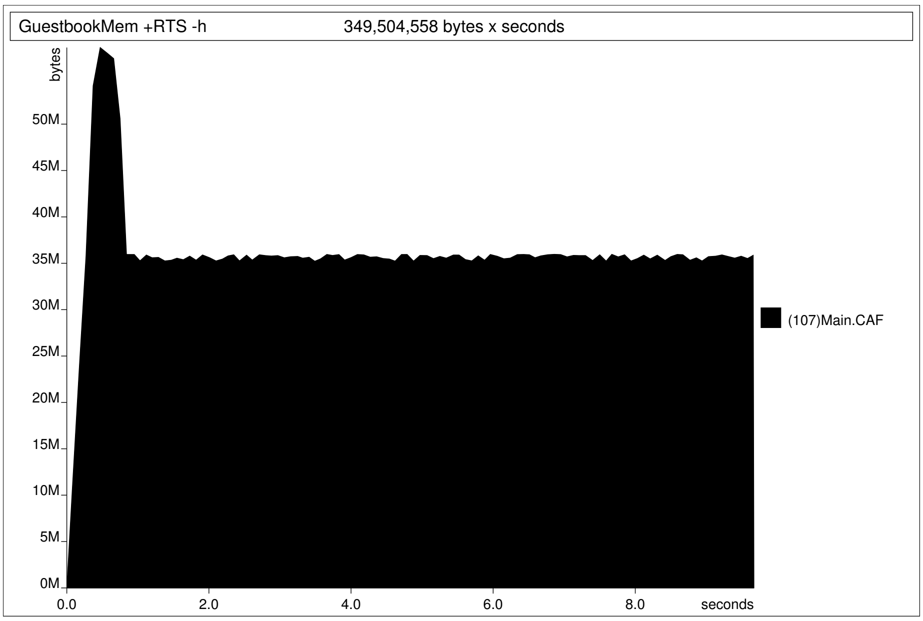

Figure 8.4: Memory profile for the guestbook example with a sequence of 1 000 delete and inserts

# 9 Discussion and conclusion

In this thesis we have described our technique for the incremental evaluation of higher-order attribute grammars. We have illustrated and evaluated the technique using two running example. For the guestbook example the results are promising, while for the C$^\sharp$ example the time benchmarks do not show the desired results. The measurements for the evaluation steps do show that certain computations are avoided, but it is clear that more work is necessary in order for this approach to be usable in practice.

We start this chapter by discussing the relevant related work (Section 9.1) and other approaches to the incremental evaluation of attribute grammars. Future work is discussed in Section 9.2 and Section 9.3 and finally we conclude this thesis in Section 9.4.

## 9.1 Related work

The automatic construction of incremental programs from declarative specifications such as attribute grammars is not a new research topic and many papers related to this topic have been written. In this section we highlight the most important related work and describe the relation to the work in this thesis.

### 9.1.1 Simple change propagation

[Demers et al., 1981] and [Yeh, 1983, Yeh and Kastens, 1988] describe a similar technique for the incremental evaluation of attribute grammars by constructing the dependency graph for the attributes and propagate changes through this graph in a breadth-first way. Conceptually this approach is similar to our approach, but it has a major drawback: higher-order attributes are not supported. Note that this restriction is not surprising as these papers were written before the introduction of higher-order attribute grammars.

The problem with this approach in combination with higher-order attributes is the same as the one we have described in this thesis. Whenever a higher-order attribute changes the connection with the previous instantiation of the higher-order child is lost and all attributes are recomputed.

A difference with our approach is that only full subtree replacements are supported, and extending the approach with derived dependency tracking for higher-order children as we do in this thesis is therefore not expected to be effective.

### 9.1.2 Synthesizer generator

A tool for constructing structure editors from declarative attribute grammar specifications is the *synthesizer generator* [Reps and Teitelbaum, 1984]. The incremental evaluation of higher-order attribute grammars in the context of the synthesizer generator is described by [Teitelbaum and Chapman, 1990]. Their algorithm performs a restricted form of derived dependency tracking for higher-order children to ensure sharing of previously computed attributes for higher-order children, such that if a subtree of a higher-order child is unchanged then no recomputation happens.

Although this approach does support higher-order attributes, it is not optimal: for some types of changes the higher-order children change in such way that recomputation happens for attributes that have not changed, while our approach is optimal in the number of attribute values being recomputed. However, the overhead of our approach is high and it is not known how often in practice changes occur for which the synthesizer generator does redundant recomputations.

### 9.1.3 Function caching

In the context of a purely functional implementation of higher-order attribute grammars, [Vogt et al., 1991] and [Saraiva et al., 2000] describe a technique for the incremental evaluation of higher-order attribute grammars using *function caching*. With function caching a global memoization table is constructed in which every visit function call is cached. Whenever a visit is performed an entry is inserted into the table with as key the subtree for which the visit was performed and the inherited attributes, and as value the synthesized attributes. For the next visit to the same the subtree, this subtree and inherited attributes are used for a table lookup, and the visit is only performed when no cache entry exists.

The advantage of function caching is that higher-order attributes are supported in a natural way without restrictions on their construction. However, there are several drawbacks to this approach. First of all there is the overhead of the lookup, which uses the whole subtree and the values of the inherited attributes as key in a large table and thus needs to perform comparisons between subtrees and attributes in order to find the corresponding element in the table. The comparison between the subtrees is however done very efficiently since all values are constructed using *hash-consing*, which ensures a unique representation in memory for each subtree such that instead of comparing the values of the subtrees themselves, only the pointers need to be compared. However, even with hash-consing many comparisons need to be performed.

Another drawback of this approach is the memory consumption. Whenever a visit is performed because its arguments are different from the previous evaluations, an entry is added to the memoization table. To avoid infinite growth of this table during the execution a *purging strategy* is used to delete entries from the table and keep its memory usage limited. Such a purging strategy can never predict future lookups to the memoization table and can therefore never be perfect; for each strategy there exist cases in which entries are removed which could have avoided recomputation.

Finally, implementing function caching efficiently requires language support. This restriction highly limits the applicability as support for function caching does not exist in modern functional languages such as Haskell.

### 9.1.4 Self-adjusting computation

Related to the incremental evaluation of attribute grammars is the technique of *self-adjusting computation* [Acar, 2005] which works for arbitrary functional programs. In a self-adjusting program the parts of the input that can change during the execution are marked, and each computation that uses such changeable data must use special primitives for reading and writing the data. During the execution of such a program the runtime machinery builds up a dependency graph of all changeable data and propagates changes efficiently through the program using that dependency graph.

The main difference to our approach is that, because the approach is more general, the dependency graph is built dynamically during execution instead of at compile time. The advantage of using a more restricted form of programs such as attribute grammars is that dependency information is available at compile time, which can then be used to statically generate more efficient evaluators. Other than this the approach is quite similar in the sense that the same kind of restrictions apply; equality is used for deciding whether or not a computation has changed, and computations need to have a specific structure in order for the technique to be effective.

### 9.1.5 Adapton

The work on Adapton [Hammer et al., 2014], a core calculus for incremental computation and corresponding OCaml library, is a more recent advancement in the field of incremental computation. The core calculus contains special operations for propagating changes through computations in a demand driven way. Because this approach is more recent than most of our work we have not used their approach in combination with attribute grammars, but it would be interesting future work to see whether the approaches can be combined.

### 9.1.6 Computational complexity

For the evaluation of our approach we have used time and memory benchmarks, and evaluation steps. These measures give a rough indication of the potential improvements of our technique, but the computational complexity could be analysed more precisely using the approach from [Çiçek et al., 2015], who develop a refinement type system for proving asymtotic bounds on incremental computation time.

## 9.2 Future work

Throughout this thesis we have already discussed several shortcomings of our approach and possible solutions to these shortcomings. In this section we list the most important parts of future work that we believe to be necessary for our approach to become applicable in practice.

### 9.2.1 Other backend

A major problem with our current approach is that the overhead is large. For the C$^\sharp$ example our incremental evaluation machinery does evaluate fewer attributes than the non-incremental evaluation machinery, but the price paid is a runtime which is 15x higher in some cases. With such an overhead the technique is not practically usable, but we believe that there are many improvements to be made.

In particular, we have implemented our evaluation machinery in Haskell using several compiler extensions specific to the GHC. It can be the case that the time overhead is related to particular implementation details in Haskell, and not to the conceptual solution presented in this thesis. Haskell is a general purpose language and the GHC can therefore fail to optimise some parts of the program for which more information is known in the attribute grammar compiler. For example, because our attribute grammars are ordered the attributes may be computed in a strict way instead of using lazy evaluation.

Another possibility, that we suggest as future work, is to let the attribute grammar compiler generate code in a backend language more tailored towards incremental computation, for example the core calculus of Adapton.

### 9.2.2 Automatic projection

As discussed in Section 7.6 the automatic projection of inherited attributes can help to avoid redundant computations. We believe that simple manual annotations from the attribute grammar programmer specifying properties of the attribute can be enough to automatically perform the projections. Furthermore, for common types such as *Map* these properties can even be even built in to the attribute grammar compiler. It is future work to investigate such an approach and verify whether useful results can be obtained in such way.

### 9.2.3 Inspectability

Our technique works only when all higher-order attributes are *inspectable*, by which we mean that the semantic rules for higher-order attributes are of a restricted form consisting only of constructors, attribute references and constants. In practical applications this is however not the case, and in Section 7.5 we have shown that arbitrary Haskell expressions can be automatically translated to attribute grammar code to solve this problem. The question does however arise whether this approach is feasible on a real-world scale.

Another possibility is to extend the language that can be used for the semantic rules to support a wider range of programs. We have already added the support for **if-then-else**-statements and it could be extended with for example pattern matching and standard operators. We suggest that in future work a language is designed which is inspectable such that the attribute grammar compiler can track changes to higher-order attributes, but is also expressive enough for many programs to be written in an appealing way.

### 9.2.4 Granularity

Our approach caches visit functions for each node in the AST, but in practice attribute computations are often small and likely to have a low runtime. We expect that our approach is much more effective when the caching only happens in specific nodes in the AST. This does result in some redundant computations, but if these computations are cheaper than the overhead of avoiding them efficiency is gained.

A profiling based approach such as [Söderberg and Hedin, 2011] use can be beneficial for automatically deciding where in the AST to do the caching. Such an approach does require real world usage data which is not always easy to obtain as we have discussed in Chapter 8. In particular, the attribute grammar programmer should provide such data which can then be used by the attribute grammar compiler to optimise the program.

### 9.2.5 Synthesized attribute equality

In our approach we only efficiently handle cases where the inherited attributes have not changed, but the same trick might be applied to the synthesized attributes. The inputs of a visit of a node are not only the inherited attributes of that visit, but also the synthesized attributes for the visits of the children of that node that are invoked. After a change to a child a visit thus is re-evaluated, even though the values of these synthesized attributes may be the same. By also requiring equality instances for our synthesized attributes it would be possible to also avoid such recomputations. However, the values of the synthesized attributes of the child are only known after doing the part of a visit in which the inherited attributes of that child are computed and the child visit is invoked. It is therefore not immediately clear how the visit functions should be changed to support this form of incrementality.

### 9.2.6 Serialisation

In this thesis we have considered incremental evaluation as a program for which the input changes over time during the execution, for example because the user edits a program in a structure editor. For the use in applications such as an incremental compiler these changes do however happen in different executions of the program. In order to support such cases the internal state of the incremental evaluation machinery needs to be stored in a file, but with our current approach using closures that is not trivial. It is therefore future work to make sure that our approach supports the serialisation of the internal state such that it can be written to the file system to be used in future executions of the program.

### 9.2.7 Correctness and soundness

An important assumption of the incremental evaluation machinery is that an incremental run returns the same result as its non-incremental counterpart. Although we have verified that our techniques work correctly for the examples, we have not yet formally proved soundness or correctness of our incremental transformation.

### 9.2.8 Utrecht Haskell Compiler

For the concrete case of the UHC there is some more engineering work to be performed next to the above suggestions. In particular, the UHC is not constructed as a single attribute grammar, but instead consists of separate attribute grammar computations which are composed at the Haskell level. In other words, higher-order children are not used for the implementation of different compiler phases but instead the higher-order kind of construction is manually implemented in Haskell.

This implementation detail is not so important for regular attribute grammar evaluation because the attribute grammar compiler does not perform optimizations that need whole program analysis. However, for our incremental evaluation machinery it is essential that the whole program is written as an attribute grammar in order to perform the dependency tracking. Unfortunately it is not trivial to rewrite the UHC in such way as other bookkeeping is performed at the top level in Haskell, and the restructuring of the UHC to consist of a single attribute grammar is therefore future work.

## 9.3 Long term future work

So far we have described the construction of incremental attribute grammar evaluation machinery by using Haskell as a backend to the attribute grammar compiler. However, when using more low-level runtime machinery we could also go the other way: use attribute grammars as a backend for the UHC. The theoretical possibility follows from Section 7.5 in which we have already described how arbitrary Haskell programs can be translated to attribute grammars, and it would be interesting to see how well that works in practice.

Such an approach gives automatic tupling for free already because of the use of attribute grammars, and if the incremental evaluation works as desired this gives incremental evaluation of arbitrary Haskell programs for free. With the current results of this thesis we do not expect any practically usable result from such an approach, but when some of the issues mentioned above are solved it can be an interesting research approach to incremental computation for functional languages in general.

## 9.4 Conclusion

The work in this thesis provides a new technique for the incremental evaluation of higher-order attribute grammars. We have shown that it can be effective in some cases, but that more work is necessary in order for this technique for be usable in practice. In this chapter we have compared our technique to other approaches and described the differences. We have suggested several topics for further research that can improve the technique and we believe that with those suggested improvements attribute grammars are a valuable tool for the construction of programs with incremental evaluation for free.

# Bibliography

[Acar, 2005] Acar, U. A. (2005). *Self-Adjusting Computation*. PhD thesis, Carnegie Mellon University.

[van Binsbergen et al., 2015] van Binsbergen, L. T., Bransen, J., and Dijkstra, A. (2015). Linearly ordered attribute grammars: With automatic augmenting dependency selection. In *Proceedings of the 2015 Workshop on Partial Evaluation and Program Manipulation*, PEPM '15, pages 49–60, New York, NY, USA. ACM.

[Bird, 1984] Bird, R. S. (1984). Using circular programs to eliminate multiple traversals of data. *Acta Informatica*, 21:239–250.

[Bove and Dybjer, 2009] Bove, A. and Dybjer, P. (2009). Dependent types at work. In Bove, A., Barbosa, L., Pardo, A., and Pinto, J., editors, *Language Engineering and Rigorous Software Development*, volume 5520 of *Lecture Notes in Computer Science*, pages 57–99. Springer Berlin Heidelberg.

[van den Brand and Klint, 2007] van den Brand, M. G. and Klint, P. (2007). ATerms for manipulation and exchange of structured data: It's all about sharing. *Inf. Softw. Technol.*, 49(1):55–64.

[Bransen et al., 2014a] Bransen, J., Dijkstra, A., and Swierstra, D. (2014a). Exploiting attribute grammars to achieve automatic tupling. Technical Report UU-CS-2014-013, Department of Information and Computing Sciences, Utrecht University.

[Bransen et al., 2014b] Bransen, J., Dijkstra, A., and Swierstra, S. D. (2014b). Lazy stateless incremental evaluation machinery for attribute grammars. In *Proceedings of the ACM SIGPLAN 2014 Workshop on Partial Evaluation and Program Manipulation*, PEPM '14, pages 145–156, New York, NY, USA. ACM.

[Bransen et al., 2015a] Bransen, J., Dijkstra, A., and Swierstra, S. D. (2015a). Incremental evaluation of higher order attributes. In *Proceedings of the ACM SIG-*

*PLAN 2015 Workshop on Partial Evaluation and Program Manipulation*, PEPM '15, New York, NY, USA. ACM.

[Bransen and Magalhães, 2013] Bransen, J. and Magalhães, J. P. (2013). Generic representations of tree transformations. In *Proceedings of the the 9th ACM SIGPLAN Workshop on Generic Programming (WGP'13)*, WGP '13.

[Bransen et al., 2012] Bransen, J., Middelkoop, A., Dijkstra, A., and Swierstra, S. D. (2012). The Kennedy-Warren algorithm revisited: ordering Attribute Grammars. In Russo, C. and Zhou, N.-F., editors, *Practical Aspects of Declarative Languages*, volume 7149 of *Lecture Notes in Computer Science*, pages 183–197. Springer Berlin Heidelberg.

[Bransen et al., 2015b] Bransen, J., van Binsbergen, L., Claessen, K., and Dijkstra, A. (2015b). Linearly ordered attribute grammar scheduling using SAT-solving. In Baier, C. and Tinelli, C., editors, *Tools and Algorithms for the Construction and Analysis of Systems*, volume 9035 of *Lecture Notes in Computer Science*, pages 289–303. Springer Berlin Heidelberg.

[Bryant and Velev, 2002] Bryant, R. E. and Velev, M. N. (2002). Boolean satisfiability with transitivity constraints. *ACM Trans. Comput. Logic*, 3(4):604–627.

[Cai et al., 2014] Cai, Y., Giarrusso, P. G., Rendel, T., and Ostermann, K. (2014). A theory of changes for higher-order languages: Incrementalizing $\lambda$-calculi by static differentiation. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '14, pages 145–155, New York, NY, USA. ACM.

[Cheney and Hinze, 2003] Cheney, J. and Hinze, R. (2003). First-class phantom types. Technical report, Cornell University.

[Çiçek et al., 2015] Çiçek, E., Garg, D., and Acar, U. (2015). Refinement types for incremental computational complexity. In Vitek, J., editor, *Programming Languages and Systems*, volume 9032 of *Lecture Notes in Computer Science*, pages 406–431. Springer Berlin Heidelberg.

[Claessen et al., 2014] Claessen, K., Duregård, J., and Pałka, M. (2014). Generating constrained random data with uniform distribution. In Codish, M. and Sumii, E., editors, *Functional and Logic Programming*, volume 8475 of *Lecture Notes in Computer Science*, pages 18–34. Springer International Publishing.

[Claessen et al., 2009] Claessen, K., Een, N., Sheeran, M., Sörensson, N., Voronov, A., and Åkesson, K. (2009). Sat-solving in practice. *Discrete Event Dynamic Systems*, 19(4):495–524.

[Codish and Zazon-Ivry, 2010] Codish, M. and Zazon-Ivry, M. (2010). Pairwise cardinality networks. In Clarke, E. and Voronkov, A., editors, *Logic for Programming, Artificial Intelligence, and Reasoning*, volume 6355 of *Lecture Notes in Computer Science*, pages 154–172. Springer Berlin Heidelberg.

[Cook, 1971] Cook, S. A. (1971). The complexity of theorem-proving procedures. In *Proceedings of the Third Annual ACM Symposium on Theory of Computing*, STOC '71, pages 151–158, New York, NY, USA. ACM.

[Demers et al., 1981] Demers, A., Reps, T., and Teitelbaum, T. (1981). Incremental evaluation for attribute grammars with application to syntax-directed editors. In *Proceedings of the 8th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '81, pages 105–116, New York, NY, USA. ACM.

[Dijkstra et al., 2009] Dijkstra, A., Fokker, J., and Swierstra, S. D. (2009). The architecture of the Utrecht Haskell Compiler. In *Proceedings of the 2nd ACM SIGPLAN symposium on Haskell*, Haskell '09, pages 93–104, New York, NY, USA. ACM.

[Dirac, 1961] Dirac, G. A. (1961). On rigid circuit graphs. *Abh. Math. Sem. Univ. Hamburg*, 25:71–76.

[Eén and Sörensson, 2004] Eén, N. and Sörensson, N. (2004). An extensible sat-solver. In Giunchiglia, E. and Tacchella, A., editors, *Theory and Applications of Satisfiability Testing*, volume 2919 of *Lecture Notes in Computer Science*, pages 502–518. Springer Berlin Heidelberg.

[Engelfriet and Filè, 1982] Engelfriet, J. and Filè, G. (1982). Simple multi-visit attribute grammars. *Journal of computer and system sciences*, 24(3):283–314.

[Gibbons, 2007] Gibbons, J. (2007). Datatype-generic programming. In Backhouse, R., Gibbons, J., Hinze, R., and Jeuring, J., editors, *Spring School on Datatype-Generic Programming*, volume 4719 of *Lecture Notes in Computer Science*. Springer-Verlag.

[Gibbons, 2013] Gibbons, J. (2013). Accumulating attributes. In Hage, J. and Dijkstra, A., editors, *Een Lawine van Ontwortelde Bomen. Liber Amicorum for S. Doaitse Swierstra, in celebration of his retirement.* Department of Information and Computing Sciences, Utrecht University, Utrecht, The Netherlands.

[Hammer et al., 2014] Hammer, M. A., Phang, K. Y., Hicks, M., and Foster, J. S. (2014). Adapton: Composable, demand-driven incremental computation. *SIGPLAN Not.*, 49(6):156–166.

[Hedin, 1994] Hedin, G. (1994). An overview of door attribute grammars. In *Proceedings of the 5th International Conference on Compiler Construction*, CC '94, pages 31–51, London, UK, UK. Springer-Verlag.

[Hedin, 2000] Hedin, G. (2000). Reference attributed grammars. *Informatica (Slovenia)*, 24(3).

[Heeren et al., 2003] Heeren, B., Leijen, D., and van IJzendoorn, A. (2003). Helium, for learning haskell. In *Proceedings of the 2003 ACM SIGPLAN Workshop on Haskell*, Haskell '03, pages 62–71, New York, NY, USA. ACM.

[Hindley, 1969] Hindley, R. (1969). The principal type-scheme of an object in combinatory logic. *Trans. Amer. Math. Soc*, 146:29–60.

[Hinze et al., 2002] Hinze, R., Jeuring, J., and Löh, A. (2002). Type-indexed data types. In *Proceedings of the 6th International Conference on Mathematics of Program Construction*, MPC '02, pages 148–174, London, UK. Springer-Verlag.

[Holdermans et al., 2006] Holdermans, S., Jeuring, J., Löh, A., and Rodriguez Yakushev, A. (2006). Generic views on data types. In *Proceedings of the 8th International Conference on Mathematics of Program Construction*, volume 4014 of *Lecture Notes in Computer Science*, pages 209–234. Springer.

[Kastens, 1980] Kastens, U. (1980). Ordered attributed grammars. *Acta Informatica*, 13:229–256.

[Kennedy and Warren, 1976] Kennedy, K. and Warren, S. K. (1976). Automatic generation of efficient evaluators for attribute grammars. In *Proceedings of the 3rd ACM SIGACT-SIGPLAN symposium on Principles on programming languages*, POPL '76, pages 32–49, New York, NY, USA. ACM.

[Kiselyov, 2011] Kiselyov, O. (2011). Generic zipper: the context of a traversal. `http://okmij.org/ftp/continuations/zipper.html`.

[Knuth, 1968] Knuth, D. E. (1968). Semantics of context-free languages. *Theory of Computing Systems*, 2(2):127–145.

[Knuth, 1971] Knuth, D. E. (1971). Semantics of context-free languages: Correction. *Theory of Computing Systems*, 5:95–96.

[Kuiper and Swierstra, 1987] Kuiper, M. F. and Swierstra, S. D. (1987). Using attribute grammars to derive efficient functional programs. In *Computing Science in the Netherlands CSN'87*.

[La Poutré and van Leeuwen, 1988] La Poutré, J. and van Leeuwen, J. (1988). Maintenance of transitive closures and transitive reductions of graphs. In Göttler, H. and Schneider, H.-J., editors, *Graph-Theoretic Concepts in Computer Science*, volume 314 of *Lecture Notes in Computer Science*, pages 106–120. Springer Berlin Heidelberg.

[Launchbury and Peyton Jones, 1994] Launchbury, J. and Peyton Jones, S. L. (1994). Lazy functional state threads. In *Proceedings of the ACM SIGPLAN 1994 conference on Programming language design and implementation*, PLDI '94, pages 24–35, New York, NY, USA. ACM.

[Lempsink et al., 2009] Lempsink, E., Leather, S., and Löh, A. (2009). Type-safe diff for families of datatypes. In *Proceedings of the 2009 ACM SIGPLAN workshop on Generic programming*, WGP '09, pages 61–72, New York, NY, USA. ACM.

[Löh and Magalhães, 2013] Löh, A. and Magalhães, J. P. (2013). Everything you always wanted to know about Doaitse. In Hage, J. and Dijkstra, A., editors, *Een Lawine van Ontwortelde Bomen. Liber Amicorum for S. Doaitse Swierstra, in celebration of his retirement.* Department of Information and Computing Sciences, Utrecht University, Utrecht, The Netherlands.

[Magalhães, 2013] Magalhães, J. P. (2013). Optimisation of generic programs through inlining. In Hinze, R., editor, *Implementation and Application of Functional Languages*, Lecture Notes in Computer Science, pages 104–121. Springer Berlin Heidelberg.

[McBride, 2001] McBride, C. (2001). The derivative of a regular type is its type of one-hole contexts. Unpublished manuscript, available at `http://strictlypositive.org/diff.pdf`.

[Middelkoop, 2012] Middelkoop, A. (2012). *Inference of Program Properties with Attribute Grammars, Revisited*. PhD thesis, Utrecht University.

[Middelkoop et al., 2011] Middelkoop, A., Dijkstra, A., and Swierstra, S. (2011). Dependently typed attribute grammars. In Hage, J. and Morazán, M., editors, *Implementation and Application of Functional Languages*, volume 6647 of *Lecture Notes in Computer Science*, pages 105–120. Springer Berlin Heidelberg.

[Middelkoop et al., 2012] Middelkoop, A., Elyasov, A. B., and Prasetya, W. (2012). Functional instrumentation of actionscript programs with asil. In Gill, A. and Hage, J., editors, *Implementation and Application of Functional Languages*, volume 7257 of *Lecture Notes in Computer Science*, pages 1–16. Springer Berlin Heidelberg.

[Milner, 1978] Milner, R. (1978). A theory of type polymorphism in programming. *Journal of Computer and System Sciences*, 17:348–375.

[van Noort et al., 2008] van Noort, T., Rodriguez Yakushev, A., Holdermans, S., Jeuring, J., and Heeren, B. (2008). A lightweight approach to datatype-generic rewriting. In *Proceedings of the ACM SIGPLAN workshop on Generic programming*, WGP '08, pages 13–24, New York, NY, USA. ACM.

[Peyton Jones, 2003] Peyton Jones, S. L. (2003). *Haskell 98, Language and Libraries. The Revised Report*. Cambridge University Press. Journal of Functional Programming Special Issue 13(1).

[Reps and Teitelbaum, 1984] Reps, T. and Teitelbaum, T. (1984). The synthesizer generator. *SIGPLAN Not.*, 19:42–48.

[Rodriguez Yakushev et al., 2009] Rodriguez Yakushev, A., Holdermans, S., Löh, A., and Jeuring, J. (2009). Generic programming with fixed points for mutually recursive datatypes. In *Proceedings of the 14th ACM SIGPLAN international conference on Functional programming*, ICFP '09, pages 233–244, New York, NY, USA. ACM.

[Saraiva, 1999] Saraiva, J. (1999). *Purely Functional Implementation of Attribute Grammars*. PhD thesis, Utrecht University.

[Saraiva et al., 2000] Saraiva, J., Swierstra, S. D., and Kuiper, M. F. (2000). Functional incremental attribute evaluation. In *Proceedings of the 9th International Conference on Compiler Construction*, CC '00, pages 279–294, London, UK. Springer-Verlag.

[Söderberg and Hedin, 2011] Söderberg, E. and Hedin, G. (2011). Automated selective caching for reference attribute grammars. In Malloy, B., Staab, S., and van den Brand, M., editors, *Software Language Engineering*, volume 6563 of *Lecture Notes in Computer Science*, pages 2–21. Springer Berlin Heidelberg.

[van Steenbergen et al., 2010] van Steenbergen, M., Magalhães, J. P., and Jeuring, J. (2010). Generic selections of subexpressions. In *Proceedings of the 6th ACM*

*SIGPLAN Workshop on Generic Programming*, WGP '10, pages 37–48, New York, NY, USA. ACM.

[Swierstra et al., 1998] Swierstra, S. D., Alcocer, P. R. A., and Saraiva, J. (1998). Designing and Implementing Combinator Languages. In *Advanced Functional Programming*, pages 150–206.

[Teitelbaum and Chapman, 1990] Teitelbaum, T. and Chapman, R. (1990). Higher-order attribute grammars and editing environments. In *Proceedings of the ACM SIGPLAN 1990 Conference on Programming Language Design and Implementation*, PLDI '90, pages 197–208, New York, NY, USA. ACM.

[Visser and Löh, 2010] Visser, S. and Löh, A. (2010). Generic storage in Haskell. In *Proceedings of the 6th ACM SIGPLAN Workshop on Generic Programming*, WGP '10, pages 25–36, New York, NY, USA. ACM.

[Vogt et al., 1989] Vogt, H. H., Swierstra, S. D., and Kuiper, M. F. (1989). Higher order attribute grammars. In *Proceedings of the ACM SIGPLAN 1989 Conference on Programming language design and implementation*, volume 24 of *PLDI '89*, pages 131–145, New York, NY, USA. ACM.

[Vogt et al., 1991] Vogt, H. H., Swierstra, S. D., and Kuiper, M. F. (1991). Efficient incremental evaluation of higher order attribute grammars. In *PLILP*, pages 231–242.

[Wadler, 1990] Wadler, P. (1990). Comprehending monads. In *Proceedings of the 1990 ACM conference on LISP and functional programming*, LFP '90, pages 61–78, New York, NY, USA. ACM.

[Xi et al., 2003] Xi, H., Chen, C., and Chen, G. (2003). Guarded recursive datatype constructors. In *Proceedings of the 30th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '03, pages 224–235, New York, NY, USA. ACM.

[Yeh, 1983] Yeh, D. (1983). On incremental evaluation of ordered attributed grammars. *BIT Numerical Mathematics*, 23:308–320.

[Yeh and Kastens, 1988] Yeh, D. and Kastens, U. (1988). Improvements of an incremental evaluation algorithm for ordered attribute grammars. *SIGPLAN Not.*, 23(12):45–50.

# Index

# Samenvatting

Computerprogramma's doen vaak berekeningen over gegevens die over tijd veranderen. Een tekstverwerker kan bijvoorbeeld automatisch een inhoudsopgave genereren en deze bijhouden terwijl een document bewerkt wordt, een e-mailprogramma kan de lijst met recente e-mails bijwerken wanneer nieuwe e-mails binnenkomen, en een programmeeromgeving kan fouten in de code tonen terwijl de programmeur de code intypt.

Gebruikers van zulke programma's verwachten dat deze snel reageren op veranderingen, maar dit levert een probleem op voor de programmeur van deze programma's: wanneer de gegevens over tijd steeds groter worden, duurt de berekening telkens langer. Om dit op te lossen kan de berekening *incrementeel* gedaan worden, waarbij kleine wijzigingen in de gegevens leiden tot een korte berekeningstijd. Deze kortere berekeningstijd kan worden bereikt door resultaten vanuit een eerdere berekening te hergebruiken.

Het programmeren van goed werkende programma's is op zichzelf al moeilijk, en het maken van een incrementele versie is nog veel moeilijker en foutgevoeliger. In dit proefschrift bekijken we daarom een alternatieve aanpak: het automatisch genereren van een incrementele versie van een programma vanuit een niet-incrementele declaratieve definitie van dat programma. Met die aanpak kan de programmeur nadenken over de standaard niet-incrementele versie van het programma, en krijgt hij de incrementele versie "gratis".

**Attributengrammatica's** We gebruiken *attributengrammatica*'s voor het op een declaratieve manier beschrijven van berekeningen over boomstructuren. Een attributengrammatica beschrijft de decoratie van bomen met attributen, welke aan knopen in de boom verbonden zijn en gebruikt worden om resultaten van berekeningen door te geven aan een ouder of een kind. Om precies te zijn bestaat een attributengrammatica uit drie verschillende componenten: een definitie van de *abstracte syntaxisboom*, een set van *attribuutdeclaraties* en een set van *semantiekregels*.

De abstracte syntaxisboom representeert de invoer van een programma. De attributengrammatica definieert welke vorm deze boom kan hebben door middel van een algebraïsch datatype. Een ontleder zet de concrete invoer van het programma om in een instantie van dat datatype, waarover vervolgens de berekening gedaan kan worden. De berekende waarden worden opgeslagen in attributen, welke van een kind naar de ouder doorgegeven kunnen worden (een *gesynthetiseerd* attribuut) of van ouder naar kind (een *overerft* attribuut). De attribuutdeclaraties geven per type knoop aan welke attributen bij dat type knoop horen, en de semantiekregels definiëren tenslotte hoe de waarde van elk attribuut kan worden berekend uit de waarden van andere attributen, bijvoorbeeld die van de ouderknoop of van de kindknopen.

Het voordeel van het gebruik van attributengrammatica's voor het automatisch genereren van incrementele programma's is dat in een attributengrammatica de afhankelijkheden tussen verschillende delen van de berekening expliciet gemaakt zijn. Hierdoor kan een automatische analyse op de attributengrammatica worden gedaan om te achterhalen welke delen van een eerdere berekening hergebruikt kunnen worden na een wijziging in de invoer van een programma. Uit eerder onderzoek is gebleken dat een dergelijke aanpak effectief kan zijn.

**Hogere-orde attributengrammatica's** De focus van dit proefschrift is de incrementele evaluatie van attributengrammatica's die *hogere-orde kinderen* bevatten (hogere-orde attributengrammatica's). De toevoeging van hogere-orde kinderen zorgt ervoor dat er meer programma's zijn die op een aantrekkelijke manier geschreven kunnen worden, maar het zorgt er ook voor dat bepaalde vormen van incrementele evaluatie niet meer leiden tot snelheidswinst. In dit proefschrift ontwikkelen we daarom een techniek voor de incrementele evaluatie van hogere-orde attributengrammatica's, zodanig dat de snelheidswinst behouden blijft wanneer hogere-orde kinderen gebruikt worden.

In een hogere-orde attributengrammatica kunnen de waarden van de attributen boomstructuren zijn waarover ook weer attributen berekend kunnen worden. Deze boomstructuren worden dus gebouwd door de attribuutberekeningen, en kunnen worden geïnstantieerd tot hogere-orde kind, waarna er over dat kind attributen berekend kunnen worden op dezelfde manier als voor reguliere kinderen. De instantiatie van een hogere-orde kind wordt in Figuur 6.1 geïllustreerd. Hogere-orde kinderen worden bijvoorbeeld gebruikt om meerdere fasen in een programma te modelleren: in de eerste fase wordt een nieuwe boom opgebouwd, waarover in de tweede fase weer andere attributen berekend worden.

Het probleem met veel vormen van incrementele evaluatie van attributengrammatica's met hogere-orde kinderen is dat alle attributen van een hogere-orde kind herberekend worden wanneer maar een deel van het hogere-orde kind is gewijzigd. In Figuur 6.2 is dit probleem gevisualiseerd, waarbij de grijze onderdelen van de boom de waarden zijn die opnieuw berekend moeten worden na een wijziging. Alhoewel de kleine wijziging op de originele boom een kleine wijziging op het hogere-orde kind tot gevolg kan hebben, gaat die informatie verloren en wordt het volledige kind als nieuw beschouwd, waardoor de snelheidswinst van incrementele evaluatie verloren gaat.

**Aanpak** De aanpak die we in dit proefschrift gebruiken is in de eerste plaats gebaseerd op een precieze representatie van wijzigingen op boomstructuren. In dit proefschrift nemen we aan dat de gebruiker met een programma werkt dat bijhoudt hoe de invoer wijzigt, bijvoorbeeld de tekstverwerker waarin een stuk tekst verplaatst wordt. Wanneer een attributengrammatica gebruikt wordt om de inhoudsopgave te genereren, moet de tekstverwerker aan de attributengrammatica-evaluator doorgeven op welke manier de invoer gewijzigd is, zodat deze de wijziging kan doorvoeren en (in veel gevallen) efficiënt de inhoudsopgave kan herberekenen.

Na enkele inleidende hoofdstukken wordt in Hoofdstuk 4 van dit proefschrift beschreven op welke manier we wijzigingen op boomstructuren representeren. De representatie is gebaseerd op het vervangen van een deelboom op een bepaalde locatie door een nieuwe boom. Deze nieuwe boom kan *verwijzingen* bevatten die aangeven dat een deel van de originele boom op die plek ingevoegd moet worden. Bijvoorbeeld, het wisselen van twee kinderen van een knoop kan gerepresenteerd worden door twee van zulke vervangingen: in het linker kind door een verwijzing naar het rechter kind, en in het rechter kind door een verwijzing naar het linker kind. Met deze representatie kan elk type wijziging zodanig worden gerepresenteerd dat de attributengrammatica-evaluator deze informatie kan gebruiken om de benodigde attributen te herberekenen.

Hoofdstuk 5 beschrijft de incrementele evaluatie van attributengrammatica's zonder hogere-orde kinderen. Er worden hiervoor twee technieken gebruikt: *wijzigingspropagatie* en *memoisatie*. De wijzigingspropagatie neemt de representatie van de wijziging en zorgt ervoor dat alle attributen die (mogelijkerwijs) gewijzigd zijn door deze wijzigingen worden herberekend. Memoisatie is de techniek die ervoor zorgt dat eerder gedane berekeningen worden opgeslagen en hergebruikt wanneer dat deel van de berekening niet is gewijzigd.

De ondersteuning voor hogere-orde kinderen voegen we toe in Hoofdstuk 6, door middel van het bijhouden van een *afgeleide wijziging*. De hogere-orde kinderen zijn een instantiatie van een attribuutwaarde, en in de evaluatie van de attributengrammatica weten we dan ook op welke manier dat kind wordt gebouwd. Van de wijziging op de originele syntaxisboom kunnen we daarom afleiden op welke manier het hogere-orde kind is gewijzigd ten opzichte van de vorige berekening. Deze wijziging wordt gepresenteerd op dezelfde manier als een wijziging op de originele syntaxisboom, waardoor de wijzigingspropagatie en memoisatie voor het hogere-orde kind op dezelfde manier kunnen worden toegepast. Dit principe wordt geïllustreerd in Figuur 6.3.

Alhoewel het idee van onze techniek is dat deze toegepast kan worden op elke hogere-orde attributengrammatica is dat niet het geval. Er zijn bepaalde restricties waaraan voldaan moet worden voordat de techniek toepast kan worden, en er zijn bepaalde patronen die goed of juist slecht kunnen zijn voor de effectiviteit van onze techniek. Deze restricties en patronen worden beschreven in Hoofdstuk 7.

**Resultaat** Het resultaat van het toepassen van de beschreven techniek op de twee voorbeelden die in dit proefschrift worden gebruikt is beschreven in Hoofdstuk 8. Op het eerste voorbeeld wordt de te verwachten snelheidswinst gehaald, maar op het tweede voorbeeld is dit niet het geval. In tegendeel: onze techniek voor incrementele evaluatie zorgt ervoor dat de code tot vijftien keer trager wordt!

Het probleem van onze techniek zoals toegepast in deze voorbeelden is dat de granulariteit te hoog is. Het tweede voorbeeld bevat, net als de meeste attributengrammatica's, veel berekeningen die maar weinig tijd kosten. Aangezien de memoisatie en wijzigingspropagatie ook tijd kosten, is het in een dergelijk geval beter om de berekening opnieuw te doen dan om te proberen de waardes van de vorige keer te hergebruiken. Dat laatste zorgt conceptueel weliswaar voor minder attribuutberekeningen, maar is in de praktijk een stuk trager.

Gelukkig betekent dit niet dat onze techniek onbruikbaar is, maar voordat deze in de praktijk toepast kan worden is er meer onderzoek nodig, bijvoorbeeld door de granulariteit kleiner te maken. In Hoofdstuk 9 wordt dit proefschrift dan ook besloten met een lijst van suggesties voor vervolgonderzoek, waarvan we denken dat het grote invloed kan hebben op de effectiviteit van onze techniek in de praktijk. Dit proefschrift kan dus worden gezien als de basis voor een nieuwe vorm van incrementele evaluatie van hogere-orde attributengrammatica's, al geeft dit nog niet de eindoplossing voor de incrementele evaluatie van hogere-orde attributengrammatica's met het doel automatisch incrementele versies van programma's te genereren uit hun declaratieve specificatie.

# Curriculum Vitae

**Jeroen Bransen**
Born on 14 September 1987 in Utrecht

**1999 – 2005**
VWO (High school) – Niftarlake College, Maarssen
Graduation date: 1 July 2005

**2005 – 2008**
Bachelor Cognitive Artificial Intelligence – Utrecht University, Utrecht
Graduation date: 31 August 2008
Thesis: *Statistical methods for Categorial Grammars*
Supervisor: Michael Moortgat

**2008 – 2010**
Master Cognitive Artificial Intelligence – Utrecht University, Utrecht
Graduation date: 31 August 2010, Cum laude
Thesis: *On the complexity of the Lambek-Grishin calculus*
Supervisors: Michael Moortgat and Rosalie Iemhoff

**2010 – 2015**
PhD student – Department of Computing and Information Sciences, Utrecht University, Utrecht
Defense date: 30 June 2015
Thesis: *On the Incremental Evaluation of Higher-Order Attribute Grammars*
Promotor: Prof. dr. S. Doaitse Swierstra
Copromotor: Dr. Atze Dijkstra

# Titles in the IPA Dissertation Series since 2009

**M.H.G. Verhoef**. *Modeling and Validating Distributed Embedded Real-Time Control Systems*. Faculty of Science, Mathematics and Computer Science, RU. 2009-01

**M. de Mol**. *Reasoning about Functional Programs: Sparkle, a proof assistant for Clean*. Faculty of Science, Mathematics and Computer Science, RU. 2009-02

**M. Lormans**. *Managing Requirements Evolution*. Faculty of Electrical Engineering, Mathematics, and Computer Science, TUD. 2009-03

**M.P.W.J. van Osch**. *Automated Model-based Testing of Hybrid Systems*. Faculty of Mathematics and Computer Science, TU/e. 2009-04

**H. Sozer**. *Architecting Fault-Tolerant Software Systems*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2009-05

**M.J. van Weerdenburg**. *Efficient Rewriting Techniques*. Faculty of Mathematics and Computer Science, TU/e. 2009-06

**H.H. Hansen**. *Coalgebraic Modelling: Applications in Automata Theory and Modal Logic*. Faculty of Sciences, Division of Mathematics and Computer Science, VUA. 2009-07

**A. Mesbah**. *Analysis and Testing of Ajax-based Single-page Web Applications*. Faculty of Electrical Engineering, Mathematics, and Computer Science, TUD. 2009-08

**A.L. Rodriguez Yakushev**. *Towards Getting Generic Programming Ready for Prime Time*. Faculty of Science, UU. 2009-9

**K.R. Olmos Joffré**. *Strategies for Context Sensitive Program Transformation*. Faculty of Science, UU. 2009-10

**J.A.G.M. van den Berg**. *Reasoning about Java programs in PVS using JML*. Faculty of Science, Mathematics and Computer Science, RU. 2009-11

**M.G. Khatib**. *MEMS-Based Storage Devices. Integration in Energy-Constrained Mobile Systems*. Faculty of Electrical

Engineering, Mathematics & Computer Science, UT. 2009-12

**S.G.M. Cornelissen**. *Evaluating Dynamic Analysis Techniques for Program Comprehension*. Faculty of Electrical Engineering, Mathematics, and Computer Science, TUD. 2009-13

**D. Bolzoni**. *Revisiting Anomaly-based Network Intrusion Detection Systems*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2009-14

**H.L. Jonker**. *Security Matters: Privacy in Voting and Fairness in Digital Exchange*. Faculty of Mathematics and Computer Science, TU/e. 2009-15

**M.R. Czenko**. *TuLiP - Reshaping Trust Management*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2009-16

**T. Chen**. *Clocks, Dice and Processes*. Faculty of Sciences, Division of Mathematics and Computer Science, VUA. 2009-17

**C. Kaliszyk**. *Correctness and Availability: Building Computer Algebra on top of Proof Assistants and making Proof Assistants available over the Web*. Faculty of Science, Mathematics and Computer Science, RU. 2009-18

**R.S.S. O'Connor**. *Incompleteness & Completeness: Formalizing Logic and Analysis in Type Theory*. Faculty of Science, Mathematics and Computer Science, RU. 2009-19

**B. Ploeger**. *Improved Verification Methods for Concurrent Systems*. Faculty of Mathematics and Computer Science, TU/e. 2009-20

**T. Han**. *Diagnosis, Synthesis and Analysis of Probabilistic Models*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2009-21

**R. Li**. *Mixed-Integer Evolution Strategies for Parameter Optimization and Their Applications to Medical Image Analysis*. Faculty of Mathematics and Natural Sciences, UL. 2009-22

**J.H.P. Kwisthout**. *The Computational Complexity of Probabilistic Networks*. Faculty of Science, UU. 2009-23

**T.K. Cocx**. *Algorithmic Tools for Data-Oriented Law Enforcement*. Faculty of Mathematics and Natural Sciences, UL. 2009-24

**A.I. Baars**. *Embedded Compilers*. Faculty of Science, UU. 2009-25

**M.A.C. Dekker**. *Flexible Access Control for Dynamic Collaborative Environments*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2009-26

**J.F.J. Laros**. *Metrics and Visualisation for Crime Analysis and Genomics*. Faculty of Mathematics and Natural Sciences, UL. 2009-27

**C.J. Boogerd**. *Focusing Automatic Code Inspections*. Faculty of Electrical Engineering, Mathematics, and Computer Science, TUD. 2010-01

**M.R. Neuhäußer**. *Model Checking Nondeterministic and Randomly Timed Systems*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2010-02

**J. Endrullis**. *Termination and Productivity*. Faculty of Sciences, Division of Mathematics and Computer Science, VUA. 2010-03

**T. Staijen**. *Graph-Based Specification and Verification for Aspect-Oriented Languages*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2010-04

**Y. Wang**. *Epistemic Modelling and Protocol Dynamics*. Faculty of Science, UvA. 2010-05

**J.K. Berendsen**. *Abstraction, Prices and Probability in Model Checking Timed Automata*. Faculty of Science, Mathematics and Computer Science, RU. 2010-06

**A. Nugroho**. *The Effects of UML Modeling on the Quality of Software*. Faculty of Mathematics and Natural Sciences, UL. 2010-07

**A. Silva**. *Kleene Coalgebra*. Faculty of Science, Mathematics and Computer Science, RU. 2010-08

**J.S. de Bruin**. *Service-Oriented Discovery of Knowledge - Foundations, Implementations and Applications*. Faculty of Mathematics and Natural Sciences, UL. 2010-09

**D. Costa**. *Formal Models for Component Connectors*. Faculty of Sciences, Division of Mathematics and Computer Science, VUA. 2010-10

**M.M. Jaghoori**. *Time at Your Service: Schedulability Analysis of Real-Time and Distributed Services*. Faculty of Mathematics and Natural Sciences, UL. 2010-11

**R. Bakhshi**. *Gossiping Models: Formal Analysis of Epidemic Protocols*. Faculty of Sciences, Department of Computer Science, VUA. 2011-01

**B.J. Arnoldus**. *An Illumination of the Template Enigma: Software Code Generation with Templates*. Faculty of Mathematics and Computer Science, TU/e. 2011-02

**E. Zambon**. *Towards Optimal IT Availability Planning: Methods and Tools*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2011-03

**L. Astefanoaei**. *An Executable Theory of Multi-Agent Systems Refinement*. Faculty of Mathematics and Natural Sciences, UL. 2011-04

**J. Proença**. *Synchronous coordination of distributed components*. Faculty of Mathematics and Natural Sciences, UL. 2011-05

**A. Moralı**. *IT Architecture-Based Confidentiality Risk Assessment in Networks of Organizations*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2011-06

**M. van der Bijl**. *On changing models in Model-Based Testing*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2011-07

**C. Krause**. *Reconfigurable Component Connectors*. Faculty of Mathematics and Natural Sciences, UL. 2011-08

**M.E. Andrés**. *Quantitative Analysis of Information Leakage in Probabilistic and Nondeterministic Systems*. Faculty of Science, Mathematics and Computer Science, RU. 2011-09

**M. Atif**. *Formal Modeling and Verification of Distributed Failure Detectors*. Faculty of Mathematics and Computer Science, TU/e. 2011-10

**P.J.A. van Tilburg**. *From Computability to Executability – A process-theoretic view on automata theory*. Faculty of Mathematics and Computer Science, TU/e. 2011-11

**Z. Protic**. *Configuration management for models: Generic methods for model comparison and model co-evolution*. Faculty of Mathematics and Computer Science, TU/e. 2011-12

**S. Georgievska**. *Probability and Hiding in Concurrent Processes*. Faculty of Mathematics and Computer Science, TU/e. 2011-13

**S. Malakuti**. *Event Composition Model: Achieving Naturalness in Runtime Enforcement*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2011-14

**M. Raffelsieper**. *Cell Libraries and Verification*. Faculty of Mathematics and Computer Science, TU/e. 2011-15

**C.P. Tsirogiannis**. *Analysis of Flow and Visibility on Triangulated Terrains*. Faculty of Mathematics and Computer Science, TU/e. 2011-16

**Y.-J. Moon**. *Stochastic Models for Quality of Service of Component Connectors*. Faculty of Mathematics and Natural Sciences, UL. 2011-17

**R. Middelkoop**. *Capturing and Exploiting Abstract Views of States in OO Verification*. Faculty of Mathematics and Computer Science, TU/e. 2011-18

**M.F. van Amstel**. *Assessing and Improving the Quality of Model Transformations*. Faculty of Mathematics and Computer Science, TU/e. 2011-19

**A.N. Tamalet**. *Towards Correct Programs in Practice*. Faculty of Science, Mathematics and Computer Science, RU. 2011-20

**H.J.S. Basten**. *Ambiguity Detection for Programming Language Grammars*. Faculty of Science, UvA. 2011-21

**M. Izadi**. *Model Checking of Component Connectors*. Faculty of Mathematics and Natural Sciences, UL. 2011-22

**L.C.L. Kats**. *Building Blocks for Language Workbenches*. Faculty of Electrical Engineering, Mathematics, and Computer Science, TUD. 2011-23

**S. Kemper**. *Modelling and Analysis of Real-Time Coordination Patterns*. Faculty of Mathematics and Natural Sciences, UL. 2011-24

**J. Wang**. *Spiking Neural P Systems*. Faculty of Mathematics and Natural Sciences, UL. 2011-25

**A. Khosravi**. *Optimal Geometric Data Structures*. Faculty of Mathematics and Computer Science, TU/e. 2012-01

**A. Middelkoop**. *Inference of Program Properties with Attribute Grammars, Revisited*. Faculty of Science, UU. 2012-02

**Z. Hemel**. *Methods and Techniques for the Design and Implementation of Domain-Specific Languages*. Faculty of Electrical Engineering, Mathematics, and Computer Science, TUD. 2012-03

**T. Dimkov**. *Alignment of Organizational Security Policies: Theory and Practice*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2012-04

**S. Sedghi**. *Towards Provably Secure Efficiently Searchable Encryption*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2012-05

**F. Heidarian Dehkordi**. *Studies on Verification of Wireless Sensor Networks and Abstraction Learning for System Inference*. Faculty of Science, Mathematics and Computer Science, RU. 2012-06

**K. Verbeek**. *Algorithms for Cartographic Visualization*. Faculty of Mathematics and Computer Science, TU/e. 2012-07

**D.E. Nadales Agut**. *A Compositional Interchange Format for Hybrid Systems: Design and Implementation*. Faculty of Mechanical Engineering, TU/e. 2012-08

**H. Rahmani**. *Analysis of Protein-Protein Interaction Networks by Means of Annotated Graph Mining Algorithms*. Faculty of Mathematics and Natural Sciences, UL. 2012-09

**S.D. Vermolen**. *Software Language Evolution*. Faculty of Electrical Engineering, Mathematics, and Computer Science, TUD. 2012-10

**L.J.P. Engelen**. *From Napkin Sketches to Reliable Software*. Faculty of Mathematics and Computer Science, TU/e. 2012-11

**F.P.M. Stappers**. *Bridging Formal Models – An Engineering Perspective*. Faculty of Mathematics and Computer Science, TU/e. 2012-12

**W. Heijstek**. *Software Architecture Design in Global and Model-Centric Software Development*. Faculty of Mathematics and Natural Sciences, UL. 2012-13

**C. Kop**. *Higher Order Termination*. Faculty of Sciences, Department of Computer Science, VUA. 2012-14

**A. Osaiweran**. *Formal Development of Control Software in the Medical Systems*

*Domain*. Faculty of Mathematics and Computer Science, TU/e. 2012-15

**W. Kuijper**. *Compositional Synthesis of Safety Controllers*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2012-16

**H. Beohar**. *Refinement of Communication and States in Models of Embedded Systems*. Faculty of Mathematics and Computer Science, TU/e. 2013-01

**G. Igna**. *Performance Analysis of Real-Time Task Systems using Timed Automata*. Faculty of Science, Mathematics and Computer Science, RU. 2013-02

**E. Zambon**. *Abstract Graph Transformation – Theory and Practice*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2013-03

**B. Lijnse**. *TOP to the Rescue – Task-Oriented Programming for Incident Response Applications*. Faculty of Science, Mathematics and Computer Science, RU. 2013-04

**G.T. de Koning Gans**. *Outsmarting Smart Cards*. Faculty of Science, Mathematics and Computer Science, RU. 2013-05

**M.S. Greiler**. *Test Suite Comprehension for Modular and Dynamic Systems*. Faculty of Electrical Engineering, Mathematics, and Computer Science, TUD. 2013-06

**L.E. Mamane**. *Interactive mathematical documents: creation and presentation*. Faculty of Science, Mathematics and Computer Science, RU. 2013-07

**M.M.H.P. van den Heuvel**. *Composition and synchronization of real-time components upon one processor*. Faculty of Mathematics and Computer Science, TU/e. 2013-08

**J. Businge**. *Co-evolution of the Eclipse Framework and its Third-party Plug-ins*. Faculty of Mathematics and Computer Science, TU/e. 2013-09

**S. van der Burg**. *A Reference Architecture for Distributed Software Deployment*. Faculty of Electrical Engineering, Mathematics, and Computer Science, TUD. 2013-10

**J.J.A. Keiren**. *Advanced Reduction Techniques for Model Checking*. Faculty of Mathematics and Computer Science, TU/e. 2013-11

**D.H.P. Gerrits**. *Pushing and Pulling: Computing push plans for disk-shaped robots, and dynamic labelings for moving points*. Faculty of Mathematics and Computer Science, TU/e. 2013-12

**M. Timmer**. *Efficient Modelling, Generation and Analysis of Markov Automata*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2013-13

**M.J.M. Roeloffzen**. *Kinetic Data Structures in the Black-Box Model*. Faculty of Mathematics and Computer Science, TU/e. 2013-14

**L. Lensink**. *Applying Formal Methods in Software Development*. Faculty of Science, Mathematics and Computer Science, RU. 2013-15

**C. Tankink**. *Documentation and Formal Mathematics — Web Technology meets Proof Assistants*. Faculty of Science, Mathematics and Computer Science, RU. 2013-16

**C. de Gouw**. *Combining Monitoring with Run-time Assertion Checking*. Faculty of Mathematics and Natural Sciences, UL. 2013-17

**J. van den Bos**. *Gathering Evidence: Model-Driven Software Engineering in Automated Digital Forensics*. Faculty of Science, UvA. 2014-01

**D. Hadziosmanovic**. *The Process Matters: Cyber Security in Industrial Control Systems*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2014-02

**A.J.P. Jeckmans**. *Cryptographically-Enhanced Privacy for Recommender Systems*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2014-03

**C.-P. Bezemer**. *Performance Optimization of Multi-Tenant Software Systems*. Faculty of Electrical Engineering, Mathematics, and Computer Science, TUD. 2014-04

**T.M. Ngo**. *Qualitative and Quantitative Information Flow Analysis for Multi-threaded Programs*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2014-05

**A.W. Laarman**. *Scalable Multi-Core Model Checking*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2014-06

**J. Winter**. *Coalgebraic Characterizations of Automata-Theoretic Classes*. Faculty of Science, Mathematics and Computer Science, RU. 2014-07

**W. Meulemans**. *Similarity Measures and Algorithms for Cartographic Schematization*. Faculty of Mathematics and Computer Science, TU/e. 2014-08

**A.F.E. Belinfante**. *JTorX: Exploring Model-Based Testing*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2014-09

**A.P. van der Meer**. *Domain Specific Languages and their Type Systems*. Faculty of Mathematics and Computer Science, TU/e. 2014-10

**B.N. Vasilescu**. *Social Aspects of Collaboration in Online Software Communities*. Faculty of Mathematics and Computer Science, TU/e. 2014-11

**F.D. Aarts**. *Tomte: Bridging the Gap between Active Learning and Real-World Systems*. Faculty of Science, Mathematics and Computer Science, RU. 2014-12

**N. Noroozi**. *Improving Input-Output Conformance Testing Theories*. Faculty of Mathematics and Computer Science, TU/e. 2014-13

**M. Helvensteijn**. *Abstract Delta Modeling: Software Product Lines and Beyond*. Faculty of Mathematics and Natural Sciences, UL. 2014-14

**P. Vullers**. *Efficient Implementations of Attribute-based Credentials on Smart Cards*. Faculty of Science, Mathematics and Computer Science, RU. 2014-15

**F.W. Takes**. *Algorithms for Analyzing and Mining Real-World Graphs*. Faculty of Mathematics and Natural Sciences, UL. 2014-16

**M.P. Schraagen**. *Aspects of Record Linkage*. Faculty of Mathematics and Natural Sciences, UL. 2014-17

**G. Alpár**. *Attribute-Based Identity Management: Bridging the Cryptographic Design of ABCs with the Real World*. Faculty of Science, Mathematics and Computer Science, RU. 2015-01

**A.J. van der Ploeg**. *Efficient Abstractions for Visualization and Interaction*. Faculty of Science, UvA. 2015-02

**R.J.M. Theunissen**. *Supervisory Control in Health Care Systems*. Faculty of Mechanical Engineering, TU/e. 2015-03

**T.V. Bui**. *A Software Architecture for Body Area Sensor Networks: Flexibility and Trustworthiness*. Faculty of Mathematics and Computer Science, TU/e. 2015-04

**A. Guzzi**. *Supporting Developers' Teamwork from within the IDE*. Faculty of Electrical Engineering, Mathematics, and Computer Science, TUD. 2015-05

**T. Espinha**. *Web Service Growing Pains: Understanding Services and Their Clients*. Faculty of Electrical Engineering, Mathematics, and Computer Science, TUD. 2015-06

**S. Dietzel**. *Resilient In-network Aggregation for Vehicular Networks*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2015-07

**E. Costante**. *Privacy throughout the Data Cycle*. Faculty of Mathematics and Computer Science, TU/e. 2015-08

**S. Cranen**. *Getting the point — Obtaining and understanding fixpoints in model checking*. Faculty of Mathematics and Computer Science, TU/e. 2015-09

**R. Verdult**. *The (in)security of proprietary cryptography*. Faculty of Science, Mathematics and Computer Science, RU. 2015-10

**J.E.J. de Ruiter**. *Lessons learned in the analysis of the EMV and TLS security protocols*. Faculty of Science, Mathematics and Computer Science, RU. 2015-11

**Y. Dajsuren**. *On the Design of an Architecture Framework and Quality Evaluation for Automotive Software Systems*. Faculty of Mathematics and Computer Science, TU/e. 2015-12

**J. Bransen**. *On the Incremental Evaluation of Higher-Order Attribute Grammars*. Faculty of Science, UU. 2015-13