PEDECIBA Informática
Instituto de Computación – Facultad de Ingeniería
Universidad de la República
Montevideo, Uruguay

Tesis de Doctorado
en Informática

First class syntax, semantics, and their composition

Marcos Viera

2014

Firsst class syntax, semantics and their
composition
ISSN 0797-6410
Tesis de Doctorado en Informática
Reporte Técnico RT 14-03
PEDECIBA
Instituto de Computación – Facultad de Ingeniería
Universidad de la República.
Montevideo, Uruguay, marzo, 2014

Título: First Class Syntax, Semantics, and Their Composition
Tesis de Doctorado
Autor: Marcos Viera
mviera@fing.edu.uy
Supervisor: Alberto Pardo
Instituto de Computación - Universidad de la República
pardo@fing.edu.uy
Orientador: Doaitse Swierstra
Utrecht University
doaitse@uu.nl
Fecha: 8 de Marzo de 2013

Resumen

Idealmente la complejidad es manejada componiendo un sistema en algunas pocas, más o menos independientes, descripciones más pequeñas de varios aspectos del artefacto general. Al describir lenguajes de programación (extensibles), las gramáticas de atributos han resultado ser una excelente herramienta para la definición modular y la integracin de sus diferentes aspectos.

En la tesis se muestra cómo construir la implementacin de un lenguaje de programación mediante la composición de una colección de fragmentos de gramáticas de atributos que describen aspectos separados del lenguaje. Más específicamente, se describe un conjunto coherente de bibliotecas y herramientas que en conjunto hacen que esto sea posible en Haskell, donde la corrección de la composición es forzada a través de la capacidad del sistema de tipos de Haskell para representar gramáticas de atributos como valores de Haskell y sus interfaces como tipos de datos. Los objetos semánticos construidos de este modo se pueden combinar con *parsers* que son construidos *on the fly* a partir de fragmentos de parsers y se representan como valores Haskell tipados. Una vez más el chequeo de tipos impide composiciones incorrectas.

A manera de caso de estudio de las técnicas propuestas en esta tesis, se implementó un compilador para el lenguaje imperativo (Pascal-like) Oberon0. A través de un diseño incremental, mostramos las capacidades de modularidad de nuestras técnicas.

Palabras clave: Gramática de primera clase, Semántica de primera clase, Gramáticas de Atributos, Construcción de Compiladores, Lenguajes extensibles, Haskell.

First Class Syntax, Semantics, and Their Composition

Marcos Viera

Author: Marcos Viera, 2012
Printed by: Wöhrmann Print Service
Cover: *El Carafon de primera* by Alessandro Spinuso, 2012

First Class Syntax, Semantics, and Their Composition

Eerste klas syntax, semantiek en hun samenstelling

(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 vrijdag 8 maart 2013 des middags te 4.15 uur

door

Marcos Viera

geboren op 9 september 1979 te San José, Uruguay

Promotoren: Prof.dr. S. D. Swierstra
Prof.dr. A. Pardo

Preface

Ideally complexity is managed by composing a system out of quite a few, more or less independent, and much smaller descriptions of various aspects of the overall artifact. When describing (extensible) programming languages, attribute grammars have turned out to be an excellent tool for modular definition and integration of their different aspects.

In this thesis we show how to construct a programming language implementation by composing a collection of attribute grammar fragments describing separate aspects of the language. More specifically we describe a coherent set of libraries and tools which together make this possible in Haskell, *where the correctness of the composition is enforced through the Haskell type system*'s ability to represent attribute grammars as plain Haskell values and their interfaces as Haskell types.

Semantic objects thus constructed can be combined with parsers which are constructed on the fly out of parser fragments and are also represented as typed Haskell values. Again the type checker prevents insane compositions.

As a small case study of the techniques proposed in this thesis, we implemented a compiler for the (Pascal-like) imperative language Oberon0. Through an incremental design, we show the modularity capacities of our techniques.

Acknowledgements. As this story has been written on two sides of the Atlantic, I feel obliged to thank in two languages. Since *ik spreek geen Nederlands*, the people I met at the eastern side of the Atlantic will be acknowledged in English.

I am very grateful to my promotor Doaitse Swierstra, who introduced me to the Haskell type-level puzzling world. It is impossible not to get contagious with his passion for our research area. He is a never ending source of ideas, which he has no problems on sharing, always starting with the sentence: "I have been thinking while biking home...". Doaitse is not only an excellent scientist. I discovered, while working with him, that he is also a very good person. Gracias, señor de Holanda!

I want to thank Wilke and Tineke Schram, who hosted me at their home during my first month in Utrecht, helping me a lot while I was taking my first baby steps into the dutch society.

The Software Technology group is a very good place where to work, its members are very talented researchers and also very kind people. Although I was there for a relatively short time, they treated my as if I was a member of the group. I thank Jurriaan for his continuous visits to my office, trying to convince me that *drops* was something that deserved to be tasted.

The foosball sessions deserve a separate chapter. We had a lot of fun producing all those weird vocalism, as a result of the excitement for this "sport". Thanks to

Américo, Arthur, José Pedro, Juan Francisco, Martijn, Reinier, Rodrigo and Sean for such a great time.

I took a couple of master courses in Utrecht to catch up with the research area I had to work on. Both courses were very helpful and inspiring. So I am very grateful to the great lecturers, Andres Löh and Jeroen Fokker.

I also want to thank my co-authors Arthur, Doaitse, Wouter, Arie and Atze, it was a pleasure and a honour to work with you and learn from you.

I am thankful to the members of the reading committee, Roland Backhouse, Koen Claessen, Johan Jeuring, Rinus Plasmeijer and Joao Saraiva, who kindly accepted to read my thesis.

Alessandro did a great work designing the cover of my thesis. He even took the risk of getting a fee on a Belgian train, for sitting in first class with a second class ticket, in order to set up what was needed to take the picture. I do not know if the material inside the thesis is that good, but I am sure its cover is a piece of art.

Finally, I want to spend some words on the group of friends who made me feel like home while I was there. Until I met them I did not know that someone can construct a friendship in such a short time. I have plenty of moments stored in my hearth, from quiet nice conversations to endless nights trying to convince me not to finish at Carafon. Alessandro, Américo, Alexey, David, Despoina, Juan Francisco, Luca, Martijn and Tizy, thanks to you, when I think about my days in Utrecht I have this feeling that the Brazilians call *saudade*.

Como esta fue una historia escrita en dos márgenes del Atlántico, me vi obligado a agradecer en dos idiomas. Luego de haber agradecido en inglés a aquellos que conocí en mi aventura europea, me toca agradecer en mi lengua a aquellos que en nuestro *paisito* me han acompañado en esta etapa tan importante de mi vida.

Repasando la sección de agradecimientos de mi tesis de maestría, con la desesperada intención de autoplagiarme un poco, tuve la grata confirmación de que a pesar de que el tiempo ha pasado, y varios pelos se han ido fugando, los afectos siguen intactos. De verdad que casi que podría copiar y pegar aquí aquellas palabras que escribí hace ya más de cinco años y, agregando algunos nuevos nombres, tendrían la misma validez. Pero no voy a ser tan mezquino, la gratitud merece ser renovada con palabras nuevas, y aquí van...

El Instituto de Computación es un lugar en el que me gusta mucho trabajar, a pesar de los 90 kilómetros que tengo que hacer para llegar hasta ahí, y eso se debe básicamente a la gente que se encuentra dispersa a lo largo de ese interminable pasillo del quinto piso y en los *kibutz* del cuerpo norte. Desde la visita por secretaría para recibir algún comentario ácido de Lucyla o Daniela, hasta las charlas de oficina, mate mediante, con Andrea, Daniel o Mónica, hacen del trabajo diario algo muy disfrutable y eso lo tengo que agradecer.

Quiero agradecer a mi supervisor, Alberto Pardo, quien me ha guiado paso a paso en este proceso de iniciación a la vida académica, estando siempre disponible para resolver cualquier cosa que le haya ido a plantear. Por suerte todavía me queda mucho por aprender de Alberto.

También quiero agradecer a aquellos con los que he compartido la escritura de (proyectos de) *papers* en esta última etapa. Gracias a Alberto, Bruno, Martín, Mauro, Pablo y Pablo (E), por considerar al menos por un rato que lo que hacemos pueda llegar a ser interesante.

En todas las instancias del curso de Programación 2 en las que he trabajado siempre me han tocado buenos grupos de compañeros. A través de sus co-responsables, Carlos y Lorena, les agradezco a todos ellos, por el buen trabajo y por la flexibilidad que siempre han mostrado cuando esta tesis le quitó algún tiempo al curso.

Quiero agradecer a todos los miembros del PEDECIBA Informática y a su secretaria María Inés. En especial quiero agradecer a los integrantes del Consejo Científico que junto con Daniel tuvimos la suerte de integrar como delegados estudiantiles.

Agradezco profundamente a todos mis amigos, los de siempre. Los que parece que nada tienen que ver con todo esto pero que en realidad son grandes culpables de lo que soy.

A mis padres, a quienes no me cansaré de decirles que me enorgullece ser su hijo, y que no me alcanzará la vida para agradecer lo que han hecho por mi.

A Carolina. Recuerdo que hace ya varios años te di aquella tarjeta que decía que “no hay nada más hermoso que el amor que ha sobrellevado las tormentas de la vida”, y la profecía se va cumpliendo, porque nuestro amor es hermoso. Primero en Montevideo, luego sobrevivir a la distancia y ahora en San José, viviendo el sueño, con el pequeño Dante, el mejor regalo que nos pudo haber dado la vida. Como se que no lo voy a poder describir mejor, le robo estas palabras a Drexler para decirte “que el corazón no miente, que afortunadamente, me haces bien, me haces bien, me haces bien”.

Sponsors. I am grateful to the European project LerNet and PEDECIBA Informática, who respectively funded the 75% and 25% of my eighteen months stay in the Netherlands. ANII financed another one month internship at the Utrecht University to be able to meet my supervisor and advance in my thesis. From 2009 to 2011, I received a grant for my PhD studies from the Engineering School of Universidad de la República.

Contents

Preface 1

1 Introduction 9
1.1 Extensible Languages . . . 10
1.1.1 Language Extension . . . 13
1.2 First-Class Syntax . . . 15
1.2.1 Grammar Representation . . . 15
1.2.2 Grammar Extensions . . . 15
1.2.3 Closed Grammars . . . 16
1.3 First-Class Semantics . . . 17
1.3.1 Definition of the Language Semantics . . . 17
1.3.2 Extending the Semantics . . . 20
1.4 Related Work . . . 23
1.5 Outline of the Thesis . . . 25

2 Constructing and Composing Efficient Top-down Parsers at Runtime 27
2.1 Introduction . . . 27
2.2 A Better Read . . . 31
2.2.1 Deriving *Gram* . . . 32
2.2.2 Grouping . . . 34
2.2.3 LC-Transformation . . . 34
2.2.4 Left-Factoring . . . 36
2.3 Representing Data Type Grammars . . . 37
2.3.1 Typed References and Environments . . . 37
2.3.2 Typed Grammar Representations . . . 38
2.3.3 Typed Grammar Representations for Data Types . . . 38
2.3.4 Representing Mutually Recursive Data Types . . . 40
2.3.5 Non Representable Data Types . . . 45
2.3.6 Deriving Data Type Grammars . . . 45
2.4 Typed Transformations . . . 46
2.4.1 Transformation Library . . . 46
2.4.2 Implementation of Grouping . . . 49
2.5 Efficiency . . . 53
2.5.1 *gread* versus *read* . . . 53
2.5.2 *gread* versus *leftcorner* . . . 55
2.6 Conclusions and Future Work . . . 55

3 Grammar Fragments Fly First-Class 57
3.1 Introduction . . . 57
3.2 Context-Free Grammar . . . 58
3.2.1 Language Extension . . . 59
3.3 Grammar Representation . . . 61
3.3.1 From Grammar to Parser . . . 62
3.3.2 Applicative Interface . . . 63
3.4 Extensible Grammars . . . 64
3.4.1 Grammar Extensions . . . 64
3.5 Closed Grammars . . . 67
3.5.1 Finding Empty Productions . . . 69
3.5.2 Removal of Empty Productions . . . 73
3.6 Fixpoint Production . . . 76
3.6.1 Fixpoint Removal . . . 77
3.7 Related Work and Conclusions . . . 77

4 Attribute Grammars Fly First-Class 79
4.1 Introduction . . . 79
4.2 HList . . . 85
4.2.1 Heterogeneous Lists . . . 86
4.2.2 Extensible Records . . . 87
4.3 Rules . . . 89
4.3.1 Rule Definition . . . 91
4.3.2 Monadic Rule Definition . . . 95
4.3.3 Applicative Rule Definition . . . 96
4.4 Aspects . . . 97
4.4.1 Aspects Combination . . . 98
4.5 Semantic Functions . . . 99
4.5.1 The Knit Function . . . 100
4.6 Common Patterns . . . 101
4.6.1 Copy Rule . . . 102
4.6.2 Other Rules . . . 103
4.7 Defining Aspects . . . 104
4.7.1 Attribute Aspects . . . 105
4.7.2 Default Aspects . . . 106
4.8 Related Work . . . 107
4.9 Conclusions . . . 108

5 Attribute Grammar Macros 111
5.1 Introduction . . . 111
5.2 Attribute Grammar Combinators . . . 113
5.3 Attribute Grammar Macros . . . 118
5.3.1 Mapping a Child to a Child . . . 120
5.3.2 Mapping a Child to a Macro . . . 120

5.3.3 Mapping a Child to a Constant . . . 121
5.4 Accessing Attributes . . . 121
5.5 Attribute Redefinitions . . . 123
5.6 Conclusions and Future Work . . . 124

6 UUAG Meets AspectAG: How to make Attribute Grammars First-Class 125
6.1 Introduction . . . 125
6.2 Attribute Grammars . . . 127
6.2.1 Initial Attribute Grammars . . . 127
6.2.2 Attribute Grammar Extensions . . . 128
6.3 From UUAG to AspectAG . . . 132
6.4 Optimizations . . . 134
6.4.1 Grouping Attributes . . . 134
6.4.2 Static Productions . . . 135
6.4.3 Benchmarks . . . 136
6.5 Conclusions and Future Work . . . 138

7 Case Study - Oberon0 139
7.1 Architecture . . . 140
7.2 Syntax . . . 142
7.3 Aspect Oriented Semantics . . . 145
7.3.1 Name analysis . . . 145
7.3.2 Type checking . . . 147
7.3.3 Source-to-source transformation . . . 149
7.3.4 Code generation . . . 150
7.4 Artifacts . . . 152
7.5 Conclusions . . . 152

8 Conclusions and Future Work 155
8.1 First Class Syntax . . . 155
8.2 First Class Semantics . . . 155
8.3 Their Composition . . . 156
8.4 Future Work . . . 156

A Oberon0 Syntax 159
A.1 Concrete Grammar . . . 159
A.1.1 L1 . . . 159
A.1.2 L2 . . . 161
A.1.3 L3 . . . 161
A.1.4 L4 . . . 162
A.2 Abstract Syntax . . . 163
A.2.1 L1 . . . 163
A.2.2 L2 . . . 165
A.2.3 L3 . . . 165

A.2.4 L4 166

Bibliography **167**

Samenvatting **173**

1 Introduction

Since the introduction of the very first programming languages, and the invention of grammatical formalisms for describing them, people have been looking into how to enable an initial language definition to be extended by someone other than the original language designers. In the extreme case a programmer, starting from an empty initial language, could thus compose his favorite language out of a collection of pre-compiled language-definition fragments. Such language fragments may range from the definition of a simple syntactic abbreviation like list comprehensions to the addition of completely new language concepts, or even extensions to the type system.

In solving the problem of how to compose a compiler, various lines of attack have been pursued. The most direct and least invasive approach, which is so widely applied that one may not recognize it as an approach to the goal sketched above, is to make use of libraries defined in the language itself, thus simulating real extensibility. Over the years this method has been very effective, and especially modern, lazily evaluated, statically typed functional languages such as Haskell serve as an ideal environment for applying this technique; the definition of many so-called combinator libraries in Haskell has shown the effectiveness of this approach, which had been characterized as the construction of *embedded domain specific languages* (EDSL). The ability to define operators and precedences can be used to mimic syntactic extensions. Unfortunately not all programming languages really support this approach very well, given the flood of so-called modeling languages and frameworks from which lots of boilerplate code is generated.

At the other extreme of the spectrum we start from a base language and the *compiler text* for that base language. Just before the compiler is compiled itself, several extra ingredients can be added textually. In this way we get great flexibility and there is virtually no limit to the things we may add. The Utrecht Haskell Compiler [19] has shown the effectiveness of this approach using attribute grammars as the composing mechanism. This approach however is not very practical when defining relatively small language extensions; we do not want every individual user to generate a completely new compiler for each small extension. Another problematic aspect of this approach is that by making the complete text of the compiler available for modification we may also loose important guarantees provided by e.g. the type system of the language being defined; we definitely do not want everyone to mess around with the delicate internals of a compiler for a complex language.

So the question arises of how we can do better than only providing powerful abstraction mechanisms without opening up the whole source of the compiler. The most commonly found approach is to introduce so-called *syntax-macros* [39], which enable the programmer to add syntactic sugar to a language by defining new notation in terms of already existing notation. Despite the fact that this approach may be very

effective, it also has severe shortcomings; as a consequence of mapping the new constructs onto existing constructs and performing any further processing such as type checking on this simpler, but often more detailed program representation, feedback from later stages is given in terms of invisible intermediate program representations. Hence the implementation details shine through, and error messages produced can be confusing or even incomprehensible.

Given the above considerations we impose some quite heavy restrictions on ourselves. In the first place extensions should go beyond merely syntactic extensions as is the case with the original syntax macros, which only map new syntax onto existing syntax; we want also to gain access to the part of the compiler which deals with the static semantics, e.g., in order to report errors in terms of the extended syntax instead of the original one. We seek extension at the semantic level, i.e. by using some sort of plug-in architecture; we will do so by constructing a core compiler as a collection of pre-compiled components, to which extra components can be added and for which existing components can be redefined at will. The questions we answer in this thesis are how to compose a compiler out of *separately compiled and statically type checked* language-definition fragments and how to construct such fragments using a domain specific language embedded in Haskell.

In this chapter we show how several related techniques we will introduce through the thesis can be combined in a unified approach to construct extensible compilers. The solution we present builds on:

- the description of typed grammar fragments as first class Haskell values, and the possibility to construct parsers out of them
- the possibility to deal with attribute grammars as first class Haskell values, which can be transformed, composed and finally evaluated.

These techniques make use of many well-known Haskell extensions, such as multi-parameter type classes, functional dependencies, generalised algebraic data types and arrow notation. For simplicity, in the rest of the chapter we will refer to this just as Haskell.

In Section 1.1 we introduce the syntax of a small language and its extension, as it is to be provided by the language definer and extender. In Section 1.2 we show the techniques we use to represent the syntax, and in Section 1.3 show the corresponding static semantics parts. We close by discussing related work and outline the rest of the thesis.

1.1 Extensible Languages

In this section we show how to express extensible languages. The architecture of our approach is depicted in Figure 1.1; boxes represent (groups of Haskell) modules and arrows are **import** relations.

In the rest of the section we will take a detailed look at each module, and how everything fits together in the construction of a compiler. Our running example will

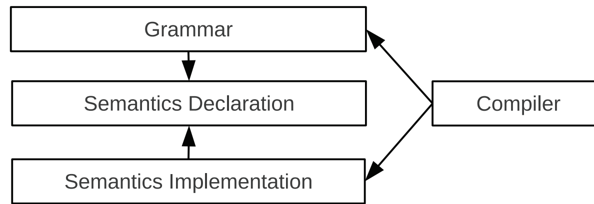

Figure 1.1: Initial Language

be a small expression language with declarations, to which we will refer as the *initial grammar*:

$$
\begin{array}{lll}
\mathit{root} & ::= & \mathit{decls}\ \texttt{"main"}\ \texttt{"="}\ \mathit{exp} \\
\mathit{decls} & ::= & \mathit{var}\ \texttt{"="}\ \mathit{exp}\ \mathit{decls} \mid \mathit{empty} \\
\mathit{exp} & ::= & \mathit{exp}\ \ \texttt{"+"}\ \mathit{term}\ \ \mid \mathit{term} \\
\mathit{term} & ::= & \mathit{term}\ \texttt{"*"}\ \mathit{factor} \mid \mathit{factor} \\
\mathit{factor} & ::= & \mathit{int} \mid \mathit{var}
\end{array}
$$

Note that this concrete grammar uses the syntactic categories *exp*, *term* and *factor* to represent operator precedences.

```
gramIni sf = proc () → do
  rec root   ← addNT ≺ ⟦ (semRoot  sf) decls "main" "=" exp ⟧
      decls  ← addNT ≺ ⟦ (semDecls sf) var   "=" exp decls ⟧
                      <|> ⟦ (semNoDecl sf) ⟧
      exp    ← addNT ≺ ⟦ (semAdd   sf) exp   "+" term   ⟧ <|> ⟦ term ⟧
      term   ← addNT ≺ ⟦ (semMul   sf) term  "*" factor ⟧ <|> ⟦ factor ⟧
      factor ← addNT ≺ ⟦ (semCst   sf) int ⟧ <|> ⟦ (semVar sf) var ⟧
  exportNTs ≺ exportList root $ export ntDecls decls . export ntExp    exp
                              . export ntTerm term . export ntFactor factor
```

Figure 1.2: Initial Language Grammar

To implement this language fragment, a language implementer has to provide the Haskell code of Figure 1.2, expressing himself using our `murder`[1] combinator library (of course one might generate this from the grammar description) and the arrow-interface[2]. This corresponds to the module *Grammar* in Figure 1.1. Without delving into details in this section, observe that the context-free grammar just given can be immediately recognized in the structure of the code. This grammar fragment description consists of a sequence of transformations, introducing new non-terminals to the

[1] MUtually Recursive Definitions Explicitly Represented: `http://hackage.haskell.org/package/murder`

[2] Using Arrow syntax [50], which is inspired by the **do**-notation for *Monads*

grammar. The notation is to be read as *output* $\leftarrow$ *transformation* $\prec$ *input*. Each non-terminal (syntactic category) of the context free grammar is introduced (using *addNT*) by defining a list of productions (alternatives) separated by <|> (choice) operators, where each production contains a sequence of elements to be recognized.

The parameter *sf* is a record containing the "semantics of the language". The type of this record is declared in the module *Semantics Declaration*, for example:

```
data SemLang decls main rs name val rest ds nds
             al ar as ml mr ms value cs var vs
  = SemLang { semRoot    :: decls → main → rs
            , semDecls   :: name → val → rest → ds
            , semNoDecl  :: nds
            , semAdd     :: al → ar → as
            , semMul     :: ml → mr → ms
            , semCst     :: value → cs
            , semVar     :: var → vs }
```

The functions contained in the record (accessed as e.g. *semMul sf*) describe how to map the semantic values associated with the abstract syntax trees corresponding to the non-terminals in the right-hand side of a production onto the semantic value of the left hand side of that production (and eventually the value associated with the root of a parse tree). We call these *semantic functions*, because they give meaning to the constructs of the language. The record is parametrized by the types that compose the types of its fields, i.e. the semantic functions of the productions. Such a record describes the abstract syntax of the language.

In Section 1.3 we show how to construct and adapt the semantic functions (module *Semantics Implementation* in Figure 1.1) using the `uuagc`-system combined with a first-class attribute grammar library. We map the abstract parse tree of the program onto a call tree of the semantic function calls. The resulting meaning of a parse tree is a function which can be seen as a mapping from the inherited to the synthesized attributes. Thus, a production is defined by a semantic function and a sequence of non-terminals and terminals (`"*"`), the latter corresponding to literals which are to be recognized.

As usual, some of the elementary parsers return values which are constructed by the scanner. For such terminals we have a couple of predefined special cases, such as *int*, which returns the integer value from the input and *var* which returns a recognized variable name.

An initial grammar is also an *extensible grammar*. It exports (with *exportNTs*) its starting point (*root*) and a list of *exportable non-terminals* each consisting of a label (by convention of the form *nt*...) and the collection of right hand sides. These right hand sides can be used and modified in future extensions.

Figure 1.3 contains a fragment of a (very simple) compiler of the example language; it corresponds to the module *Compiler* of Figure 1.1. The function *genCompiler* closes a grammar and generates a parser integrated with the semantics for the language starting from the first non-terminal, which in our case is *root*. The left-corner

```
import Grammar        (gramIni)
import SemanticsImpl  (semIni)
compiler = genCompiler (gramIni semIni)
```

Figure 1.3: Initial Language Compiler

transform is applied to remove possible left recursion from the grammar, in order to use straightforward top-down parsing techniques in the actual parsing process.

1.1.1 Language Extension

The language (and thus compiler of that language) can be extended without having either to re-compile or to inspect the grammar and semantic components of the compiler for the initial language. Figure 1.4 shows the structure of a compiler produced as an extension of an initial language including the introduction of new syntax. In this case both the grammar and the semantics are being extended.

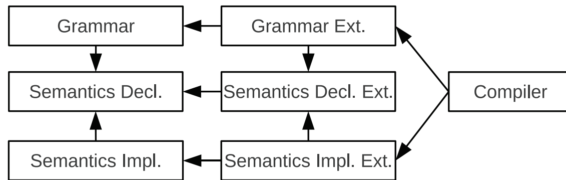

Figure 1.4: Language Extension

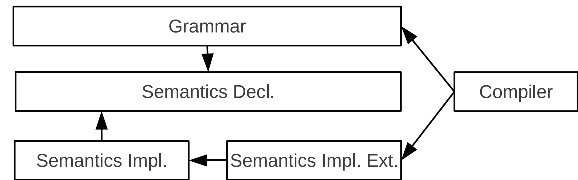

Figure 1.5: Lang. Semantics Modification

If the extension only involves modification of the semantics (e.g. to add new aspects or redefine existing ones), then it suffices to add an extension to the module containing the *Semantics Implementation* (Figure 1.5).

In the rest of the section we show how to extend the language just defined by adding new kinds of expressions such as conditional expressions and new syntactic categories such as conditions:

```
factor ::= ... | "if" cond "then" exp "else" exp
cond   ::= exp "==" exp | exp ">" exp
```

The grammar extension *gramExt* is again defined as a Haskell value, which *imports* an existing set of productions and builds an extended set, as shown in Figure 1.6. In this case the type of the *sf* record, defined in the module *Semantics Declaration Extension*, is:

```
data SemLangExt cnd thn els is el er es gl gr gs
  = SemLangExt { semIf :: cnd → thn → els → is
               , semEq :: el → er → es
               , semGr :: gl → gr → gs }
```

```
gramExt sf = proc imported → do
  let exp    = getNT ntExp    imported
  let factor = getNT ntFactor imported
  rec addProds ≺ (factor, ⟦ (semIf  sf) "if" cond "then" exp "else" exp ⟧)
      cond  ← addNT  ≺ ⟦ (semEq sf) exp "==" exp ⟧
                     <|> ⟦ (semGr sf) exp ">"  exp ⟧
  exportNTs ≺ extendExport imported (export ntCond cond)
```

Figure 1.6: Language Extension

We first show how to combine previously defined productions with the newly defined productions into an extended grammar: for each non-terminal to be extended, or used in an extension, we retrieve its list of productions (using *getNT*) from the *imported* non-terminals, and add new productions to this list using *addProds*. For example, for *factor* the new **if** ... **then** ... **else**... production is added by:

```
let exp    = getNT ntExp    imported
let factor = getNT ntFactor imported
rec addProds ≺ (factor, ⟦ (semIf sf) "if" cond "then" exp "else" exp ⟧)
```

Extra non-terminals can be added as well using *addNT*; in the example we add the non-terminal *cond* with its two productions to represent some simple conditions:

```
cond ← addNT ≺ ⟦ (semEq sf) exp "==" exp ⟧ <|> ⟦ (semGr sf) exp ">" exp ⟧
```

Finally, we extend the list of exportable non-terminals with (some of) the newly added non-terminals, so they can be extended by further fragments elsewhere:

```
exportNTs ≺ extendExport imported (export ntCond cond)
```

Because both *gramIni* and *gramExt* are proper Haskell values, which are separately defined in different modules which can be compiled separately, we claim that the term *first-class grammar fragments* is justified.

```
import GrammarExt        (gramIni, gramExt)
import SemanticsImplExt (semIni,  semExt)
compiler = genCompiler (gramIni semIni +>> gramExt semExt)
```

Figure 1.7: Extended Language Compiler

The extended language compiler is shown in Figure 1.7. The operator (+>>) composes an initial grammar with its extension, returning a new (initial) grammar. The function *genCompiler* makes sure that all existing references to non-terminals eventually refer to the final version of the definitions for these non-terminals.

1.2 First-Class Syntax

In this section we introduce the `murder` library, which we use to define and combine grammars. The library is based on the typed representation of grammars and the typed transformations [6] of these grammars. A detailed description of the library will be provided in Chapter 3.

1.2.1 Grammar Representation

We use a representation of grammars as typed abstract syntax [7] based on the use of Generalized Algebraic Data Types [54]. The idea is to indirectly refer to non-terminals via references encoded as types. Such references type-index into an environment holding the actual collection of productions for non-terminals. This enforces that the productions occurring in an environment can only contain references to non-terminals that belong to the environment in question. A grammar is a value with type *Grammar a*, where the type *a* is the type of a witness of a complete successful parse starting from the root nonterminal of the grammar.

1.2.2 Grammar Extensions

Grammar definitions (Figure 1.2) and extensions (Figure 1.6) are typed transformations of values of type *Grammar*, implemented using the library `TTTAS`[3] (Typed Transformations of Typed Abstract Syntax). `TTTAS` enables us to represent typed transformation steps, (possibly) extending a typed environment. In other words, by using typed transformations when adding non-terminals and productions to a grammar, we will always construct a grammar that is assured to be well-typed again.

In `TTTAS` the transformations are represented as *Arrow*s [29]. *Arrow*s are a generalization of *Monad*s, modeling a computation that takes inputs and produces outputs. In our case the computation maintains a state containing the environment mapping the non-terminals of the grammar onto the thus far defined productions. We use the input and output of the arrows to read and write data controlling the transformation process.

Both extensible grammars and grammar extensions have to export their starting point and their list of *exportable non-terminals*, which can be used and/or modified by future extensions. We encode this data in a value of type *Export*, which is constructed using the function *exportList*, and extended with *extendExport*.

The only difference between extensible grammars and grammar extensions is that a grammar extension has to import the list of non-terminals it will extend, while an initial grammar does not import anything.

Thus, the definition of an extensible grammar, like the one in Figure 1.2, has the following shape:

$$
\begin{aligned}
&\mathit{gramIni} = \mathbf{proc}\ ()\ \rightarrow \mathbf{do}\ \ ...\\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad \mathit{exportNTs} \prec \mathit{exported}
\end{aligned}
$$

[3] `http://hackage.haskell.org/package/TTTAS`

where the **proc** () part indicates that *gramIni* is a typed transformation that takes just () as input and returns as output a value (*exported*) of type *Export*. With *exportNTs* we inject the *Export* value in the transformation in order to return it as output.

The definition of a grammar extension, like the one in Figure 1.6, has the shape:

$$gramExt = \textbf{proc}\ imported \rightarrow \textbf{do}\ \ldots$$
$$exportNTs \prec exported$$

Now in order to extend a grammar with a grammar extension all we have to do is to compose both transformations by connecting the output of the first to the input of the second. This is the role of the operator (+>>), used in Figure 1.7, which is a (type) specialized version of the *Arrow*'s composition (>>>).

To add a new non-terminal to the grammar boils down to adding a new term to the environment using the transformation *addNT*. The input to *addNT* is the initial list of alternative productions for the non-terminal and the output is a non-terminal symbol, i.e. the index of the newly added non-terminal in the new grammar. Thus, when in Figure 1.2 we write:

$$exp \leftarrow addNT \prec productions$$

we are adding the non-terminal for the expressions, with the list of productions *productions* passed as a parameter, and we bind to *exp* a symbol holding the reference to the added non-terminal so it can be used in the definition of this or other non-terminals. The list of alternative *productions* is expressed in an applicative style; i.e. in terms of *pure*, (<*>), (<*) and (<|>), or using the brackets $[\![$ and $]\!]$. These brackets are inspired by the *idioms* approach as introduced by McBride [44]. The brackets $[\![$ and $]\!]$ are the LaTeX representations of the Haskell identifiers *iI* and *Ii*, which come with a collection of Haskell class and instance declarations which together allow us to write $[\![$ (*semMul sf*) *term* "*" *factor* $]\!]$ instead of the more elaborate text:

pure (*semMul sf*) <*> *sym term* <* *tr* "*" <*> *sym factor*

Adding new productions to an existing non-terminal boils down to appending the extra productions to the list of existing productions of that non-terminal. Figure 1.6 contains an example of adding a production to the non-terminal *factor*. The transformation *addProds* takes as input a pair with a reference to the non-terminal to be extended and the list of productions to add:

$$addProds \prec (nonterminal, productions)$$

In this case the output is irrelevant, since no new references are created as a result of this extension.

1.2.3 Closed Grammars

We can run the transformation by closing the grammar; i.e. all references are made to point to the latest version of their corresponding non-terminals. Thus, a call to

closeGram starts with an empty grammar, and applies to it all the transformations defined in the grammar description to obtain the defined grammar.

$genCompiler = (parse \ . \ generate \ . \ leftcorner) \ closeGram$

The type of a closed grammar is *Grammar a*, where *a* is a phantom type [28] representing the type of the start non-terminal.

Since this grammar can be left-recursive we have to apply the *leftcorner* [7] typed transformation in order to remove potential left-recursion:

$leftcorner :: Grammar \ a \rightarrow Grammar \ a$

The function *generate* generates a parser integrated with the semantics for the language starting from the first non-terminal, which in our case is *root*.

$generate :: Grammar \ a \rightarrow Parser \ Token \ a$

Finally, *parse* parses the input program while computing the meaning of that program. Currently we can generate either `uulib`[4] or `uu-parsinglib`[5] parsers.

$parse :: Parser \ Token \ a \rightarrow [Token] \rightarrow ParseResult \ a$

1.3 First-Class Semantics

In this section we complete the example by showing how we use attribute grammars to define the static semantics of the initial language and how such definitions can be redefined when the language is extended.

An Attribute Grammar describes for a context-free grammar how each node in a parse tree is to be decorated with a collection of values, called *attributes*. For each attribute we have a defining expression in which we may refer to other "nearby" attributes, thus defining a data-flow graph based on the abstract syntax tree. An attribute grammar evaluator schedules the computation of these expressions, such that the attributes we are interested in eventually get computed.

1.3.1 Definition of the Language Semantics

To define the static semantics of a language we use the `AspectAG`[6] embedding of attribute grammars in Haskell. We introduce `AspectAG` in Chapter 4. In order to be able to redefine attributes or to add new attributes later, it encodes the lists of inherited and synthesized attributes of a non-terminal as an *HList*-encoded [35] value, indexed by types using the Haskell class mechanism. In this way the *closure test* of

[4] `http://hackage.haskell.org/package/uulib`
[5] `http://hackage.haskell.org/package/uu-parsinglib`
[6] `http://hackage.haskell.org/package/AspectAG`

```
SemanticsImpl.agi

  DATA Root  | Root decls : Decls main : Expr
  DATA Decls | Decl name : String val : Expr rest : Decls
             | NoDecl
  DATA Expr  | Add al : Expr ar : Expr
             | Mul ml : Expr mr : Expr
             | Cst value : Int
             | Var var : String


  ATTR Root Decls  SYN spp  : PP_Doc
  ATTR Root Expr   SYN sval : Int
  ATTR Decls Expr  INH ienv : [(String, Int)]
  ATTR Decls       SYN senv : [(String, Int)]
```

Figure 1.8: Language semantics

the attribute grammar (each attribute has exactly one definition) is realized through the Haskell class system. Thus, attribute grammar fragments can be individually type-checked, compiled, distributed and composed to construct a compiler. Albeit easy to use for the experienced Haskell programmer, it has a rather steep learning curve for the uninitiated. A further disadvantage is that the approach is relatively expensive: once the language gets complicated (in our Haskell compiler `UHC` [19] some non-terminals have over 20 attributes), the cost of accessing attributes may eventually overshadow the cost of the actual computations.

For those reasons we define in Chapter 6 an extension to the `uuagc` compiler [64], that generates `AspectAG` code fragments from original `uuagc` sources. This tool enables a couple of optimizations to the `AspectAG` code: we limit both our reliance on the *HList*-encoding, resulting in a considerable speed improvement, and allow existing uuagc code to be reused in a flexible environment.

With the `--aspectag` option we make `uuagc` generate `AspectAG` code out of a set of `.ag` files and their corresponding `.agi` files. An `.agi` file includes the declaration of a grammar and its attributes (the interface), while the **SEM** blocks specifying the computation of these attributes are included in the `.ag` file (the implementation).

In the rest of the chapter we will show examples written in the `uuagc` language. Although another valid option would have been to implement the semantic functions directly in `AspectAG`, or to use a hybrid approach.

Figure 1.8 shows the `.agi` file for the semantics of our initial language. Notice that the grammar defined here is not exactly the same as the context-free grammar of the language, since our attribute grammars are built on top of the *abstract syntax* of the language. We define attributes for the following aspects: pretty printing, realized by the synthesized attribute *spp*, which holds a pretty printed document

SemanticsImpl.ag

```
SEM Root  | Root     lhs.spp   = decls.spp <-> "main =" <#> main.spp
SEM Decls | Decl     lhs.spp   = name <#> "=" <#> val.spp <-> rest.spp
          | NoDecl   lhs.spp   = empty
SEM Expr  | Add      lhs.spp   = al.spp <#> "+" <#> ar.spp
          | Mul      lhs.spp   = ml.spp <#> "*" <#> mr.spp
          | Cst      lhs.spp   = pp (show value)
          | Var      lhs.spp   = pp var

SEM Root  | Root     lhs.sval  = main.sval
SEM Expr  | Add      lhs.sval  = al.sval + ar.sval
          | Mul      lhs.sval  = ml.sval * mr.sval
          | Cst      lhs.sval  = value
          | Var      lhs.sval  = case lookup var lhs.ienv of
                                   Just v   → v
                                   Nothing  → 0

SEM Root  | Root     decls.ienv = [ ]
                     main.ienv  = decls.senv
SEM Decls | Decl     val.ienv   = [ ]
                     rest.ienv  = (name, val.sval) : lhs.ienv

SEM Decls | Decl     lhs.senv = rest.senv
          | NoDecl   lhs.senv = lhs.ienv
```

Figure 1.9: Language semantics

of type *PP_Doc*, and expression evaluation, realized by the synthesized attribute *sval* of type *Int*, which holds the result of an expression, and an inherited attribute *ienv* which holds the environment ([(*String*, *Int*)]) in which an expression is to be evaluated. *Synthesized attributes* take their definition "from below", using the values of the synthesized attributes of the children of the node the attribute is associated with and the inherited attributes of the node itself. An *inherited attribute* is defined "from above": in its defining expression we may refer to the inherited attributes of its parent and the synthesized attributes of its siblings.

Keep in mind that we chose these trivial semantics in order to keep the example simple, and focus on the features of the technique. A real compiler should involve more complex tasks such as type-checking, optimization and code generation.

Figure 1.9 shows the `.ag` file including the implementation of the attributes declared above. In a **SEM** block we specify how attributes of a production are to be computed out of the attributes from the left hand side and children of the production. The

defining expressions at the right hand side of the = signs are almost plain Haskell code, using minimal syntactic extensions to refer to attributes. We refer to a synthesized attribute of a child using the notation *child.attribute* and to an inherited attribute of the production itself (the left-hand side) as **lhs**.*attribute*. Terminals are referred to by the name introduced in the **DATA** declaration. For example, the rule for the attribute *ienv* for the child *rest* of the production *Decl* extends the inherited list *ienv* by a pair composed of the *name* used in the declaration and the value *sval* of the child with name *val* (*val.sval*).

The pretty-printing attribute is defined for each production by combining the pretty printed children using the pretty printing combinators from the `uulib` library: (`<#>`) for horizontal (beside) composition, (`<->`) for vertical (above) composition, and *pp* to pretty print a string.

The semantics of the expression evaluation (*sval*) is intuitive. Variables of the main expression are located in an environment constructed as follows:

- the declarations sub-tree (*decls*) receives an empty environment *ienv* and extends it through the list of declarations with the values resulting from the evaluation of the expression in the right hand side of each declaration
- the complete environment is passed "up" to the root in the attribute *senv*
- this environment is distributed into the *main* expression as *ienv*

The rules to describe the computation of the attribute *ienv* for the productions *Add* and *Mul* of the non-terminal *Expr* are omitted. In this case, rules that copy the attribute (unchanged) to the children are inserted automatically by `uuagc`. The library `AspectAG` includes a function *copy* that implements the same behaviour.

Notice that the expressions of the declarations (*Decl*) should be closed, since they are (in our current definition) evaluated in an empty environment.

A semantic function (*sem_Prod*) is generated for each production (*Prod*) of the grammar. Thus, to complete our initial language of Section 1.1 we only need to construct the record *semIni* with these semantic functions:

$$
\begin{aligned}
semIni = SemLang \{ & semRoot \ = sem_Root \\
, & semDecls = semDecls, semNoDecl = sem_NoDecl \\
, & semAdd \ \ = sem_Add, semMul \quad = sem_Mul \\
, & semCst \ \ \ = sem_Cst, \ semVar \quad \ = sem_Var \}
\end{aligned}
$$

1.3.2 Extending the Semantics

Having first-class attribute grammars enables us to have a compiled definition of the semantics of a language and to introduce relatively small extensions to it later, without the need to either reconstruct the whole compiler, or to require the sources of the core language to be available.

SemanticsImplExt.agi

```
EXTENDS "SemanticsImpl"
ATTR Root Decls Expr SYN serr USE {++} {[]} : [String]
```

SemanticsImplExt.ag

```
SEM Decls | Decl lhs.serr = (case lookup name lhs.ienv of
                               Just _   → [name ++ " duplicated"]
                               Nothing → []) ++ val.serr ++ rest.serr
SEM Expr  | Var  lhs.serr = case lookup var lhs.ienv of
                               Just _   → []
                               Nothing → [var ++ " undefined"]
```

Figure 1.10: Language Extension: Errors

In this subsection we show, by using some simple examples, how extensions can be defined.

The use of variables and declarations in the example language can be erroneous. Thus we introduce in Figure 1.10 an extra synthesized attribute (*serr*) in which we collect error messages corresponding to duplicated definitions and referring to undefined variables. This extension to the language corresponds to the kind of extensions described in Figure 1.5, because it only involves a change at the semantic level; no new syntax is added.

The keyword **EXTENDS** in Figure 1.10 is used to indicate which attribute grammar is being extended. The **USE** clause included in the declaration of the synthesized attribute *serr* indicates that, for the productions where the definition is omitted, the attribute will be computed by *collecting* the synthesized attributes *serr* of the children of the production. If the collection is empty (*NoDecl* and *Cst*) the value of the attribute is []. In the other case (*Root*, *Add* and *Mul*) the values are combined with the operator (++). The same can be done in `AspectAG` using the function *use*.

In Section 1.1.1 we extended the initial language with a conditional expression. The implementation of the semantics of this extension, which corresponds to the extensions depicted in Figure 1.4, is shown in Figure 1.11. In this case not only new attributes are added, but we also extend the abstract syntax with a new kind of node, and define a new production for the existing non-terminal *Expr*. Since semantics extensions are pairwise incremental, we also have to define the computation of the attribute *serr* for the newly included productions.

`AspectAG` enables the *redefinition* of already existing attributes. In `uuagc` (extended to generate `AspectAG`) we use :=, instead of =, to declare attribute redefinitions. For example, in the extension of Figure 1.12, the attribute *ienv* is redefined to allow the use of variables in the expressions of the declarations. Notice how a very small change to the attribute grammar definitions may influence the overall language considerably.

```
SemanticsImplExt2.agi

  EXTENDS "SemanticsImplExt"
  DATA Expr  | If cnd : Cond thn : Expr els : Expr
  DATA Cond | Eq el : Expr er : Expr
            | Gr gl : Expr gr : Expr
  ATTR Cond SYN sval : Bool
  ATTR Cond SYN spp  : PP_Doc
  ATTR Cond SYN serr USE {++} {[]} : [String]
```

```
SemanticsImplExt2.ag

  SEM Expr  | If  lhs.sval = if cnd.sval then thn.sval else els.sval
  SEM Cond | Eq lhs.sval = el.sval ≡ er.sval
                lhs.spp  = el.spp <#> "==" <#> er.spp
           | Gr lhs.sval = gl.sval > gr.sval
                lhs.spp  = gl.spp <#> ">" <#> gr.spp
```

Figure 1.11: Language Extension: If

```
SemanticsImplExt3.agi

  EXTENDS "SemanticsImplExt2"
```

```
SemanticsImplExt3.ag

  SEM Decls | Decl val.ienv := rest.senv
```

Figure 1.12: Language Extension: Variables in declarations

Usually we do not want to define the complete semantics of a syntactic extension from scratch. If we limited ourselves to a syntax-macro like mechanism, where new syntax is mapped onto existent syntax, it would be useful to have a way to express this mapping at the semantic level. In Chapter 5 we extend `AspectAG` with an *agMacro* combinator that enables us to define the attribute computations of a new production in terms of the attribute computations of existing productions. Thus, we can define the extensions *Sq*, computing the square of an expression, *Pyth* for the sum of the squares of two expressions, and *Db* to double an expression as in Figure 1.13. The fragment *Sq se* : *Expr* ⇒ (*Mul se se*) defines a production *Sq* with a child *se*, where the computation of its semantics is based on the computation of the semantics of the production *Mul*, but mapping both children to *se*. In the case of *Pyth*, macros are

```
SemanticsImplExt4.agi

  EXTENDS "SemanticsImplExt3"
  DATA Expr | Sq se : Expr                ⇒ (Mul se se)
            | Pyth pl : Expr pr : Expr    ⇒ (Add (Sq pl) (Sq pr))
            | Db de : Expr                ⇒ (Mul (Cst 2) de)
```

Figure 1.13: Language Extension: Sq and Pyth

```
SemanticsImplExt4.ag

  SEM Expr | Sq   lhs.spp := "sq" <#> se.spp
           | Pyth lhs.spp := "pyth" <#> pl.spp <#> pr.spp
           | Db   lhs.spp := "db" <#> de.spp
```

Figure 1.14: Pretty printing redefinition

used recursively to define the mapping of the children of the production *Add*.

Sometimes we will need to define a special semantics for certain attributes of the production. For example, with the definition of *Sq* of Figure 1.13, if we pretty print the expression **pyth** 3 4 the result will be $3 * 3 + 4 * 4$, since that was the abstract syntax tree to which it was mapped in order to compute its semantics. This however is likely not to be the desired behavior. Fortunately we are able to redefine attribute computations in such cases! Thus, in the corresponding `.agi` file (Figure 1.14) we redefine the semantics of the pretty printing aspect.

1.4 Related Work

Although syntax extensions are not commonly supported in typed languages, there is a long tradition in languages like Lisp [75], Scheme [2], Prolog [1], and more recently Stratego [11]. For these syntactically very parsimonious languages a pressing need for such a facility exists, and the absence of a rich type system does not provide a burden for its implementation. We quote Fisher and Shivers [22] who say *"Once one has become accustomed to such a powerful tool, it is hard to give up. When we find ourselves writing programs in languages such as Java, SML, or C, that is, that lack Scheme's syntax extension ability- we find that we miss it greatly"*. Having made this observation they introduce the Ziggurat [23] system, which aims at the same goal as this thesis; the underlying technology is completely different though. They use a delegation based system with which the semantics associated with the node in an abstract syntax tree can be updated. By using Lisp as their implementation language they do not have to cope with the problems posed by the Haskell type system; on the other hand the users of the Ziggurat system do not have the advantages associated

with having a typed implementation language. We believe that having a statically typed implementation language is a great advantage, and we happily rephrase the above quote: *"Once one has become accustomed to the advantages of a static type system, it is hard to give up. When we find ourselves writing programs in languages such as Lisp, PHP, Ruby and JavaScript, that lack Haskell's type and class system-we find that we miss it greatly"*.

Another distinguishing feature is that our underlying technology for describing the static semantics is based on attribute grammars. Attribute grammars have proven themselves extremely useful for compositional language definitions. Adams [3] proposed a set of tools for modular syntax and modular attribute grammars in an untyped setting. Among many others, the attribute grammars systems LISA [45], JastAdd [21], Silver [68] and Kiama [60], have successfully tackled the problem of defining modular extensible compilers in a typed context.

Most of these systems, like `uuagc`, have a generative approach to compositionality; i.e. take the sources of all the composing modules and generate a monolithic system in a host language. Therefore, they do not provide separate compilation. An exception is Kiama, which is embedded as a library in Scala, and supports composition by using *mixins* and *traits*. From this point of view, Kiama is closely related to `AspectAG`, although the former is not able to perform well-formedness checks (such as the closure test) to a composed grammar, unless the grammar is declared as non-extensible. The design of `AspectAG` is inspired by [16], which represents attributions using Rémy-style records, instead of the type-level programming techniques.

LISA and Silver include parser generators to construct parsers out of the composed grammars. Since we do not have access to the source of the composing grammars, we use typed grammar transformations and parser combinators to generate (left-recursion free) top-down parsers *on the fly*. Neither Kiama nor JastAdd provides support for concrete syntax specification and parsing.

All the systems support synthesized and inherited attributes, but some of them extended the model with some new features. Silver includes *forwarding*, to allow productions to implicitly define the computation of some attributes by translation. This functionality is very similar to the provided in `AspectAG` by the combination of *agMacro*s and attribute redefinitions. JastAdd and Kiama support *reference attributes*, i.e. attributes that refer to other tree nodes. This is useful in writing compilers, because it allows one to model language relations (such as the use and declaration of variables and types) as references inside the abstract syntax tree. We do not support this kind of attribute.

Finally, we use Haskell, a strongly-typed pure functional programming language, to define the attribute computations. We think it fits perfectly to the declarative nature of attribute grammars. In cases where imperative languages like Java (JastAdd, LINDA) are used, it becomes impossible to control the absence of side-effects. Silver defines its own language which is declarative and strongly-typed, although more limited.

1.5 Outline of the Thesis

The rest of the thesis is organized as follows:

- In Chapter 2 we describe our approach to construct efficient parsers for the language of data types in a compositional way. This chapter is an adapted version of our Haskell 2008 paper "Haskell, Do You Read Me?: Constructing and composing efficient top-down parsers at runtime" [71].
- In Chapter 3 we extend our technique to first-class context-free grammars, introducing an unrestricted, applicative interface for constructing them. This chapter is an adapted and extended version of our LDTA 2012 paper "Grammar Fragments Fly First-Class" [70].
- In Chapter 4 we introduce an embedding of attribute grammars in Haskell. This chapter is an adapted and extended version of our ICFP 2009 paper "Attribute Grammars Fly First-Class: How to do aspect oriented programming in Haskell" [73].
- In Chapter 5 we extend our embedding with attribute grammar macros, a mechanism which allows us to express semantics in terms of already existing semantics. This chapter is an adapted and extended version of our SBLP 2012 paper "Attribute Grammar Macros" [69].
- In Chapter 6 we describe an extension of `uuagc` to generate (and optimize) attribute grammars expressed in our embedding. This chapter is an adapted version of our LDTA 2012 paper "UUAG Meets AspectAG: How to make Attribute Grammars First-Class" [72].
- In Chapter 7 we present a small case study, which we implemented to participate in the LDTA 2011 Tool Challenge. A reduced version of this chapter will be included in a joint paper which is planned to be submitted to Science of Computer Programming.
- Finally, in Chapter 8 we conclude and discuss some directions for future work.

As a formal detail, the Association for Computing Machinery (ACM) has copyright on the paper version of Chapter 2, Chapter 3, Chapter 4 and Chapter 6. Springer has copyright on the paper version of Chapter 5

2 Haskell, Do You Read Me? Constructing and Composing Efficient Top-down Parsers at Runtime

The Haskell definition and implementation of *read* is far from perfect. In the first place *read* is not able to handle the associativities defined for infix operators. Furthermore, it puts constraints on the way *show* is defined, and especially forces it to generate far more parentheses than expected to be able to read back such printed values. Lastly, it may give rise to exponential parsing times. All this is due to the compositionality requirement for *read* functions, which imposes a top-down parsing strategy.

We propose a different approach, based on typed abstract syntax, in which grammars describing the data types are composed dynamically. Using the `TTTAS` transformation library these syntax descriptions are combined and transformed into parsers at runtime, from which the required *read* functions are constructed. In this way we obtain linear parsing times, achieve consistency with the defined associativities, and may use a version of *show* which generates far fewer parentheses, thus improving readability of printed values.

The described transformation algorithm can be incorporated in a Haskell compiler, thus moving most of the work involved to compile time.

2.1 Introduction

In this chapter we propose a solution[1] to a few long standing, related problems in the design of the Haskell *Read* and *Show* classes. We start by explaining the current design, which was considered an optimal point in the design space available at the time of the design of Haskell98 [52].

Consider the following data type, together with the fixity declarations of the operators involved:

```
infixl 5 :<:
infixr 6 :>:
data T1 = T1 :<: T1 | T1 :>: T1 | C1
          deriving (Read, Show)

v  = C1 :>: C1 :<: C1 :<: C1
w = (read "C1 :>: C1 :<: C1 :<: C1" :: T1)
x  = (read (show v)                 :: T1)
```

[1] Available as the `ChristmasTree` library: `http://hackage.haskell.org/package/ChristmasTree`.

Given the fixity declarations, the definition of v is fine. Unfortunately the evaluation of w leads to a runtime error, because *read* is ignorant of the associativities of `:>:` and `:<:`. It is a sad observation that despite all the effort that went into the design of the language, we cannot just take a constant expression out of the program, put it in a file and read it back. Surprisingly, the definition of *show* is such that x is well-defined again.

The second problem relates to the efficiency of the standard implementation of *read*. In a `GHC` bug ticket [51] it is explained why, with the current implementation of *read* and *show*, the following expression takes a long time to be processed, and on some systems may not run at all:

```
read "(((((((((((C1)))))))))))" :: T1
```

To understand what is going on we delve into the internals of the implementation, and the definitions of *read* and *show* from the Haskell98 Report, using a small example.

```
T1 (n) → T1 (5) ":<:" T1 (6)        (n ⩽ 5)
       | T1 (7) ":>:" T1 (6)        (n ⩽ 6)
       | "C1"
       | "(" T1 (0) ")"
```

Figure 2.1: Grammar of the type *T1*

Consider the grammar of Figure 2.1, in which the parameter indicates the priority level at which the non-terminal may occur in an expression. Note how the associativity of the operators is encoded by this parameter: for the first alternative the second occurrence of *T1* in the right hand side has a higher priority.

A second observation is that for $n = 5$, this grammar is actually left recursive because of the first alternative, and thus cannot be parsed by conventional top-down parsing methods, based on recursive descent techniques.

The Haskell98 Report describes how left recursion is avoided by using a modified grammar, in which the priorities of the children are always higher than the priority of the left hand side of the production. So the language which is actually recognised by the generated *read* function is described by the non left-recursive grammar:

```
T1 (n) → T1 (6) ":<:" T1 (6)        (n ⩽ 5)
       | T1 (7) ":>:" T1 (7)        (n ⩽ 6)
       | "C1"
       | "(" T1 (0) ")"
```

Note that this grammar treats all operators as non-associative. The derived instance for *read* is:

```
left_prec  = 5
right_prec = 6
app_prec   = 10
instance Read T1 where
  readsPrec n r
    = readParen (n > left_prec)
        (\r -> [(u :<: v, w) |
            (u, s) <- readsPrec (left_prec + 1) r,
            (":<:", t) <- lex s,
            (v, w) <- readsPrec (left_prec + 1) t]
        ) r
    ++ readParen (n > right_prec)
        (\r -> [(u :>: v, w) |
            (u, s) <- readsPrec (right_prec + 1) r,
            (":>:", t) <- lex s,
            (v, w) <- readsPrec (right_prec + 1) t]
        ) r
    ++ readParen (n > app_prec)
        (\r -> [(C1, s) |
            ("C1", s) <- lex r]
        ) r
```

The function *readParen* requires a pair of parentheses around its parser argument, if its first argument evaluates to *True*. The price we have to pay for avoiding left-recursive grammars, is that we have to place many more parentheses in our expressions. The good news, and the reason that the aforementioned x is well-defined, is that the derived *show* function generates these extra parentheses; the derived *read* is helped to perform its task by the derived *show*, such that $read.show = id$.

By taking a closer look at this code we can now understand the source of the potential exponential parsing times; all three alternatives happily start by accepting a `"("`-symbol – the first one expecting to see a `:<:` after having seen the corresponding closing parenthesis, the second one expecting a `:>:`, and the third one expecting nothing– and if the second input symbol is a `"("` too, all three have three more ways to proceed, leading to an exponential growth in parsing time.

Now consider a expression of the form `C1 :>: (C1 :>: (...))`. Here we do not have the problem of the opening parentheses, but for expressions with more than 10 `C1`s the parsing time grows exponentially too. What is happening? If we split the grammar according to the precedences we can see the problem:

```
T1 (0..5)  -> T1 (6) ":<:" T1 (6) | T1 (6)
T1 (6)     -> T1 (7) ":>:" T1 (7) | T1 (7)
T1 (7..10) -> "C1"
            | "(" T1 (0) ")"
```

Due to the division of the non-terminal *T1* into three non-terminals, new alterna-

tives pointing directly to the next level have to be added to *T1* (0 . . 5) and *T1* (6). Nonterminals *T1* (0 . . 5) and *T1* (6) have a common prefix into their productions. So, each `"C1"` will be parsed twice before making a decision between the alternatives *T1* (7) `":>:"` *T1* (7) and *T1* (7); and, even worse, this process is performed twice before deciding between *T1* (6) `":<:"` *T1* (6) and *T* (6).

One might expect that there is a simple cure for these problems, since the Haskell compiler itself is able to parse the equivalent expression. In the example case a compiler could indeed spend a bit more time in analysing the data type and constructing an equivalent grammar which does not have the identified shortcomings. This leads, using the applicative parser interface [44], straightforwardly to the following combinator based parser for *T1*, using the parser combinators *pChainl* and *pChainr*:

```
infixr 7`pChainl`,`pChainr`
pT1 = (":<:", (:<:)) `pChainl`
      (":>:", (:<:)) `pChainr`
                     (pParens pT1 <|> pToken "C1")
```

Both combinators combine an operator, described by its string representation and a binary function defining its semantics, and a parser for the operands into a parser which recognises a sequence of operands separated by operators. When parsing is completed the combinator *pChainl* builds the result for a left-associative operator and *pChainr* for a right-associative operator.

Unfortunately however the situation is not always so easy to solve. Consider the following definition:

```
infix 5 :+:
infix 6 :*:
data T2 a = T2 a :+: T2 a
          | a    :*: T2 a
          | C2
```

When deriving *read* for *T2*, a Haskell implementation does not generate a parser, but a function that maps a parser (coming from the *Read* dictionary) recognising values of some parameter type *a*, to a parser which recognises values of type *T2 a*. In this way we can handle the situation where the complete grammar is not at hand when building parsers: the parameter of *T2* might be defined in another module, or may not be defined at all.

It now also becomes clear why the strategy chosen in Haskell works; we have limited our languages to a class for which we can build parsers by composing parsers whenever we define new languages by composing languages. Each module happily generates its own instances of the class *Read*, and these values can straightforwardly be combined into the required parser. So the question we answer in this chapter is:

> *"How can we construct efficient parsers for the language of data types in a compositional way?"*.

In the rest of this chapter we show how these problems can be overcome, using a library for transforming typed abstract syntax, the design of which has been described in [6].

Before delving into the technical details we start out by sketching the solution. Parser generators usually perform some form of grammar analysis, but unfortunately the result of such analyses cannot easily be combined into the analysis result for a combined grammar [11, 9]. Since there is no easy way to compose parsers, we take one step back and compose grammars instead, and thus we have to represent grammars as Haskell values. Once the grammars are composed we can build the required parser.

In order to make grammars first-class values we introduce a polymorphic data type *DGrammar a* (DataGrammar), describing grammatical structures which describe *String* values corresponding to values of type *a*. By making values of this data type member of a class:

```
class Gram a where
  grammar :: DGrammar a
```

we can now provide the type of our *read* function, *gread*[2]:

```
read  :: Read  a ⇒ String → a   -- the original
gread :: Gram  a ⇒ String → a   -- our version
```

In Section 2.2 we give a top-level overview of the steps involved. In Section 2.3 we describe how to represent grammars using typed abstract syntax, thus preparing the grammars for the transformations in Section 2.4. In Section 2.5 we spend some words on the efficiency of the overall approach and describe a few open problems and details to pursue further, whereas in Section 2.6 we conclude.

2.2 A Better Read

We obtain a parser for rules of data type *t* by taking the following steps.

deriveGrammar Generate an instance of the class *Gram*. We provide a function *deriveGrammar*, defined using Template Haskell [58], which performs this step, although we would eventually expect a compiler to take care of this. The instance *Gram T1*, describing the structure of the type *T1* is generated by calling:

```
$ (deriveGrammar “T1)
```

In this generated description precedences and associativities are reflected by annotating uses of non-terminals in the right hand side with the precedence of the position at which they occur, and by annotating productions with the level at which they may be applied (as in Figure 2.1). This is similar to the description given in the Haskell98 report.

[2] Available at: `http://hackage.haskell.org/package/ChristmasTree`

group When a grammar refers to other grammars, which are generated separately and probably in a different module, we have to remove these references by combining the separate grammars into a single complete grammar; this corresponds to the dictionary passing for *Read*. Once this is done we know all the precedences of all the non-terminals involved, and we may construct a new grammar using a sufficient number of new non-annotated non-terminals, in which the precedences and associativities are represented in the grammar itself.

leftcorner For all resulting left-recursive grammars (or parts thereof) we perform the Left-Corner transform [7]. The LC-transform is a relatively straightforward transformation which maps a grammar onto an equivalent grammar which is not left-recursive.

leftfactoring Apply left-factoring to the resulting grammar, in order to remove the source of inefficiencies we have seen in section 1.

compile Convert the grammar into a parser. We use the parser combinators included in the `uulib` [62] package in order to construct a fast parser.

parse Add a call to this parser, a check for a successful result and the generation of an error message in case of failure.

All these steps are visible as individual functions in *gread*:

$$
\begin{aligned}
&gread :: (Gram\ a) \Rightarrow String \rightarrow a \\
&gread = (parse\ .\ compile\ .\ leftfactoring\ .\ leftcorner\ .\ group)\ grammar
\end{aligned}
$$

Since all these steps, except the first one, are performed at runtime, we have achieved true runtime compositionality. Modules can be compiled separately, and the final parsing function is generated just in time. In the next subsections we look at each step in more detail.

2.2.1 Deriving *Gram*

The data type *DGrammar* describes grammars, and we postpone its detailed discussion to Section 2.3. In Figure 2.2 we give the instance of the class *Gram*, containing a value of type *DGrammar T1*, which is generated for the data type *T1* from Figure 2.1. Without going into the implementation details, it is easy to see the direct relationship between the data type *T1* and its *DGrammar T1* representation. For example the part of the grammar:

$$| \ T1\ (7)\ \texttt{":>:"}\ T1\ (6) \qquad\qquad (n \leqslant 6)$$

which corresponds to the second alternative ($n \leqslant 6$) in the data type definition, is represented by the pair:

```
instance Gram T1 where
  grammar = DGrammar _0 envT1
envT1 :: Env DGram (T1, ()) (T1, ())
envT1 = consD (nonts _0) Empty
  where
    nonts _T1 = DLNontDefs
      [ (DRef (_T1, 5)
        , DPS [ dNont (_T1, 5) .#. dTerm ":<:" .#.
                dNont (_T1, 6) .#. dEnd infixL]
        )
      , (DRef (_T1, 6)
        , DPS [ dNont (_T1, 7) .#. dTerm ":>:" .#.
                dNont (_T1, 6) .#. dEnd infixR]
        )
      , (DRef (_T1, 10)
        , DPS [ dTerm "C1" .#. dEnd (const C1)
              , dTerm "(" .#. dNont (_T1, 0) .#.
                dTerm ")" .#. dEnd parenT]
        )
      ]
    infixL e1 _ e2 = e2 :<: e1
    infixR e1 _ e2 = e2 :>: e1
```

Figure 2.2: Representation of the grammar of type *T1*

```
(DRef (_T1, 6)
, DPS [ dNont (_T1, 7) .#. dTerm ":>:" .#.
        dNont (_T1, 6) .#. dEnd infixR]
)
```

In the first component of this pair we specify the non-terminal and its precedence level (which corresponds to a guard behind a set of production rules), while in the second component we find the set of corresponding productions (in this case a singleton list). Each right-hand side consists of a sequence of terminals (*dTerm*) and non-terminals (*dNont*), separated by an operator .#. indicating sequential composition. The sequence finishes with a call to *dEnd f*, where *f* (in this case *infixR*) is a function which takes the parsing results of the right-hand side elements into a value of type *T1*.

$$
\begin{array}{llll}
T2\ (n) \rightarrow & T2\ (6)\ \texttt{":+:"}\ T2\ (6) & & (n \leqslant 5) \\
\mid & A\ (7)\ \ \texttt{":*:"}\ T2\ (7) & & (n \leqslant 6) \\
\mid & \texttt{"C2"} & & \\
\mid & \texttt{"("}\ T2\ (0)\ \texttt{")"} & &
\end{array}
$$

Figure 2.3: Grammar of the type *T2 a*

2.2.2 Grouping

The first transformation we apply to the grammar is to split it according to precedences actually used. The result of grouping the grammar for the type *T1* (Figure 2.1) is:

$$
\begin{array}{l}
A \rightarrow A\ \texttt{":<:"}\ B \mid B \\
B \rightarrow C\ \texttt{":>:"}\ B \mid C \\
C \rightarrow \texttt{"C1"} \mid \texttt{"("}\ A\ \texttt{")"}
\end{array}
$$

where A groups all non-terminals from level 0 to 5, B corresponds to the non-terminal of level 6 and C all non-terminals from level 7 up-to 10. The original reference to *T1* (0) between parentheses is mapped to a reference to A. For non-terminals representing levels less than 10 (A and B) a new alternative that points to the next level is added.

When a grammar contains references to non-terminals of other grammars, we include all the referred grammars. Hence, if we have the grammar of *T2 a* (Figure 2.3), the result of grouping *T2 T1* is:

$$
\begin{array}{l}
A \rightarrow B\ \texttt{":+:"}\ B \mid B \\
B \rightarrow F\ \texttt{":*:"}\ C \mid C \\
C \rightarrow \texttt{"C2"} \mid \texttt{"("}\ A\ \texttt{")"} \\
D \rightarrow D\ \texttt{":<:"}\ E \mid E \\
E \rightarrow F\ \texttt{":>:"}\ E \mid F \\
F \rightarrow \texttt{"C1"} \mid \texttt{"("}\ D\ \texttt{")"}
\end{array}
$$

Note that the non-terminal names of the split grammar of *T1* have changed from A, B and C to D, E and F, respectively.

Of course a compiler could do this statically for those types for which all necessary information is already available; but in the general case this is something which has to be done dynamically.

2.2.3 LC-Transformation

Consider the grammar of the data type *T1* after applying *group*. The production:

$$A \rightarrow A\ \texttt{":<:"}\ B \mid B$$

is left-recursive. So, this grammar cannot be parsed by a top-down parser. We remedy this by applying a Left-Corner transformation [30], for which a typed implementation is given in [7]. Since the complete implementation is given in that paper, we only give a short description of this transformation.

We use the following notational convention for grammar meta-variables. Lower-case letters (a, b, etc.) denote terminal symbols. Low-order upper-case letters (A, B, etc.) denote non-terminals, while high-order upper-case letters (X, Y, Z) denote symbols that can either be terminals or non-terminals. Greek lower-case symbols (α, β, etc.) denote sequences of terminals and non-terminals.

A *direct left-corner* of a non-terminal A is a symbol X so that there exists a production for A with X as the left-most symbol on the right-hand side. The *left-corner* relation is defined as the transitive closure of the direct left-corner relation. So, a non-terminal being left-recursive is equivalent to being a left-corner of itself.

For each (left-recursive) non-terminal A of the original grammar, the function *leftcorner* applies the following rules to build new productions for A and productions for new non-terminals A_X, where X is a left-corner of A and a non-terminal A_X stands for that part of an A after having seen an X.

1. For each production $A \rightarrow X\ \alpha$ of the source grammar add $A_X \rightarrow \alpha$ to the target grammar, and add X to the set of left-corners found for A.
2. For each newly found left-corner X of A:
   a) If X is a terminal symbol b add $A \rightarrow b\ A_b$ to the transformed grammar.
   b) If X is a non-terminal B then for each original production $B \rightarrow Y\ \beta$ add the production $A_Y \rightarrow \beta\ A_B$ to the transformed grammar and add Y to the left-corners of A.

This transformation can not deal with alternatives of the form $A \rightarrow X\ \alpha$ if X is an empty element; i.e. alternatives with empty elements at the beginning. This is not a restriction for us, since the grammars we are working with (data types grammars) do not include such kind of alternatives. In Chapter 3 we will show how to overcome this problem when dealing with more general grammars.

The left-corner transformation for the type *T1* yields the grammar:

$$\begin{array}{ll}
A & \rightarrow \texttt{"C1"}\ A_C1 \mid \texttt{"("}\ A_(\\
A_A & \rightarrow \texttt{":<:"}\ B\ A_A \mid \texttt{":<:"}\ B \\
A_B & \rightarrow A_A \mid \epsilon \\
A_C & \rightarrow \texttt{":>:"}\ B\ A_B \mid A_B \\
A_C1 & \rightarrow A_C \\
A_(& \rightarrow A\ \texttt{")"}\ A_C \\
B & \rightarrow \texttt{"C1"}\ B_C1 \mid \texttt{"("}\ B_(\\
B_C & \rightarrow \texttt{":>:"}\ B \mid \epsilon \\
B_C1 & \rightarrow B_C
\end{array}$$

$$
\begin{array}{ll}
B_(& \rightarrow A\ \texttt{")"}\ B_C \\
C & \rightarrow \texttt{"C1"}\ C_C1 \mid \texttt{"("}\ C_(\\
C_C1 & \rightarrow \epsilon \\
C_(& \rightarrow A\ \texttt{")"}
\end{array}
$$

2.2.4 Left-Factoring

Looking at the grammar of *T1* after the LC-transform, we see that a common prefix has shown up in the productions for the non-terminal A_A. This overlap leads to inefficient parsers, since we have to parse the same part of the input more than once. The function *leftfactoring* removes such common prefixes by applying the following rule until all left-factors have been removed.

- For each set of productions $C = \{A \rightarrow X\ \alpha_1,\ ...,\ A \rightarrow X\ \alpha_n\}$, with $n > 1$, add the productions $(A \rightarrow X\ A__X,\ A__X \rightarrow \alpha_1,\ ...,\ A__X \rightarrow \alpha_n)$ to the grammar, and remove the productions in C.

So, by applying *leftfactoring* to the grammar after the LC-transform we obtain its optimised version:

$$
\begin{array}{ll}
A & \rightarrow \texttt{"C1"}\ A_C1 \mid \texttt{"("}\ A_(\\
A_A & \rightarrow \texttt{":<:"}\ A_A__lt \\
A_A__lt & \rightarrow B\ A_A__lt__B \\
A_A__lt__B & \rightarrow A_A \mid \epsilon \\
A_B & \rightarrow A_A \mid \epsilon \\
A_C & \rightarrow \texttt{":>:"}\ B\ A_B \mid A_B \\
A_C1 & \rightarrow A_C \\
A_(& \rightarrow A\ \texttt{")"}\ A_C \\
B & \rightarrow \texttt{"C1"}\ B_C1 \mid \texttt{"("}\ B_(\\
B_C & \rightarrow \texttt{":>:"}\ B \mid \epsilon \\
B_C1 & \rightarrow B_C \\
B_(& \rightarrow A\ \texttt{")"}\ B_C \\
C & \rightarrow \texttt{"C1"}\ C_C1 \mid \texttt{"("}\ C_(\\
C_C1 & \rightarrow \epsilon \\
C_(& \rightarrow A\ \texttt{")"}
\end{array}
$$

2.3 Representing Data Type Grammars

We represent the grammars as *typed abstract syntax*, encoded using Generalised Algebraic Data Types [54]. In the following subsections we introduce this representation and the issues involved in deriving it from a data type. The main problem to be solved is how to represent the typed references, and how to maintain a type correct representation during the transformation processes.

2.3.1 Typed References and Environments

Pasalic and Linger [49] introduced an encoding *Ref* of typed references to an environment containing values of different type. A *Ref* is labeled with the type of the referenced value and the type of an environment (a nested Cartesian product extending to the right) the value lives in:

```
data Ref a env where
  Zero ::                Ref a (env', a)
  Suc  :: Ref a env' → Ref a (env', b)
```

The constructor *Zero* expresses that the first element of the environment has to be of type a. The constructor *Suc* does not care about the type of the first element in the environment (it is polymorphic in the type b), and remembers a position in the rest of the environment.

Baars et al. [5, 6] extend this idea such that environments do not contain values of mixed type but instead terms (expressions) describing such values; these terms take an extra type parameter describing the environment to which references to other terms occurring in the term may point. In this way we can describe typed terms containing typed references to other terms. As a consequence, a GADT *Env* is introduced, which may be used to represent an environment, consisting of a collection of possibly mutually recursive definitions (in our case grammars). The environment stores a heterogeneous list of terms of type *t a use*, which are the right-hand expressions of the definitions. References to elements are represented by indices in the list, implemented by values of type *Ref*.

```
data Env t use def where
  Empty :: Env t use ()
  Ext   :: Env t use def' → t a use → Env t use (def', a)
```

The type parameter *def* is a nested product containing the type labels a of the terms of type *t a use* occurring in the environment. When the constructor *Ext* is used to extend an environment *Ent t use def′* with a term *t a use* the type label of the resulting environment is extended with this a, resulting in (def', a). The type *use* describes the types that may be referenced from within terms of type *t a use* using *Ref a use* values. When the types *def* and *use* coincide the type system ensures that the references in the terms do not point to values outside the environment; i.e. the environment is closed. A type *FinalEnv* forces an environment to be closed:

```
type FinalEnv t usedef = Env t usedef usedef
```

The function *lookupEnv* takes a reference and an environment. The reference is used as an index in the environment to locate the referenced value. The types guarantee that the lookup succeeds, and that the value found is indeed labeled with the type with which the *Ref* argument was labeled:

```
lookupEnv :: Ref a env → Env t s env → t a s
lookupEnv Zero    (Ext _ p) = p
lookupEnv (Suc r) (Ext ps _) = lookupEnv r ps
```

2.3.2 Typed Grammar Representations

A *Grammar* consists of a root symbol, represented by a value of type *Ref a*..., where *a* is the type of the witness of a successful parse, and a closed environment *FinalEnv*, containing for each non-terminal of the grammar its list of alternative productions. Because the internal structure of the grammar is not of interest it is made existential. This enables us to add or remove non-terminals without changing the visible type of the grammar as such.

```
data Grammar a = ∀ env.Grammar (Ref a env)
                                (FinalEnv Productions env)
newtype Productions a env = PS {unPS :: [Prod a env]}
```

A production is a sequence of symbols, and a symbol is either a terminal with *Token* as its witness or a non-terminal, encoded by a reference.

```
data Token = Keyw String | Open | Close
data Symbol a env where
  Nont :: Ref a env → Symbol a     env
  Term :: Token     → Symbol Token env
data Prod a env where
  Seq :: Symbol b env → Prod (b → a) env
                      → Prod a       env
  End :: a            → Prod a       env
```

The right hand side sequence of symbols terminated by an *End f* element. The function *f* accepts the parsing results of the right hand side elements as arguments, and builds the parsing result for the left-hand side non-terminal.

2.3.3 Typed Grammar Representations for Data Types

For a grammar corresponding to a Haskell data type the situation is a bit different, since we actually have a whole collection of non-terminals: for each non-terminal the

set is indexed by the precedences. Furthermore in productions of a non-terminal we can have references to non-terminals of both the grammar (i.e. data type) being defined as well as other grammars, corresponding to parameters of the data type. For example, the grammar of the type *T2 a* (Figure 2.3) has a reference to the 7th precedence level of the grammar of the type parameter *a*.

We coin the non-terminal we are finally interested in the *main non-terminal*, and our new grammar representation type *DGrammar* starts with a reference to the main non-terminal in the environment. Note that this is the only non-terminal that can be referred to from outside the grammar!

```
data DGrammar a = ∀ env.DGrammar (Ref a env)
                                  (FinalEnv DGram env)
data DGram a env = DGD (DLNontDefs a env)
                 | DGG (DGrammar a)
```

Other non-terminals definitions may be included in the environment as further *DGD*'s, and all the non-terminals labeled by *DGD* can be mutually recursive. In order to be able to refer to other grammars (such as introduced by a type parameter) we introduce an extra kind of non-terminal (*DGG*), which is the starting symbol of a completely new grammar. This imposes a tree like hierarchy on our non-terminals, with the *DGrammar* nodes representing mutually recursive sets of non-terminals.

A reference to a non-terminal has to indicate the place in the environment where the non-terminal is defined (which can either be an internal non-terminal or another grammar) and the level of precedence at the referring position:

```
newtype DRef a env = DRef (Ref a env, Int)
```

A non-terminal is defined by a list of productions available at each precedence level. An occurrence (*DRef* (r, n), *prods*) tells us that the alternatives *prods* of the non-terminal r are available for the levels from 0 to n. For efficiency reasons we order the list in increasing order of precedence.

```
newtype DLNontDefs a env
  = DLNontDefs [(DRef a env, DProductions a env)]
```

The list of alternative productions *DProductions* is defined similar to *Productions*:

```
newtype DProductions a env = DPS {unDPS :: [DProd a env]}
data DProd a env where
  DSeq :: DSymbol b env → DProd (b → a) env
                        → DProd a       env
  DEnd :: a             → DProd a       env
data DSymbol a env where
  DNont :: DRef a env → DSymbol a     env
  DTerm :: Token      → DSymbol Token env
```

In order to make our grammar definitions look a bit nicer we introduce:

```
infixr 5 .#.
(.#.)                    = DSeq
consG   g es             = Ext     es (DGG g)
consD   g es             = Ext     es (DGD g)
dNont   nt               = DNont  (DRef nt)
dTerm   t | t ≡ "("      = DTerm Open
          | t ≡ ")"      = DTerm Close
          | otherwise    = DTerm (Keyw t)
dEnd    f                = DEnd   f
parenT  p1  e p2         = e

_0 = Zero
_1 = Suc _0
_2 = Suc _1
```

Figure 2.4 shows the *DGrammar* (*T2 a*) representation of the grammar *T2 a* (Figure 2.3). It consists of an environment with the production of *T2 a* represented at position *_0* and the grammar of the type *a* at position *_1*. So *DRef* (*_0*, n) refers to *T2 a* at level n and *DRef* (*_1*, n) refers the grammar of the type a at level n. Due to the type signature of the environment, the type system guarantees that the *grammar* we store as the second component in the environment is of type *DGrammar a*.

2.3.4 Representing Mutually Recursive Data Types

When performing the grammar transformations, we expect the grammars to be complete, i.e. all referred grammars are inlined in the grammar from which we want to derive a *gread*. In case of mutually recursive data types, like *T3* and *T4* of Figure 2.5, if we derive the instances:

```
instance Gram T3 where
  grammar = DGrammar _0 envT3
instance Gram T4 where
  grammar = DGrammar _1 envT4
```

we get an unbounded number of copies of each grammar when trying to inline them. This happens because the generation of the grammars is mutually recursive too.

Mutual recursion occurs if there is a cycle of data types mentioned explicitly. When trying to define the representation of a type it can be detected, by constructing a directed graph with the explicit calls to other types. If the type belongs to a strongly connected component there is a cyclic type dependency with the other components.

We have solved the problem of cyclic dependencies using the idea of binding groups [52]. When a strongly connected component is found, the definitions of all the components

```
instance Gram a ⇒ Gram (T2 a) where
  grammar = DGrammar _0 envT2
envT2 :: (Gram a) ⇒ FinalEnv DGram (((), a), T2 a)
envT2 = consD (nonts _0 _1) $
          consG grammar Empty
  where
    nonts _T2 _A = DLNontDefs
      [ (DRef (_T2, 5)
        , DPS [ dNont (_T2, 6) .#. dTerm ":+:" .#.
                dNont (_T2, 6) .#. dEnd infixP ]
        )
      , (DRef (_T2, 6)
        , DPS [ dNont (_A,  7) .#. dTerm ":*:" .#.
                dNont (_T2, 7) .#. dEnd infixT ]
        )
      , (DRef (_T2, 10)
        , DPS [ dTerm "C2"  .#. dEnd (const C2)
              , dTerm "("   .#. dNont (_T2, 0) .#.
                dTerm ")"   .#. dEnd parenT ]
        )
      ]
    infixP e1 _ e2 = e2 :+: e1
    infixT e1 _ e2 = e2 :*: e1
```

Figure 2.4: Representation of the grammar of type *T2 a*

types are tupled together into a single environment. Remember that our environments (*Env*) have no problem in describing mutually recursive definitions. So, in the case of *T3* and *T4*, we build the environment:

```
envT3T4 :: FinalEnv DGram (((), T4), T3)
envT3T4 = consD (nonts3 _0 _1) $
            consD (nonts4 _1 _0) Empty
  where
    nonts3 _T3 _T4 = DLNontDefs
      [ (DRef (_T3, 10)
        , DPS [ dTerm "T3" .#. dNont (_T4, 0) .#.
                dEnd consT3
              , dTerm "C3" .#. dEnd (const C3)
              , dTerm "(" .#. dNont (_T3, 0) .#.
                dTerm ")" .#. dEnd parenT
              ]
```

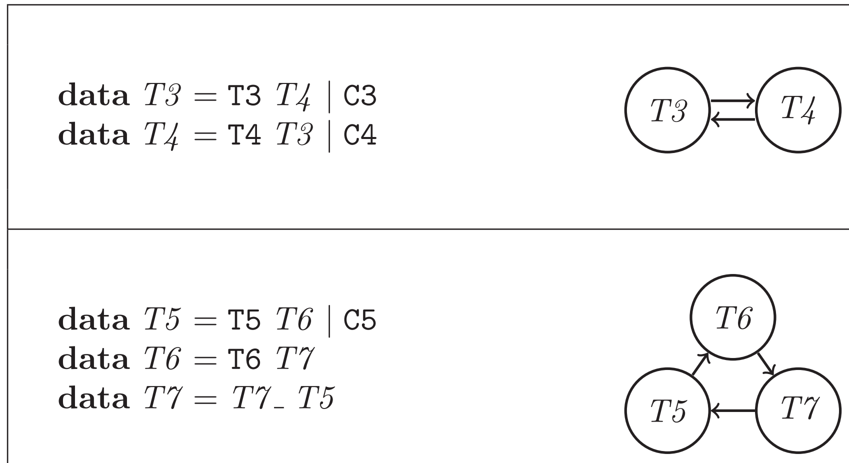

Figure 2.5: Mutually recursive types with graph representation

```
      )
    ]
  nonts4 _T4 _T3 = DLNontDefs
    [(DRef (_T4, 10)
     , DPS [ dTerm "T4" .#. dNont (_T3, 0) .#.
              dEnd consT4
            , dTerm "C4" .#. dEnd (const C4)
            , dTerm "(" .#. dNont (_T4, 0) .#.
              dTerm ")" .#. dEnd parenT
            ]
     )
    ]
  consT3 a = const (T3 a)
  consT4 a = const (T4 a)
```

Note that when defining *T3* we pass the location of *T4* in the environment, and vice versa. For both types the instances can now be created using the same environment, only using different references for the root symbols.

```
instance Gram T3 where
  grammar = DGrammar _0 envT3T4
instance Gram T4 where
  grammar = DGrammar _1 envT3T4
```

As we can see in Figure 2.6, there are some cases where a type is a member of a strongly connected component, but it does not contain explicit references to the other members of its component. This happens when we have a parametrised type that is instantiated with a member of the component. This relation is expressed in the figure

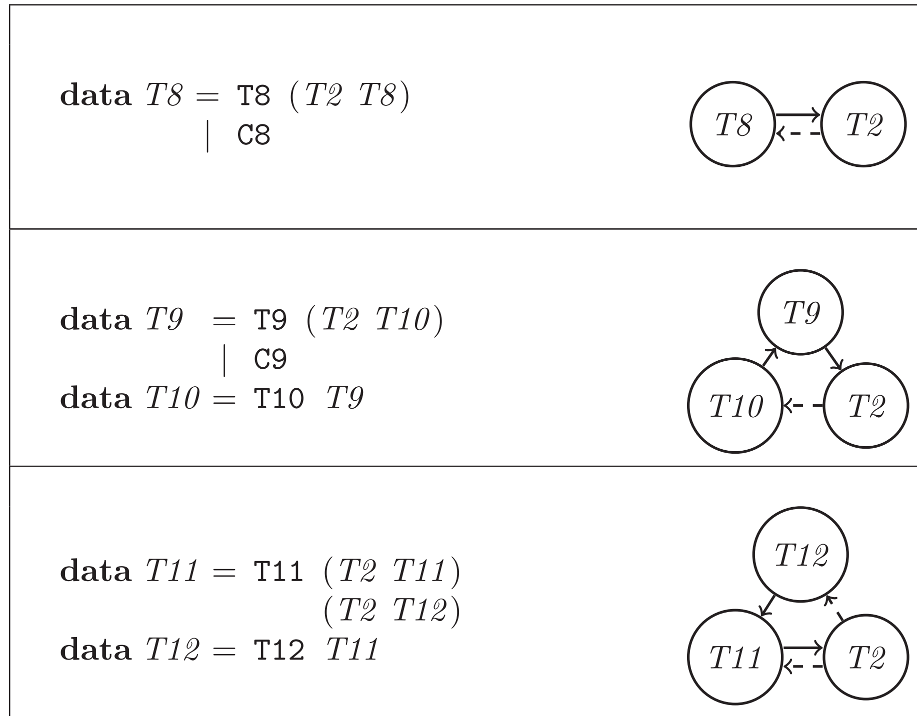

Figure 2.6: Mutually recursive components with *weak* edges

as a dashed edge in the graph. We call such edges *weak* edges, and the types pointing from a such an edge a *weak member*.

These types, in the examples *T2*, generate the cyclic type dependencies but they do not form part of it: the grammar for *T2* is generated without referring to *T8*, *T9*, *T10* or *T11*. But, for example, to generate the grammar of *T9* (or *T10*) the definition of (*T2 T10*) has to be made part of the environment. So in order to define the environment for the instances of *T9* and *T10*:

```
instance Gram T9 where
  grammar = DGrammar _0 envT9T10
instance Gram T10 where
  grammar = DGrammar _1 envT9T10
```

We include a copy of the definition of the non-terminals of *T2 a* instantiated with *T10*:

```
envT9T10 :: FinalEnv DGram ((((), T2 T10), T10), T9)
envT9T10 = consD (nonts9  _0 _2) $
           consD (nonts10 _1 _0) $
           consD (nonts2  _2 _1) Empty
  where
    nonts9 _T9 _T2 = DLNontDefs
```

```
    [(DRef (_T9, 10)
     , DPS [dTerm "T9" .#. dNont (_T2, 0) .#.
             dEnd consT9
           , dTerm "C9" .#. dEnd (const C9)
           , dTerm "(" .#. dNont (_T9, 0) .#.
             dTerm ")" .#. dEnd parenT
           ]
     )
    ]
  nonts10 _T10 _T9 = DLNontDefs
    [(DRef (_T10, 10)
     , DPS [dTerm "T10" .#. dNont (_T9, 0) .#.
             dEnd consT10
           , dTerm "(" .#. dNont (_T10, 0) .#.
             dTerm ")" .#. dEnd parenT
           ]
     )
    ]
  nonts2 _T2 _T10 = DLNontDefs
    [(DRef (_T2, 5)
     , DPS [dNont (_T2, 6)  .#. dTerm ":+:" .#.
             dNont (_T2, 6)  .#. dEnd infixP]
     )
    , (DRef (_T2, 6)
     , DPS [dNont (_T10, 7) .#. dTerm ":*:" .#.
             dNont (_T2, 7)  .#. dEnd infixT]
     )
    , (DRef (_T2, 10)
     , DPS [dTerm "C2"  .#. dEnd (const C2)
           , dTerm "("   .#. dNont (_T2, 0) .#.
             dTerm ")"   .#. dEnd parenT]
     )
    ]
  consT9  a = const (T9  a)
  consT10 a = const (T10 a)
  infixP e1 _ e2 = e2 :+: e1
  infixT e1 _ e2 = e2 :*: e1
```

Note that the instance of *Gram T2* does not occur in this environment; the instance of *Gram T2* is the one defined in Section 2.3.3.

We have to include all the instances of *weak* edges into a binding group. In the case of *T11* there are two *weak* edges from *T2*. Hence both (*T2 T11*) and (*T2 T12*) are included.

```
envT11T12 :: FinalEnv DGram (((((), T2 T12), T2 T11), T12), T11)
```

```
envT11T12 = consD (nonts11 _0 _2 _3) $
            consD (nonts12 _1 _0) $
            consD (nonts2  _2 _0) $
            consD (nonts2  _3 _1) Empty
```

2.3.5 Non Representable Data Types

There are some cases in which we cannot define a representation of the grammar. In the presence of non uniform data types, we cannot avoid the generation of infinite grammars. Consider the data type:

```
data T13 a = T13 (T13 (a, a)) | C13 a
```

To generate the grammar of *T13 a*, we need the grammar of *T13* (a, a), that needs the grammar of *T13* $((a, a), (a, a))$, and so on. Note that all grammars are of different type, so we cannot use the approach defined before.

Another type that cannot be represented with our approach, because is also a kind of non uniform type, is the fix-point type:

```
data Fix f = In (f (Fix f))
```

In these cases we have to resort to the current way the read function works.

2.3.6 Deriving Data Type Grammars

To automatically derive the data type grammars, we use Template Haskell. While you can do most of the introspection needed also with *Data.Generics* [37, 38], we specifically need the fixity information of infix constructors for our grammar, which is not available from *Data.Generics*.

We first need to find out if the type is part of a mutually recursive group. Then we generate code for all types in the group, but only construct an instance for the type *deriveGrammar* was called on.

Calculating binding groups

The algorithm that finds the set of types that is mutually recursive is pretty straightforward: recursively getting the information of the types used in the constructors, while building a graph of types.

To make sure we do not loop, we stop when we find a type that is already in the graph. This works fine, but for types of a kind other than $*$, we need to take the type arguments into account . We bind the arguments in the environment and we do not recurse if we have done so with the same arguments before.

Generating *Gram* instances

Using the binding group, we generate the *DLNontDefs* for each of the types. This is straightforward: for a normal constructor we add a non-terminal at precedence level 10, using the constructor as term and it is arguments as references to non-terminals. For infix constructors we use the precedence and associativity information to add the at the right precedence. For each types we add a special non-terminal for parentheses.

When we need a reference to another grammar we use a naming scheme using the type, bindings (if applicable) and a prefix. For references to grammars that are not known at compile time we use the argument name, prefixed by the type and a general prefix.

When we have all the generated *DLNontDefs* we can chain them together using *consD*. For types that take arguments, we add a *consG grammar* for each argument. In the resulting environment, there will still be variables for references to grammars that are not defined yet. We solve this by wrapping the definitions in two lambda expressions. The inner expression makes the mapping from the 'polymorphic' grammars to names (using explicit polymorphic signatures in the patterns). The outer lambda is used to create the mappings for the parametrised grammars.

As an example, when calling $(*deriveGrammar* “*T8*) the generated code looks like Figure 2.7.

2.4 Typed Transformations

In this section we present the approach used in implementing the transformations:

$$
\begin{array}{ll}
\mathit{group} & :: \mathit{DGrammar}\ a \to \mathit{Grammar}\ a \\
\mathit{leftcorner} & :: \mathit{Grammar}\ \ a \to \mathit{Grammar}\ a \\
\mathit{leftfactoring} & :: \mathit{Grammar}\ \ a \to \mathit{Grammar}\ a
\end{array}
$$

All these functions are implemented by using the typed transformation library `TTTAS` [6]. In the following subsections we introduce the library and describe the implementation of the function *group*. The function *leftcorner* has been presented in the mentioned paper, and *leftfactoring* has a quite similar structure.

2.4.1 Transformation Library

The library is based on the *Arrow* type *Trafo*, which represents typed transformation steps, (possibly) extending an environment *Env*:

data *Trafo m t s a b*

The arguments are the types of: the meta-data m (i.e., state other than the environment we are constructing), the terms t stored in the environment, the final environment s, the arrow-input a and arrow-output b. Thus, instances of the classes

```
instance Gram T8
  where grammar = DGrammar Zero
    ((λ_t_T8 _t_T2'T8 →
       (λ(_nonts_T8 ::   ∀ env.Ref T8 env
                        → Ref (T2 T8) env
                        → DLNontDefs T8 env)
         (_nonts_T2 ::   ∀ env a_0.Ref (T2 a_0) env
                        → Ref a_0 env
                        → DLNontDefs (T2 a_0) env)
          →
         consD (_nonts_T8 _t_T8 _t_T2'T8)
               (consD (_nonts_T2 _t_T2'T8 _t_T8) Empty))
       (λ_r_T8 _r_T2'T8 → DLNontDefs
         [(DRef (_r_T8, 10), DPS [
             ((.#.) $ dTerm "T8")
             ((.#.) (dNont (_r_T2'T8, 0))
             (dEnd (λarg1 _ → T8 arg1)))
           , ((.#.) $ dTerm "C8") (dEnd (\_ → C8))
           , dTerm "("
             .#. (dNont (_r_T8, 0)
             .#. (dTerm ")" .#. dEnd parenT))
             ])])
       (λ_r_T2 _r_T2_a → DLNontDefs [...]))
     Zero (Suc Zero)
     :: FinalEnv DGram (((), T2 T8), T8)
```

Figure 2.7: Generated grammar of type *T8*

Category and *Arrow* are implemented for (*Trafo m t s*), which provides a set of functions for constructing and combining *Trafo*s. Some of these functions which we will refer to are:

- $(\ggg) :: \mathit{Category}\ \mathit{cat} \Rightarrow \mathit{cat}\ a\ b \rightarrow \mathit{cat}\ b\ c \rightarrow \mathit{cat}\ a\ c$
  Left to right composition.
- $\mathit{arr} :: \mathit{Arrow}\ a \Rightarrow (b \rightarrow c) \rightarrow a\ b\ c$
  Lift a function to an arrow.
- $\mathit{first} :: \mathit{Arrow}\ a \Rightarrow a\ b\ c \rightarrow a\ (b, d)\ (c, d)$
  Use the first component for the argument arrow and copy the second component unchanged.
- $\mathit{second} :: \mathit{Arrow}\ a \Rightarrow a\ b\ c \rightarrow a\ (d, b)\ (d, c)$
  Use the second component for the argument arrow and copy the first component unchanged.
- $\mathit{returnA} :: \mathit{Arrow}\ a \Rightarrow a\ b\ b$
  Identity arrow.

There also exists a convenient notation [50] for *Arrow*s, which is inspired by the **do**-notation for *Monad*s. For example, writing:

$$
\begin{aligned}
&\mathbf{proc}\ x \rightarrow \mathbf{do}\ y \leftarrow f \prec x \\
&\qquad\qquad\quad\ z \leftarrow g \prec x \\
&\qquad\qquad\quad\ \mathit{returnA} \prec y + z
\end{aligned}
$$

is equivalent to:

$$\mathit{arr}\ (\lambda x \rightarrow (x, x)) \ggg \mathit{first}\ f \ggg \mathit{second}\ g \ggg \mathit{arr}\ (\lambda(y, z) \rightarrow y + z) \ggg \mathit{returnA}$$

The class *ArrowLoop* is instantiated to provide feedback loops with:

$$\mathit{loop} :: \mathit{ArrowLoop}\ a \Rightarrow a\ (b, d)\ (c, d) \rightarrow a\ b\ c$$

The second component of the output (with type d) is fed back as the second component of the input. In Arrow notation this is represented with the **rec** value recursion construct.

A transformation is run with *runTrafo*, starting with an empty environment and an initial value of type a. The universal quantification over the type s ensures that transformation steps cannot make any assumptions about the type of the (yet unknown) final environment.

$$\mathit{runTrafo} :: (\forall\ s.\mathit{Trafo}\ m\ t\ s\ a\ (b\ s)) \rightarrow m\ () \rightarrow a \rightarrow \mathit{Result}\ m\ t\ b$$

The result of running a transformation is encoded by the type *Result*, containing the final meta-data, the output type and the final environment. It is existential in the

final environment, because in general we do not know how many definitions will be introduced by a transformation and which are their associated types. Note that the final environment has to be closed (hence the use of *FinalEnv*).

data *Result m t b* = ∀ *s*.*Result* (*m s*) (*b s*) (*FinalEnv t s*)

New terms can be added to the environment by using the function *newSRef*. It takes the term of type *t a s* to be added as input and yields as output a reference of type *Ref a s* that points to this term in the final environment:

newSRef :: *Trafo Unit t s* (*t a s*) (*Ref a s*)
data *Unit s* = *Unit*

The type *Unit* is used to express the fact that this transformation does not record any meta-information.

Functions of type (*FinalEnv t s* → *FinalEnv t s*) which update the final environment of a transformation can be lifted into the *Trafo* and composed using *updateFinalEnv*. All functions lifted using *updateFinalEnv* will be applied to the final environment once it is created.

updateFinalEnv :: *Trafo m t s* (*FinalEnv t s* → *FinalEnv t s*) ()

If we have, for example:

proc () → **do** *updateFinalEnv* ≺ *upd1*
...
updateFinalEnv ≺ *upd2*

the function (*upd2*.*upd1*) will be applied to the final environment, produced by the transformation.

The combinator *sequenceA* composes a list of *Trafo*s with input *a* and output *b*, as a *Trafo* with input *a* and output a list of outputs generated sequentially by each *Trafo* of the composed list.

sequenceA :: [*Trafo m t s a b*] → *Trafo m t s a* [*b*]

2.4.2 Implementation of Grouping

The function *group* splits the grammar into parts, depending on the precedence, while changing the representation of the grammar to the one used in the implementation of the left-corner transform:

group :: *DGrammar a* → *Grammar a*

References Mapping

The transformation has to map references in a *DGrammar* with explicitly indicated precedences to a representation where all elements represent normal non-terminals. So, we have to transform the *DRef*s references into the old representation to *Ref*s into the new environment. We introduce a *DRef*-transformer for this conversion, where *env1* describes the types of the old non-terminals and *env2* those of the new non-terminals:

```
newtype DT env1 env2 = DT {unDT :: ∀ a.DRef a env1 → Ref a env2}
```

With this transformer we map each production into its new representation using references into new environment. This is done by applying *unDT* to each non-terminal reference in the production:

```
mapDP2Prod :: DT env1 env2 → DProd a env1 → Prod a env2
mapDP2Prod t (DEnd x)              = End x
mapDP2Prod t (DSeq (DNont x) r) = Seq  (Nont (unDT t x))
                                       (mapDP2Prod t r)
mapDP2Prod t (DSeq (DTerm x) r) = Seq  (Term x)
                                       (mapDP2Prod t r)
```

The function *dp2prod* lifts *mapDP2Prod* using the combinator *arr*. Thus, it takes a *DProd* and returns a transformation that has as output a *Prod*, which is a production in the new environment.

```
type GTrafo = Trafo Unit Productions
dp2prod :: DProd a env → GTrafo s (DT env s) (Prod a s)
dp2prod p = arr (λenv2s → mapDP2Prod env2s p)
```

The type of the resulting *Trafo* indicates that the transformation creates an environment of *Productions* (a *Grammar*).

Each precedence level definition is converted to a non-terminal in the new grammar, using the function *ld2nt*. This function takes a pair (*DRef a env*, *DProductions a env*), that defines a level of precedence, and creates the new non-terminal, returning a reference to it. The transformation made by *dp2prod* is applied to all the elements of the list of alternative productions (*DProductions*) using *sequenceA*, in order to obtain a list of alternative productions in the new grammar (*Productions*). In parallel, the function *mkNxtLev* creates a new production to add to the list, that directly refers to the next level of precedence, if the represented level is less than 10.

```
ld2nt :: (DRef a env, DProductions a env) → GTrafo s (DT env s) (DRef a s)
ld2nt (DRef (rnt, i), DPS lp)
      =    proc env2s →
             do ps        ← sequenceA (map dp2prod lp) ≺ env2s
```

```
                (PS nl) ← mkNxtLev ≺ env2s
                r       ← newSRef  ≺ PS $ nl ++ ps
                returnA ≺ DRef (r, i)
    where
      mkNxtLev = arr $ λt → PS $
        if (i < 10)
          then [Seq (Nont $ unDT t $ DRef (rnt, i + 1)) (End id)]
          else  []
```

Then the possible new production (or an empty list otherwise) is appended to the mapped alternative productions, generating the list that is combined with the creation of a new reference. This new reference is the new non-terminal, which stores its productions. The reference and the precedence level that represents are the output of the transformation.

By applying this transformation to a list of definitions of precedence levels we obtain a list of *DRef*s:

```
newtype ListDR a s = ListDR {unListDR :: [DRef a s]}
```

We now apply this transformation to all the defined levels of precedence in all the non-terminal definitions and recursively to all the referenced grammars. In this way we construct a mapping from the references in the original environment to references in the transformed one.

```
newtype DMapping o n = DMapping {unDMapping :: Env ListDR n o}
```

A *DRef*-transformer can be obtained from the *DMapping* by constructing a function that takes a *DRef a env*, looks up the reference in the environment and subsequently locates the appropriate precedence level in the list:

```
dmap2trans :: DMapping env s → DT env s
dmap2trans (DMapping env)
   = DT (λ(DRef (r, i)) → case (lookupEnv r env) of
                               ListDR rs → (plookup i rs))
```

Having an ordered list of *DRef*s, the function *plookup* returns the first reference (*Ref*) that applies to a given preference level.

```
plookup :: Int → [DRef a s] → Ref a s
plookup i ((DRef (r, p)) : drs) | i ⩽ p      = r
                                | otherwise = plookup i drs
```

Transformation

The function *group* runs a *Trafo* that generates the new environment and returns as output the reference of the starting point (precedence level 0 in the main non-terminal). We construct the new grammar by taking the output and the constructed environment from the *Result*.

```
group :: DGrammar a → Grammar a
group gram
   = let trafo = proc x → do (ListDR rs) ← (gGrammar gram) ≺ x
                             returnA ≺ plookup 0 rs
     in case runTrafo trafo Unit ⊥ of
          Result _ r grm → Grammar r grm
```

The function *gGrammar* implements the grammar transformation. It takes a *DGrammar* and returns a transformation that constructs the "grouped" environment and has as output the list of new references of the main non-terminal.

```
gGrammar :: DGrammar a → GTrafo s t (ListDR a s)
gGrammar (DGrammar r gram) = proc _ → do
                       rec let env_s = dmap2trans menv_s
                             menv_s ← gDGrams gram ≺ env_s
                       returnA ≺ lookupEnv r (unDMapping menv_s)
```

The function applies the transformation returned by *gDGrams* to the elements of the environment. This transformation takes as input a *DRef*-transformer, mapping all non-terminals from the original environment to the newly generated one. The output is a *DMapping* which remembers the new locations of the non-terminals from the original grammar. To obtain the needed *DRef*-transformer for this transformation, the function *gGrammar* uses a feed-back loop using the *DMapping* returned by the transformation itself. To obtain the list of mapped references for the main non-terminal it just looks up the reference in the *DMapping*.

The function *gDGrams* iterates (by induction) over the environment that contains the non-terminal definitions and the grammars referenced by them.

```
gDGrams :: Env DGram env env' → GTrafo s (DT env s) (DMapping env' s)
gDGrams = mapTrafoEnv tr
  where
    tr (DGG gram)                  = gGrammar gram
    tr (DGD (DLNontDefs nonts)) = proc env_s → do
                             r ← sequenceA (map ld2nt nonts)) ≺ env_s
                             returnA ≺ ListDR r
```

In the case of a grammar, the function *gGrammar* is invoked. The output of this transformation is the list of new references assigned to the main non-terminal of this grammar. The list is added to the *DMapping* in the place of the grammar.

In the case of a list of precedences (a non-terminal), we *map* the function *ld2nt* to the list, obtaining a list of transformations. Each transformation adds a new non-terminal to the new grammar and returns the new reference and the precedence level that represents. We execute this transformations sequentially (using *sequenceA*) and add the resulting list of references to the *DMapping*.

The iteration over the environment is performed by the function *mapTrafoEnv*.

$$
\begin{aligned}
&mapTrafoEnv :: (\forall a.t\ a\ env \to GTrafo\ s\ i\ (ListDR\ a)) \\
&\qquad\qquad\qquad \to Env\ t\ env\ env' \to GTrafo\ s\ i\ (DMapping\ env') \\
&mapTrafoEnv\ _\ Empty \qquad = \mathbf{proc}\ _ \to \mathbf{do} \\
&\qquad\qquad\qquad\qquad\qquad returnA \prec DMapping\ Empty \\
&mapTrafoEnv\ t\ (Ext\ x\ xs) = \mathbf{proc}\ i \to \mathbf{do} \\
&\qquad\qquad\qquad\qquad\qquad tx \ \leftarrow t\ x \qquad\qquad\quad \prec i \\
&\qquad\qquad\qquad\qquad\qquad txs \leftarrow mapTrafoEnv\ t\ xs \prec i \\
&\qquad\qquad\qquad\qquad\qquad returnA \prec DMapping\ (Ext\ tx\ \$\ unDMapping\ txs)
\end{aligned}
$$

2.5 Efficiency

In this section we show some experimental results about the efficiency of our approach[3]. First of all we compare *read* and *gread* in the presence of infix constructors. Finally we show how the presence of the left-factoring optimisation influences efficiency.

2.5.1 *gread* versus *read*

In Figure 2.8 we show the execution times resulting from the use of *read* and *gread* to parse an expression of the form `C1 :>: (C1 :>: ...)`, where n is the number of constructors `C1` the expression has.

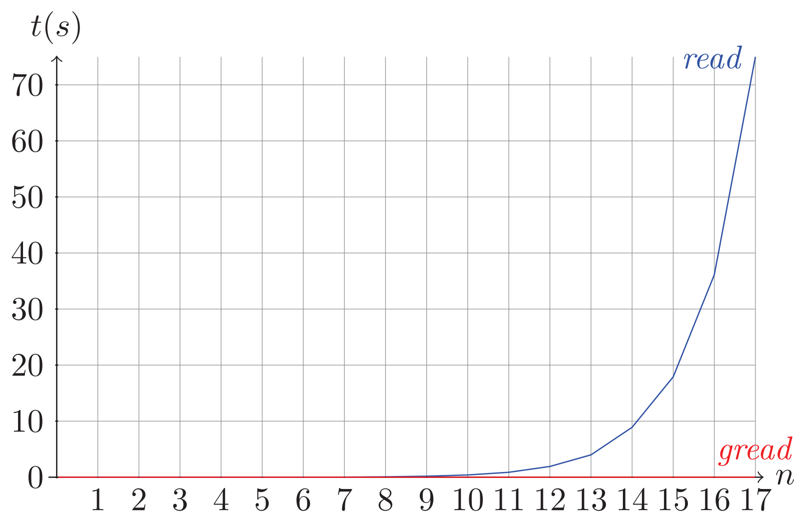

Figure 2.8: Execution times of reading `C1` :> (`C1 :>: ...`)

The function *read* clearly has an exponential behaviour. It takes 75 seconds to resolve the case with 17 `C1`s and does not run after 18. On the other hand, the function *gread*

[3]The tests were run in a computer with 1.6 GHz Intel Core Duo processor and 1 GB RAM.

maintains negligible times. If we do not use parentheses we can read 50000 `C1`s within a second.
We obtain similar behaviour with (... `:<: C1`) `:<: C1`. Note that this is a bad case for the function *read*, due to the opening parentheses. The function *read* takes 23 seconds to resolve the case with 9 `C1`s (does not run after 10), while the function *gread* requires negligible times: more than 40000 `C1`s can be read within a second, without the extra parentheses.

Data type grammars are usually very small, but in order to test our approach in its worst case, we defined a large data type of the form:

data *TBig t1 t2 t3 t4 t5 t6 t7 t8 t9 t10*
  = `CB`
  | `TB1` (*TBig t1 t2 t3 t4 t5 t6 t7 t8 t9 t10*)
  | ...
  | `TB`n (*TBig t1 t2 t3 t4 t5 t6 t7 t8 t9 t10*)

where n is a number between 10 and 100. Note that the type has 10 parameters and no infix constructors. So a relatively large combination and transformation effort is needed, while the optimisations do not add anything. We tested this type with an expression `TB`n (...(`TB`n `CB`)...) with 10000 constructors.

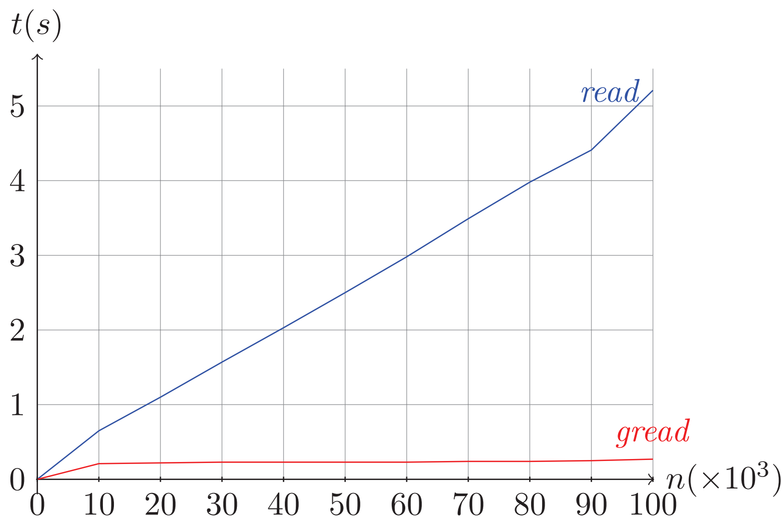

Figure 2.9: Execution times of reading a large data type

We can see in Figure 2.9 that the function *gread* has linear behaviour after a constant (n-dependent) startup time. From this case we can conclude that the time needed to perform the transformations is almost negligible. We have performed the same tests using the expressions `TB1` (...(`TB1 CB`)...) and `TB`$\frac{n}{2}$ (...(`TB`$\frac{n}{2}$ `CB`)...) obtaining similar results.

2.5.2 *gread* versus *leftcorner*

We have shown that the *gread* function has efficient behaviour in comparison with the Haskell *read*. But what happens if we do not include the left-factoring?

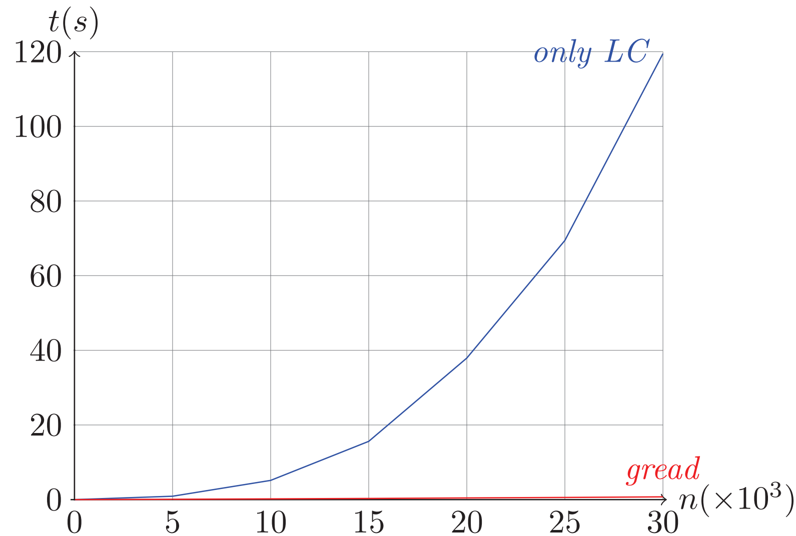

Figure 2.10: Execution times of reading `C1:<:...:<:C1` with and without left-factoring

As we can see in Figure 2.10 (expression `C1 :<: ... :<: C1`) the inclusion of left-factoring improves the efficiency by avoiding duplicate parsing.

We have tested both functions in situations where the left-factoring is not needed and they behave in a similar way; so the extra transformation work and the few extra non-terminals add little to the total cost of parsing. For example, in the case of Table 2.1 there are no common prefixes in the evaluated productions while only applying the LC-transform.

2.6 Conclusions and Future Work

We have shown an alternative way to implement the *read* (and consequently also the *show*) functions. We read data in linear time, generate shorter output, and the overhead caused by generating the read functions at runtime does not seem to be a problem; not even for very large data types. Unfortunately we are not able to handle nested data types which have infix constructors; for these one has to write the parsing functions by hand. Note that this problem only occurs if the nested data type occurs at the left-hand side of an infix type constructor, and that in such cases also the conventional solution is problematic.

Besides the completely dynamic implementation which we have presented in which we compose all grammars at runtime, a large part of the work could be done by the Haskell compiler at compilation time.

n ($\times 10^3$)	*gread* (s)	*only LC* (s)
10	0.14	0.14
20	0.33	0.32
30	0.57	0.56
40	0.82	0.82
50	1.13	1.11
60	1.47	1.47
70	1.82	1.81
80	2.23	2.22
90	2.68	2.67
100	3.18	3.15

Table 2.1: Execution times of reading `C1 :>: ... :>: C1` with and without left-factoring

We consider the Template Haskell implementation to be a prototype. Further optimisations are to tuple grammars with their corresponding parser. If we know there are no problems with common prefixes or left-recursion we can resort to simpler parsing methods, and generate parsers only once by sharing them.

Straightforward extensions are the inclusion of a generator for record constructors. An open research problem is how to merge in the techniques for parsing record fields in arbitrary order, since the proposed solution [4] critically depends on the dynamic generation of parsers; we expect lazy evaluation to save us here. Finally, we need a more robust naming scheme to deal with problems due a similarly named types coming from different modules.

3 Grammar Fragments Fly First-Class

We present a Haskell library for expressing (fragments of) grammars using typed abstract syntax with references. We can analyze and transform such representations and generate parsers from them. What makes our approach special is that we can combine embedded grammar fragments on the fly, i.e. after they have been compiled. Thus grammar fragments have become fully typed, first-class Haskell values.

We show how we can extend an initial, limited grammar embedded in a compiler with new syntactic constructs, either by introducing new non-terminals or by adding new productions for existing non-terminals. We do not impose any restrictions on the individual grammar fragments, nor on the structure as a whole.

3.1 Introduction

There are many different ways to represent grammars and grammatical structures *in a typeful way*: be it in implicit form using *conventional parser combinators* or more explicitly in the form of *typed abstract syntax*. Each approach has its own advantages and disadvantages. The former, being a domain specific embedded language, makes direct use of the typing, abstraction and naming mechanisms of the host language. This implicit representation however does have its disadvantages: we can only perform a limited form of grammar analysis and transformation since references to non-terminals in the embedded language are implemented by variable references in the host language. The latter approach, which does give us full access to the complete domain specific program, comes with a more elaborate naming system which gives us the possibility to identify references to non-terminals; however transforming such programs in Haskell necessitates to provide proofs (in our case encoded through the Haskell type system) that the types remain correct during transformation.

One of the applications of the latter approach is where one wants to compose grammar fragments. This is needed when a user can extend the syntax of a base language. In doing so he has to extend the underlying context-free grammar and he has to define the semantics for these new constructs. Once all extensions have become available the parser for the complete language is constructed by joining the newly defined or redefined semantics with the already existing parts. In a more limited way such things can be done by e.g. the quasi quoting mechanism as available through Template Haskell [41, 58], which however has its limitations: the code using the new syntax can be clearly distinguished from the host language. Furthermore the TH code, which is run in a separate phase, is not guaranteed to generate type correct code; only after the code is expanded type checking takes place, often leading to hard to understand error messages.

In the previous chapter we have shown how to compose grammar fragments for a limited class of grammars, i.e. those describing the output format of Haskell data types. These latter grammars have a convenient property: productions will never derive the empty string, which is a pre-condition for the Left-Corner Transform (LCT) [7] which is to be applied later to remove left-recursion from the grammar which arises from the use of infix data constructors.

In this chapter introduce the library `murder`[1] for expressing first-class context-free grammars. We describe an *unrestricted*, *applicative* interface for constructing such grammar descriptions, we describe how they can be combined, and how they can be transformed so they fulfill the precondition of the LCT. The final result can safely be mapped onto a top-down parser, constructed using a conventional parser combinator library.

In section 3.2 we describe the "user-interface" to our library. In section 3.3 we introduce the types used to represent our fragments, whereas in section 3.4 we describe the internal data structures. In section 3.6 we extend our grammar representation with a fixpoint-like combinator. In section 3.7 we discuss some related work and conclude.

3.2 Context-Free Grammar

In this section we show how to express a context free grammar fragment. Our running example is the simple expression language we introduced in Chapter 1

Figure 3.1 shows the concrete grammar of the *initial language* and the almost isomorphic Haskell code encoding of this language fragment in terms of our combinator library and the *Arrow*-interface [29, 50]. A grammar description is an *Arrow*, representing the introduction of its composing non-terminals. Since non-terminals can be mutually recursive, they are declared using a **rec** block. The function *addNT* introduces a new non-terminal together with some initial productions (alternatives) separated by `<|>` operators. Each alternative (right hand side of a production) consists of a sequence of elements, expressed in so-called *applicative style*, using the idiomatic brackets[2] $[\![$ and $]\!]$ which delineate the description of a production from the rest of the Haskell code. The brackets $[\![$ and $]\!]$ are syntactic sugar for the Haskell function *iI* and constant *Ii*. A production consists of a call to a *semantic function*, which maps the results of the trailing non-terminals to the result of this production, and a sequence of non-terminals and terminals, the latter corresponding to literals which are to be recognized. Since terminal symbols like `"main"` and `"*"` do not bear any meaning our idioms automatically discard these results: the expression $[\![$ *semMul term* `"*"` *factor* $]\!]$ is equivalent to *pure* $(\lambda l\ _\ r \rightarrow$ *semMul l r*$)$ `<*>` *sym term* `<*>` *tr* `"*"` `<*>` *sym factor* in the *Applicative* interface [44]. The semantic functions are defined elsewhere (using monad transformers, attribute grammars or embedded AG code, as we will see in Chapter 4). By convention we will let their names start with *sem*. For elementary

[1] Available at: `http://hackage.haskell.org/package/murder`
[2] `http://www.haskell.org/haskellwiki/Idiom_brackets`

```
Grammar:

  root   ::= decls "main" "=" exp
  decls  ::= var "=" exp decls | empty
  exp    ::= exp  "+" term  | term
  term   ::= term "*" factor | factor
  factor ::= int | var

Haskell code:

  prds = proc () → do
     rec root   ← addNT  ≺ ⟦ semRoot decls "main" "=" exp ⟧
         decls  ← addNT  ≺ ⟦ semDecls var "=" exp decls ⟧
                         <|> ⟦ semNoDecl ⟧
         exp    ← addNT  ≺ ⟦ semAdd exp  "+" term  ⟧ <|> ⟦ term ⟧
         term   ← addNT  ≺ ⟦ semMul term "*" factor ⟧ <|> ⟦ factor ⟧
         factor ← addNT  ≺ ⟦ semCst int ⟧ <|> ⟦ semVar var ⟧
     exportNTs ≺ exportList root $ export ntDecls  decls
                                 . export ntExp    exp
                                 . export ntTerm   term
                                 . export ntFactor factor
  gram = closeGram prds
```

Figure 3.1: Initial language

parsers which return values which are constructed by the scanner we provide a couple of predefined special cases, such as *int* which returns the integer value from the input and *var* which returns a recognized variable name.

An initial grammar is also an *extensible grammar*. It exports (with *exportNTs*) its starting point (*root*) and a list of *exportable non-terminals* which actually stand for a collection of productions to be used and modified in future extensions. Each *export*-ed non-terminal is labeled by a unique value of a unique type (by convention starting with *nt*). The function *closeGram* takes the list of productions, and converts it into a compiler; in our case a parser integrated with the semantics for the language derived from the starting symbol *root*.

3.2.1 Language Extension

We now extend the language with an extra non-terminal for conditions (Boolean expressions) and an extra production for conditional expressions:

$exp\ ::= ...$ | "if" $cond$ "then" exp "else" exp
$cond ::= exp$ "==" exp | exp ">" exp

This language extension $prds'$ is defined as a closed Haskell value by itself, which accesses an already existing set of productions (*imported*) and builds an extended set, as shown in Figure 3.2.

```
prds' = proc imported → do
  let exp    = getNT ntExp    imported
  let factor = getNT ntFactor imported
  rec addProds ≺ (factor, ⟦ (semIf  sf) "if" cond "then" exp "else" exp ⟧)
      cond ← addNT ≺ ⟦ (semEq sf) exp "==" exp ⟧
                 <|> ⟦ (semGr sf) exp ">"  exp ⟧
  exportNTs ≺ extendExport imported (export ntCond cond)
gram' = closeGram (prds +>> prds')
```

Figure 3.2: Language Extension

For each non-terminal to be extended we retrieve its current list of productions (using *getNT*) from the *imported* non-terminals, and add new productions to this list using *addProds*. The **if**-expression is e.g. added by:

```
let exp    = getNT ntExp    imported
let factor = getNT ntFactor imported
rec addProds ≺ (factor, ⟦ (semIf sf) "if" cond "then" exp "else" exp ⟧)
```

New non-terminals can be added as well using *addNT*; in the example we add the non-terminal *cond*:

```
cond ← addNT ≺ ⟦ semEq exp "==" exp ⟧ <|> ⟦ semGr exp ">" exp ⟧
```

Finally, we extend the list of exportable non-terminals with (some of) the newly added non-terminals, so they can be extended by further fragments elsewhere:

```
exportNTs ≺ extendExport imported (export ntCond cond)
```

The original grammar *prds* is extended with $prds'$ using the combinator (+>>). Because both *prds* and $prds'$ are proper Haskell values which can be separately defined in different modules and compiled separately we claim that the term *first class grammar fragments* is justified here. It is important to note that all these productions are well-typed Haskell values, of which the type is parameterized with the type of values the expressions represent; so, based on the type of *semIf* the Haskell compiler will be

able to check that the non-terminal *cond* indeed parses Boolean expressions! By being able to compile a language and its extensions separately, a framework for extensible compilers can be defined which allows a language to be extended without providing full access to its source and without having to re-compile the whole compiler.

3.3 Grammar Representation

Having described how a user describes and combines individual language fragments, we now embark on the description of the internals of our library.

Just as in Chapter 2, a *Grammar* consists of a closed environment, containing a list of productions for each non-terminal, and a reference (*Ref a env*) to the root symbol, where *a* is the type of the witness of a successful parse. Note that the type *env* is hidden using existential quantification, so changes to the structure of the grammar can be made, by adding or removing non-terminals, without having to change the visible part of its type.

```
data EG
data CG
data Grammar s a = ∀ env.Grammar (Ref a env)
                                 (FinalEnv (Productions s) env)
newtype Productions s a env = PS {unPS :: [Prod s a env]}
```

The type *s* encodes the state of the grammar, that is: *EG* if the grammar can contain empty productions and *CG* (closed grammar) if the grammar does not.

For productions we choose a representation which differs slightly from the one used in the previous chapter. Here we represent productions really in an *applicative-style*; i.e. using constructors *Pure* and *Seq* analogous to the *pure* function and `<*>` operator of applicative functors:

```
data Prod s a env where
  Pure    :: a                     → Prod s    a env
  Seq     :: Prod s (a → b)    env
          → Prod s a           env → Prod s    b env
  Sym     :: Symbol a t env        → Prod s    a env

  FlipSeq :: Prod CG a         env
          → Prod CG (a → b) env → Prod CG b env
```

FlipSeq is a variant of *Seq* with its arguments in reverse order. By imposing *s* to be *CG*, we restrict *FlipSeq* to be included only in grammars not containing empty productions. *Sym* is a special case of *pure* that lifts a symbol to a production. A symbol is either a terminal or a non-terminal encoded by a reference pointing to one of the elements in an environment labeled with *env*. A normal terminal contains the literal string it represents. We define a category of *attributed terminals*, which are not

fixed by a literal string. Every attributed terminal refers to some lexical structure. In contrast to the normal terminals which do not bear a semantic value, for attributed terminals the parsed values are used and the type *a* instantiates to the type of the parsed value.

```
data TTerm; data TNonT; data TAttT
data Symbol a t env where
  Term      :: String →      Symbol (DTerm String) TTerm  env
  TermInt   ::               Symbol (DTerm Int)    TAttT  env
  TermVarid ::               Symbol (DTerm String) TAttT  env
  Nont      :: Ref a env →   Symbol a              TNonT  env
```

The type parameter *t* indicates, at the type-level, whether a *Symbol* is a terminal (type *TTerm*) for which the result is (usually) discarded, a non-terminal (*TNonT*) or an attributed terminal (*TAttT*) in the value of which we are interested. In order to make our code more readable we introduce the smart constructors *trm*, *int* and *var*, for the terminals *Term*, *TermInt* and *TermVarid*, respectively. The data type *DTerm* couples the value of a terminal with its position in the source (code) from where it is obtained:

```
data DTerm a = DTerm {pos :: Pos, value :: a}
```

3.3.1 From Grammar to Parser

A grammar can be compiled into a top-down parser with an *Applicative* (and *Alternative*) interface. We show how a grammar can be translated to the `uu-parsinglib` parser combinator library [63], which can then be used to *parse* a String into a *ParseResult* containing a semantic value of type *a*:

```
generate :: Set String → Grammar CG a → Parser a
parse    :: Parser a → String → ParseResult a
```

The function *generate* translates a *Productions* list as a sequence of parsers combined by `<|>`. The *Prod* constructors *Seq*, *FlipSeq* and *Pure* are translated to `<*>`, `<**>` and *pure*, respectively. Terminals are translated to terminal parsers and non-terminal references are retrieved from an environment containing the translated productions for each non-terminal. Notice that only *CG* grammars can be compiled.

```
newtype Const f a s = C {unC :: f a}
generate :: Set String → Grammar CG a → Parser a
generate reserved (Grammar (start :: Ref a env) rules)
  = id <$ pSpaces <*> (unC (lookupEnv start result))
  where
    result = mapEnv (λ(PS ps) → C (foldr1 (<|>) [comp p | p ← ps])) rules
```

```
comp :: Prod CG t env → Parser t
comp (Seq x y)              = comp x <*> comp y
comp (FlipStar x y)         = comp x <**> comp y
comp (Pure x)               = pure x
comp (Sym (Term t))         = DTerm <$> pPos <*> pTerm t
comp (Sym (Nont n))         = unC (lookupEnv n result)
comp (Sym TermInt)          = DTerm <$> pPos <*> pInt
comp (Sym TermVarid)        = DTerm <$> pPos <*> (pVar reserved)
```

```
mapEnv :: (∀ a.f a s → g a s) → Env f s env → Env g s env
mapEnv _ Empty       = Empty
mapEnv f (Ext r v) = Ext (mapEnv f r) (f v)
```

Since the `uu-parsinglib` performs a breadth-first search it will parse a large class of grammars without any further *try* or *cut*-like annotations; the only requirement it imposes is that the grammar is not left-recursive (which holds since we will apply the LCT before compiling the grammar into a parser) and that the grammar is unambiguous. This latter property is unfortunately undecidable; fortunately it is trivial to generate a parser version which can handle ambiguous grammars too, since the `uu-parsinglib` library contains provisions for this.

3.3.2 Applicative Interface

We want the type *Productions* to be an instance of the Haskell classes *Applicative* and *Alternative* themselves as we have seen in the examples. However, this is impossible due to the order of its type parameters; we need a to be the last parameter[3]. Thus, we define the type *PreProductions* for descriptions of (possibly empty) alternative productions.

```
newtype PreProductions env a = PP {unPP :: [Prod EG a env]}
```

The translation from *PreProductions* to *Productions* is trivial:

```
prod :: PreProductions env a → Productions EG a env
prod (PP ps) = PS ps
```

Now we can define the (*PreProductions env*) instances of *Applicative* and *Alternative*:

```
instance Applicative (PreProductions env) where
  pure f = PP [Pure f]
  (PP f) <*> (PP g) = PP [Seq f' g' | f' ← f, g' ← g]
instance Alternative (PreProductions env) where
```

[3]We cannot just redefine *Productions* with this order, because we need the current order for the transformations we will introduce later.

$empty = PP\ [\,]$
$(PP\ f)$ <|> $(PP\ g) = PP\ (f +\!\!+\ g)$

We are dealing with lists of alternative productions, thus the alternative operator (<|>) takes two lists of alternatives and just appends them. In the case of sequential application (<*>) a list of productions is generated with all the possible combinations of the operands joined with a *Seq*.

We also defined smart constructors for symbols: *sym* for the general case and *tr* for the special case where the symbol is a terminal.

```
sym :: Symbol a t env → PreProductions env a
sym s = PP [Sym s]
tr   :: String → PreProductions env String
tr   s = sym (Term s)
```

3.4 Extensible Grammars

In this section we present our approach to define and combine *extensible grammars* (like the one in Figure 3.1) and *grammar extensions* (Figure 3.2). The key idea is to see the definition, and possibly future extensions, of a grammar as a typed transformation that introduces new non-terminals into a typed grammar (*Grammar*). For example, both *prds* and *prds′* of Figures 3.1 and 3.2 are typed transformations: while *prd* starts with an empty context-free grammar and transforms it by adding the non-terminals *root*, *decls*, *exp*, *term* and *factor*, the grammar extension *prd′* continues the transformation started by *prd* and modifies the definition of one of the non-terminals and adds a new one. Notice that a *Grammar* is a collection of mutually recursive typed structures; thus, performing transformations while maintaining the whole collection well-typed is non-trivial. We use TTTAS (introduced in Section 2.4.1) to implement our transformations.

3.4.1 Grammar Extensions

In this subsection we present the API of a library for defining and combining *extensible grammars* (like the one in Figure 3.1) and *grammar extensions* (Figure 3.2).

A grammar extension can be seen as a series of typed transformation steps that can add new non-terminals to a typed grammar and/or modify the definition of already existing non-terminals. We define an extensible grammar type (*ExtGram*) for constructing initial grammars from scratch and a grammar extension type (*GramExt*) as a typed transformation that extends a typed extensible grammar. In both cases a *Trafo* uses the *Productions* as the type of terms defined in the environment being carried.

```
type ExtGramTrafo = Trafo Unit (Productions EG)
```

```
type ExtGram env            start' nts'
   = ExtGramTrafo env ()                    (Export start' nts' env)
type GramExt env start nts start' nts'
   = ExtGramTrafo env (Export start nts env) (Export start' nts' env)
```

Exportable non-terminals

Both extensible grammars and grammar extensions have to export the starting point *start'* and a list of *exportable non-terminals nts'* to be used in future extensions. The only difference between them is that a grammar extension has to import the elements (*start* and *nts*) exported by the grammar it is about to extend, whereas an extensible grammar, given that it is an initial grammar, does not need to import anything.

The exported (and imported, in the case of grammar extensions) elements have type *Export start nts env*, including the starting point (a non-terminal, with type *Symbol start TNonT env*) and the list of exportable non-terminals (*nts env*).

```
data Export start nts env = Export (Symbol start TNonT env) (nts env)
```

The list of exportable non-terminals has to be passed in a *NTRecord*, which is an implementation of extensible records very similar to the one in the *HList* library [35], with the difference that it has a type parameter *env* for the environment where the non-terminals point into. A field ($l \in v$) relates a (first-class) non-terminal label l with a value v. A *NTRecord* can be constructed with the functions (.*.), for record extension, and *ntNil*, for empty records. The function *getNT* is used to retrieve the value part corresponding to a specific non-terminal label from a record. We have defined some functions to construct *Export* values:

```
exportList r ext = Export r (ext ntNil)
export l nt      = (.*.) (l ∈ nt)
```

Thus, the export list in Figure 3.1 is equivalent to:

```
Export root (ntDecls ∈ decls .*. ntExp    ∈ exp    .*.
             ntTerm ∈ term .*. ntFactor ∈ factor .*. ntNil)
```

In order to finally export the starting point and the exportable non-terminals we chain an *Export* value through the transformation in order to return it as output.

```
exportNTs :: NTRecord (nts env)
          ⇒ ExtGramTrafo env (Export start nts env) (Export start nts env)
exportNTs = returnA
```

Thus, the definition of an extensible grammar (like the one in Figure 3.1) has the following shape, where *exported_nts* is a value of type *Export*:

```
prds = proc () → do ...
                     exportNTs ≺ exported_nts
```

The definition of a grammar extension, like the one in Figure 3.2, has the shape:

$$prds' = \mathbf{proc}\ (imported_nts) \rightarrow \mathbf{do}\ ...$$
$$exportNTs \prec exported_nts$$

where *imported_nts* and *exported_nts* are both of type *Export*. We have defined a function to extend (imported) exportable lists:

$$extendExport\ (Export\ r\ nts)\ ext = Export\ r\ (ext\ nts)$$

Adding Non-terminals

To add a new non-terminal to the grammar we need to add a new term to the environment.

$$addNT :: ExtGramTrafo\ env\ (PreProductions\ env\ a)\ (Symbol\ a\ TNonT\ env)$$
$$addNT = \mathbf{proc}\ p \rightarrow \mathbf{do}\ r \leftarrow newSRef \prec prod\ p$$
$$returnA \prec Nont\ r$$

The input to *addNT* is the initial list of alternative productions (*PreProductions*) for the non-terminal and the output is a non-terminal symbol, i. e. a reference to the non-terminal in the grammar. Thus, when in Figure 3.1 we write:

$$term \leftarrow addNT \prec [\![\ semMul\ term\ \texttt{"*"}\ factor\ ]\!]\ \texttt{<|>}\ [\![\ factor\ ]\!]$$

we are adding the non-terminal for terms, with the alternative productions $[\![\ factor\ ]\!]$ and $[\![\ semMul\ term\ \texttt{"*"}\ factor\ ]\!]$, and we bind to *factor* a symbol holding the reference to the added non-terminal thus making it available to be used in the definition of this or other non-terminals. Because *Trafo* instantiates *ArrowLoop*, we can define mutually recursive non-terminals using the keyword **rec**, like in figures 3.1 and 3.2.

Updating Productions

The list of productions of an existing non-terminal can be updated by a function f:

$$updProds :: ExtGramTrafo\ env$$
$$(Symbol\ a\ TNonT\ env$$
$$,PreProductions\ env\ a \rightarrow PreProductions\ env\ a)$$
$$()$$
$$updProds = \mathbf{proc}\ (Nont\ r, f) \rightarrow \mathbf{do}$$
$$updateFinalEnv \prec updateEnv\ (\lambda ps \rightarrow PS\ \$\ (unPP\ .\ f)\ (PP\ \$\ unPS\ ps))\ r$$

Thus, adding new productions to a non-terminal translates into the concatenation of the new productions to the existing list of productions of the non-terminal.

$$ppAppend\ pp1\ pp2 = PP\ \$\ (unPP\ pp1) +\!\!+ (unPP\ pp2)$$

```
addProds = proc (nont, prds) → do
  updProds ≺ (nont, ppAppend prds)
```

In Figure 3.2 we have seen an example of adding productions to the non-terminal *factor*. It is also easy to define a function *remProds*, to remove all the productions of a non-terminal, based on *updProds*. The function which updates the productions in this case is (*const* $ *PP* [ ]). Note that removing the non-terminal completely would be much harder.

Grammar Extension and Composition

Extending a grammar boils down to composing two transformations, the first one constructing an extensible grammar and the second one representing a grammar extension.

```
(+>>) :: (NTRecord (nts env), NTRecord (nts′ env))
        ⇒ ExtGram env start  nts → GramExt env start nts start′ nts′
        → ExtGram env start′ nts′
g +>> sm = g >>> sm
```

We defined (+>>) to restrict the types of the composition. Two grammar extensions can be composed just by using the (>>>) operator from the *Arrow* class.

If we want to compose two extensible grammars *g1* and *g2* with disjoint non-terminals sets, we have to sequence them, obtain their start points *s1* and *s2* and add a new starting point *s* with *s1* and *s2* as productions.

```
(<++>) :: (NTUnion nts1 nts2 nts)
         ⇒ ExtGram env start nts1 → ExtGram env start nts2
         → ExtGram env start nts
g1 <++> g2 = proc () → do (Export s1 ns1) ← g1 ≺ ()
                          (Export s2 ns2) ← g2 ≺ ()
                          s ← addNT ≺ sym s1 <|> sym s2
                          returnA ≺ Export s (ntUnion ns1 ns2)
```

The function *ntUnion* performs the union of the non-terminal labels both at value and type level. It introduces the constraint *NTUnion*, which also ensures the disjointedness of the sets. If we have a function *ntIntersection*, returning for each intersecting non-terminal its position on each grammar (*nt1*, *nt2*), then we can define a general composition of grammars. We have to extend (<++>) with the transformation (*addProds* (*nt1*, ⟦ *id nt2* ⟧) >>> *addProds* (*nt2*, ⟦ *id nt1* ⟧)) for each non-terminal belonging to the intersection.

3.5 Closed Grammars

To close a grammar we run the *Trafo*, in order to obtain the grammar to which we apply the Left-Corner Transform. By applying *leftcorner* we prevent the resulting

grammar to be left-recursive, so it can be parsed by a top-down parser. Such a step is essential since we cannot expect from a large collection of language fragments, that the resulting grammar will be e.g. LALR(1) or non-left-recursive. The type of the start non-terminal *a* is the type of the resulting grammar.

```
closeGram :: (∀ env.ExtGram env a nts) → Grammar CG a
closeGram prds = case runTrafo prds Unit () of
  Result _ (Export (Nont r) _) gram
     → (leftCorner . removeEmpties) (Grammar r gram)
```

The *leftcorner* function is an adaptation to our representation of *Prod* of the transformation proposed in [7] which preprocesses the grammar such that empty parts at the beginning of productions have been removed:

```
removeEmpties :: Grammar EG a → Grammar CG a
leftCorner    :: Grammar CG a → Grammar CG a
```

The function *removeEmpties* takes a grammar that can have empty productions (*Grammar EG a*) and returns an equivalent grammar (*Grammar CG a*) without empty productions and without left-most empty elements. Since this transformation does not introduce new non-terminals to the grammar, we do not need to use `TTTAS` to implement it.

First, the possibly empty production of each non-terminal is located using the function *findEmpties*, that takes the environment with the productions of the grammar and returns an isomorphic environment with values of type *HasEmpty*. If a non-terminal has an empty production, then the position of the resulting environment corresponding to this non-terminal contains a value *HasEmpty f*, where *f* is the semantic value associated with this empty case. If a non-terminal does not have empty productions then the environment contains a *HasNotEmpty* on its corresponding position.

After locating the empty productions we remove them from the grammar using the function *removeEmptiesEnv*, where the empty production of each non-terminal is removed and added to the contexts where the non-terminal is referenced. Thus, if the root symbol has an empty production, allowing the parsing of the empty string, this behavior will not be present after the removal. For simplicity reasons we avoid this situation by disallowing empty productions for the root symbol of the grammars we deal with, and yield an error message in this case. It is easy to remove this constraint by adding a production (*Pure f*) to the start non-terminal of the grammar resulting from the whole (*leftCorner.removeEmpties*) transformation, where (*HasEmpty f*) is the result of looking-up the start point in the environment of empty productions. But in practice we do not expect this to be necessary.

```
data HasEmpty a env = Unknown | HasNotEmpty | HasEmpty a

removeEmpties :: Grammar EG a → Grammar CG a
```

```
removeEmpties (Grammar start prds) =
  let empties = findEmpties prds
  in  case lookupEnv start empties of
        HasNotEmpty → Grammar start (removeEmptiesEnv empties prds)
        _           → error "Empty prod at start point"
```

In the following sub-sections we will explain the empty productions removal algorithm on more detail. For that we will use the following example grammar:

```
A → pure fA <*> tr "a" <*> sym B
B → sym C <*> sym D <*> pure fB
C → pure fC1 <*> sym C <*> tr "c" <|> pure fC2
D → pure fD <|> tr "d"
```

3.5.1 Finding Empty Productions

The function *findEmpties* constructs an environment with values of type *HasEmpty*. It starts with an initial environment of found empties without information, created by *initEmpties*, and iterates updating this environment until a fixpoint is reached. The function *stepFindEmpties* implements one step of this iteration, returning a triple with: the environment with the found empty productions thus far, a Boolean value indicating whether this step introduced changes to the environment, and a Boolean value that tells us whether the returned environment still has any *Unknown* non-terminals.

```
type GramEnv s = Env (Productions s)

findEmpties :: GramEnv EG env env → Env HasEmpty env env
findEmpties prods = findEmpties' prods (initEmpties prods)
findEmpties' prds empties
  = case stepFindEmpties empties prds empties of
          (empties', True, _     ) → findEmpties' prds empties'
          (empties', False, False) → empties'
          (_,        False, True ) → error "Incorrect Grammar"
```

If we arrive at a fixpoint, and still have remaining *Unknown* non-terminals, then the grammar is incorrect, so we get a soundness check for the grammar for free. Such non-terminals will not be able to derive a finite sentence, as in the following example:

```
A → term "a" <*> sym A
```

The initial environment of the algorithm is an environment with an *Unknown* value for each non-terminal of the grammar. In the example, the initial environment is the one of the `Step 0` in Figure 3.3.

```
initEmpties :: GramEnv EG use def → Env HasEmpty use def
initEmpties Empty         = Empty
initEmpties (Ext nts _) = Ext (initEmpties nts) Unknown
```

On each *step* we take the actual environment of found empty productions and we go through all the non-terminals of the grammar, trying to find out if the information about the existence of empty productions for this non-terminal can be updated (*updEmpty*).

```
stepFindEmpties :: Env HasEmpty use use
                → GramEnv EG use def
                → Env HasEmpty use def
                → (Env HasEmpty use def, Bool, Bool)
stepFindEmpties empties (Ext rprd prd) (Ext re e)
                = let (re', rchg, runk) = stepFindEmpties empties rprd re
                      (e',  chg,  unk)  = updEmpty empties prd e
                  in  (Ext re' e', chg ∨ rchg, unk ∨ runk)
stepFindEmpties _ Empty Empty = (Empty, False)
```

Step 0	Step 1
A → *Unknown* *B* → *Unknown* *C* → *Unknown* *D* → *Unknown*	*A* → *HasNotEmpty* *B* → *Unknown* *C* → *HasEmpty fC2* *D* → *HasEmpty fD*
Step 2	**Step 3**
A → *HasNotEmpty* *B* → *HasEmpty* (*fC2 fD fB*) *C* → *HasEmpty fC2* *D* → *HasEmpty fD*	*A* → *HasNotEmpty* *B* → *HasEmpty* (*fC2 fD fB*) *C* → *HasEmpty fC2* *D* → *HasEmpty fD*

Figure 3.3: Results of Finding Empties Steps for the Example

We only have to update the *HasEmpty* value associated with a non-terminal if in the actual environment it is *Unknown*. In the other cases we already know whether this non-terminal has any empty productions.

```
updEmpty :: Env HasEmpty use use → Productions EG a use → HasEmpty a use
         → (HasEmpty a use, Bool, Bool)
updEmpty empties prds Unknown = case hasEmpty empties prds of
                                    Unknown → (Unknown, False, True)
```

```
                                   e          → (e,         True, False)
updEmpty _ _ e = (e, False, False)
```

The new *HasEmpty* information for a non-terminal is computed out of the previous environment and the list of productions of the non-terminal. The *HasEmpty* information is retrieved for each production using *isEmpty*, and those results are combined. If any of the productions is empty then we have found that the non-terminal may derive the empty string. If we find more than a single empty production for a non-terminal then the grammar is ambiguous. If all the productions are not empty, then we return *HasNotEmpty*. In other cases, the information for this non-terminal remains still *Unknown*.

```
hasEmpty :: Env HasEmpty env env → Productions EG a env → HasEmpty a env
hasEmpty empties (PS ps)
  = foldr (λp re → combine (isEmpty p empties) re) HasNotEmpty ps
combine :: HasEmpty a env → HasEmpty a env → HasEmpty a env
combine (HasEmpty _) (HasEmpty _) = error "Ambiguous Grammar"
combine _            (HasEmpty f) = HasEmpty f
combine (HasEmpty f) _            = HasEmpty f
combine HasNotEmpty HasNotEmpty = HasNotEmpty
combine _            _            = Unknown
```

An empty production is obtained from: a production (*Pure a*), a reference to a non-terminal that has an empty production, or a sequence of two empty productions. In this case we construct a *HasEmpty a* value with a the semantic action associated to this empty alternative; for a sequential composition of actions f and x the associated semantic action is $(f\ x)$, given that *pure f* `<*>` *pure x* $\equiv$ *pure* $(f\ x)$. If a production is a terminal symbol, a reference to a non-terminal that has no empty production, or a sequence of two productions where at least one of them is not empty, then this production is not empty and we return *HasNotEmpty*. We obtain the information of the referenced non-terminals from the environment created thus far. Thus, it can happen that in a certain step there is not enough information to take a decision about a production, remaining it *Unknown*. This is the case of a reference to a non-terminal that is still *Unknown* and a sequence of two productions where the first production is empty and the second *Unknown* or the first is *Unknown* and the second not empty.

```
isEmpty :: Prod EG a env → Env HasEmpty env env → HasEmpty a env
isEmpty (Pure a)          _       = HasEmpty a
isEmpty (Sym (Nont r)) empties = lookupEnv r empties
isEmpty (Sym _)           _       = HasNotEmpty
isEmpty (Seq pl pr)       empties
   = case isEmpty pl empties of
         HasNotEmpty → HasNotEmpty
         HasEmpty f  → case isEmpty pr empties of
```

```
                              HasEmpty x    → HasEmpty (f x)
                              e             → e
    Unknown        → case isEmpty pr empties of
                              HasNotEmpty → HasNotEmpty
                              _             → Unknown
```

Let us look at our example grammar; in Figure 3.3 we show the results of the steps taken to find the empty productions.

In `Step 1` we find useful information for non-terminals A, C and D. In the case of A we have only one production:

Pure fA ‘*Seq*‘ *Sym* (*Term* `"a"`) ‘*Seq*‘ *Sym B*

looking at the left part of the sequence:

Pure fA ‘*Seq*‘ *Sym* (*Term* `"a"`)

we have another sequence with an empty left part and a non-empty right part. Since one of its components is not empty, the whole sequence is not empty; and the same applies to its containing sequence.

In the cases of C and D, it can be seen that both have two productions. In both cases one production is empty and the other is not empty; hence we have immediately located an empty production for both non-terminals.

The non-terminal B has a single production:

Sym C ‘*Seq*‘ *Sym D* ‘*Seq*‘ *Pure fB*

if we look at the left part:

Sym C ‘*Seq*‘ *Sym D*

it is a sequence of two non-terminals. Thus we have to look for the information we have from the previous step, in this case the `Step 0` (the initial environment). For both C and D we still do not have any information, thus the information of the left part of the production is *Unknown*. Since the right part of the production is empty (*Pure fB*), we cannot assume anything yet about the existence of empty productions for B.

In `Step 2` we take another look at B. Now we know that C has an empty production with semantic action *fC2* and D has an empty production with semantic action *fD*. Therefore from the left part of the sequence we can construct a *HasEmpty* (*fC2 fD*), having finally found the empty production *HasEmpty* (*fC2 fD fB*).

The `Step 3` does not perform any changes to the environment of found empty productions, since no non-terminal is *Unknown*. Thus, we have found the empty productions of the grammar.

3.5.2 Removal of Empty Productions

Once the empty productions are found, we can proceed to remove them. The function *removeEmptiesEnv* traverses the environment with the productions of the non-terminals, removing the empty productions, transforming productions which start with an empty element into productions starting with non-empty elements, and transforming the contexts where the non-terminals with empty productions are referenced.

```
removeEmptiesEnv :: Env HasEmpty use use
                 → GramEnv EG use def → GramEnv CG use def
removeEmptiesEnv empties = mapEnv (removeEmpty empties)
```

To remove the possibly empty production from a non-terminal, we apply the function *splitEmpty* to every production, concatenating the resulting alternative productions.

```
removeEmpty :: Env HasEmpty env env
            → Productions EG a env → Productions CG a env
removeEmpty empties (PS prds)
   = PS $ foldr ((++) . remEmptyProd) [] prds
     where remEmptyProd = fst . splitEmpty empties
```

The function *splitEmpty* takes a production, and the environment of found empty productions, and returns a pair with a list of alternative productions generated from removing the empty part of the input production, and the possibly empty part of the production. While removing the empty productions in *removeEmpty* we use the generated non-empty productions and ignore the empty part.

```
splitEmpty :: Env HasEmpty env env → Prod EG a env
           → ([Prod CG a env], Maybe (Prod CG a env))
```

In the case of non-terminal symbols, the generated non-empty production is a reference to the symbol itself, since the algorithm will remove its possible empty production. The empty production, if it exists, is looked-up in the environment of found empty productions.

```
splitEmpty empties (Sym (Nont r))
   = case lookupEnv r empties of
          HasEmpty f → ([Sym $ Nont r], Just (Pure f))
          _          → ([Sym $ Nont r], Nothing)
```

Terminal symbols are non-empty productions, thus the generated non-empty production is the symbol itself, not having any empty part. On the other hand, a *Pure a* production is an empty production without non-empty part.

```
splitEmpty _ (Sym s)  = ([Sym s], Nothing)
splitEmpty _ (Pure a) = ([], Just (Pure a))
```

In the example, when removing the empty productions of D, *splitEmpty* is invoked for the alternative productions (*Pure fD*) and (*Sym* (*Term* "d")), that result in the respective pairs ([*Sym* (*Term* "d")], *Nothing*) and ([ ], *Just* (*Pure fD*)). Thus, after the removal D only has the production (*Sym* (*Term* "d")).
The non-empty productions generated by the transformation of a sequence f `<*>` g are:

- *fne_gne*: Sequences of the combination of the non-empty productions generated from f and g.
- *fne_ge*: Sequences of the non-empty productions generated from f and the empty production of g.
- *fe_gne*: Sequences of the empty production of f and the non-empty productions generated from g. Here we introduce the *FlipSeq* "reversed" sequence, translating (*fe* `<*>` *gne*) to (*gne* `<**>` *fe*), in order to move the non-empty part of the sequence to the left. Thus we avoid introducing left-most empty elements.

The possible empty production generated from f`<*>`g is *fe_ge*, a production *Pure* (*fv gv*) where *fv* and *gv* are the semantic actions associated to the empty productions of f and g, respectively. Notice the use of the *Maybe Monad*.

```
splitEmpty empties (Seq f g)
          = let (fne, fe)  = splitEmpty empties f
                (gne, ge)  = splitEmpty empties g
                fne_gne    = [Seq fv gv | fv ← fne, gv ← gne]
                fne_ge     = case ge of
                               Nothing → [ ]
                               Just gv  → [Seq      fv  gv | fv ← fne ]
                fe_gne     = case fe of
                               Nothing → [ ]
                               Just fv  → [FlipSeq gv fv  | gv ← gne]
                fe_ge      = do (Pure fv) ← fe
                               (Pure gv) ← ge
                               return $ Pure (fv gv)
            in  (fne_gne ++ fne_ge ++ fe_gne, fe_ge)
```

The function *splitEmpty* takes a production of type *Prod EG a env* as argument, and thus productions of the form (*FlipSeq g f*) are not possible as input. However, as we have seen before, this kind of productions can be generated out of the transformation (case *fe_gne*) because the returned productions have type *Prod CG a env*. In the example, during the removal of the empty production of B, we call *splitEmpty* for:

```
Sym C `Seq` Sym D `Seq` Pure fB
```

that calls *splitEmpty* for *Sym C* ‘*Seq*‘ *Sym D* and *Pure fB*. Let us see what happens in the evaluation for the first sub-production. The function *splitEmpty* is again called for *Sym C* and *Sym D*, resulting in:

```
fne ⇒ [Sym C]
fe  ⇒ Just (Pure fC2)
gne ⇒ [Sym D]
ge  ⇒ Just (Pure fD)
```

Thus, the empty part of the sub-production is (*Pure* (*fC2 fD*)), and the non-empty generated productions are:

```
[Sym C ‘Seq‘      Sym D
,Sym C ‘Seq‘      Pure fD
,Sym D ‘FlipSeq‘ Pure fC2]
```

Finally, given that the result of *splitEmpty* for (*Pure fB*) is ([ ], *Just* (*Pure fB*)), the empty part of *B* coincides with the one found with *findEmpties* and the transformed *B* is:

```
B → PS [Sym C ‘Seq‘      Sym D     ‘Seq‘ Pure fB
       ,Sym C ‘Seq‘      Pure fD   ‘Seq‘ Pure fB
       ,Sym D ‘FlipSeq‘ Pure fC2 ‘Seq‘ Pure fB]
```

Note that now *B* does not contain any empty production, includes productions for the empty and non-empty part of every referenced non-terminal, and has no left-most empty element.

The result of the transformation over the whole grammar example, using the smart constructors to make it easier to read, is:

```
A  → tr "a" <**> pure fA   <*> sym B
   <|> tr "a" <**> pure fA   <*> pure (fC2 fD fB)
B  → sym C  <*>  sym D    <*> pure fB
   <|> sym C  <*>  pure fD   <*> pure fB
   <|> sym D <**> pure fC2 <*> pure fB
C  → sym C <**> pure fC1 <*> tr "c"
D  → tr "d"
```

Notice how our brute-force approach generates grammars which have productions which start with the same sequence of elements. These will be taken care of by the left-factoring which is done as the last step of the Left Corner Transform. A slight different approach would be to extend our formalism to allow for nested structures, where we have a special kind of non-terminals, i.e. those which are only referenced once, and which we substitute directly in the grammar. This will lead to a grammar with a rule:

```
A → tr "a" <**> pure fA <*> (sym B <|> pure (fC2 fD fB))
```

Unfortunately this will make the formulation of the Left Corner Transform more complicated.

3.6 Fixpoint Production

In order to be able to define recursive productions, we have added a sort of fixpoint combinator to our productions representation. The data type *Prod* is extended with the constructors *Fix*, for the fixpoint combinator, and *Var*, for references to the fixed point.

```
data FL a
data Prod s a env where
   ...
   Fix :: Productions (FL a) a env → Prod EG a env
   Var :: Prod (FL a) a env
```

The type parameter *s* is used to restrict: *Fix* to be used only at "top-level", *Var* to be used only at "fixpoint-level" (*FL*), productions *Var* to have the same type of their containing *Fix*.

Thus, by defining some smart constructors:

```
varPrd   :: PreProductions (FL a) env a
varPrd   = PP [Var]
fixPrd   :: PreProductions (FL a) env a → PreProductions EG env a
fixPrd p = PP [(Fix . prod) p]
```

we can, for example, represent the useful EBNF-like combinators *pSome* and *pMany*.

```
pSome :: PreProductions (FL [a]) env a → PreProductions EG env [a]
pSome p = fixPrd (one <|> more)
   where one  = (:[]) <$> p
         more = (:)   <$> p <*> varPrd

pMany :: PreProductions (FL [a]) env a → PreProductions EG env [a]
pMany p = fixPrd (none <|> more)
   where none = pure []
         more = (:)   <$> p <*> varPrd
```

Another useful combinator is *pFoldr*, which is a generalized version of *pMany*, where the semantic functions have to be passed as an argument.

```
pFoldr :: (a → b → b, b)
       → PreProductions (FL b) env a → PreProductions EG env b
pFoldr (c, e) p = fixPrd (none <|> more)
   where none = pure e
         more = c <$> p <*> varPrd
```

Thus, in Figure 3.1 instead of writing:

```
decls ← addNT  ≺  ⟦ semDecls var "=" exp decls ⟧
                 <|> ⟦ semNoDecl ⟧
```

we could have written:

```
decls ← addNT ≺ pFoldr (semDecls, semNoDecl) ⟦ var "=" exp ⟧
```

3.6.1 Fixpoint Removal

The semantics of *Fix* and *Var* are provided by a new transformation *removeFix*, that we add to the grammar closing pipeline.

```
closeGram :: (∀ env.ExtGram env a nts) → Grammar CG a
closeGram prds = case runTrafo prds Unit () of
  Result _ (Export (Nont r) _) gram
     → (leftCorner . removeEmpties . removeFix) (Grammar r gram)
```

The function *removeFix* takes a grammar which can have *Fix* and *Var* productions, and returns a new grammar without them.

```
removeFix :: Grammar EG a → Grammar EG a
```

What we basically do is to traverse the input environment returning a copy of the productions in every case but *Fix*. When a (*Fix prds*) is found, we use *addNT* to add a new non-terminal to the grammar and return its reference. The productions we add to this non-terminal result of replacing *Var* by the non-terminal reference in *prds*. Thus, for example:

```
rec A ← addNT ≺ fixPrd (pure fA1 <|> pure fA2 <* trm "a" <*> varPrd)
```

is equivalent to do:

```
rec R ← addNT ≺ pure fA1 <|> pure fA2 <* trm "a" <*> sym R
    A ← addNT ≺ sym R
```

3.7 Related Work and Conclusions

This work builds on our previous work on typed transformations of typed grammars [6, 7], although we here stuck more to the conventional applicative style in order to make it more accessible to the everyday programmer who knows Haskell. The major contribution of this chapter is the introduction of a set of combinators to describe, extend and combine grammar fragments using arrow notation. In order to avoid problems with the constructed grammars we have introduced a preprocessing step before applying the LCT.

Of course there exist a myriad of other approaches to represent context-free grammars and grammar fragments, but we are not aware of the existence of a *typeful way of representing grammar fragments using an embedded domain specific language* as we have presented here. Because of the embeddedness it remains possible to define one's own grammar constructs such as sequences, optional elements and chains of elements, thus keeping all the advantages commonly found in combinator parser based approaches.

Devriese and Piessens [17] propose a model for explicitly recursive grammars in Haskell, which provides an applicative interface to describe productions. By using generic programming techniques from [77] their representation supports a wide range of grammar algorithms, including the Left-Corner Transform.

Brink et al. [13] introduced a framework to represent grammars and grammar transformations in the dependently typed programming language Agda. In such a language it is possible to prove correctness properties of the transformations, more than the preservation of semantic types.

On one hand both claim to be less complex than our technique, but on the other hand, they are both based on closed non-terminal domains, and thus they lack grammar extension and composition which form the core of this paper.

Finally note that the way in which we eventually construct parsers out of the constructed grammar in no way precludes other approaches. So it is a trivial extension to generate parsers which can deal with ambiguous grammars by using suitable combinators (like the *amb* combinator from the `uu-parsinglib` library). If one wants to use very general parsing techniques or scannerless parsing techniques [11] there is nothing that prevents one from doing so. No information is lost in the final representation.

4 Attribute Grammars Fly First-Class

Attribute Grammars (AGs), a general-purpose formalism for describing recursive computations over data types, are a useful tool for implementing language semantics. In this chapter we present `AspectAG`, a typed embedding of AGs in Haskell. The key lies in using HList-like typed heterogeneous collections (extensible polymorphic records) and expressing AG well-formedness conditions as type-level predicates (i.e., type-class constraints). By further type-level programming we can also express common programming patterns, corresponding to the typical use cases of monads such as *Reader*, *Writer* and *State*.

4.1 Introduction

Functional programs can easily be extended by defining extra functions. If however a data type is extended with a new alternative, each parameter position and each case expression where a value of this type is matched has to be inspected and modified accordingly. In object oriented programing the situation is reversed: if we implement the alternatives of a data type by sub-classing, it is easy to add a new alternative by defining a new subclass in which we define a method for each part of desired global functionality. If however we want to define a new function for a data type, we have to inspect all the existing subclasses and add a method describing the local contribution to the global computation over this data type. This problem was first noted by Reynolds [56] and later referred to as "the expression problem" by Wadler [74]. We start out by showing how the use of AGs overcomes this problem.

As running example we use the classic *repmin* function [8]; it takes a tree argument, and returns a tree of similar shape, in which the leaf values are replaced by the minimal value of the leaves in the original tree (see Figure 4.1). The program (Figure 4.2) was originally introduced to describe so-called circular programs, i.e. programs in which part of a result of a function is again used as one of its arguments. We will use this example to show that the computation is composed of three so-called *aspects*: the computation of the minimal value as the first component of the result of *sem_Tree* (*asp_smin*), passing down the globally minimal value from the root to the leaves as the parameter *ival* (*asp_ival*), and the construction of the resulting tree as the second component of the result (*asp_sres*).

Now suppose we want to change the function *repmin* into a function *repavg* which replaces the leaves by the average value of the leaves. Unfortunately we have to change almost every line of the program, because instead of computing the minimal value we have to compute both the sum of the leaf values and the total number of leaves. At the root level we can then divide the total sum by the total number of leaves to

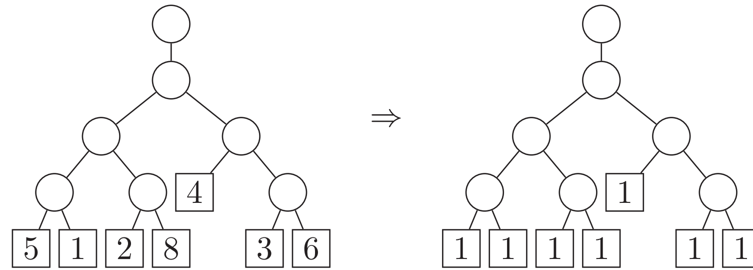

Figure 4.1: *repmin* replaces leaf values by their minimal value

```
data Root = Root Tree
data Tree = Node Tree Tree | Leaf Int

repmin = sem_Root

sem_Root (Root tree) =              let (smin, sres)     = (sem_Tree tree) smin
                                    in  (sres)
sem_Tree (Node l r)  = λival →      let (lmin, lres)     = (sem_Tree l    ) ival
                                        (rmin, rres)     = (sem_Tree r    ) ival
                                    in  (min lmin rmin, Node lres rres    )
sem_Tree (Leaf i)    = λival →          (i, Leaf ival)
```

Figure 4.2: *repmin* as a circular program

```
DATA Root | Root tree
DATA Tree | Node l, r : Tree
          | Leaf i : {Int}
SYN Tree  [smin : Int]
SEM Tree
      | Leaf lhs .smin = i
      | Node lhs .smin = min l.smin r.smin
INH Tree [ival : Int]
SEM Root
      | Root tree.ival  = tree.smin
SEM Tree
      | Node l    .ival  = lhs.ival
             r    .ival  = lhs.ival
SYN Root Tree [sres : Tree]
SEM Root
      | Root lhs .sres  = tree.sres
SEM Tree
      | Leaf lhs .sres  = Leaf (lhs.ival)
      | Node lhs .sres  = Node (l.sres) (r.sres)
```

Figure 4.3: AG specification of repmin

compute the average leaf value. However, the traversal of the tree, the passing of the value to be used in constructing the new leafs and the construction of the new tree all remain unchanged. What we are now looking for is a way to define the function *repmin* as:

$$repmin = sem_Root\ (asp_smin \oplus asp_ival \oplus asp_sres)$$

so we can easily replace the aspect *asp_smin* by *asp_savg*:

$$repavg = sem_Root\ (asp_savg \oplus asp_ival \oplus asp_sres)$$

In Figure 4.3 we have expressed the solution of the *repmin* problem in terms of a domain specific language, i.e., as an `uuagc` attribute grammar [64]. Attributes are values associated with tree nodes. We will refer to a collection of (one or more) related attributes, with their defining rules, as an aspect. After defining the underlying data types by a few **DATA** definitions, we define the different aspects: for the two “result” aspects we introduce synthesized attributes (**SYN** *smin* and **SYN** *sres*), and for the “parameter” aspect we introduce an inherited attribute (**INH** *ival*).

Note that attributes are introduced separately, and that for each attribute/alternative pair we have a **SEM** rule which contains a separate piece of code describing

what to compute. The defining expressions at the right hand side of the =-signs are all written in Haskell, using minimal syntactic extensions to refer to attribute values; we refer to a synthesized attribute of a child using the notation *child.attribute* and to an inherited attribute of the production itself (the left-hand side) as **lhs**.*attribute*. These expressions are copied directly into the generated program: only the attribute references are replaced by references to values defined in the generated program. The attribute grammar system only checks whether for all attributes a definition has been given. Type checking of the defining expressions is left to the Haskell compiler when compiling the generated program (given in Figure 4.2).

As a consequence type errors are reported in terms of the generated program. Although this works reasonably well in practice, the question arises whether we can define a set of combinators which enables us to embed the AG formalism directly in Haskell, thus making the separate generation step uncalled for and immediately profiting from Haskell's type checker and getting error messages referring to the original source code.

A first approach to such an embedded attribute grammar notation was made by de Moor et al. [16]. Unfortunately this approach, which is based on extensible records [25], necessitates the introduction of a large set of combinators, which encode positions of children-trees explicitly. Furthermore combinators are indexed by a number which indicates the number of children a node has where the combinator is to be applied. The *first contribution* of this chapter is that we show how to overcome these shortcomings by making use of the Haskell class system.

The *second contribution* is that we show how to express the previous solution in terms of heterogeneous collections, thus avoiding the use of Hugs-style extensible records are not supported by the main Haskell compilers.

Attribute grammars exhibit typical design patterns; an example of such a pattern is the inherited attribute *ival*, which is distributed to all the children of a node, and so on recursively. Other examples are attributes which thread a value through the tree, or collect information from all the children which have a specific attribute and combine this into a synthesized attribute of the father node. In normal Haskell programming this would be done by introducing a collection of monads (*Reader*, *State* and *Writer* monad respectively), and by using monad transformers to combine these in to a single monadic computation. Unfortunately this approach breaks down once too many attributes have to be dealt with, when the data flows backwards, and especially if we have a non-uniform grammar, i.e., a grammar which has several different non-terminals each with a different collection of attributes. In the latter case a single monad will no longer be sufficient.

One way of making such computational patterns first-class is by going to a universal representation for all the attributes, and packing and unpacking them whenever we need to perform a computation. In this way all attributes have the same type at the attribute grammar level, and non-terminals can now be seen as functions which map dictionaries to dictionaries, where such dictionaries are tables mapping *String*s representing attribute names to universal attribute values [15]. Although this provides us with a powerful mechanism for describing attribute flows by Haskell functions, this

```
data Root = Root {tree :: Tree}
            deriving Show
data Tree = Node {l :: Tree, r :: Tree}
            | Leaf {i :: Int}
            deriving Show


$(deriveAG ''Root)
$(attLabels ["smin", "ival", "sres"])


asp_smin = synthesize smin    at {Tree}
           use        min 0   at {Node}
           define at  Leaf = i
asp_ival = inherit    ival    at {Tree}
           copy   at  {Node}
           define at  Root.tree = tree.smin
asp_sres = synthesize sres             at {Root, Tree}
           use        Node (Leaf 0)    at {Node}
           define at  Root = tree.sres
                      Leaf = Leaf lhs.ival


asp_repmin = asp_smin ⊕ asp_sres ⊕ asp_ival
repmin t = select sres from compute asp_repmin t
```

Figure 4.4: *repmin* in our embedded DSL

comes at a huge price; all attributes have to be unpacked before the contents can be accesses, and to be repacked before they can be passed on. Worse still, the check that verifies that all attributes are completely defined, is no longer a static check, but rather something which is implicitly done at run-time by the evaluator, as a side-effect of looking up attributes in the dictionaries. The *third contribution* of this chapter is that we show how patterns corresponding to the mentioned monadic constructs can be described, again using the Haskell class mechanism.

The *fourth contribution* of this chapter is that it presents yet another large example of how to do type-level programming in Haskell, and what can be achieved with it. In our conclusions we will come back to this.

Before going into the technical details we want to give an impression of what our embedded Domain Specific Language (DSL) looks like. In Figure 4.4 we give our definition of the *repmin* problem in a lightly sugared notation.

To completely implement the *repmin* function the user of our library[1] needs to undertake the following steps (Figure 4.4):

- define the Haskell data types involved;
- optionally, generate some boiler-plate code using calls to Template Haskell;
- define the aspects, by specifying whether the attribute is inherited or synthesized, with which non-terminals it is associated, how to compute its value if no explicit definition is given (i.e., which computational pattern it follows), and providing definitions for the attribute at the various data type constructors (productions in grammar terms) for which it needs to be defined, resulting in *asp_repmin*;
- composing the aspects into a single large aspect *asp_repmin*
- define the function *repmin* that takes a *tree*, calls the semantic function for the tree and the aspect *asp_repmin*, and selects the synthesized attribute *sres* from the result.

Together these rules define for each of the productions a so-called Data Dependency Graph (DDG). A DDG is basically a data-flow graph (Figure 4.5), with as incoming values the inherited attributes of the father node and the synthesized attributes of the children nodes (indicated by closed arrows), and as outputs the inherited attributes of the children nodes and the synthesized attributes of the father node (open arrows). The semantics of our DSL is defined as the data-flow graph which results from composing all the DDGs corresponding to the individual nodes of the abstract syntax tree. Note that the semantics of a tree is thus represented by a function which maps the inherited attributes of the root node onto its synthesized attributes.

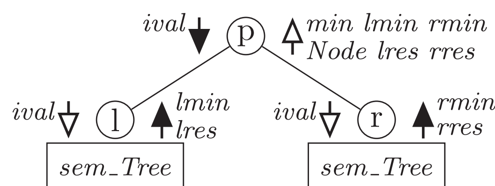

Figure 4.5: The DDG for *Node*

The main result of this chapter is a combinator based implementation of attribute grammars in Haskell; it has statically type checked semantic functions, it is statically checked for correctness at the attribute grammar level, and high-level attribute evaluation patterns can be described.

In Section 4.2 we introduce the heterogeneous collections, which are used to combine a collection of inherited or synthesised attributes into a single value. In Section 4.3

[1] Available at: `http://hackage.haskell.org/package/AspectAG`.

we show how individual attribute grammar rules are represented. In Section 4.4 we introduce the aforementioned $\oplus$ operator which combines the aspects. In Section 4.5 we introduce a function *knit* which takes the DDG associated with the production used at the root of a tree and the mappings (*sem_...* functions) from inherited to synthesised attributes for its children (i.e. the data flow over the children trees) and out of this constructs a data flow computation over the combined tree. In Section 4.6 we show how the common patterns can be encoded in our library, and in Section 4.7 we show how default aspects can be defined. In Section 4.8 we discuss related work, and in Section 4.9 we conclude.

4.2 HList

The library HList [35] implements typeful heterogeneous collections (lists, records, etc.), using techniques for dependently typed programming in Haskell [26, 43] which in turn make use of Haskell 98 extensions for multi-parameter classes [53] and functional dependencies [33]. The idea of *type-level programming* is based on the use of types to represent type-level values, and classes to represent type-level types and functions.

In order to be self-contained we start out with a small introduction. To represent Boolean values at the type level we define a new type for each of the Boolean values. The class *HBool* represents the type-level type of Booleans. We may read the instance definitions as "the type-level values *HTrue* and *HFalse* have the type-level type *HBool*":

class *HBool x*

data *HTrue*; *hTrue* $= \bot :: HTrue$
data *HFalse*; *hFalse* $= \bot :: HFalse$

instance *HBool HTrue*
instance *HBool HFalse*

Since we are only interested in type-level computation, we defined *HTrue* and *HFalse* as empty types. By defining an inhabitant for each value we can, by writing expressions at the value level, construct values at the type-level by referring to the types of such expressions.

Multi-parameter classes can be used to describe type-level *relations*, whereas functional dependencies restrict such relations to functions. As an example we define the class *HOr* for type-level disjunction:

class $(HBool\ t, HBool\ t', HBool\ t'') \Rightarrow HOr\ t\ t'\ t'' \mid t\ t' \rightarrow t''$
  where $hOr :: t \rightarrow t' \rightarrow t''$

The context $(HBool\ t, HBool\ t', HBool\ t'')$ expresses that the types t, t' and t'' have to be type-level values of the type-level type *HBool*. The functional dependency $t\ t' \rightarrow t''$ expresses that the parameters t and t' uniquely determine the parameter t''. This implies that once t and t' are instantiated, the instance of t'' must be

uniquely inferable by the type-system, and that thus we are defining a type-level function from t and t' to t''. The type-level function itself is defined by the following non-overlapping instance declarations:

```
instance HOr HFalse HFalse HFalse
  where hOr _ _ = hFalse
instance HOr HTrue HFalse HTrue
  where hOr _ _ = hTrue
instance HOr HFalse HTrue HTrue
  where hOr _ _ = hTrue
instance HOr HTrue HTrue HTrue
  where hOr _ _ = hTrue
```

If we write (*hOr hTrue hFalse*), we know that t and t' are *HTrue* and *HFalse*, respectively. So, the second instance is chosen to select *hOr* from and thus t'' is inferred to be *HTrue*.

Despite the fact that is looks like a computation at the value level, its actual purpose is to express a computation at the type-level; no interesting value level computation is taking place at all. If we had defined *HTrue* and *HFalse* in the following way:

```
data HTrue  = HTrue ; hTrue = HTrue  :: Htrue
data HFalse = HFalse; hFalse = HFalse :: HFalse
```

then the same computation would also be performed at the value level, resulting in the value *HTrue* of type *HTrue*.

4.2.1 Heterogeneous Lists

Heterogeneous lists are represented with the data types *HNil* and *HCons*, which model the structure of a normal list both at the value and type level:

```
data HNil       = HNil
data HCons e l = HCons e l
```

The sequence *HCons True* (*HCons* "bla" *HNil*) is a correct heterogeneous list with type *HCons Bool* (*HCons String HNil*). We introduce the class *HList* and its instances to prevent incorrect expressions, such as *HCons True False*, where the second argument of *HCons* is not a type-level list. This constraint is expressed by adding a context condition to the *HCons*... instance:

```
class HList l
instance HList HNil
instance HList l ⇒ HList (HCons e l)
```

The library includes a multi-parameter class *HExtend* to model the extension of heterogeneous collections.

$$\textbf{class}\ \mathit{HExtend}\ e\ l\ l' \mid e\ l \rightarrow l', l' \rightarrow e\ l$$
$$\quad \textbf{where}\ \mathit{hExtend} :: e \rightarrow l \rightarrow l'$$

The functional dependency $e\ l \rightarrow l'$ makes that *HExtend* is a type-level function, instead of a relation: once e and l are fixed l' is uniquely determined. It fixes the type l' of a collection, resulting from extending a collection of type l with an element of type e. The member *hExtend* performs the same computation at the level of values. The instance of *HExtend* for heterogeneous lists includes the well-formedness condition:

$$\textbf{instance}\ \mathit{HList}\ l \Rightarrow \mathit{HExtend}\ e\ l\ (\mathit{HCons}\ e\ l)$$
$$\quad \textbf{where}\ \mathit{hExtend} = \mathit{HCons}$$

The main reason for introducing the class *HExtend* is to make it possible to encode constraints on the things which can be *HCons*-ed; here we have expressed that the second parameter should be a list again. In the next subsection we will see how to make use of this facility.

4.2.2 Extensible Records

In our code we will make heavy use of non-homogeneous collections: grammars are a collection of productions, and nodes have a collection of attributes and a collection of children nodes. Such collections, which can be extended and shrunk, map typed labels to values and are modeled by an *HList* containing a heterogeneous list of fields, marked with the data type *Record*. We will refer to them as records from now on:

$$\textbf{newtype}\ \mathit{Record}\ r = \mathit{Record}\ r$$

An empty record is a *Record* containing an empty heterogeneous list:

$$\mathit{emptyRecord} :: \mathit{Record}\ \mathit{HNil}$$
$$\mathit{emptyRecord} = \mathit{Record}\ \mathit{HNil}$$

A field with label l (a phantom type [28]) and value of type v is represented by the type:

$$\textbf{newtype}\ \mathit{LVPair}\ l\ v = \mathit{LVPair}\ \{\mathit{valueLVPair} :: v\}$$

Labels are now almost first-class objects, and can be used as type-level values. We can retrieve the label value using the function *labelLVPair*, which exposes the phantom type parameter:

$$\mathit{labelLVPair} :: \mathit{LVPair}\ l\ v \rightarrow l$$
$$\mathit{labelLVPair} = \bot$$

Since we need to represent many labels, we introduce a polymorphic type *Proxy* to represent them; by choosing a different phantom type for each label to be represented we can distinguish them:

```
data Proxy e; proxy = ⊥ :: Proxy e
```

Thus, the following declarations define a record (*myR*) with two elements, labelled by *Label1* and *Label2*:

```
data Label1; label1 = proxy :: Proxy Label1
data Label2; label2 = proxy :: Proxy Label2
field1 = LVPair True  :: LVPair (Proxy Label1) Bool
field2 = LVPair "bla" :: LVPair (Proxy Label2) [Char]
myR = Record (HCons field1 (HCons field2 HNil)
```

Since our lists will represent collections of attributes we want to express statically that we do not have more than a single definition for each attribute occurrence, and so the labels in a record should be all different. This constraint is represented by requiring an instance of the class *HRLabelSet* to be available when defining extendability for records:

```
instance HRLabelSet (HCons (LVPair l v) r)
   ⇒ HExtend (LVPair l v) (Record r) (Record (HCons (LVPair l v) r))
  where hExtend f (Record r) = Record (HCons f r)
```

The class *HasField* is used to retrieve the value part corresponding to a specific label from a record:

```
class HasField l r v | l r → v where
  hLookupByLabel :: l → r → v
```

At the type-level it is statically checked that the record r indeed has a field with label l associated with a value of the type v. At value-level the member *hLookupByLabel* returns the value of type v. So, the following expression returns the string `"bla"`:

```
hLookupByLabel label2 myR
```

The possibility to update an element in a record at a given label position is provided by:

```
class HUpdateAtLabel l v r r′ | l v r → r′ where
  hUpdateAtLabel :: l → v → r → r′
```

In order to keep our programs readable we introduce infix operators for some of the previous functions:

```
(.*.)    = hExtend
_ .=. v  = LVPair v
r # l    = hLookupByLabel l r
```

Furthermore we will use the following syntactic sugar to denote lists and records in the rest of the thesis:

- $\{ \; v1, ..., vn \; \}$ for $(v1 \; .*. \; ... \; .*. \; vn \; .*. \; HNil)$
- $\{\{ \; v1, ..., vn \; \}\}$ for $(v1 \; .*. \; ... \; .*. \; vn \; .*. \; emptyRecord)$

So, for example the definition of *myR* can now be written as:

```
myR = {{ label1 .=. True, label2 .=. "bla" }}
```

4.3 Rules

In this subsection we show how attributes and their defining rules are represented. An *attribution* is a finite mapping from attribute names to attribute values, represented as a *Record*, in which each field represents the name and value of an attribute.

```
type Att att val = LVPair att val
```

The labels[2] (attribute names) for the attributes of the example are:

```
data Att_smin; smin = proxy :: Proxy Att_smin
data Att_ival;  ival  = proxy :: Proxy Att_ival
data Att_sres;  sres  = proxy :: Proxy Att_sres
```

When inspecting what happens at a production we see that information flows from the inherited attribute of the parent and the synthesized attributes of the children (henceforth called in the *input* family) to the synthesized attributes of the parent and the inherited attributes of the children (together called the *output* family from now on). Both the input and the output attribute family is represented by an instance of:

```
data Fam c p = Fam c p
```

A *Fam* contains a single attribution for the parent and a collection of attributions for the children. Thus the type *p* will always be a *Record* with fields of type *Att*, and the type *c* a *Record* with fields of the type:

```
type Chi ch atts = LVPair ch atts
```

where *ch* is a label that represents the name of that child and *atts* is again a *Record* with the fields of type *Att* associated with this particular child. In our example the *Root* production has a single child *Ch_tree* of type *Tree*, the *Node* production has two children labelled by *Ch_l* and *Ch_r* of type *Tree*, and the *Leaf* production has a single child called *Ch_i* of type *Int*. As we will see later it comes in handy if, besides the name of an element, we also encode its type in the label. Thus we generate, using template Haskell:

[2] These and all needed labels can be generated automatically by Template Haskell functions available in the library

$$
\begin{array}{llll}
\textbf{data}\ \mathit{Ch_tree}; & \mathit{ch_tree} & = \mathit{proxy} :: \mathit{Proxy}\ (\mathit{Ch_tree}, \mathit{Tree}) \\
\textbf{data}\ \mathit{Ch_r}; & \mathit{ch_r} & = \mathit{proxy} :: \mathit{Proxy}\ (\mathit{Ch_r}, \mathit{Tree}) \\
\textbf{data}\ \mathit{Ch_l}; & \mathit{ch_l} & = \mathit{proxy} :: \mathit{Proxy}\ (\mathit{Ch_l}, \mathit{Tree}) \\
\textbf{data}\ \mathit{Ch_i}; & \mathit{ch_i} & = \mathit{proxy} :: \mathit{Proxy}\ (\mathit{Ch_i}, \mathit{Int})
\end{array}
$$

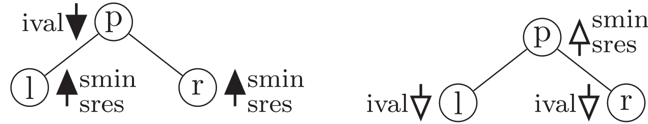

Figure 4.6: Repmin's input and output families for Node

Families are used to model the input and output attributes of attribute computations. For example, Figure 4.6 shows the input (black arrows) and output (white arrows) attribute families of the repmin problem for the production Node. We now give the attributions associated with the output family of the *Node* production, which are the synthesized attributes of the parent (*SP*) and the inherited attributions for the left and right child (*IL* and *IR*):

$$
\begin{array}{lll}
\textbf{type}\ \mathit{SP} & = \mathit{Record}\ (& \mathit{HCons}\ (\mathit{Att}\ (\mathit{Proxy}\ \mathit{Att_smin})\ \mathit{Int}) \\
 & & \mathit{HCons}\ (\mathit{Att}\ (\mathit{Proxy}\ \mathit{Att_sres})\ \ \mathit{Tree}) \\
 & & \mathit{HNil}) \\
\textbf{type}\ \mathit{IL} & = \mathit{Record}\ (& \mathit{HCons}\ (\mathit{Att}\ (\mathit{Proxy}\ \mathit{Att_ival})\ \ \mathit{Int}) \\
 & & \mathit{HNil}) \\
\textbf{type}\ \mathit{IR} & = \mathit{Record}\ (& \mathit{HCons}\ (\mathit{Att}\ (\mathit{Proxy}\ \mathit{Att_ival})\ \ \mathit{Int}) \\
 & & \mathit{HNil})
\end{array}
$$

The next type collects the last two children attributions into a single record:

$$
\begin{array}{lll}
\textbf{type}\ \mathit{IC} & = \mathit{Record}\ (& \mathit{HCons}\ (\mathit{Chi}\ (\mathit{Proxy}\ (\mathit{Ch_l},\ \mathit{Tree}))\ \mathit{IL}) \\
 & & \mathit{HCons}\ (\mathit{Chi}\ (\mathit{Proxy}\ (\mathit{Ch_r}, \mathit{Tree}))\ \mathit{IR}) \\
 & & \mathit{HNil})
\end{array}
$$

We now have all the ingredients to define the output family for *Node*-s.

$$\textbf{type}\ \mathit{Output_Node} = \mathit{Fam}\ \mathit{IC}\ \mathit{SP}$$

Attribute computations are defined in terms of *rules*. As defined by [15], a rule is a mapping from an input family to an output family. In order to make rules composable we define a rule as a mapping from an input family to a function which extends an output family with the new elements defined by this rule:

$$\textbf{type}\ \mathit{Rule}\ \mathit{sc}\ \mathit{ip}\ \mathit{ic}\ \mathit{sp}\ \mathit{ic}'\ \mathit{sp}' = \mathit{Fam}\ \mathit{sc}\ \mathit{ip} \rightarrow \mathit{Fam}\ \mathit{ic}\ \mathit{sp} \rightarrow \mathit{Fam}\ \mathit{ic}'\ \mathit{sp}'$$

Thus, the type *Rule* states that a rule takes as input the synthesized attributes of the children *sc* and the inherited attributes of the parent *ip* and returns a function from the output constructed thus far (inherited attributes of the children *ic* and synthesized attributes of the parent *sp*) to the extended output.

The composition of two rules is the composition of the two functions we get by applying each of them to the input family first:

```
ext :: Rule sc ip ic' sp' ic'' sp'' → Rule sc ip ic sp ic' sp' → Rule sc ip ic sp ic'' sp''
(f `ext` g) input = f input.g input
```

4.3.1 Rule Definition

We now introduce the functions *syndef* and *inhdef*, which are used to define primitive rules which define a synthesized or an inherited attribute respectively. Figure 4.7 lists all the rule definitions for our running example. The naming convention is such that a rule with name *prod_att* defines the attribute *att* for the production *prod*. Without trying to completely understand the definitions we suggest the reader to compare them with their respective **SEM** specifications in Figure 4.3.

```
leaf_smin  (Fam chi par) = syndef smin (chi # ch_i)
node_smin  (Fam chi par) = syndef smin (min ((chi # ch_l) # smin)
                                            ((chi # ch_r) # smin))

root_ival  (Fam chi par) = inhdef ival  { nt_Tree }
                                        {{ ch_tree .=. (chi # ch_tree) # smin }}
node_ival  (Fam chi par) = inhdef ival  { nt_Tree }
                                        {{ ch_l    .=. par # ival
                                         , ch_r    .=. par # ival }}

root_sres  (Fam chi par) = syndef sres  ((chi # ch_tree) # sres)
leaf_sres  (Fam chi par) = syndef sres  (Leaf (par # ival))
node_sres  (Fam chi par) = syndef sres  (Node ((chi # ch_l) # sres)
                                              ((chi # ch_r) # sres))
```

Figure 4.7: Rule definitions for repmin

The function *syndef* adds the definition of a synthesized attribute. It takes a label *att* representing the name of the new attribute, a value *val* to be assigned to this attribute, and it builds a function which updates the output constructed thus far.

```
syndef :: HExtend (Att att val) sp sp'
       ⇒ att → val → (Fam ic sp → Fam ic sp')
syndef att val (Fam ic sp) = Fam ic (att .=. val .*. sp)
```

The record *sp* with the synthesized attributes of the parent is extended with a field with name *att* and value *val*, as shown in Figure 4.8. If we look at the type of the function, the check that we have not already defined this attribute is done by the constraint *HExtend* (*Att att val*) *sp* sp', meaning that sp' is the result of adding the field (*Att att val*) to *sp*, which cannot have any field with name *att*. Thus the Haskell type system is statically preventing duplicated attribute definitions.

Figure 4.8: Synthesized attribute definition

Let us take a look at the rule definition *node_smin* of the attribute *smin* for the production *Node* in Figure 4.7. The children *ch_l* and *ch_r* are retrieved from the input family so we can subsequently retrieve the attribute *smin* from these attributions, and construct the computation of the synthesized attribute *smin*. This process is demonstrated in Figure 4.9. The attribute *smin* is required (underlined) in the children *l* and *r* of the input, and the parent of the output is extended with *smin*.

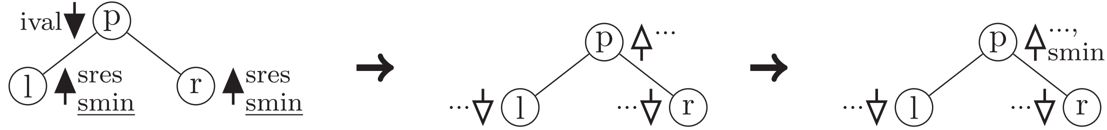

Figure 4.9: Rule *node_sres*

If we take a look at the type which is inferred for *node_sres* we find back all the constraints which are normally checked by an off-line attribute grammar system, i.e., an attribute *smin* is made available by each child (accessed by its name) and an attribute *smin* can be safely added to the current synthesized attribution of the parent: [3]

$$
\begin{aligned}
\mathit{node_sres} :: \; & (\mathit{HasField}\ (\mathit{Proxy}\ (\mathit{Ch_l}, \mathit{Tree}))\ \mathit{sc}\ \mathit{scl} \\
& , \mathit{HasField}\ (\mathit{Proxy}\ \mathit{Att_smin})\ \mathit{scl}\ \mathit{Int} \\
& , \mathit{HasField}\ (\mathit{Proxy}\ (\mathit{Ch_r}, \mathit{Tree}))\ \mathit{sc}\ \mathit{scr} \\
& , \mathit{HasField}\ (\mathit{Proxy}\ \mathit{Att_smin})\ \mathit{scr}\ \mathit{Int} \\
& , \mathit{HExtend}\ (\mathit{Att}\ (\mathit{Proxy}\ \mathit{Att_smin})\ \mathit{Int})\ \mathit{sp}\ \mathit{sp}') \\
\Rightarrow \; & \mathit{Rule}\ \mathit{sc}\ \mathit{ip}\ \mathit{ic}\ \mathit{sp}\ \mathit{ic}\ \mathit{sp}'
\end{aligned}
$$

The function *inhdef* introduces a new inherited attribute for a collection of non-terminals. It takes the following parameters:

[3] In order to keep the explanation simple we will suppose that *min* is not overloaded, and takes *Int*'s as parameter.

att the attribute which is being defined;

nts the non-terminals with which this attribute is being associated;

vals a record labelled with child names and containing values, describing how to compute the attribute being defined at each of the applicable child positions.

The parameter *nts* takes over the role of the **INH** declaration in Figure 4.3. Here this extra parameter seems to be superfluous, since its value can be inferred, but adds an additional restriction to be checked (yielding better error messages) and it will be used in the introduction of default rules later. The names for the non-terminals of our example are:

```
nt_Root = proxy :: Proxy Root
nt_Tree = proxy :: Proxy Tree
```

The result of *inhdef* again is a function which updates the output constructed thus far.

```
inhdef :: Defs att nts vals ic ic'
       ⇒ att → nts → vals → (Fam ic sp → Fam ic' sp)
inhdef att nts vals (Fam ic sp) = Fam (defs att nts vals ic) sp
```

The class *Def* is defined by induction over the record *vals* containing the new definitions. The function *defs* inserts each definition into the attribution of the corresponding child.

```
class Defs att nts vals ic ic' | vals ic → ic' where
  defs :: att → nts → vals → ic → ic'
```

We start out with the base case, where we have no more definitions to add. In this case the inherited attributes of the children are returned unchanged.

```
instance Defs att nts (Record HNil) ic ic where
  defs _ _ _ ic = ic
```

The instance for *HCons* given below first recursively processes the rest of the definitions by updating the collection of collections of inherited attributes of the children *ic* into *ic'*. A helper type level function *SingleDef* (and its corresponding value level function *singledef*) is used to incorporate the single definition (*pch*) into *ic'*, resulting in a new set *ic''*. The type level functions *HasLabel* and *HMember* are used to statically check whether the child being defined (*lch*) exists in *ic'* and if its type (*t*) belongs to the non-terminals *nts*, respectively. The result of both functions are *HBool*s (either *HTrue* or *HFalse*) which are passed as parameters to *SingleDef*. We are now ready to give the definition for the non-empty case:

```
instance (Defs att nts (Record vs) ic ic'
         , HasLabel (Proxy (lch, t)) ic' mch
```

```
           , HMember (Proxy t) nts mnts
           , SingleDef mch mnts att (Chi (Proxy (lch, t)) vch) ic' ic'')
    => Defs att nts (Record (HCons (Chi (Proxy (lch, t)) vch) vs)) ic ic''
  where
    defs att nts ~(Record (HCons pch vs)) ic = singledef mch mnts att pch ic'
      where ic'  = defs att nts (Record vs) ic
            lch  = labelLVPair pch
            mch  = hasLabel lch ic'
            mnts = hMember (sndProxy lch) nts
```

The class *Haslabel* can be encoded straightforwardly, together with a function which retrieves part of a phantom type:

```
class HBool b => HasLabel l r b | l r -> b
instance HasLabel l r b => HasLabel l (Record r) b
instance (HEq l lp b, HasLabel l r b', HOr b b' b'')
    => HasLabel l (HCons (LVPair lp vp) r) b''
instance HasLabel l HNil HFalse

hasLabel :: HasLabel l r b => l -> r -> b
hasLabel = ⊥

sndProxy :: Proxy (a, b) -> Proxy b
sndProxy _ = ⊥
```

We only show the instance with both *mch* and *mnts* equal to *HTrue*, which is the case we expect to apply in a correct attribute grammar definition: we do not refer to children which do not exist, and this child has the type we expect.[4]

```
class SingleDef mch mnts att pv ic ic' | mch mnts pv ic -> ic'
  where singledef :: mch -> mnts -> att -> pv -> ic -> ic'

instance (HasField lch ic och
         , HExtend (Att att vch) och och'
         , HUpdateAtLabel lch och' ic ic')
    => SingleDef HTrue HTrue att (Chi lch vch) ic ic'
  where
    singledef _ _ att pch ic = hUpdateAtLabel lch (att .=. vch .*. och) ic
      where lch = labelLVPair  pch
            vch = valueLVPair pch
            och = hLookupByLabel lch ic
```

We will guarantee that the collection of attributions *ic* (inherited attributes of the children) contains an attribution *och* for the child *lch*, and so we can use *hUpdateAtlabel*

[4]The instances for error cases could just be left undefined, yielding to "undefined instance" type errors. In our library we use a class *Fail* (as defined in [35], section 6) in order to get more instructive type error messages.

to extend the attribution for this child with a field (*Att att vch*), thus binding attribute *att* to value *vch*. The type system checks, thanks to the presence of *HExtend*, that the attribute *att* was not defined before in *och*.

4.3.2 Monadic Rule Definition

To make definitions look somewhat "prettier", we have defined the functions *syndefM* and *inhdefM*, that are versions of *syndef* and *inhdef* that use a *Reader* monad to access the attributes. In Figure 4.10 we show the rules of Figure 4.7 written in terms of *syndefM* and *inhdefM*. The monad keeps the input family; we use *at* to lookup an

```
leaf_smin = syndefM smin $ liftM id (at ch_i)
node_smin = syndefM smin $ do l ← at ch_l
                              r ← at ch_r
                              return $ min (l # smin) (r # smin)

root_ival  = inhdefM ival { nt_Tree } $
                 do tree ← at ch_tree
                    return {{ ch_tree .=. tree # smin }}
node_ival  = inhdefM ival { nt_Tree } $
                 do lhs ← at lhs
                    return {{ ch_l   .=. lhs # ival
                            , ch_r   .=. lhs # ival }}

root_sres  = syndefM sres  $ liftM (#sres) (at ch_tree)
leaf_sres  = syndefM sres  $ do lhs ← at lhs
                               return $ Leaf (lhs # ival)
node_sres  = syndefM sres  $ do l ← at ch_l
                               r ← at ch_r
                               return $ Node (l # sres) (r # sres)
```

Figure 4.10: Monadic Rule definitions for repmin

attribution given a label.

```
class At l m v | l → v where
  at :: l → m v
```

The type of the returned attribute depends on the label. If the label indicates a child of the production (i.e. it has type (*Proxy* (*lch*, *nt*))) the attribution of this child is returned.

```
instance (HasField (Proxy (lch, nt)) chi v
         , MonadReader (Fam chi par) m)
```

```
              ⇒ At (Proxy (lch, nt)) m v where
   at lbl = liftM (λ(Fam chi _) → chi # lbl) ask
```

If the label is *lhs* (with type (*Proxy Lhs*)) then the returned attribution is the one corresponding to the left-hand side.

```
data Lhs; lhs = proxy :: Proxy Lhs
instance MonadReader (Fam chi par) m
              ⇒ At (Proxy Lhs) m par where
   at _ = liftM (λ(Fam _ par) → par) ask
```

We can pass the input family to a monadic attribute computation by running (*runReader*) the monad.

```
def :: Reader (Fam chi par) val → ((Fam chi par) → val)
def = runReader
```

Thus, the functions *syndefM* and *inhdefM* receive a monadic computation, run it, and apply the respective non-monadic function to the obtained attribute computation.

```
syndefM :: (HExtend (Att att val) sp sp')
          ⇒ att → Reader (Fam sc ip) val → Rule sc ip ic sp ic sp'
syndefM att mval inp = syndef att (def mval inp)

inhdefM :: (Defs att nts vals ic ic')
          ⇒ att → nts → Reader (Fam sc ip) vals → Rule sc ip ic sp ic' sp
inhdefM att nts mvals inp = inhdef att nts (def mvals inp)
```

4.3.3 Applicative Rule Definition

Given an instance of *Applicative* for *Reader*, we can define rules in an applicative way using the same functions we defined for the monadic case. In Figure 4.11 we show the applicative version of our example.

We define *attr* to lift the value of an attribute *att* of a given attribution *ch*:

```
attr :: (HasField att chi val, At ch (Reader (Fam sc ip)) chi)
     ⇒ ch → att → Reader (Fam sc ip) val
attr ch att = (#att) <$> at ch
```

We also define some combinators to lift the construction of records. Thus the rules for inherited attributes, like *node_ival*, can be written in a prettier way.

```
infixr 2 .*..
(.*..) :: HExtend f r r'
   ⇒ Reader (Fam sc ip) f → Reader (Fam sc ip) r → Reader (Fam sc ip) r'
```

```
leaf_smin = syndefM smin $ at ch_i
node_smin = syndefM smin $ min <$> attr ch_l smin <*> attr ch_r smin

root_ival = inhdefM ival { nt_Tree } $
                         {{{ ch_tree .=.. attr ch_tree smin }}}
node_ival = inhdefM ival { nt_Tree } $
                         {{{ ch_l    .=.. attr lhs ival
                         ,   ch_r    .=.. attr lhs ival }}}

root_sres = syndefM sres $ attr ch_tree sres
leaf_sres = syndefM sres $ Leaf <$> attr lhs ival
node_sres = syndefM sres $ Node <$> attr ch_l sres <*> attr ch_r sres
```

Figure 4.11: Applicative Rule definitions for repmin

```
x .*.. xs = (.*.) <$> x <*> xs
infixr 3 .=..
(.=..) :: ch → Reader (Fam sc ip) val → Reader (Fam sc ip) (LVPair ch val)
l .=.. f = (.=.) <$> pure l <*> f
emptyRecordA :: Reader (Fam sc ip) (Record HNil)
emptyRecordA = pure emptyRecord
```

In Figure 4.11 we used some syntax sugar ({{{...}}}) to represent lifted records, similar to the syntax we used for records.

4.4 Aspects

We represent aspects as records which contain for each production a rule field:

```
type Prd prd rule = LVPair prd rule
```

For our example we thus introduce fresh labels to refer to repmin's productions:

```
data P_Root; p_Root = proxy :: Proxy P_Root
data P_Node; p_Node = proxy :: Proxy P_Node
data P_Leaf; p_Leaf = proxy :: Proxy P_Leaf
```

We now can define the aspects of repmin as records with the rules of Figure 4.7.[5]

```
asp_smin = {{ p_Leaf .=. leaf_smin
            , p_Node .=. node_smin }}
```

[5] We assume that the monomorphism restriction has been switched off.

```
asp_ival  = {{ p_Root .=. root_ival
             , p_Node .=. node_ival }}
asp_sres  = {{ p_Root .=. root_sres
             , p_Node .=. node_sres
             , p_Leaf .=. leaf_sres }}
```

4.4.1 Aspects Combination

We define the class *Com* which will provide the instances we need for combining aspects:

```
class Com r r' r'' | r r' → r''
  where (⊕) :: r → r' → r''
```

With this operator we can now combine the three aspects which together make up the repmin problem:

```
asp_repmin = asp_smin ⊕ asp_ival ⊕ asp_sres
```

Combination of aspects is a sort of union of records where, in case of fields with the same label (i.e., for rules for the same production), the rule combination (*ext*) is applied to the values. To perform the union we iterate over the second record, inserting the next element into the first one if it is new and combining it with an existing entry if it exists:

```
instance Com r (Record HNil) r where
  r ⊕ _ = r
instance (HasLabel lprd r b
         , ComSingle b (Prd lprd rprd) r r'''
         , Com r''' (Record r') r'')
  ⇒ Com r (Record (HCons (Prd lprd rprd) r')) r'' where
  r ⊕ (Record (HCons prd r')) = r''
    where b    = hasLabel (labelLVPair prd) r
          r''' = comsingle b prd r
          r''  = r''' ⊕ (Record r')
```

We use the class *ComSingle* to insert a single element into the first record. The type-level Boolean parameter b is used to distinguish those cases where the left hand operand already contains a field for the rule to be added and the case where it is new.[6]

```
class ComSingle b f r r' | b f r → r'
  where comsingle :: b → f → r → r'
```

[6]This parameter can be avoided by allowing overlapping instances, but we prefer to minimize the number of Haskell extensions we use.

If the first record has a field with the same label *lprd*, we update its value by composing the rules.

```
instance (HasField lprd r (Rule sc ip ic' sp' ic'' sp'')
          , HUpdateAtLabel lprd (Rule sc ip ic sp ic'' sp'') r r')
  ⇒ ComSingle HTrue (Prd lprd (Rule sc ip ic sp ic' sp')) r r' where
  comsingle _ f r = hUpdateAtLabel n ((r # n) `ext` v) r
    where n = labelLVPair f
          v = valueLVPair f
```

In case the first record does not have a field with the label, we just insert the element in the record.

```
instance ComSingle HFalse f (Record r) (Record (HCons f r)) where
  comsingle _ f (Record r) = Record (HCons f r)
```

4.5 Semantic Functions

Our overall goal is to construct a *Tree*-algebra and a *Root*-algebra. For the domain associated with each non-terminal we take the function mapping its inherited to its synthesized attributes. The hard work is done by the function *knit*, the purpose of which is to combine the data flow defined by the DDG –which was constructed by combining all the rules for this production– with the semantic functions of the children (describing the flow of data from their inherited to their synthesized attributes) into the semantic function for the parent.

With the attribute computations as first-class entities, we can now pass them as an argument to functions of the form *sem_* < *nt*>. The following code follows the definitions of the data types at hand: it contains recursive calls for all children of an alternative, each of which results in a mapping from inherited to synthesized attributes for that child followed by a call to knit, which stitches everything together:

```
sem_Root asp (Root t)     = knit (asp # p_Root) {{ ch_tree .=. sem_Tree asp t }}
sem_Tree asp (Node l r) = knit (asp # p_Node) {{ ch_l    .=. sem_Tree asp l
                                               ,  ch_r    .=. sem_Tree asp r }}
sem_Tree asp (Leaf i )    = knit (asp # p_Leaf) {{ ch_i    .=. sem_Lit i }}


sem_Lit e (Record HNil) = e
```

Since this code is completely generic we provide a Template Haskell function *deriveAG* which automatically generates the functions such as *sem_Root* and *sem_Tree*, together with the labels for the non-terminals and labels for referring to children. Thus, to completely implement the *repmin* function we need to undertake the following steps:

- Generate the semantic functions and the corresponding labels by using:

  $\$\ (deriveAG\ \text{“}Root)$

- Define and compose the aspects as shown in the previous sections, resulting in *asp_repmin*.

- Define the function *repmin* that takes a *tree*, executes the semantic function for the tree and the aspect *asp_repmin*, and selects the synthesized attribute *sres* from the result.

  $repmin\ tree = sem_Root\ asp_repmin\ (Root\ tree)\ ()\ \#\ sres$

4.5.1 The Knit Function

As said before the function *knit* takes the combined rules for a node and the semantic functions of the children, and builds a function from the inherited attributes of the parent to its synthesized attributes. We start out by constructing an empty output family, containing an empty attribution for each child and one for the parent. To each of these attributions we apply the corresponding part of the rules, which will construct the inherited attributes of the children and the synthesized attributes of the parent (together forming the output family). Rules however contain references to the input family, which is composed of the inherited attributes of the parent *ip* and the synthesized attributes of the children *sc*.

```
knit :: (Empties fc ec, Kn fc ic sc)
     ⇒ Rule sc ip ec (Record HNil) ic sp → fc → ip → sp
knit rule fc ip = let ec          = empties fc
                      (Fam ic sp) = rule (Fam sc ip) (Fam ec emptyRecord)
                      sc          = kn fc ic
                  in  sp
```

The function *kn*, which takes the semantic functions of the children (*fc*) and their inputs (*ic*), computes the results for the children (*sc*). The functional dependency $fc \rightarrow ic\ sc$ indicates that *fc* determines *ic* and *sc*, so the shape of the record with the semantic functions determines the shape of the other records:

```
class Kn fc ic sc | fc → ic sc where
  kn :: fc → ic → sc
```

We declare a helper instance of *Kn* to remove the *Record* tags of the parameters, in order to be able to iterate over their lists without having to tag and untag at each step:

```
instance Kn fc ic sc ⇒ Kn (Record fc) (Record ic) (Record sc) where
  kn (Record fc) (Record ic) = Record $ kn fc ic
```

When the list of children is empty, we just return an empty list of results.

```
instance Kn HNil HNil HNil where
  kn _ _ = hNil
```

The function *kn* is a type level *zipWith* ($), which applies the functions contained in the first argument list to the corresponding element in the second argument list.

```
instance Kn fcr icr scr ⇒ Kn (HCons (Chi lch (ich → sch)) fcr)
                             (HCons (Chi lch ich         ) icr)
                             (HCons (Chi lch sch         ) scr) where
  kn ~(HCons pfch fcr) ~(HCons pich icr) =
    let scr = kn fcr icr
        lch = labelLVPair  pfch
        fch = valueLVPair pfch
        ich = valueLVPair pich
    in  HCons (newLVPair lch (fch ich)) scr
```

The class *Empties* is used to construct the record, with an empty attribution for each child, which we have used to initialize the computation of the input attributes with.

```
class Empties fc ec | fc → ec where
  empties :: fc → ec
```

In the same way that *fc* determines the shape of *ic* and *sc* in *Kn*, it also tells us how many empty attributions *ec* to produce and in which order:

```
instance Empties fc  ec ⇒ Empties (Record fc) (Record ec) where
  empties (Record fc) = Record $ empties fc
instance Empties fcr ecr ⇒ Empties (HCons (Chi lch fch)            fcr)
                                   (HCons (Chi lch (Record HNil)) ecr)
  where
  empties ~(HCons pch fcr) = let ecr = empties fcr
                                 lch = labelLVPair pch
                             in  HCons (newLVPair lch emptyRecord) ecr
instance Empties HNil HNil where
  empties _ = hNil
```

4.6 Common Patterns

At this point all the basic functionality of attribute grammars has been implemented. In practice however we want more. If we look at the code in Figure 4.3 we see that the rules for *ival* at the production *Node* are “free of semantics”, since the value is

copied unmodified to its children. If we were dealing with a tree with three children instead of two the extra line would look quite similar. When programming attribute grammars such patterns are quite common and most attribute grammar systems contain implicit rules which automatically insert such "trivial" rules. As a result descriptions can decrease in size dramatically. The question now arises whether we can extend our embedded language to incorporate such more high level data flow patterns.

4.6.1 Copy Rule

The most common pattern is the copying of an inherited attribute from the parent to all its children. We capture this pattern with the an operator *copy*, which takes the name of an attribute *att* and an heterogeneous list of non-terminals *nts* for which the attribute has to be defined, and generates a copy rule for this. This corresponds closely to the introduction of a *Reader* monad.

```
copy :: (Copy att nts vp ic ic', HasField att ip vp)
     ⇒ att → nts → Rule sc ip ic sp ic' sp
```

Thus, for example, the rule *node_ival* of Figure 4.7 can now be written as:

```
node_ival input = copy ival { nt_Tree } input
```

The function *copy* uses a function *defcp* to define the attribute *att* as an inherited attribute of its children. This function is similar in some sense to *inhdef*, but instead of taking a record containing the new definitions it gets the value *vp* of the attribute which is to be copied to the children:

```
copy att nts (Fam _ ip) = defcp att nts (ip # att)

defcp :: Copy att nts vp ic ic' ⇒ att → nts → vp → (Fam ic sp → Fam ic' sp)
defcp att nts vp (Fam ic sp) = Fam (cpychi att nts vp ic) sp
```

The class *Copy* iterates over the record *ic* containing the output attribution of the children, and inserts the attribute *att* with value *vp* if the type of the child is included in the list *nts* of non-terminals and the attribute is not already defined for this child.

```
class Copy att nts vp ic ic' | ic → ic' where
  cpychi :: att → nts → vp → ic → ic'


instance Copy att nts vp (Record HNil) (Record HNil)
  where cpychi _ _ _ _ = emptyRecord
instance (Copy att nts vp (Record ics) ics'
         , HMember (Proxy t) nts mnts
         , HasLabel att vch mvch
```

$$
\begin{array}{l}
\quad\quad , Copy'\ mnts\ mvch\ att\ vp\ (Chi\ (Proxy\ (lch, t))\ vch)\ pch \\
\quad\quad , HExtend\ pch\ ics'\ ic) \\
\Rightarrow Copy\ att\ nts\ vp\ (Record\ (HCons\ (Chi\ (Proxy\ (lch, t))\ vch)\ ics))\ ic \\
\textbf{where} \\
cpychi\ att\ nts\ vp\ (Record\ (HCons\ pch\ ics)) = cpychi'\ mnts\ mvch\ att\ vp\ pch\ .{*}.\ ics' \\
\quad \textbf{where}\ ics' = cpychi\ att\ nts\ vp\ (Record\ ics) \\
\quad\quad lch = sndProxy\ (labelLVPair\ pch) \\
\quad\quad vch = valueLVPair\ pch \\
\quad\quad mnts = hMember\ lch\ nts \\
\quad\quad mvch = hasLabel\ att\ vch
\end{array}
$$

The function $cpychi'$ updates the field pch by adding the new attribute:

$$
\begin{array}{l}
\textbf{class}\ Copy'\ mnts\ mvch\ att\ vp\ pch\ pch' \mid mnts\ mvch\ pch \to pch' \\
\quad \textbf{where}\ cpychi' :: mnts \to mvch \to att \to vp \to pch \to pch'
\end{array}
$$

When the type of the child doesn't belong to the non-terminals for which the attribute is defined we define an instance which leaves the field pch unchanged.

$$
\begin{array}{l}
\textbf{instance}\ Copy'\ HFalse\ mvch\ att\ vp\ pch\ pch\ \textbf{where} \\
\quad cpychi'\ _\ _\ _\ _\ pch = pch
\end{array}
$$

We also leave pch unchanged if the attribute is already defined for this child.

$$
\begin{array}{l}
\textbf{instance}\ Copy'\ HTrue\ HTrue\ att\ vp\ pch\ pch\ \textbf{where} \\
\quad cpychi'\ _\ _\ _\ _\ pch = pch
\end{array}
$$

In other case the attribution vch is extended with the attribute $(Att\ att\ vp)$.

$$
\begin{array}{l}
\textbf{instance}\ HExtend\ (Att\ att\ vp)\ vch\ vch' \\
\quad \Rightarrow Copy'\ HTrue\ HFalse\ att\ vp\ (Chi\ lch\ vch)\ (Chi\ lch\ vch')\ \textbf{where} \\
\quad cpychi'\ _\ _\ att\ vp\ pch = lch\ .{=}.\ (att\ .{=}.\ vp\ .{*}.\ vch) \\
\quad\quad \textbf{where}\ lch = labelLVPair\ pch \\
\quad\quad\quad vch = valueLVPair\ pch
\end{array}
$$

4.6.2 Other Rules

In this section we introduce two more constructs of our DSL, without giving their implementation. Besides the *Reader* monad, there is also the *Writer* monad. Often we want to collect information provided by some of the children into an attribute of the parent. This can be used to e.g. collect all identifiers contained in an expression. Such a synthesized attribute can be declared using the *use* rule, which combines the attribute values of the children in similar way as Haskell's *foldr1*. The *use* rule takes the following arguments: the attribute to be defined, the list of non-terminals for which the attribute is defined, a monoidal operator which combines the attribute

values, and a unit value to be used in those cases where none of the children has such an attribute.

$$\begin{aligned} &use :: (Use\ att\ nts\ a\ sc, HExtend\ (Att\ att\ a)\ sp\ sp')\\ &\quad \Rightarrow att \to nts \to (a \to a \to a) \to a \to Rule\ sc\ ip\ ic\ sp\ ic\ sp' \end{aligned}$$

Using this new combinator the rule *node_smin* of Figure 4.7 becomes:

$$node_smin = use\ smin\ \{\ nt_Tree\ \}\ min\ 0$$

A third common pattern corresponds to the use of the *State* monad. A value is threaded in a depth-first way through the tree, being updated every now and then. For this we have chained attributes (both inherited and synthesized). If a definition for a synthesized attribute of the parent with this name is missing we look for the right-most child with a synthesized attribute of this name. If we are missing a definition for one of the children, we look for the right-most of its left siblings which can provide such a value, and if we cannot find it there, we look at the inherited attributes of the father.

$$\begin{aligned} &chain :: (Chain\ att\ nts\ val\ sc\ ic\ sp\ ic'\ sp', HasField\ att\ ip\ val)\\ &\quad \Rightarrow att \to nts \to Rule\ sc\ ip\ ic\ sp\ ic'\ sp' \end{aligned}$$

4.7 Defining Aspects

Now we have both implicit rules to define attributes, and explicit rules which contain explicit definitions, we may want to combine these into a single *attribute aspect* which contains all the definitions for single attribute. We now refer to Figure 4.12 which is a desugared version of the notation presented in the introduction.

An inherited attribute aspect, like *asp_ival* in Figure 4.12, can be defined using the function *inhAspect*. It takes as arguments: the name of the attribute *att*, the list *nts* of non-terminals where the attribute is defined, the list *cpys* of productions where the copy rule has to be applied, and a record *defs* containing the explicit definitions for some productions:

$$\begin{aligned} inhAspect\ att\ nts\ cpys\ defs = &\ (defAspect\ (FnCpy\ att\ nts)\ cpys)\\ &\oplus (attAspect\ (FnInh\ att\ nts)\ \ defs) \end{aligned}$$

The function *attAspect* generates an attribute aspect given the explicit definitions, whereas *defAspect* constructs an attribute aspect based in a common pattern's rule. Thus, an inherited attribute aspect is defined as a composition of two attribute aspects: one with the explicit definitions and other with the application of the copy rule. In the following sections we will see how *attAspect* and *defAspect* are implemented.

A synthesized attribute aspect, like *asp_smin* and *asp_sres* in Figure 4.12, can be defined using *synAspect*. Here the rule applied is the use rule, which takes *op* as the monoidal operator and *unit* as the unit value.

```
asp_smin
   = synAspect smin { nt_Tree }                              -- synthesize at
               min 0 { p_Node }                              -- use at
               {{ p_Leaf .=. (λ(Fam chi _) → chi # ch_i) }}  -- define at
asp_ival
   = inhAspect ival  { nt_Tree }                             -- inherit
               { p_Node }                                    -- copy at
               {{ p_Root .=. (λ(Fam chi _) →
                     {{ ch_tree .=. (chi # ch_tree) # smin }}) }}  -- define at
asp_sres
   = synAspect sres  { nt_Root, nt_Tree el
               Node (Leaf 0) { p_Node }
               {{ p_Root .=. (λ(Fam chi _   ) → (chi # ch_tree) # sres)
               ,  p_Leaf .=. (λ(Fam _   par) → Leaf (par # ival)) }}
```

Figure 4.12: Aspects definition for repmin

```
synAspect att nts op unit uses defs = (defAspect (FnUse att nts op unit) uses)
                                    ⊕ (attAspect (FnSyn att)              defs)
```

A chained attribute definition introduces both an inherited and a synthesized attribute. In this case the pattern to be applied is the chain rule.

```
chnAspect att nts chns inhdefs syndefs = (defAspect (FnChn att nts) chns)
                                       ⊕ (attAspect (FnInh att nts) inhdefs)
                                       ⊕ (attAspect (FnSyn att)     syndefs)
```

4.7.1 Attribute Aspects

Consider the explicit definitions of the aspect *asp_sres*. The idea is that, when declaring the explicit definitions, instead of completely writing the rules, like:

```
{{ p_Root .=. (λinput → syndef sres ((chi input # ch_tree) # sres))
,  p_Leaf .=. (λinput → syndef sres (Leaf (par input # ival))) }}
```

we just define a record with the functions from the input to the attribute value:

```
{{ p_Root .=. (λinput → (chi input # ch_tree) # sres)
,  p_Leaf .=. (λinput → Leaf (par input # ival)) }}
```

By mapping the function ((.) (*syndef sres*)) over such records, we get back our previous record containing rules. The function *attAspect* updates all the values of a record by applying a function to them:

```
class AttAspect rdef defs rules | rdef defs → rules
      where attAspect :: rdef → defs → rules
instance (AttAspect rdef (Record defs) rules
             , Apply rdef def rule
             , HExtend (Prd lprd rule) rules rules')
         ⇒ AttAspect rdef (Record (HCons (Prd lprd def) defs)) rules'
   where
      attAspect rdef (Record (HCons def defs)) =
         let lprd = (labelLVPair def)
         in  lprd .=. apply rdef (valueLVPair def) .*. attAspect rdef (Record defs)
instance AttAspect rdef (Record HNil) (Record HNil) where
      attAspect _ _ = emptyRecord
```

The class *Apply* (from the HList library) models the function application, and it is used to add specific constraints on the types:

```
class Apply f a r | f a → r
   where apply :: f → a → r
```

In the case of synthesized attributes we apply ((.) (*syndef att*)) to values of type (*Fam sc ip* → *val*) in order to construct a rule of type (*Rule sc ip ic sp ic sp'*). The constraint *HExtend* (*LVPair att val*) *sp sp'* is introduced by the use of *syndef*. The data type *FnSyn* is used to determine which instance of *Apply* has to be chosen.

```
data FnSyn att = FnSyn att
instance HExtend (LVPair att val) sp sp'
    ⇒ Apply (FnSyn att) (Fam sc ip → val) (Rule sc ip ic sp ic sp') where
   apply (FnSyn att) f = syndef att . f
```

In the case of inherited attributes the function ((.) (*inhdef att nts*)) is applied to define the rule.

```
data FnInh att nt = FnInh att nt
instance Defs att nts vals ic ic'
    ⇒ Apply (FnInh att nts) (Fam sc ip → vals) (Rule sc ip ic sp ic' sp) where
   apply (FnInh att nts) f = inhdef att nts . f
```

4.7.2 Default Aspects

The function *defAspect* is used to construct an aspect given a rule and a list of production labels.

```
class DefAspect deff prds rules | deff prds → rules
   where defAspect :: deff → prds → rules
```

It iterates over the list of labels *prds*, constructing a record with these labels and a rule determined by the parameter *deff* as value. For inherited attributes we apply the copy rule *copy att nts*, for synthesized attributes *use att nt op unit* and for chained attributes *chain att nts*. The following types are used, in a similar way than in *attAspect*, to determine the rule to be applied:

data *FnCpy att nts* = *FnCpy att nts*
data *FnUse att nt op unit* = *FnUse att nt op unit*
data *FnChn att nt* = *FnChn att nt*

Thus, for example in the case of the aspect *asp_ival*, the application:

defAspect (*FnCpy ival* { *nt_Tree* }) { *p_Node* }

generates the default aspect:

{{ *p_Node* .=. *copy ival* { *nt_Tree* } }}

4.8 Related Work

There have been several previous attempts at incorporating first-class attribute grammars in lazy functional languages. To the best of our knowledge all these attempts exploit some form of extensible records to collect attribute definitions. They however do not exploit the Haskell class system as we do. De Moor et al. [16] introduce a whole collection of functions, and a result it is no longer possible to define copy, use and chain rules. Other approaches fail to provide some of the static guarantees that we have enforced [15].

The exploration of the limitations of type-level programming in Haskell is still a topic of active research. For example, there has been recent work on modelling relational data bases using techniques similar to those applied in this paper [59].

As to be expected the type-level programming performed here in Haskell can also be done in dependently typed languages such as Agda [47, 48]. By doing so, we use Boolean values in type level-functions, thereby avoiding the need for a separate definition of the type-level Booleans. This would certainly simplify certain parts of our development. On the other hand, because Agda only permits the definition of total functions, we would need to maintain even more information in our types to make it evident that all our functions are indeed total.

An open question is how easy it will be to extend the approach taken to more global strategies of accessing attributes definitions; some attribute grammars systems allow references to more remote attributes [55, 10]. Although we are convinced that we can in principle encode such systems too, the question remains how much work this turns out to be.

Another thing we could have done is to make use of associated types [14] in those cases where our relations are actually functions; since this feature is still experimental and has only recently become available we have refrained from doing so for the

moment. Future work includes this and the use of promotion of data types to kinds [78] to avoid the use of type classes (e.g. *HList*) to represent the types of type level values.

4.9 Conclusions

In the first place we remark that we have achieved all four goals stated in the introduction:

1. removing the need for a whole collection of indexed combinators as used in [16]
2. replacing extensible records completely by heterogeneous collections
3. the description of common attribute grammar patterns in order to reduce code size, and making them almost first class objects
4. give a nice demonstration of type level programming

We have extensive experience with attribute grammars in the construction of the Utrecht Haskell compiler [19]. The code of this compiler is completely factored out along the two axes mentioned in the introduction [20, 24, 18], using the notation used in Figure 4.3. In doing so we have found the possibility to factor the code into separate pieces of text indispensable.

We also have come to the conclusion that the so-called monadic approach, although it may seem attractive at first sight, in the end brings considerable complications when programs start to grow [32]. Since monad transformers are usually type based we already run into problems if we extend a state twice with a value of the same type without taking explicit measures to avoid confusion. Another complication is that the interfaces of non-terminals are in general not uniform, thus necessitating all kind of tricks to change the monad at the right places, keeping information to be reused later, etc. In our generated Haskell compiler [19] we have non-terminals with more than 10 different attributes, and glueing all these together or selectively leaving some out turns out to be impossible to do by hand.

In our attribute grammar system (`uuagc` on Hackage), we perform a global flow analysis, which makes it possible to schedule the computations explicitly [34]. Once we know the evaluation order we do not have to rely on lazy evaluation, and all parameter positions can be made strict. When combined with a uniqueness analysis we can, by reusing space occupied by unreachable attributes, get an even further increase in speed. This leads to a considerable, despite constant, speed improvement. Unfortunately we do not see how we can perform such analyses with the approach described in this chapter: the semantic functions defining the values of the attributes in principle access the whole input family, and we cannot find out which functions only access part of such a family, and if so which part.

Of course a straightforward implementation of extensible records will be quite expensive, since basically we use nested pairs to represent attributions. We think however that a not too complicated program analysis will reveal enough information to

be able to transform the program into a much more efficient form by flattening such nested pairs. Note that thanks to our type-level functions, which are completely evaluated by the compiler, we do not have to perform any run-time checks as in [15]: once the program type-checks there is nothing which will prevent it to run to completion, apart form logical errors in the definitions of the attributes.

Concluding we think that the library described here is quite useful and relatively easy to experiment with. We notice furthermore that a conventional attribute grammar restriction, stating that no attribute should depend on itself, does not apply since we build on top of a lazily evaluated language. An example of this can be found in online pretty printing [61, 65]. Once we go for speed it may become preferable to use more conventional off-line generators. Ideally we should like to have a mixed approach in which we can use the same definitions as input for both systems.

5 Attribute Grammar Macros

Having extensible languages is appealing, but raises the question of how to construct extensible compilers and how to compose compilers out of a collection of pre-compiled components.

Being able to deal with attribute grammar fragments as described in Chapter 4 makes it possible to describe semantics in a compositional way; this leads naturally to a plug-in architecture, in which a core compiler can be constructed as a (collection of) pre-compiled component(s), and to which extra components can safely be added as need arises.

We extend `AspectAG` with a set of combinators that make it easy to describe semantics in terms of already existing semantics in a macro-like style, just as syntax macros extend the syntax of a language. We also show how existing semantics can be redefined, thus adapting some aspects from the behavior defined by the macros.

5.1 Introduction

Since the introduction of the very first programming languages, and the invention of grammatical formalisms for describing them, people have investigated how an initial language definition can be extended by someone else than the original language designer by providing separate language-definition fragments.

The simplest approach starts from the *text* which describes a compiler for the base language. Just before the compiler is compiled, several extra ingredients may be added textually. In this way we get great flexibility and there is virtually no limit to the things we may add. The Utrecht Haskell Compiler [19] has shown the effectiveness of this approach by composing a large number of attribute grammar fragments textually into a complete compiler description. This approach however is not very practical when defining relatively small language extensions; we do not want an individual user to have to generate a completely new compiler for each small extension. Another problematic aspect of this approach is that, by making the complete text of the compiler available for modification or extension, we also loose important safety guarantees provided by e.g. the type system; we definitely do not want everyone to mess around with the delicate internals of a compiler for a complex language.

So the question arises how we can reach the effect of textual composition, but without opening up the whole compiler source. The most commonly found approach is to introduce so-called *syntax macros* [39], which enable the programmer to add *syntactic sugar* to a language by defining new notation *in terms of already existing syntax*.

In this chapter we will focus on how to provide such mechanisms *at the semantic level* [40] too. As a running example we take a minimal expression language described by the grammar:

$$
\begin{array}{lll}
expr & \rightarrow & \texttt{"let"}\ var\ \texttt{"="}\ expr\ \texttt{"in"}\ expr \mid term\ \texttt{"+"}\ expr \mid term \\
term & \rightarrow & factor\ \texttt{"*"}\ term \mid factor \\
factor & \rightarrow & int \mid var
\end{array}
$$

with the following abstract syntax (as a Haskell data type):

$$
\begin{array}{lllll}
\mathbf{data}\ Root & = & Root & \{ expr :: Expr \} & \\
\mathbf{data}\ Expr & = & Cst & \{ cv\ \ :: Int \} \mid Var\ \{ vnm :: String \} & \\
 & \mid & Mul & \{ me1 :: Expr, & me2 :: Expr \} \\
 & \mid & Add & \{ ae1\ :: Expr, & ae2\ :: Expr \} \\
 & \mid & Let & \{ lnm\ :: String, & val\ \ :: Expr, body :: Expr \}
\end{array}
$$

Suppose we want to extend the language with an extra production for defining the square of a value. A syntax macro aware compiler might accept definitions of the form $square\ (se :: Expr) \Rightarrow Mul\ se\ se$, translating the new syntax into the existing abstract syntax.

Although this approach may be very effective and seems attractive, such transformational programming [12] has its shortcomings too. As a consequence of mapping the new constructs onto existing constructs and performing any further processing on this simpler, but often more detailed program representation, feedback from later stages of the compiler is given in terms of the intermediate program representations in which the original program structure if often hard to recognise. For example, if we do not change the pretty printing phase of the compiler, the expression $square\ 2$ will be printed as $2 * 2$, and worse, type-checking the expression $square\ True$ will lead to more than a single error message. Hence the implementation details shine through, and the produced error messages can be confusing or even incomprehensible. Similar problems show up when defining embedded domain specific languages: the error messages from the type system are typically given in terms of the underlying representation [27].

In Chapter 4 we introduced `AspectAG`: a Haskell library of first-class attribute grammars, which can be used to implement a language semantics and its extensions in a safe way, i.e. by constructing a core compiler as a (collection of) pre-compiled component(s), to which extra components can safely be added at will. In this chapter we show how we can define the semantics of the right hand side in terms of existing semantics, in the form of *attribute grammar macros*.

We also show how, by using first class attribute grammars, the already defined semantics can easily be *redefined* at the places where it makes a difference, e.g. in pretty printing and generating error messages.

The functionality provided by the combination of attribute grammar macros and redefinition is similar to the *forwarding attributes* [67] technique for higher-order attribute grammars, implemented in the Silver AG system [68]. We however implement

```
  -- Pretty-Printing
sppRoot  = syndefM spp $ liftM (#spp) (at ch_expr)
...
sppAdd   = syndefM spp $ do e1 ← at ch_ae1
                            e2 ← at ch_ae2
                            return $ e1 # spp >#< "+" >#< e2 # spp
...
  -- Environment
ienvRoot = inhdefM ienv { nt_Expr } $
           do return {{ ch_expr .=. ([ ] :: [(String, Int)]) }}
...
ienvLet  = inhdefM ienv { nt_Expr } $
           do lnm ← at ch_lnm
              val ← at ch_val
              lhs ← at lhs
              return {{ ch_val  .=. lhs # ienv
                      , ch_body .=. (lnm, val # sval) : lhs # ienv }}
  -- Value
svalRoot = syndefM sval $ liftM (#sval) (at ch_expr)
...
svalVar  = syndefM sval $ do vnm ← at ch_vnm
                             lhs ← at lhs
                             return $ fromJust (lookup vnm (lhs # ienv))
...
```

Figure 5.1: Fragments of the specification of the example's semantics using the `AspectAG` library

our proposal as a set of combinators embedded in Haskell, such that the consistency of the composite system is checked by the Haskell type checker.

In Section 5.2 we give a top-level overview of our approach. In Section 5.3 we show how to define semantic macros and in Section 5.5 how to redefine attributes. We close by presenting our conclusions and future work.

5.2 Attribute Grammar Combinators

Before delving into the technical details, we show in this section how the semantics of our running example language and some simple extensions can be implemented using our approach. We have chosen our example to be very simple, in order to

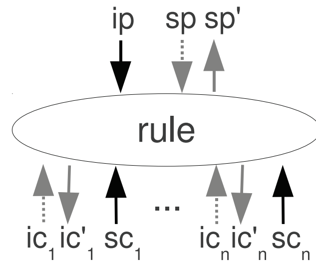

Figure 5.2: Rule: black arrows represent input and gray arrows represent output; dotted gray arrows represent the already constructed output which can be used to compute further output elements (hence the direction of the arrow)

help the understanding of the technique. For a more involved example, including an implementation of the Oberon-0 language [76] using macros to represent the **FOR** and **CASE** statements in terms of a core sub-language, we refer to Chapter 7.

The semantics are defined by two aspects: pretty printing, realized by a synthesized attribute *spp*, which holds a pretty printed document, and expression evaluation, realized by two attributes: a synthesized *sval* of type *Int*, which holds the result of an expression, and an inherited *ienv* which holds the environment ($[(String, Int)]$) in which an expression is to be evaluated. We show how the attributes are directly definable in Haskell using the functions *syndefM* and *inhdefM* from the `AspectAG` library, which define a single synthesized or inherited attribute respectively. Figure 5.1 lists some of the rule definitions of the semantics of our example. In our naming convention a rule with name *attProd* defines the attribute *att* for the production *Prod*. The rule *sppAdd* for the attribute *spp* of the production *Add* looks for its children attributions and binds them ($e_i \leftarrow at\ ch_ae_i$) and then combines the pretty printed children $e_i \# spp$ with the string `"+"` using the pretty printing combinator (`>#<`) for horizontal (beside) composition, from the `uulib` library. The rule *ienvLet* specifies that the *ienv* value coming from the parent (**lhs** stands for “left-hand side”) is copied to the *ienv* position of the child *val*; the *ienv* attribute of the *body* is this environment extended with a pair composed of the name (*lnm*) associated with the first child and the value (the *sval* attribute) of the second child.

As defined in Chapter 4, a rule is a mapping from an input family (the inherited attributes of the parent and the synthesized attributes of the children) to a function which extends the output family (the inherited attributes of the children and the synthesized attributes of the parent) with the new elements defined by this rule.

$$\mathbf{type}\ Rule\ sc\ ip\ ic\ sp\ ic'\ sp' = Fam\ sc\ ip \rightarrow (Fam\ ic\ sp \rightarrow Fam\ ic'\ sp')$$

Figure 5.2 shows a graphic representation of a rule; each rule describes a node of a data flow graph which has an underlying tree-shaped structure induced by the abstract syntax tree at hand.

The composition of two rules is the composition of the two functions resulting from applying each of them to the input family:

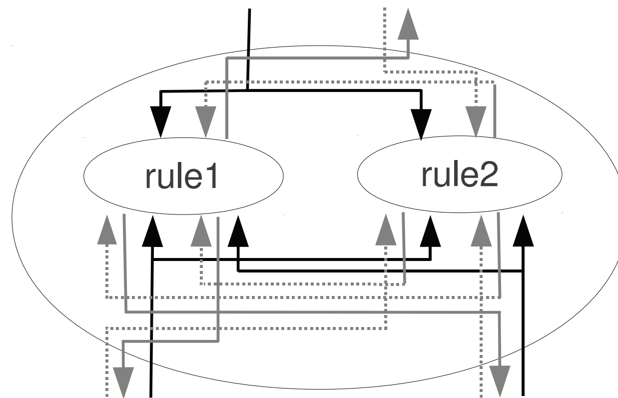

Figure 5.3: Rules Composition: produces a new rule, represented by the external oval

$$ext :: Rule\ sc\ ip\ ic'\ sp'\ ic''\ sp'' \to Rule\ sc\ ip\ ic\ sp\ ic'\ sp' \to Rule\ sc\ ip\ ic\ sp\ ic''\ sp''$$
$$(rule1\ `ext`\ rule2)\ input = rule1\ input.rule2\ input$$

Figure 5.3 represents a composition *rule1* ‘*ext*‘ *rule2*, of rules with two children. By inspecting the labyrinths of this figure, it can be seen how the inputs (black arrows) are shared and the outputs are combined by using the outputs of *rule2* (solid gray) as output constructed thus far of *rule1* (dotted gray). Thus, the outputs constructed thus far (dotted gray) of the composed rule are passed to *rule2* and the resulting outputs (solid gray) of the composed rule are equivalent to the resulting outputs of *rule1*. In Figure 5.4 we show for each production of the example how we combine the various aspects introduced by the attributes using the function *ext*.

The semantics we associate with an abstract syntax tree is a function which maps the inherited attributes of the root node to its synthesized attributes. So for each production that may be applied at the root node of the tree we have to construct a function that takes the semantics of its children and uses these to construct the semantics of the complete tree. We will refer to such functions as *semantic functions*. The hard work is done by the function *knit*, that “ties the knot”, combining the attribute computations (i.e. the data flow at the node) with the semantics of the children trees (describing the flow of data from their inherited to their synthesized

$$\begin{aligned}
aspRoot &= sppRoot\ `ext`\ svalRoot\ `ext`\ ienvRoot \\
aspCst &= sppCst\ `ext`\ svalCst \\
aspVar &= sppVar\ `ext`\ svalVar \\
aspMul &= sppMul\ `ext`\ svalMul\ `ext`\ ienvMul \\
aspAdd &= sppAdd\ `ext`\ svalAdd\ `ext`\ ienvAdd \\
aspLet &= sppLet\ `ext`\ svalLet\ `ext`\ ienvLet
\end{aligned}$$

Figure 5.4: Composition of the semantics

attributes) into the semantic function for the parent. The following code defines the semantic functions of the production *Add*:

$$semExpr_Add\ sae1\ sae2 = knit\ aspAdd\ \{\{\ ch_ae_1 \mathrel{.=.} sae1, ch_ae_2 \mathrel{.=.} sae2\ \}\}$$

where the function *knit* is applied to the combined attributes for the production.

The resulting semantic functions can be associated with the concrete syntax by using parser combinators [63] in an applicative style[1]:

```
pExpr   = semExpr_Let  <$ pKeyw "let" <*> pString
                       <* pKeyw "="   <*> pExpr
                       <* pKeyw "in"  <*> pExpr
        <|> semExpr_Add <$> pTerm  <* pKeyw "+" <*> pExpr  <|> pTerm
pTerm   = semExpr_Mul <$> pFactor <* pKeyw "*" <*> pTerm <|> pFactor
pFactor = semExpr_Cst  <$> pInt <|> semExpr_Var <$> pString
```

Thus far we have described a methodology to define the static semantics of a language. The goal of this chapter is to show how we can define new productions by combining existing productions, while probably updating some of the aspects. We want to express the semantics of new productions *in terms of already existing semantics* and *by adapting parts of the semantics* resulting from such a composition.

To show our approach we will extend the language of our example with some extra productions; one for defining the square of a value, one for defining the sum of the squares of two values, and one for doubling a value:

```
expr → ... | "square" expr | "pyth" expr expr | "double" expr
```

In the rest of this section we define the semantic functions *semExpr_Sq*, *semExpr_Pyth* and *semExpr_Double*, of the new productions, in a macro style, although providing specific definitions for the pretty-printing attributes. Thus, if the expressions' parser is extended with these new productions:

```
pExpr = ... <|> semExpr_Sq     <$ pKeyw "square" <*> pExpr
            <|> semExpr_Pyth   <$ pKeyw "pyth"   <*> pExpr <*> pExpr
            <|> semExpr_Double <$ pKeyw "double" <*> pExpr
```

the semantic action associated to parse, for example, `"square 2"` returns the value 2 for the attribute *sval* and `"square 2"` for *spp*.

Thus far, when extending the example language with a *square* production, we would have to define its semantics from scratch, i.e we had to define all its attributes in the same way we did for the original language. Thus, if the semantics of a language are defined by about twenty attributes[2] (to perform pretty-printing, name binding, type checking, optimizations, code generation, etc.), a definition of all these twenty

[1] The parsers can be generated using the technique introduced in Chapter 3.

[2] As is the case in the UHC Haskell compiler.

```
aspSq     = agMacro (aspMul  ,   ch_me1 ↪→ ch_se
                             <.> ch_me2 ↪→ ch_se)
aspPyth   = agMacro (aspAdd  ,   ch_ae1 ⟹ (aspSq,  ch_se ↪→ ch_pe1)
                             <.> ch_ae2 ⟹ (aspSq,  ch_se ↪→ ch_pe2))
aspDouble = agMacro (aspMul  ,   ch_me1 ⟹ (aspCst, ch_cv ⤳ 2)
                             <.> ch_me2 ↪→ ch_de)
```

Figure 5.5: Language Extension

```
sppSq      = synmodM spp $ do de ← at ch_de
                              return $ "square" >#< de # spp
aspSq'     = sppSq `ext` aspSq
sppPyth    = synmodM spp $ do e1 ← at ch_pe1
                              e2 ← at ch_pe2
                              return $ "pyth" >#< e1 # spp >#< e2 # spp
aspPyth'   = sppPyth `ext` aspPyth
sppDouble  = synmodM spp $ do de ← at ch_de
                              return $ "double" >#< de # spp
aspDouble' = sppDouble `ext` aspDouble
```

Figure 5.6: Redefiniton of the *spp* attribute

attributes has to be provided. To avoid this, we introduce *attribute grammar macros* in Figure 5.5 to define the extensions of the example.

The square of a value is the multiplication of this value by itself. Thus, the semantics of multiplication can be used as a basis, by passing to it the semantics of the only child (*ch_se*) of the square production both as ch_me_1 and ch_me_2. We do so in the definition of *aspSq* in Figure 5.5; we declare an attribute grammar macro based on the attribute computations for the production *Mul*, defined in *aspMul*, with its children (ch_me_1 and ch_me_2) mapped to the new child *ch_se*.

Attribute macros can map children to other macros, and so on. For example, in the definition of *aspPyth* (sum of the squares of ch_pe_1 and ch_pe_2) the children are mapped to macros based on the semantics of square (*aspSq*).

When defining a macro based on the semantics of a production which has literal children, these children can be mapped to literals. In the definition of *aspDouble* the child ch_me_1 of the multiplication is mapped to a constant, which is mapped to the literal 2.

In some cases we may want to introduce a specialized behavior for some specific

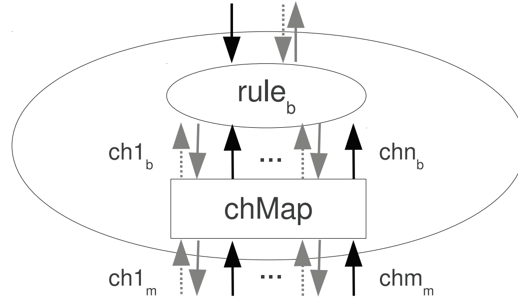

Figure 5.7: AG Macro

attributes of an aspect defined by a macro. For example, the pretty printing attribute *spp* of the macros of Figure 5.5 currently is expressed in terms of the base rule. Thus when pretty printing *square* x, instead $x * x$ will be shown. Fortunately it turns out to be very easy to overwrite the definition of some specific attribute instead of adding a new one. This is implemented by the functions *synmodM* and *inhmodM*.

In Figure 5.6 we show how the pretty printing attributes of the language extensions we defined in Figure 5.5 can be redefined to reflect their original appearance in the input program.

5.3 Attribute Grammar Macros

An attribute grammar macro is determined by a pair with the *body rule* ($rule_b$) of the macro and the mapping (*chMap*) between the children of this rule and their newly defined semantics, and returns a *macro rule*. As shown in Figure 5.7, *chMap* (rectangle) is an interface between the children of the body rule (inner oval) and the children of the macro rule (outer oval). The number of children of the macro rule (below *chMap* in the figure) does not need to be the same as the number of children of the body rule.

The function *agMacro* constructs the macro rule; it performs the "knitting" of $rule_b$, by applying this rule to its input and the output produced thus far. These elements have to be obtained from the corresponding elements of the macro rule and the mapping *chMap*. To keep the code clear, we will use the subindex b for the elements of the body rule and m for the elements of the macro rule. Thus, the macro rule takes as input the family ($Fam\ sc_m\ ip_m$) and updates the output family constructed thus far ($Fam\ ic_m\ sp_m$) to a new output family ($Fam\ ic''_m\ sp'_m$):

$$
\begin{array}{l}
agMacro\ (rule_b, chMap)\ (Fam\ sc_m\ ip_m)\ (Fam\ ic_m\ sp_m) = \\
\quad \mathbf{let}\ ip_b = ip_m \\
\qquad sp_b = sp_m \\
\qquad (Fam\ ic'_b\ sp'_b) = rule_b \quad (Fam\ sc_b\ ip_b)\ (Fam\ ic_b\ sp_b) \\
\qquad (ic'_m, ic_b, sc_b) = chMap\ (sc_m, ic_m)\ (ic'_b, emptyRecord, emptyRecord) \\
\qquad ic''_m = hRearrange\ (recordLabels\ ic_m)\ ic'_m \\
\qquad sp'_m = sp'_b \\
\quad \mathbf{in}\ \ (Fam\ ic''_m\ sp'_m)
\end{array}
$$

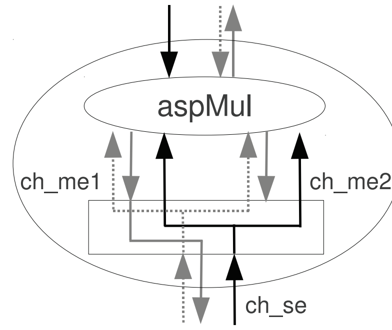

Figure 5.8: aspSq

The inherited and synthesized attributes of the parent of the body rule (ip_b and sp_b) respectively correspond to ip_m and sp_m, the inherited and synthesized attributes of the parent of the macro rule. The inherited and synthesized attributes of the children of the body rule (ic_b and sc_b), as well as the updated inherited attributes of the children of the macro rule (ic'_m), are generated by the children mapping function *chMap*. The function *chMap* takes as input a pair (sc_m, ic_m) with the synthesized attributes and the inherited attributes constructed thus far of the children of the macro rule, and returns a function that updates a triple with the updated inherited attributes (ic'_m) of the children of the macro rule and the inherited (ic_b) and synthesized (sc_b) attributes of the children of the body rule. We start with an "initial" triple composed of the updated inherited attributes of the children of the body rule (ic'_b), which has been converted into ic'_m, and two empty records (to be extended to ic_b and sc_b). Notice that the attributes we pass to *chMap* are effectively the ones indicated by the incoming arrows in Figure 5.7.

The rearranging of ic'_m is just a technical detail stemming from the use of HList; by doing this we make sure that the children in ic_m and ic'_m are in the same order, thus informing the type system that both represent the same production. The synthesized attributes of the parent of the macro rule (sp'_m) are just sp'_b, the synthesized attributes of the parent of the body rule.

Mapping functions resemble rules in the sense that they take an input and return a function that updates its "output", that in this case is the triple (ic'_m, ic_b, sc_b) instead of an output family. Thus, they can be combined in the same way as rules are combined; the combinator (`<.>`), used in Figure 5.5, is exactly the same as the *ext* function but with a different type:[3]

$$
\begin{aligned}
(\texttt{<.>}) &:: ((sc_m, ic_m) \rightarrow ((ic'1_m, ic1_b, sc1_b) \rightarrow (ic'2_m, ic2_b, sc2_b))) \\
&\rightarrow ((sc_m, ic_m) \rightarrow ((ic'0_m, ic0_b, sc0_b) \rightarrow (ic'1_m, ic1_b, sc1_b))) \\
&\rightarrow ((sc_m, ic_m) \rightarrow ((ic'0_m, ic0_b, sc0_b) \rightarrow (ic'2_m, ic2_b, sc2_b)))
\end{aligned}
$$

$$(chMap1 \; \texttt{<.>} \; chMap2) \; inp = chMap1 \; inp.chMap2 \; inp$$

[3] To avoid confusion with rule combination, instead of using apostrophes to denote updates we use numeric suffixes

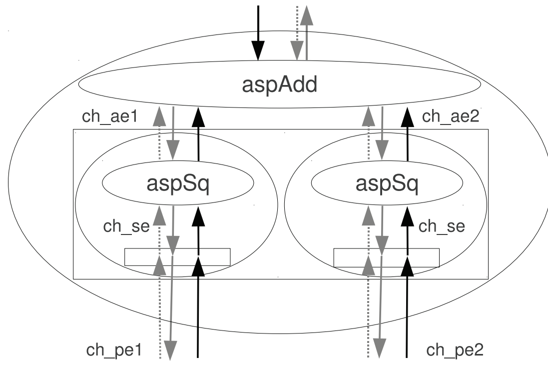

Figure 5.9: aspPyth

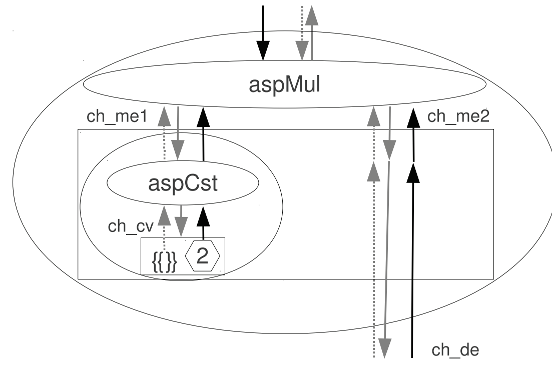

Figure 5.10: aspDouble

5.3.1 Mapping a Child to a Child

We use the combinator ($\hookrightarrow$) to map a child lch_b of the body rule to a child lch_m of the macro rule.

$$
\begin{aligned}
& lch_b \hookrightarrow lch_m = \lambda(sc_m, ic_m)\ (ic'0_m, ic0_b, sc0_b) \rightarrow \\
& \quad \textbf{let}\ ic'1_m = hRenameLabel\ lch_b\ lch_m\ (hDeleteAtLabel\ lch_m\ ic'0_m) \\
& \qquad\quad ic1_b\ \ = lch_b \mathbin{.=.} (ic_m \mathbin{\#} lch_m) \mathbin{.*.} ic0_b \\
& \qquad\quad sc1_b\ \ = lch_b \mathbin{.=.} (sc_m \mathbin{\#} lch_m) \mathbin{.*.} sc0_b \\
& \quad \textbf{in}\ \ (ic'1_m, ic1_b, sc1_b)
\end{aligned}
$$

The updated inherited attributes for the child lch_m correspond to the updated inherited attributes of the child lch_b. Thus, the new ic'_m ($ic'1_m$) is the original one with the field lch_b renamed to lch_m. Since more than a single child of the body rule can be mapped to a child of the macro rule, like in *aspSq* of Figure 5.5, we have to avoid duplicates in the record by deleting a possible previous occurrence of lch_m. This decision fixes the semantics of multiple occurrences of a child in a macro: the child will receive the inherited attributes of its left-most mapping. We represent this behavior in Figure 5.8 with the gray arrow, which corresponds to the inherited attributes of ch_me_2, pointing nowhere outside the mapping. In the cases of the initial inherited attributes and the synthesized attributes, they have to be extended with a field corresponding to the child lch_b with the attributions for the child lch_m from the inherited and synthesized attributes, respectively, of the macro rule.

5.3.2 Mapping a Child to a Macro

Inside a macro a child can be mapped to some other macro ($rule_c$, $chMap$), where the subindex c stands for child. This is the case of the definitions of *aspPyth* and *aspDouble*, graphically represented in Figure 5.9 and Figure 5.10, where the rectangles representing the children mappings have rules (ovals) inside.

$$
\begin{array}{l}
lch_b \Longrightarrow (rule_c, chMap) = \lambda(sc_m, ic_m)\ (ic'0_m, ic0_b, sc0_b) \rightarrow \\
\quad \textbf{let}\ (Fam\ ic'_c\ sp'_c) = agMacro\ (rule_c, chMap)\ (Fam\ sc_m\ (ic'0_m \# lch_b)) \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad (Fam\ ic_m\ emptyRecord) \\
\qquad ic'1_m = hLeftUnion\ ic'_c\ (hDeleteAtLabel\ lch_b\ ic'0_m) \\
\qquad ic1_b \ \ = lch_b \mathrel{.=.} emptyRecord \mathrel{.*.} ic0_b \\
\qquad sc1_b \ \ = lch_b \mathrel{.=.} sp'_c \qquad\qquad\ \mathrel{.*.} sc0_b \\
\quad \textbf{in}\ \ (ic'0_m, ic1_b, sc1_b)
\end{array}
$$

In this case, the inner macro has to be evaluated using *agMacro*. The children of the inner macro will be included in the children of the outer macro; thus the synthesized attributes of the inner macro are included in sc_m, and the new inherited attributes of the children have to extend ic_m. The inherited attributes of the parent of the inner macro are the inherited attributes of the child *lch_b* of the body rule of the outer macro. The synthesized attributes of the parent of the inner macro are initialized with an empty attribution. The child lch_b is removed from $ic'0_m$, because the macro rule will not include it. On the other hand, the inherited attributes of the children of the inner macro (ic'_c) have to be added to the inherited attributes of the children of the macro. With the function *hLeftUnion* from HList we perform an union of records, choosing the elements of the left record in case of duplication. We initialize the inherited attributes for lch_b with an empty attribution, since it cannot be seen "from the outside". The synthesized attributes are initialized with the resulting synthesized attributes of the inner rule.

5.3.3 Mapping a Child to a Constant

With the combinator ($\multimap$) we define a mapping from a child with label *lch* to a literal value *cst*. For the body rule, the initial synthesized attributes of the child lch_b are fixed to the literal *cst*.

$$
\begin{array}{l}
lch_b \multimap cst = \lambda(_, _)\ (ic'0_m, ic0_b, sc0_b) \rightarrow \\
\quad \textbf{let}\ ic'1_m = hDeleteAtLabel\ lch\ ic'0_m \\
\qquad ic1_b \ \ = lch_b \mathrel{.=.} emptyRecord \mathrel{.*.} ic0_b \\
\qquad sc1_b \ \ = lch_b \mathrel{.=.} cst \qquad\qquad\quad \mathrel{.*.} sc0_b \\
\quad \textbf{in}\ \ (ic'1_m, ic1_b, sc1_b)
\end{array}
$$

The (internal) macro associated to the mapping of the child ch_me_1 in Figure 5.10 shows the semantics of the combinator ($\multimap$). The synthesized attributes of *ch_cv* are fixed to the constant (hexagon) 2. Since the child is mapped to a constant, the inherited attributes are ignored (the arrow points nowhere). Nevertheless, we have to provide a (empty) set of inherited attributes constructed thus far to the rule *aspCst*.

5.4 Accessing Attributes

We defined a couple of combinators *withLhsAtt* and *withChildAtt* to provide access to the values of the attributes composing the input family.

$$
\begin{array}{l}
aspCond = withChildAtt\ ch_cnd\ sval\ \$\ \lambda cnd \rightarrow \\
\quad \textbf{let}\ opt1 = (aspLet \quad ,\ ch_lnm \rightsquigarrow \texttt{"is-true"} \\
\qquad\qquad\qquad\qquad \texttt{<.>}\ ch_val \hookrightarrow ch_pe_1 \\
\qquad\qquad\qquad\qquad \texttt{<.>}\ ch_body \Longrightarrow opt2) \\
\qquad\ opt2 = (aspLet \quad ,\ ch_lnm \rightsquigarrow \texttt{"is-false"} \\
\qquad\qquad\qquad\qquad \texttt{<.>}\ ch_val \hookrightarrow ch_pe_2 \\
\qquad\qquad\qquad\qquad \texttt{<.>}\ ch_body \Longrightarrow res) \\
\qquad\ res \quad = (aspVar \quad ,\ ch_vnm \rightsquigarrow \textbf{if}\ cnd \not\equiv 0\ \textbf{then}\ \texttt{"is-true"} \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \textbf{else}\ \ \texttt{"is-false"}) \\
\quad \textbf{in}\ \ withoutChild\ ch_cnd\ (agMacro\ opt1)
\end{array}
$$

Figure 5.11: Language Extension: Conditionals

The function *withLhsAtt* takes as arguments the label *att* of an inherited attribute of the parent and a function *frule* that uses the value of this attribute to return a rule.

$$
\begin{array}{l}
withLhsAtt :: HasField\ att\ ip\ v \\
\qquad\qquad \Rightarrow att \rightarrow (v \rightarrow Rule\ sc\ ip\ ic\ sp\ ic'\ sp') \\
\qquad\qquad \rightarrow Rule\ sc\ ip\ ic\ sp\ ic'\ sp' \\
withLhsAtt\ att\ frule\ (Fam\ sc\ ip)\ (Fam\ ici\ spi) \\
\quad = frule\ (ip \# att)\ (Fam\ sc\ ip)\ (Fam\ ici\ spi)
\end{array}
$$

The value of the attribute *att* is located in the record *ip* of the input family, containing the inherited attributes of the parent.

The function *withChildAtt* is similar to *withLhsAtt*, but adding to its parameters the label of child whose synthesized attribute we want to use.

$$
\begin{array}{l}
withChildAtt :: (HasField\ lch\ sc\ r, HasField\ att\ r\ v) \\
\qquad\qquad \Rightarrow lch \rightarrow att \rightarrow (v \rightarrow Rule\ sc\ ip\ ic\ sp\ ic'\ sp') \\
\qquad\qquad \rightarrow Rule\ sc\ ip\ ic\ sp\ ic'\ sp' \\
withChildAtt\ lch\ att\ frule\ (Fam\ sc\ ip)\ (Fam\ ici\ spi) \\
\quad = frule\ ((sc \# lch) \# att)\ (Fam\ sc\ ip)\ (Fam\ ici\ spi)
\end{array}
$$

Thus we first lookup the attribution corresponding to the child *lch*, and obtain the value of the attribute *att* from this attribution.

We can use these combinators to discriminate between the values of a given attribute when defining a macro. For example, suppose we want to extend the example language with a conditional expression:

$$expr \rightarrow ... \mid expr\ \texttt{"?"}\ expr\ \texttt{":"}\ expr$$

The first sub-expression is the condition and the other two are the true branch (to the left of the `":"` symbol) and the false branch (to the right).

In Figure 5.11 we show how such an extension can be implemented, using macros, in terms of variable bindings and their use. We basically bind the true sub-expression (ch_pe_1) to a variable `"is-true"`, the false sub-expression (ch_pe_2) to `"is-false"` and decide which variable to use in the body of the inner binding based on the result of the evaluation (*sval*) of the condition (*ch_cnd*). We use the combinator *withChildAtt* to obtain the value of the attribute *sval* of the child *ch_cnd*.

In this case, the macro (*agMacro opt1*) does not consider the child *ch_cnd*. The combinator *withoutChild* takes a child label *lch* and a rule *rule1* without this child and returns a rule with this child.

```
withoutChild lch rule1 (Fam sc ip) (Fam ici spi) =
  let spi1 = spi
      ip1  = ip
      sc1  = hDeleteAtLabel lch sc
      ici1 = hDeleteAtLabel lch ici
      (Fam ico1 spo1) = rule1 (Fam sc1 ip1) (Fam ici1 spi1)
      ico  = lch .=. (ici # lch) .*. ico1
      ico' = hRearrange (recordLabels ici) ico
      spo  = spo1
  in  (Fam ico' spo)
```

To invoke *rule1* we have to remove the child from the synthesized attributes (*sc*) and the inherited attributes produced thus far (*ici*). Then we *reinsert* the attributes of this child to the inherited attributes produced thus far (ico') by the resulting rule.

5.5 Attribute Redefinitions

We have shown how to introduce new syntax and how to express its meaning in terms of existing constructs. In this section we show how we can *redefine* parts of the just defined semantics by showing how to redefine attribute computations.

The function *synmod* (and its monadic version *synmodM*) modifies the definition of an existing synthesized attribute:

```
synmod :: HUpdateAtLabel att val sp sp' => att -> val -> Fam ic sp -> Fam ic sp'
synmod att val (Fam ic sp) = Fam ic (hUpdateAtLabel att val sp)
```

Note that the only difference between *syndef*, from Section 4.3, and *synmod*, is that the latter updates an existing field of the attribution *sp*, instead of adding a new field. With the use of the HList's function *hUpdateAtLabel* we enforce (by type class constraints) the record *sp*, which contains the synthesized attributes of the parent constructed thus far, indeed contains a field labeled *att*. Thus, a rule created using *synmod* has to extend, using *ext*, some other rule that has already defined the synthesized attribute this rule is *redefining*.

The `AspectAG` library also provides functions *inhmodM* and *inhmod*, analogous to *inhdefM* and *inhdef*, that modify the definition of an inherited attribute for all children coming from a specified collection of semantic categories.

A generalized version of the redefinition functions, to update an attribute value given its previous definition, is also provided. For example, a synthesized attribute can be updated with:

$$
\begin{aligned}
&\mathit{synupd} :: (\mathit{HasField}\ \mathit{att}\ \mathit{sp}\ \mathit{val}, \mathit{HUpdateAtLabel}\ \mathit{att}\ \mathit{val}'\ \mathit{sp}\ \mathit{sp}')\\
&\qquad\quad \Rightarrow \mathit{att} \rightarrow (\mathit{val} \rightarrow \mathit{val}') \rightarrow \mathit{Fam}\ \mathit{ic}\ \mathit{sp} \rightarrow \mathit{Fam}\ \mathit{ic}\ \mathit{sp}'\\
&\mathit{synupd}\ \mathit{att}\ f\ (\mathit{Fam}\ \mathit{ic}\ \mathit{sp}) = \mathit{Fam}\ \mathit{ic}\ (\mathit{hUpdateAtLabel}\ \mathit{att}\ \mathit{val}'\ \mathit{sp})\\
&\quad \textbf{where}\ \mathit{val}' = f\ (\mathit{sp}\ \#\ \mathit{att})
\end{aligned}
$$

5.6 Conclusions and Future Work

Building on top of a set of combinators that allow us to formulate extensions to semantics as first class attribute grammars (i.e. as plain typed Haskell values), we introduced in this chapter a mechanism which allows us to express semantics in terms of already existing semantics, without the need to use higher order attributes.

The programmer of the extensions does not need to know the details of the implementation of every attribute. In order to implement a macro or a redefinition for a production he only needs the names of the attributes used and the names of the children of the production, the latter being provided by the definition of the abstract syntax tree.

This work is part of a bigger plan, involving the development of a series of techniques to deal with the problems involved in both syntactic and semantic extensions of a compiler by composing compiled and type-checked Haskell values. In this way we leverage the type checking capabilities of the Haskel world into such specifications, and we profit from all the abstraction mechanisms Haskell provides.

We already think that the current approach is to be preferred over stacking more and more monads when defining a compositional semantics as is conventionally done in the Haskell world [57].

6 UUAG Meets AspectAG: How to make Attribute Grammars First-Class

The Utrecht University Attribute Grammar Compiler (`UUAGC`) takes attribute grammar declarations from *multiple source files* and generates an attribute grammar evaluator consisting of *a single Haskell source text*. The problem with such generative approaches is that, once the code is generated and compiled, neither new attributes can be introduced nor existing ones can be modified without providing access to all the source code and without having to regenerate and recompile the entire program.

In contrast to this textual approach we presented in earlier chapters the Haskell combinator library `AspectAG` with which one can construct attribute grammar fragments as a *Haskell value*. Such descriptions can be individually type-checked, compiled, distributed and composed to construct a compiler. This method however results in rather inefficient compilers, due to the extra indirection caused by this increased flexibility.

We show how to combine the two approaches by generating `AspectAG` code fragments from `UUAGC` sources, thus making it possible to trade between efficiency and flexibility, enabling a couple of optimizations for `AspectAG` resulting in a considerable speed improvement and making existing `UUAGC` code reusable in a flexible environment.

6.1 Introduction

The key advantage of using attribute grammar systems is that they allow us to describe the semantics of a programming language *in an aspect oriented way*. A complete evaluator can be assembled from a large collection of attribute grammar fragments, each describing a specific aspect of the language at hand.

Solutions to the quest for composable language description can be found at the textual level, as done by most attribute grammar systems [45, 21, 19], or at the semantic level, where language descriptions become first class values, which can be composed to build a complete language description.

The first approach is supported by, amongst many others, the Utrecht University Attribute Grammar System (`UUAGC`) [20], which reads in a complete language definition from a large collection of files, each describing a separate aspect of the language. These fragments are assembled and analyzed together, leading to a large, monolithic and eficiënt compiler, which however cannot easily be adapted once generated and compiled. In the object-oriented world we find similar weaving based approaches in e.g. Lisa [45] and Jastadd [21], which are both Java based.

At the other extreme we find the attempts to assemble the semantics from individual fragments, which, in case of Haskell, use monad transformers to stack a large collection of relatively independent computations over the syntax tree, each taking care of one of the aspects that together make up a complete compiler [32, 57]. Unfortunately, the monad-based approach comes with its own problems: one gets easily lost in the stack of monads, one is sometimes obliged to impose an order which does not really make sense, and the type system makes it hard to e.g. compose state out of a number of individual states which probably carry the same type. Furthermore, the implicit order in which attributes have to be evaluated becomes very explicit in the way the monads are composed.

In earlier chapters we have presented a completely different, non monad-based, approach to describing first-class language definition fragments; using a collection of combinators (the `AspectAG` Haskell package) it becomes possible to express attribute grammars using an Embedded Domain-Specific Language in Haskell (Chapter 4); unfortunately it is both a bit more verbose than the specific syntax as provided by the `UUAGC` system and relatively expensive. In order to provide the possibility to redefine attributes or to add new attributes elsewhere, we encode the lists of inherited and synthesized attributes of a non-terminal as an `HList`-encoded [35] value, indexed by types using the Haskell class mechanism. In this way checking the well-formedness of the attribute grammar is realized through the Haskell class system. Once the language gets complicated (in our Haskell compiler `UHC` [19] some non-terminals have over 20 attributes), the cost of accessing attributes may become noticeable. Note that, in contrast to the weaving based approaches, this approach supports *separate compilation of individual aspects*: each generated fragment is individually checked to be well-typed and once compiled its source is not modifiable by other extensions. Once it is used however it has to "dynamically link" itself to the other components, which induces an extra source of inefficiency.

In this chapter we seek to alleviate the aforementioned verbosity and inefficiency by generating `AspectAG` code from the original `UUAGC` code. We furthermore take the opportunity to group collections of attributes which are not likely to be adapted, so we can shorten the `HList` values, thus relieving the costs of the extra available expressibility. Only the attributes which are to be adapted by other language fragments have to be made part of these `HList` values at the top level; hence we only pay for the extra flexibility when needed.

In section 6.2 we describe the way the `UUAGC` represents grammars and introduce our running example, which consists of an initial language fragment and a small extension. In section 6.3 we describe how to generate `AspectAG` code out of the `UUAGC` sources and in section 6.4 we describe how we optimize the generated code.

```
LangDef.ag

DATA Root  | Root decls : Decls main : Expr
DATA Decls | Decl name : String val : Expr rest : Decls
           | NoDecl
DATA Expr  | Add al : Expr ar : Expr
           | Mul ml : Expr mr : Expr
           | Cst value : Int
           | Var var : String

ATTR Root Expr SYN sval : Int
SEM Root  | Root    lhs.sval  = main.sval
SEM Expr  | Add     lhs.sval  = al.sval + ar.sval
          | Mul     lhs.sval  = ml.sval * mr.sval
          | Cst     lhs.sval  = value
          | Var     lhs.sval  = case lookup var lhs.ienv of
                                  Just v  -> v
                                  Nothing -> 0

ATTR Decls Expr INH ienv : [(String, Int)]
SEM Root  | Root    decls.ienv = []
                    main.ienv  = decls.senv
SEM Decls | Decl    val.ienv   = []
                    rest.ienv  = (name, val.sval) : lhs.ienv

ATTR Decls SYN senv : [(String, Int)]
SEM Decls | Decl    lhs.senv = rest.senv
          | NoDecl  lhs.senv = lhs.ienv
```

Figure 6.1: AG specification of the language semantics

6.2 Attribute Grammars

6.2.1 Initial Attribute Grammars

An Attribute Grammar is a context-free grammar where the nodes in the parse tree are decorated with a (usually quite large) number of values, called *attributes*. As running example we revisit the example language of Chapter 1.

In Figure 6.1 we show the semantics in terms of `UUAGC` input. Attributes define semantics for the language in terms of the grammar and in their defining expression may refer to other attributes. A tree-walk evaluator generated from the AG computes values for these attributes, and thus provides implementations for the semantics in the form of compilers and interpreters. In our example (Figure 6.1) we use three at-

```
LangExt.ag

  ATTR Root Decls Expr SYN serr USE {++} {[]} : [String]
  SEM Decls | Decl lhs.serr = (case lookup name lhs.ienv of
                                 Just _    → [name ++ " duplicated"]
                                 Nothing → []) ++ val.serr ++ rest.serr
  SEM Expr | Var  lhs.serr = case lookup var lhs.ienv of
                                 Just _    → []
                                 Nothing → [var ++ " undefined"]
```

Figure 6.2: Semantics extended with an attribute that collects errors

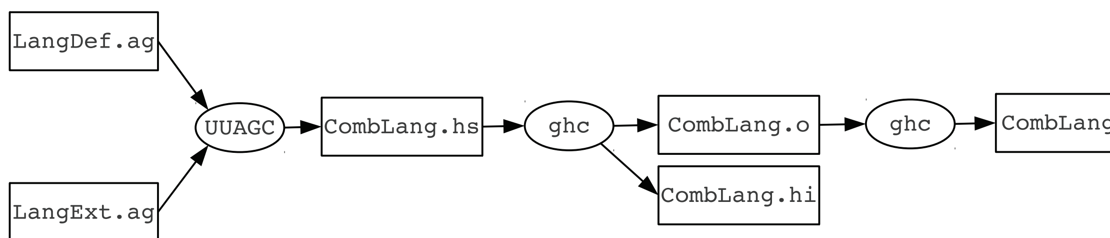

Figure 6.3: Compilation Process with UUAGC

tributes: one attribute (**SYN** *sval*) holding the result value, one attribute (**INH** *ienv*) in which we assemble the environment from the declarations (*ienv*) and one attribute (**SYN** *senv*) for passing the final environment back to the *Root* to be used in the main expression.

In the **SEM** blocks we specify how attributes are to be computed out of other attributes. These computations are expressed in terms of almost plain Haskell code, using minimal syntactic extensions to refer to the attributes.

When the `UUAGC` compiler weaves its input files into a Haskell program the rules' expressions are copied almost verbatim into the generated program: only the attribute references are replaced by references to values defined in the generated program. The `UUAGC` compiler checks whether a definition has been given for each attribute, whereas type checking of the defining expressions is left to the Haskell compiler when compiling the generated program.

6.2.2 Attribute Grammar Extensions

In this subsection we show how we can extend the given language *without touching the code written* (neither the generated nor the compiled code). In our compiler we want to generate error messages, so we introduce an extra synthesized attribute (*serr*, Figure 6.2), in which we report occurences of dual declarations (*name* is already an element of the *ienv*) and absent declarations (*name* is not an element of *ienv*).

To compile this code using `UUAGC` and `GHC` we follow the process described in Figure 6.3; i.e. use `UUAGC` to generate a completely fresh Haskell file out of the two related

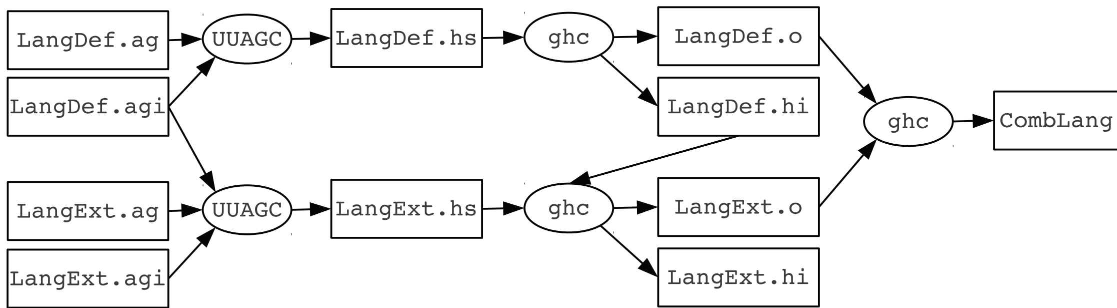

Figure 6.4: Compilation Process with our extension of UUAGC

attribute grammar sources, compile the composite result with `GHC` and link it with yet another call to `GHC`. Keep in mind that by doing so we only generate the semantic part of the compiler, which has to be completed with a few lines of main program containing the parsers from which refer to the generated semantic part.

To use `AspectAG` almost the same code has to be written, but by passing some extra flags to `UUAGC` we generate *human-readable* `AspectAG` code. This enables a completely different construction process (Figure 6.4), which makes it possible to have a compiled definition of the semantics of a core language and to introduce relative small extensions to it later, without neither the need to reconstruct the whole compiler, nor even requiring the sources of the core language to be available! Thus, for example, a core language compiler and a set of optional extensions can be distributed (without sources), such that the user can link his own extended compiler together. Such extensions could also be written in `AspectAG` directly.

To switch on this extension in `UUAGC` we pass the flag `--aspectag`:

```
uuagc -a --aspectag LangDef
uuagc -a --aspectag LangExt
```

With `--aspectag` we make `UUAGC` generate `AspectAG` code out of a set of `.ag` files and their corresponding `.agi` files, as we show in the following sections.

An `.agi` file includes the declaration of a grammar and its attributes (the interface), while the **SEM** blocks, which specify the computation of these attributes, end up in the `.ag` file (the implementation). Figure 6.5 shows the attribute grammar specification of Figure 6.1 adapted to our approach. Notice that the code is exactly the same, although distributed over a file `Langdef.agi` containing **DATA** and **ATTR** declarations, and a file `Langdef.ag` with the rules.

In Figure 6.6 we adapt the extension of Figure 6.2. In this case a new keyword **EXTENDS** is used to indicate which attribute grammar is being extended. Extensions are incremental. Thus, if we define yet another extension (Figure 6.7) which adds a new production representing negating expressions to the attribute grammar resulting from the previous extension *LangExt*, the specific rules for the attributes *sval*, *ienv* and *serr* have to be defined (in case they differ from the otherwise generated copy rules).

```
LangDef.agi

DATA Root  | Root decls : Decls main : Expr
DATA Decls | Decl name : String val : Expr rest : Decls
           | NoDecl
DATA Expr  | Add al : Expr ar : Expr
           | Mul ml : Expr mr : Expr
           | Cst value : Int
           | Var var : String

ATTR Root Expr  SYN sval  : Int
ATTR Decls Expr INH ienv : [(String, Int)]
ATTR Decls      SYN senv : [(String, Int)]
```

```
LangDef.ag

SEM Root  | Root   lhs.sval  = main.sval
SEM Expr  | Add    lhs.sval  = al.sval + ar.sval
          | Mul    lhs.sval  = ml.sval * mr.sval
          | Cst    lhs.sval  = value
          | Var    lhs.sval  = case lookup var lhs.ienv of
                                 Just v   → v
                                 Nothing → 0

SEM Root  | Root   decls.ienv  = [ ]
                   main.ienv = decls.senv
SEM Decls | Decl   val.ienv    = [ ]
                   rest.ienv   = (name, val.sval) : lhs.ienv

SEM Decls | Decl   lhs.senv = rest.senv
          | NoDecl lhs.senv = lhs.ienv
```

Figure 6.5: Language semantics

```
LangExt.agi

  EXTENDS "LangDef"
  ATTR Root Decls Expr SYN serr USE {++} {[]} : [String]
```

```
LangExt.ag

  SEM Decls | Decl lhs.serr = (case lookup name lhs.ienv of
                                 Just _    -> [name ++ " duplicated"]
                                 Nothing -> []) ++ val.serr ++ rest.serr
  SEM Expr  | Var  lhs.serr = case lookup var lhs.ienv of
                                 Just _    -> []
                                 Nothing -> [var ++ " undefined"]
```

Figure 6.6: Language Extension: Errors

```
LangExt2.agi

  EXTENDS "LangExt"
  DATA Expr | Neg expr : Expr
```

```
LangExt2.ag

  SEM Expr | Neg lhs.sval   = -expr.sval
  SEM Expr | Neg expr.ienv = lhs.ienv
  SEM Expr | Neg lhs.serr   = expr.serr
```

Figure 6.7: Language Extension: Negation

```
LangExt3.agi

  EXTENDS "LangExt2"
```

```
LangExt3.ag

  SEM Decls | Decl val.ienv := rest.senv
```

Figure 6.8: Language Extension: Attribute *ienv* redefined to allow the use of variables in declarations

An important feature of the `AspectAG` library is that, besides adding new attributes or productions, existing definitions for attributes can be overwritten. If we want to extend the example language in such a way that an expression in a declaration may refer to sibling declarations, we can do so by redefining the definition for the environment we pass to such right-hand side expressions. In Figure 6.8 we show how this can be done using := instead of =, the `UUAGC` syntactic form of *attribute redefinitions*.

6.3 From UUAG to AspectAG

The translation to `AspectAG` is quite straightforward. In the rest of this section we will show, with some examples, its most important aspects.

Grammar

Since we use extensible records, labels have to be generated to refer to the children of the productions of the grammar. For example, the child label generated out of the `.agi` file of Figure 6.7 is *ch_expr_Neg_Expr*. A label in an `HList` is represented by a plain Haskell value of a singleton type.

Attribute Definition

A collection of synthesized or inherited attributes is an extensible record, too. Thus, for each **ATTR** declaration in the `.agi` file, a label has to be generated to refer to the defined attribute. The declaration **ATTR** *Decls* **SYN** *senv* : {[(*String*, *Int*)]}, in Figure 6.6, generates the label *att_senv*.

The `AspectAG` function *syndefM* adds the definition of a synthesized attribute. It constructs a rule *Rule sc ip ic sp ic sp′*, where *sp′* is the record *sp* extended with a field representing the new attribute. We use *syndefM* to generate the code for the rules for the synthesized attributes, like:

```
SEM Decls | Decl lhs.senv = rest.senv
```

Resulting in the code:

```
senv_Decls_Decl = syndefM att_senv $ do rest ← at ch_rest_Decls_Decl
                                        return $ rest # att_senv
```

where *at ch_rest_Decls_Decl* locates the *rest* child in the record *sc* of the environment with type *Fam sc ip* using the label *ch_rest_Decls_Decl*. Having this record bound to *rest* the `HList` lookup operator # is used to locate the value of the attribute *att_senv*. The uses of such calls to *at* will inform the type system that the input family *Fam sc ip* has to have a child *ch_rest_Decls_Decl* with a defined attribute *att_senv*. Such constraints turn up as class constraints, to be checked by the Haskell type checker.

The same procedure is followed to generate code for the inherited attributes, but using the function *inhdefM*. For the declarations:

```
SEM Decls | Decl val.ienv  = []
                 rest.ienv = (name, val.sval) : lhs.ienv
```

The following code is generated:

```
ienv_Decls_Decl = inhdefM att_ienv nts_ienv $
  do
    lhs  ← at lhs
    name ← at ch_name_Decls_Decl
    val  ← at ch_val_Decls_Decl
    return {{ ch_val_Decls_Decl  .=. []
            , ch_rest_Decls_Decl .=. (name, val # att_sval) : lhs # att_ienv }}
```

The parameter *nts_ienv* is a list of labels representing the non-terminals for which the attribute *ienv* is defined (generated out of the **ATTR** declarations). The function *lhs* returns the record *ip* (inherited attributes of the parent) from the input family *Fam sc ip*. The defined computations for each child are returned in an extensible record,which is iterated by a "type-level function" (implemented by a type class called *Defs*) to extend the corresponding records in *ic*.

Generating the Semantic Functions

Thus, when generating `AspectAG` code, all the rules for the attributes of each production are composed. In the example of Figure 6.5 the following composition is generated for the production *Decl*:

```
atts_Decls_Decl = ienv_Decls_Decl `ext` senv_Decls_Decl
```

The semantic functions of the non-terminal *Decl* is:

```
sem_Decls_Decl   = knit atts_Decls_Decl
sem_Decls_NoDecl = knit atts_Decls_NoDecl
```

This code is generated out of the **DATA** declarations.

Extensions

The keyword **EXTENDS** indicates that an attribute grammar declaration *extends* an existing attribute grammar. In an extension we can both add new attributes or productions or redefine the computation of existing attributes.

When the code of an extension is generated, the names of context-free grammar and the previously defined attributes have to be imported from the code generated for the system to extend. We take this information from the (chain of) `.agi` file(s) of the extended module.

We also generate a qualified import of the whole module, so we can refer to already defined rules without name clashes:

```
import qualified LangDef
```

So, when introducing new attributes, we can perform the composition for each production where the attribute is defined, and knit it again. For example:

```
atts_Decls_Decl = serr_Decls_Decl `ext` LangDef.atts_Decls_Decl
sem_Decls_Decl = knit atts_Decls_Decl
```

When an attribute is overwritten using :=, a similar approach as when defining new attributes is taken. Instead of using *syndefM* and *inhdefM* to define attributes, the functions *synmodM* and *inhmodM* are used, which are almost identical to their respective *def* functions, with the difference that instead of extending a record with a new attribute, the value of an existing attribute is updated in the record.

6.4 Optimizations

The flexibility provided by the use of list-like structures to represent collections of attributes (and children) of productions has its consequences in terms of performance. In this section we propose a couple of optimizations, based on changing some of the extensible records we use by normal records (Cartesian products). Both optimizations can be performed automatically by the transformation.

6.4.1 Grouping Attributes

If some attributes are fixed and will not be redefined, the use of extensible records is not necessary: in those cases we can group such a collection of synthesized attributes into a single attribute *att_syn* and such a collection of inherited attributes into an attribute *att_inh*. The type of a grouping attribute is a (non extensible) record containing the grouped attributes.

Attributes defined in extensions cannot be grouped with the original attributes. Thus, in our running example applying grouping does not make much sense, since every group will have only one attribute. But if the specifications in Figures 6.5 and 6.6 were joined in the generation process we will have the attributes *att_inh* and *att_syn* for *Decls* with types:

```
data Inh_Decls = Inh_Decls {ienv_Inh_Decls :: [(String, Int)]}
data Syn_Decls = Syn_Decls {senv_Syn_Decls :: [(String, Int)]
                           , serr_Syn_Decls :: [String]}
```

To define and access the grouped attributes, one more level of indirection is added. Thus, the definition of *att_syn* for the production *Decl* is:

```
syn_Decls_Decl = syndefM att_syn $
  do rest ← at ch_rest_Decls_Decl
     return Syn_Decls {senv_Syn_Decls = (senv_Syn_Decls (rest # att_syn))}
```

By default, all the attributes of every production are grouped, but grouped attributes cannot be redefined without having to make changes to the entire group. The flag `--nogroup` lets us specify the list of attributes we do not want to be included in the grouping. For example, the following call to `uuagc` generates the `AspectAG` code for the example with all the attributes grouped except *ienv*, which will be redefined in the extensions.

```
uuagc -a --aspectag --nogroup=ienv LangDef.ag
```

6.4.2 Static Productions

If we do not need the possibility to change the definition of already existing productions (note that our flexible approach did not forbid this thus far), a less flexible approach to represent productions can also be taken. The flag `--static` activates an optimization where the collection of child attributions are represented as records instead of extensible records. Thus, instead of defining the labels for the children of the productions, we define for each production a record with the children as fields. For example:

```
data Ch_Decls_Decl _name _val _rest
  = Ch_Decls_Decl {ch_name :: _name, ch_val :: _val, ch_rest :: _rest}
```

In this case, the generic *knit* function, which uses the type class mechanism to iterate over an HList, cannot be used anymore and thus specific *knit* functions are generated for such productions:

```
knit_Decls_Decl rule fc ip = sp
  where ec = Ch_Decls_Decl {{ }} {{ }} {{ }}
        (Fam ic sp) = rule (Fam sc ip) (Fam ec {{ }})
        sc = Ch_Decls_Decl ((ch_name fc) (ch_name ic))
                           ((ch_val  fc) (ch_val  ic))
                           ((ch_rest fc) (ch_rest ic))
```

Also the semantic functions are a bit different:

```
sem_Decls_Decl sn sv sr
  = knit_Decls_Decl atts_Decls_Decl (Ch_Decls_Decl sn sv sr)
```

We cannot use the generic type-level function *Defs* to define inherited attributes. We must define a specific instance of *Defs* for each production:

```
instance (HExtend (LVPair att v2) ic2 ic'2
         ,HExtend (LVPair att v3) ic3 ic'3)
```

```
  ⇒ Defs att nts (Ch_Decls_Decl v1               v2   v3  )
                 (Ch_Decls_Decl (Record HNil) ic2  ic3  )
                 (Ch_Decls_Decl (Record HNil) ic′2 ic′3) where
  defs att nts vals ic = Ch_Decls_Decl (ch_name ic)
                                       (att .=. ch_val  vals .*. ch_val  ic)
                                       (att .=. ch_rest vals .*. ch_rest ic)
```

and adapt the rule definitions to the use of records. For example:

```
ienv_Decls_Decl = inhdefM att_ienv nts_ienv $
  do lhs  ← at lhs
     name ← at ch_name_Decls_Decl
     val  ← at ch_val_Decls_Decl
     return Ch_Decls_Decl
              { ch_val_Decls_Decl  = []
              , ch_rest_Decls_Decl = (name, val # att_sval) : lhs # att_ienv }
```

6.4.3 Benchmarks

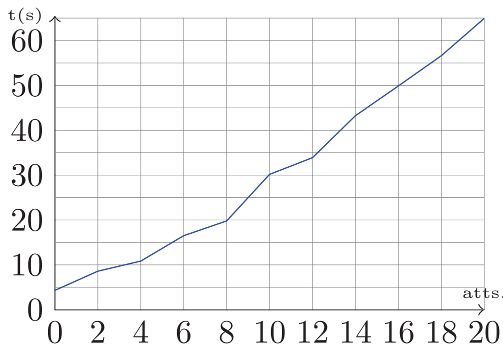

Figure 6.9: Grouping Syn. Attrs.

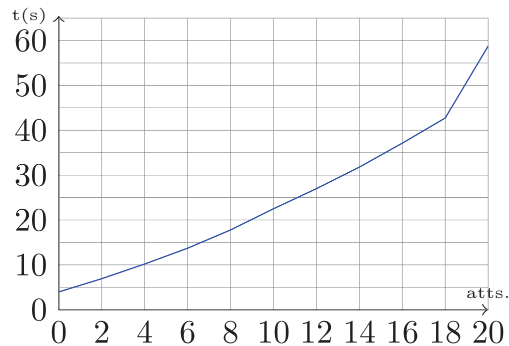

Figure 6.10: Grouping Inh. Attrs.

We benchmarked our optimizations against `AspectAG` and `UUAGC`, in order to analyze their performance impact.[1] Figures 6.9 and 6.10 show the effect of grouping attributes in a grammar represented by a binary tree. Note that these represent worst-case scenarios, since we hardly perform any work in the rules themselves. The y-axis represents the execution time (in seconds) and the x-axis the number of ungrouped attributes (the rest are grouped) in a full tree with 15 levels. In Figure 6.9 we show the results for a system with twenty synthesized attributes. Figure 6.10 shows the results for twenty inherited attributes and one synthesized attribute to collect them. In both cases the effect of grouping attributes becomes clear; for a relative large number of attributes the grouping optimization achieves good speedup, since

[1] Information available at: `http://www.cs.uu.nl/wiki/bin/view/Center/Benchmarks`

whenever an attribute is needed it is located in constant time instead of linear time in the number of attributes.

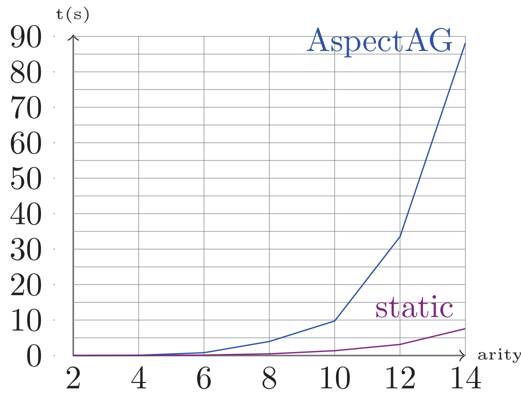

Figure 6.11: Static: Syn.

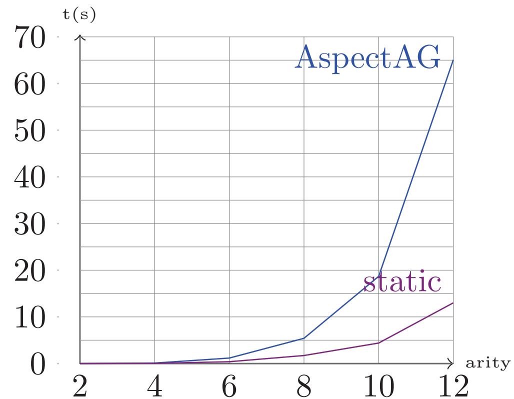

Figure 6.12: Static: Syn. and Inh.

In figures 6.11 and 6.12 we show the performance impact of the "static productions" optimization, as the number of children of the nodes increases. We tested with complete trees with depth 5; the x-axis represents the arity of the tree. Figure 6.11 shows the results for one synthesized attribute, while the results of Figure 6.12 include one synthesized and one inherited attribute. Thus, the optimization helps, and has a big impact on productions with many children, because we are avoiding iterations over lists of children when evaluating the semantics for each node. In figures 6.13 and

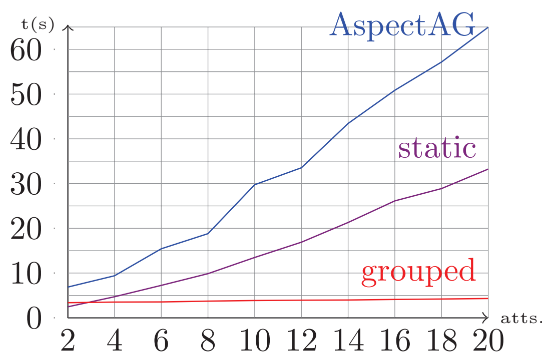

Figure 6.13: Static vs Grouped: Syn.

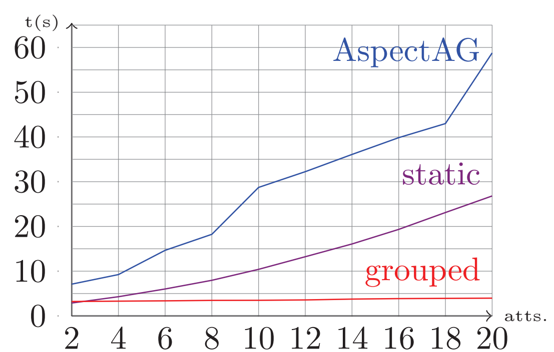

Figure 6.14: Static vs Grouped: Inh.

6.14 we compared the performance of both optimizations and `AspectAG` in a simple grammar represented by a binary tree. In this case the x-axis represents the number of (synthesized or inherited) attributes. As the number of attributes increases, the grouping optimization has a bigger performance impact. If we apply both optimizations together (figures 6.15 and 6.16) we obtain better times, although we are still quite far from the performance of `UUAGC`.

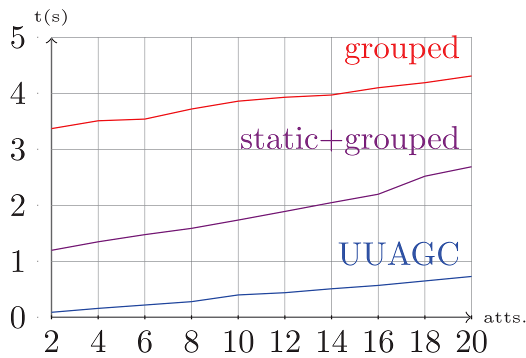

Figure 6.15: Static + Grouped: Syn.

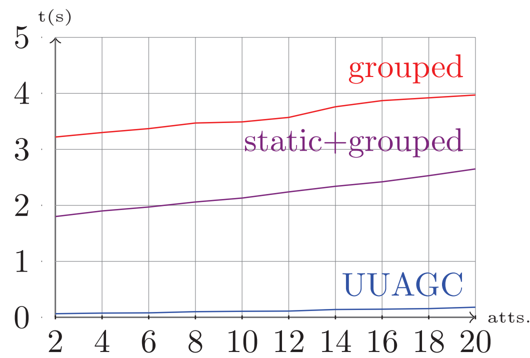

Figure 6.16: Static + Grouped: Inh.

6.5 Conclusions and Future Work

We have shown an approach to write `AspectAG` code, i.e. a strongly-typed flexible attribute system, in a less verbose Domain-Specific Language style.

We provide a framework for the generation of flexible compilers by taking a hybrid approach to their architecture. The core part is composed by a single monolithic part, which is evaluated efficiently, and a set of *redefinable* aspects. Extensions (and redefinitions) can be plugged into this core, albeit at a certain cost. The syntax macros-like mechanism we proposed in Chapter 5 follows this idea. The semantics of newly introduced syntax is defined in terms of already existing semantics. Some existing aspects, e.g. pretty-printing, may be redefined to provide accurate feedback.

Summarizing, it can be seen that flexibility still has its cost, but the application of the optimizations is a good option as the number of attributes and/or children of the productions increases. One should keep in mind that the actual computations done in our examples in the rule functions is trivial. Hence in a real compiler, where most of the work is actually done in the rules, the overhead coming with the extra flexibility will usually be far less of a burden. The numbers we have presented relate to a worst case situation.

Possible future work is to add a new optimization, consisting in the generation of a less type-safe code than AspectAG. This involves the addition of a Haskell type-checking phase to UUAGC. Furthermore, a drawback of that approach is that it ties us to the use of `UUAGC`, not allowing the introduction of extensions written directly in the target language (e.g. `AspectAG`).

7 Case Study - Oberon0

As a case study of the techinques proposed in this thesis, we participated in the LDTA 2011 Tool Challenge[1]. The challenge was to implement a compiler for Oberon0, a small (Pascal-like) imperative language designed by Nicolas Wirth as an example language for his book “Compiler Construction” [76].

The goal of the challenge is to contribute to “a better understanding, among tool developers and tool users, of relative strengths and weaknesses of different language processing tools, techniques, and formalisms”. The challenge is divided into a set of incremental sub-problems, that can be seen as points in a two dimensional space. The first dimension (Table 7.1) defines a series of language levels, each building on the previous one by adding some new features. The second dimension (Table 7.2) consists

L1	Oberon0 without procedures and with only primitive types.
L2	Add a Pascal-style for-loop and a Pascal-style case statement.
L3	Add Oberon0 Procedures.
L4	Add Oberon0 Arrays and Records.

Table 7.1: Language Levels.

of several traditional language processing tasks, such as parsing, pretty-printing, static analysis, optimizations and code generation.

T1	Parsing and Pretty-Printing
T2	Name Binding
T3	Type Checking
T4	Desugaring
T5	C Code Generation

Table 7.2: Processing Tasks.

This incremental design has two main reasons. First, participants were able to provide partial solutions, choosing the most suitable tasks to show the characteristics and features of their tool or technique. The possible software artifacts generated to solve any of the 25 proposed problems range between a L1 T1, a parser and pretty-printer of a simple subset of Oberon0, and L4 T1-5, the full proposed system. In order to be able to compare participant’s artifacts, a list of suggested software artifacts to be completed (Table 7.3) is provided. The second reason of the design is to show how

[1] `http://ldta.info/tool.html`

Artifact	*Level*	*Tasks*	*Description*
A1	L2	T1-2	Core language with pretty-printing and name binding
A2a	L3	T1-2	A1 plus pretty-printing and name binding for procedures
A2b	L2	T1-3	A1 plus type checking
A3	L3	T1-3	A2a and A2b
A4	L4	T1-5	Full language and all tasks

Table 7.3: Artifacts.

the different techniques for modularity provided by the participants can be used in the implementation of a growing system.

We have provided an implemention of all the proposed problems, and made it available in Hackage as the `oberon0`[2] package.

7.1 Architecture

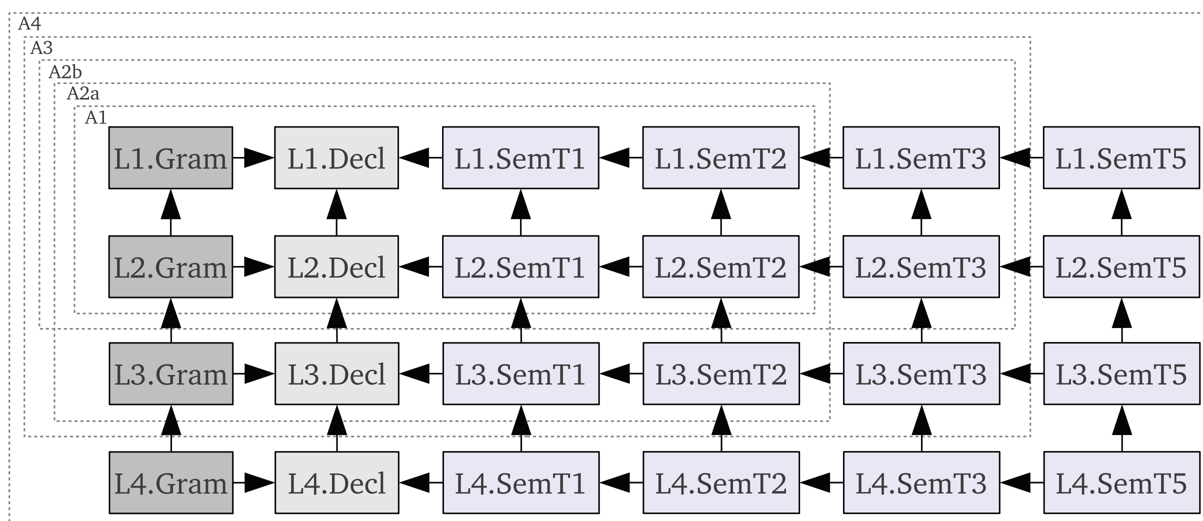

Figure 7.1: Architecture of the Oberon0 Implementation

The architecture of our implementation of Oberon0 is given in Figure 7.1; boxes represent Haskell modules and arrows are **import** relations[3], where every module can be compiled separately and results in a set of normal Haskell value definitions. The design is incremental: rows corresponds to syntactic extensions (language levels) and

[2] `http://hackage.haskell.org/package/oberon0`.

[3] For example, module *L2.SemT1* imports from (i.e. depends on) modules *L2.Decl* and *L1.SemT1*.

columns corresponds to semantic extensions (tasks); each artifact in the challenge corresponds to a dashed box surrounding the modules involved in it. For each language level *L1* to *L4*:

- *Gram* modules contain syntax definition in the form of first-class grammar fragments, as introduced in Chapter 3

- *Decl* modules contain the definition of the type of the semantics' record, and thus the interface to the corresponding part of the abstract syntax of the language at hand

- *Sem* modules implement the semantics of each task in the form of rules which together construct an attribute grammar

Notice that we do not include modules to implement Task 4. In Subsection 7.3.3 we will explain how by using attribute grammar macros when defining *L2* we get this task almost for free.

To build a compiler, e.g. Artifact 4 (Figure 7.2), we import the syntax fragments (*l1*, *l2*, *l3* and *l4* from *L4.Gram*) and their respective semantics (*l1t4*, *l2t4*, *l3t4* and *l4t4* from *L4.Sem*), combine them and build the compiler in the form of a parser which calls semantic functions. In Figure 7.3 we show how the parser of Artifact 4

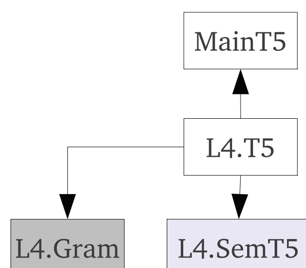

Figure 7.2: Architecture of Artifact 4

```
gl4t5 = closeGram $ emptyGram +>>
                    l1 l1t5   +>>
                    l2 l2t5   +>>
                    l3 l3t5   +>>
                    l4 l4t5
pA4 = (parse . generate kws) gl4t5
```

Figure 7.3: A Parser for Artifact 4

is generated. The left-associative operator (+>>) composes an initial grammar with an extension; we start with an empty grammar (*emptyGram*) and extend it with the different language fragments. The function *closeGram* closes the constructed grammar and applies the *left-corner transform* in order to remove potential left-recursion; as a consequence straightforward combinator-based top-down parsing techniques can be used in building the parser. Then *generate kws* generates a parser integrated with the semantics for the language starting from the first non-terminal, where the list *kws* is a list of keywords extracted from the grammar description. This takes care of the problem caused by the fact that some identifiers in earlier challenges may become keywords in later challenges. The function *parse* performs the parse of the input program and computes the meaning of that program. In the actual implementation of Oberon0 we generate scanner-less `uu-parsinglib` parsers.

7.2 Syntax

Using our combinator library `murder` we describe the concrete syntax of each language fragment as a Haskell value. A fragment of the *code constructing the CFG* of the initial language L1 (module *L1.Gram*) is given in Figure 7.4; the complete definition of the concrete grammar of the four languages can be found in Appendix A.1. The parameter

```
l1 sf = proc _ → do
  rec
    modul  ← addNT ≺ ...
    ...
    ss     ← addNT ≺ ⟦ (pSeqStmt sf)
                         stmt
                         (pFoldr (pSeqStmt sf, pEmptyStmt sf)
                                 (⟦ ";" stmt ⟧)) ⟧
    stmt   ← addNT ≺ ⟦ (pAssigStmt  sf) ident ":=" exp ⟧
                  <|> ⟦ (pIfStmt     sf)
                         "IF" cond
                         (pFoldr (pCondStmtL_Cons sf, pCondStmtL_Nil sf)
                                 (⟦ "ELSIF" cond ⟧))
                         mbelse
                         "END" ⟧
                  <|> ⟦ (pWhileStmt  sf) "WHILE" exp "DO" ss "END" ⟧
                  <|> ⟦ (pEmptyStmt sf) ⟧
    cond   ← addNT ≺ ⟦ (pCondStmt sf) exp "THEN" ss ⟧
    mbelse ← addNT ≺ pMaybe (pMaybeElseStmt_Nothing sf
                            , pMaybeElseStmt_Just sf)
                            (⟦ "ELSE" ss ⟧)
    exp    ← addNT ≺ ...
    ...
  exportNTs ≺ exportList modul $ export cs_Expression    exp
                               . export cs_StmtSeq       ss
                               . export cs_Statement     stmt
                               . export cs_MaybeElseStmt mbelse
                               . ...
```

Figure 7.4: Fragment of the concrete syntax specification of L1

sf contains the "semantics of the language"; its type is defined in the module *L1.Decl* and is derived from the abstract syntax of which we show a fragment in Figure 7.5. The full abstract syntax of the four languages can be found in Appendix A.2. We use

```
data Statement  = AssigStmt { id_AssigStmt   :: String
                            , exp_AssigStmt  :: Expression }
                | IfStmt    { if_IfStmt      :: CondStmt
                            , elsif_IfStmt   :: CondStmtL
                            , else_IfStmt    :: MaybeElseStmt }
                | WhileStmt { exp_WhileStmt  :: Expression
                            , ss_WhileStmt   :: Statement }
                | SeqStmt   { s1_SeqStmt     :: Statement
                            , s2_SeqStmt     :: Statement }
                | EmptyStmt
type CondStmtL = [CondStmt]
data CondStmt   = CondStmt { exp_CondStmt :: Expression
                           , ss_CondStmt  :: Statement }
type MaybeElseStmt = Maybe Statement
data Expression = ...
```

Figure 7.5: AS of the statements of L1

the Template Haskell function *deriveLang*[4] to derive the type of the record, given the list of data types together composing the abstract syntax tree. For example, for the example fragment we call:

```
$ (deriveLang "L1" [''Module, ''Statement, ''Expression
                   , ''CondStmtL, ''CondStmt, ''MaybeElseStmt])
```

For each production of the abstract syntax tree a field is produced, with name the name of the production prefixed by a *p* and as type the type of the semantic function, which is defined in terms of the semantics associated with the children of the production. For example, the field generated for the production *AssigStmt* is:

```
pAssigStmt :: sf_id_AssigStmt → sf_exp_AssigStmt → sf_AssigStmt
```

For the cases of *List* or *Maybe* type aliases, fields are produced using the name of the non-terminal (i.e. the type) to disambiguate. In our example, for *CondStmtL* we generate the fields *pCondStmtL_Cons* and *pCondStmtL_Nil*, and for *MaybeElseStmt* we generate *pMaybeElseStmt_Just* and *pMaybeElseStmt_Nothing*.

The code of Figure 7.4 defines the context free grammar of the language fragment, using the record *sf* to add semantics to it. We use the `murder` combinators *pFoldr* and *pMaybe* to model repetition and option, respectively. These combinators are analogous to the respective *foldr* and *maybe* functions.

[4] Provided by the package `AspectAG`.

Grammars defined in this way are *extensible*, since further transformations may be applied to the grammar under construction in other modules. Each grammar exports (with *exportNTs*) its starting point (e.g. *modul*) and a table of *exported non-terminals*, each consisting of a label (by convention of the form *cs_...*) and a reference to the current definition of that non-terminal, again a plain Haskell value which can be used and modified in future extensions. Figure 7.6 contains a fragment of the definition of L2 (from module *L2.Gram*), which extends the L1 grammar with a **FOR**-loop statement. We start by retrieving references to all non-terminals which

```
l2 sf = proc imported → do
  let ss    = getNT cs_StmtSeq    imported
  let stmt  = getNT cs_Statement  imported
  let exp   = getNT cs_Expression imported
  let ident = getNT cs_Ident      imported
  ...
  rec
    addProds ≺ (stmt    , ⟦ (pForStmt sf) "FOR" ident ":=" exp dir exp mbexp
                                          "DO" ss "END" ⟧)
    dir      ← addNT ≺ ⟦ (pTo sf) "TO" ⟧ <|> ⟦ (pDownto sf) "DOWNTO" ⟧
    mbexp    ← addNT ≺ pMaybe (pCst1Exp sf, id) (⟦ "BY" exp ⟧)
    ...
  exportNTs ≺ imported
```

Figure 7.6: Fragment of the grammar extension L2

are to be extended or used (using *getNT*) from the *imported* non-terminals. We add new productions to existing non-terminals with *addProds*; this does not lead to references to new non-terminals. New non-terminals can still be introduced as well using *addNT*. The Haskell type-system ensures that the *imported* list indeed contains a table with entries *cs_StmtSeq*, *cs_Statement*, *cs_Expression* and *cs_Ident*, and that the types of these non-terminals coincide with their use in the semantic functions of the extensions.

The definition in Figure 7.6 may look a bit verbose, caused by the interface having been made explicit. Using some Template Haskell this can easily be overcome.

Figure 7.7 shows the abstract syntax tree fragment corresponding to the **FOR**-loop extension. The prefix *EXT_* indicates that this definition is extending a given non-terminal.

```
data EXT_Statement
   = ForStmt {id_ForStmt :: String, start_ForStmt :: Expression
             , dir_ForStmt :: ForDir, stop_ForStmt :: Expression
             , step_ForStmt :: Expression, ss_ForStmt :: Statement}
   | ...
data ForDir = To | Downto
data EXT_Expression = Cst1Exp
...
```

Figure 7.7: AST of the **FOR**-loop of L2

7.3 Aspect Oriented Semantics

The semantics of Oberon0 were implemented using the `AspectAG` embedding of attribute grammars in Haskell. In order to be able to redefine attributes or to add new attributes later, it encodes the lists of inherited and synthesized attributes of a non-terminal as an `HList`-encoded [35] value; each attribute is associated with a unique type which is used as an index in such a "list". The lookup process is performed by the Haskell class mechanism. In this way the *closure test* of the attribute grammar (each attribute has a single definition) is implicitly realised by the Haskell compiler when trying to build the right instances of the classes. Thus, attribute grammar fragments can be individually type-checked, compiled, distributed and composed to construct a compiler.

7.3.1 Name analysis

Error messages produced by the name analysis are collected in a synthesized attribute called *serr*.The default behaviour of this attribute for most of the productions is to combine (append) the errors produced by the children of the production. This behaviour is captured by the function *use* from the `AspectAG` library, which takes as arguments the label of the attribute to be defined (*serr*), the Haskell list of non-terminals (labels) for which the attribute is defined (*serrNTs*), an operator for combining the attribute values (++), and a unit value to be used when none of the children has such an attribute ([] :: *String*).

```
serrRule = use serr serrNTs (++) ([] :: [String])
```

When a new name is defined we check for multiple declarations and at name uses we check for incorrect uses or uses of undefined identifiers, producing error messages when appropriate. The code below shows the definition of *serr* for the use of an

identifier represented by a production *IdExp*, which has a child named *ch_id_IdExp* of type (*DTerm String*)[5].

```
serrIdExp = syn serr $ do
   lhs ← at lhs
   nm ← at ch_id_IdExp
   return $ checkName nm (lhs # ienv) ["Var", "Cst"] "an expression"
```

With the (plain Haskell) function *checkName* we lookup the name (*nm*) in the symbol table (inherited attribute *ienv* coming from the left-hand side) and, if it is defined, we verify that the name represents either a variable ("Var") or a constant ("Cst") and generate a proper error message if not.

The symbol table is implemented by the pair of attributes *senv* and *ienv*. The synthesized attribute *senv* collects the information from the name declarations and the inherited attribute *ienv* distributes this information through the tree.

In order to perform the name analysis, the type of the symbol table could have been *Map String NameDef*, which is a map from names to values of type *NameDef* representing information about the bound name. However, since we want to use the same symbol table for future extensions, we keep the type "non-closed" by using a list-like structure:

```
data SymbolInfo b a = SI b a
type NMap a = Map String (SymbolInfo NameDef a)
```

For the current task the symbol table includes values of type *NMap a*, parametric in *a*, the "the rest of the information we might want to store for this symbol". In the example below, for declarations of constants, the table consists of a map from the introduced name to a *SymbolInfo* which includes the information needed by the name analysis (constructed using *cstDef*) and some other (yet unknown) information, which is represented by the argument the rule receives:

```
senvCstDecl r = syn senv $ do
   nm ← at ch_id_CstDecl
   return $ Map.singleton (value nm) (SI (cstDef $ pos nm) r)
```

Similarly to how we used *use* for the default cases of synthesized attributes, we capture the behaviour of distributing an inherited attribute to the children of a production with the function *copy*:

```
ienvRule _ = copy ienv ienvNTs
```

The various aspects introduced by the attributes are combined using the function *ext*:

[5] *DTerm a* is the type used by murder to represent *attributed terminals* (i.e. identifiers, values); it encodes the value (*value*) and position in the source code (*pos*) of the terminal.

$$\mathit{aspCstDecl}\ r = \mathit{senvCstDecl}\ r\ \text{`}\mathit{ext}\text{`}\ \mathit{ienvCstDecl}\ r\ \text{`}\mathit{ext}\text{`}\ \mathit{serrCstDecl}\ \text{`}\mathit{ext}\text{`}\ \mathit{T1.aspCstDecl}$$

In this case, for the production *CstDecl*, we extend *T1.aspCstDecl*, which is imported from *L1.SemT1* and includes the pretty-printing attribute, with the attributes implementing the name analysis task (*serr*, *ienv* and *senv*).

Once the attributes definitions are composed, the semantic functions for the productions may be computed using the function *knit*. For example, the semantic function of the production *CstDecl* in the case of *L1.SemT2* is *knit* (*aspCstDecl* ()). The use of () (unit) here is just to "close the symbol table", since no further information needs to be recorded for Task 2.

7.3.2 Type checking

Type error messages are collected in the synthesized attribute *sterr*. For type checking we extend the symbol table with the type information (*TInfo*) of the declared names. This is done by *updating* the value of the attribute *senv* with the function *synupdM*, which is similar to *syn* but redefines it making use of its current definition. In the following example we update the symbol table information for the production *VarDecl*, where *sty* is an attribute defined for expressions and types, computing their type information:

```
senvVarDecl' r = synupdM senv $ do
    typ ← at ch_typ_VarDecl
    return $ Map.map (λ(SI nd _) → (SI nd $ SI (typ # sty) r))
```

The previous definition of the type information is just ignored and only used to indicate the type of the symbol table. Thus, thanks to lazy evaluation, when extending the aspects of Task 2 we only need to pass an undefined value of type *SymbolInfo TInfo a*, where *a* is the type of even further information to be stored in the symbol table (for future extensions):

```
undTInfo :: a → SymbolInfo TInfo a
undTInfo = const ⊥
aspVarDecl r = (senvVarDecl' r) `ext` sterrRule `ext`
               (T2.aspVarDecl $ undTInfo r)
```

To represent type information we have to deal again with the lack of open data types in Haskell, since we want to keep some specific information for each of the types of the extensible type system we are implementing, and we have decided to resort to the use of Haskell's *Dynamic* type. A *TInfo*, with the information of a certain type, consists of: the representation *trep* of the given type, encapsulated as a *Dynamic* value, a *String* with its pretty-printing (*tshow*), and a function *teq* that, given another type information indicates if the actual type is compatible with the given one.

```
data TInfo = TInfo { trep  :: Dynamic
                   , tshow :: String
                   , teq   :: (TInfo → Bool) }
```

The main task we perform during type checking is to verify whether the actual type of an expression is compatible with the type expected by its context. For example if the condition of an **IF** statement has type `BOOLEAN`.

```
check pos expected got
  = if (teq expected got) ∨ (teq got unkTy) ∨ (teq expected unkTy)
    then []
    else [show pos ++ ": Type error. Expected " ++ tshow expected ++
          ", but got " ++ tshow got]
```

If either the expected or the obtained type is unknown (*unkTy*) we do not report a type error, because unknown types are generated by errors that have been already detected by the name analysis process.

A very simple case of type information is the elementary type `BOOLEAN`, where we do not provide any extra information than the type itself. Thus, the type representation is implemented with a singleton type *BoolType*.

```
data BoolType = BoolType
boolTy = let d   = toDyn BoolType
             bEq = (≡) (dynTypeRep d) . dynTypeRep . trep . baseType
         in  TInfo d "BOOLEAN" bEq
```

To construct the corresponding *TInfo* we convert a *BoolType* value into a *Dynamic* with the function *toDyn*. A type is compatible with `BOOLEAN` if its base type[6] is also `BOOLEAN`, i.e. is compatible if both types are represented with *BoolType* values. With the function *dynTypeRep* we extract a concrete representation of the type of the value inside a *Dynamic* that provides support for equality.

There exist some other cases were a more involved type representation is needed. For example, in the case of **ARRAY** we include the type information of its elements and the length of the array, if it can be statically computed.

```
data ArrType = ArrType (Maybe Int) TInfo
```

Then, by using the type-safe cast function *fromDynamic* we can get access to this information provided the dynamic typed value represents an array. Thus, when trying to index a variable, we can for example check if the index is out of range; in case the cast does not succeed we indicate that the variable we are trying to access is not an array:

```
checkSelArray pos ty ind
  = case (fromDynamic . trep . baseType) ty of
```

[6] In case of a user type, the type it denotes.

```
    Just (ArrType l _) → checkIndex pos ind l
    _                  → [show pos ++ ": Accessed variable is not an array"]
```

We use the same technique to keep information about the fields of a **RECORD** and the parameters of a **PROCEDURE**.

7.3.3 Source-to-source transformation

In Chapter 5 we extended `AspectAG` with an *agMacro* combinator that enables us to define the attribute computations of a new production in terms of the attribute computations of existing productions. We defined the semantics of the extensions of the language level L2 using this macro mechanism. The **FOR**-loop is implemented as a **WHILE**-loop and the **CASE** statement is defined in terms of an **IF**-**ELSIF**-**ELSE** cascade.

Figure 7.8 contains the macro definition for the **FOR**-loop, which is parametrized by the attributes (semantics) of:

- *SeqStmt*: sequence of statements
- *AssigStmt*: assign statement
- *IntCmpExp*: integer comparison expression
- *IdExp*: identifier expression
- *IntBOpExp*: integer binary operation expression

We use the combinator *withChildAtt* to obtain the value of the *self* attribute of the child *ch_dir_ForStmt*, with the direction of the iteration. In case the value is *To* the loop counter is incremented (*Plus*) on each step while is less or equal (*LECmp*) the stop value. In other case (*Downto*) we use *Minus* to decrement the counter and *GECmp* (greater or equal) to compare it it with the stop value. In Figure 7.9 we show the structure of the macro (i.e. the **FOR**-loop in terms of the original AST) for the *To* case. That can be seen as a code translation from:

```
FOR id := start TO stop BY step DO
  ss
END
```

to:

```
id := start;
WHILE id <= stop DO
  ss;
  id := id + step
END
```

In the cases were specialized behaviour is needed, like for example pretty-printing, it is still possible to redefine the attributes involved on these aspects. As such, our mechanism is much more expressive than conventional macro mechanisms, which only perform a structure transformation. Using the library we get Task 4 almost for free.

```
macroForStmt aspSeqStmt aspAssigStmt aspWhileStmt
             aspIntCmpExp aspIdExp aspIntBOpExp
 = withChildAtt ch_dir_ForStmt self $ λdir →
let (op_stop, op_step) = case dir of
                           To     → (LECmp, Plus)
                           Downto → (GECmp, Minus)
    initStmt  = (aspAssigStmt  ,  ch_id_AssigStmt   ↪ ch_id_ForStmt
                               <.> ch_exp_AssigStmt  ↪ ch_start_ForStmt)
    whileStmt = (aspWhileStmt  ,  ch_exp_WhileStmt  ⟹ condWhile
                               <.> ch_ss_WhileStmt   ⟹ bodyWhile)
    condWhile = (aspIntCmpExp  ,  ch_op_IntCmpExp   ⇝ op_stop
                               <.> ch_e1_IntCmpExp   ⟹ idExp
                               <.> ch_e2_IntCmpExp   ↪ ch_stop_ForStmt)
    idExp     = (aspIdExp      ,  ch_id_IdExp       ↪ ch_id_ForStmt)
    bodyWhile = (aspSeqStmt    ,  ch_s1_SeqStmt     ↪ ch_ss_ForStmt
                               <.> ch_s2_SeqStmt     ⟹ stepWhile)
    stepWhile = (aspAssigStmt  ,  ch_id_AssigStmt   ↪ ch_id_ForStmt
                               <.> ch_exp_AssigStmt  ⟹ expStep)
    expStep   = (aspIntBOpExp  ,  ch_op_IntBOpExp   ⇝ op_step
                               <.> ch_e1_IntBOpExp   ⟹ idExp
                               <.> ch_e2_IntBOpExp   ↪ ch_step_ForStmt)
in  withoutChild ch_dir_ForStmt
             (agMacro (aspSeqStmt ,  ch_s1_SeqStmt ⟹ initStmt
                                  <.> ch_s2_SeqStmt ⟹ whileStmt))
```

Figure 7.8: Macro definition of the **FOR**-loop

Our approach is not very suitable for some other kind of source-to-source transformations like optimizations, because we do not represent the AST with values (if we want to keep the AST extensible) and we (still) do not have higher-order attributes. Although a possible approach is to generate an AST of a fixed core language and perform the optimizations in this language.

7.3.4 Code generation

We generate the C abstract syntax representation provided by the `language-c`[7] package. This package also includes a pretty-printing function for the abstract syntax.

[7] `http://hackage.haskell.org/package/language-c`

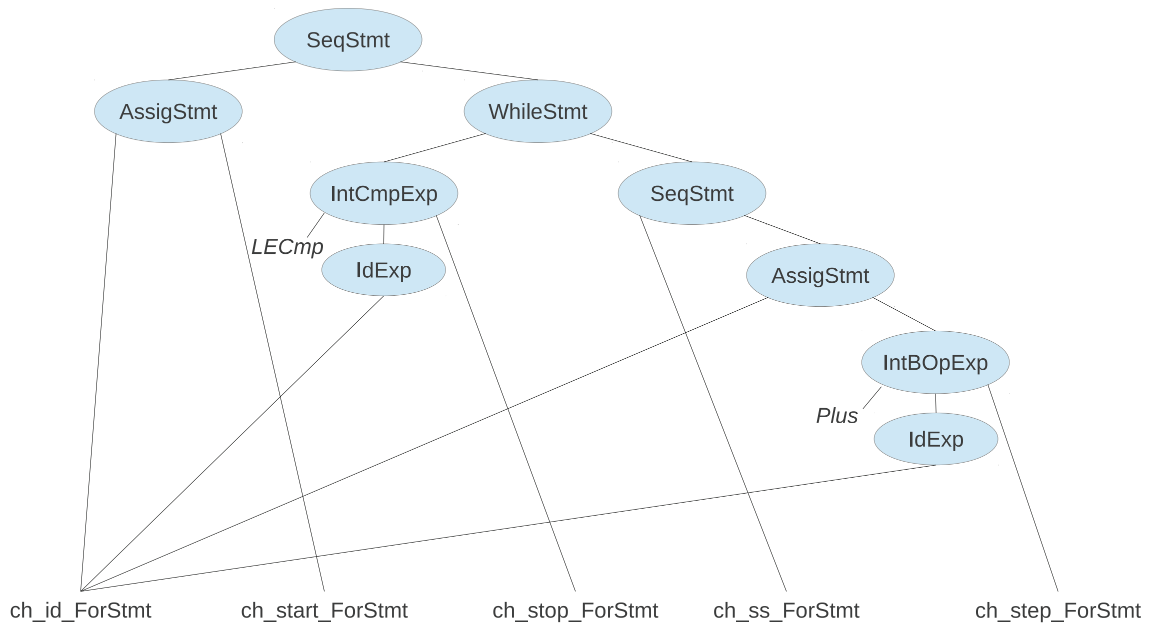

Figure 7.9: The **FOR**-loop in terms of the original AST

Since ANSI C does not include nested functions we have to lift all the procedures, types and constants definitions to top-level when generating the C code required by the challenge (note that the lifting as specified is trivial, since the exercise does not require bindings to be lifted properly). In order to avoid name clashes with C keywords or due to the lifting process, we rename every identifier to make it unique. New names are composed by: a character `'_'` (assuring no clashes with C keywords), the path (module and procedure names) to the scope were the name is defined and the actual name. Thus, if we have the following Oberon0 program:

```
MODULE A;
  VAR BC : INTEGER;
  PROCEDURE B;
    PROCEDURE C;
    END C
  END B
END A.
```

The names are mapped: the variable name BC to A_BC, the procedure name B to A_B and the procedure name C to A_B_C. Since underscore is not allowed in Oberon0 identifiers, this renaming does not introduce new clashes, like the one we could have had with C if the variable BC was called B_C.

To implement the renaming we extend the symbol table with the name mapping.

Lang. / Task	*Common*	*T1*	*T2*	*T3*	*T5*	*Total*
Common	-	42	14	-	23	79
L1	128	156	147	220	228	879
L2	187	98	69	65	56	475
L3	94	75	75	134	145	523
L4	48	67	56	197	95	463
Total	457	438	361	616	547	2419

Table 7.4: Code sizes (in lines of code) of the components of the compiler

7.4 Artifacts

In Table 7.4 we show the complexity (in lines of code without comments) of our implementation of the compiler, disaggregated into the different tasks and language levels. The *Common* column includes the *Gram* and *Decl* files, while the *Common* row includes some code used by the *Main* modules.

The code includes 26 lines of Template Haskell, calling functions defined in the libraries to avoid some boilerplate.

We have implemented all the combinations from L1-T1 to L4-T5, including the artifacts proposed by the challenge.

7.5 Conclusions

The most important aspect of our approach is the possibility to construct a compiler out of a collection of pre-compiled, statically type-checked, possibly mutually dependent language-definition fragments written in Haskell, but with a DSL taste.

When looking at all the aspects we have covered we can conclude that we managed to find solutions for all aspects of the problems; we were rescued by the fact that we could always fall back to plain Haskell, in case our libraries were not providing a standard solution for the problem at hand. We have seen such solutions for dealing with flexible symbol tables, generating new identifiers and types.

We mention again that our implementation is quite verbose, since each module contains quite some code "describing its interface" in the collection of co-operating modules. This is the price we have to pay for getting the extreme degree of flexibility we are providing. By collapse the modules the amount of linking information shrinks considerably. Other option to reduce verbosity is to use `uuagc` to generate `AspectAG` code (Chapter 6).

Another cause of the verbosity is that we have not used the system itself or Template Haskell to capture common patterns. We have chosen to reveal the underlying mechanisms, the role of the type system, the full flexibility provided, and have left open the possibility for further extensions.

The lack of open data types in Haskell makes it hard to implement AST trans-

formations in extensible languages using our technique. Semantic macros solve some of these problems. A possible approach is to use our technique to implement the front-end of a compiler, translating to a core fixed language, and then use other more traditional approaches (like `uuagc`) to implement the back-end. Another option is to use *data types à la carte* [66] to simulate open data types (and functions) in Haskell.

8 Conclusions and Future Work

With the combination of the techniques we have developed over the years our dream is close to coming true: the possibility to construct a complete compiler out of a collection of pre-compiled, statically type-checked, possibly mutually dependent language-definition fragments.

Summarizing the conclusions of the previous chapters, we tackled the problem of how to construct a composable compiler both at syntactic and semantic level.

8.1 First Class Syntax

We started by using typed grammars and typed transformations to implement an alternative version of the read functions; i.e. parsers for Haskell data types. By dealing with grammars as Haskell values we were able to compose, analyze and transform them (to apply for example the left-corner transform) to construct efficient parsers. With the use of typed transformations we mantain a type correct representation of the grammars during the transformation processes. It is important to point out that grammar fragments are combined on the fly; i.e. after they have been compiled. This work is based on a previous joint work with Arthur Baars and Doaitse Swierstra [6, 7]. Then we generalized our approach for expressing first-class context-free grammars. We introduced a set of combinators to describe, extend and compose grammar fragments using arrow notation, while expressing the productions in an applicative style.

8.2 First Class Semantics

At the semantic level, we introduced an embedding of attribute grammars in Haskell. We use strongly typed heterogenuous collections (`HList`) to model the collections of attribute computations that decorate the nodes of the abstract syntax tree. With the use of such structure, attribute grammar well-formedness conditions are expressed as type-level predicates, being the Haskell type-system responsible for checking them. We used type-level programming to describe some common attribute grammar patterns, in order to reduce code size. Attribute computations can be defined and composed; different attributes can be stored in different modules, compiled and then composed. An importan characteristic of our embedding is that attribute computations can also be redefined, in order to specialize (or modify the behaviour of) parts of already defined (and compiled) semantics.

We also introduced a macro-like mechanism to the attribute grammars embedding, to be able to express semantics in terms of already existing semantics. To program a

language extension using macros, the programmer does not need to know the implementation details of every attribute of the system; only the used attributes and the structure of the abstract syntax have to be known.

Since the use of our approach may sometimes feel a bit verbose to an attribute grammar programmer, we defined an extension to `UUAGC` (the Utrecht Attribute Grammars Compiler) to transform its code to our embedding. This provides a way to write a strongly-typed flexible attribute grammar system in a Domain-Specific Language style. We also defined a couple of optimizations that, at the cost of some flexibility (some attributes can not be redefined), result in a considerable speed improvement.

8.3 Their Composition

We have already seen in the Introduction how to compose a compiler using all our techniques, describing the syntax of the language using first class grammars and implementing the semantic functions as first class attribute grammars. We developed a small case study, implementing a compiler for the small Pascal-like Oberon0 language, to put this to work.

We think we were able to meet all aspects of the challenge in a satisfactory way. The embedding nature of our approach was very helpful, since we were able to use plain Haskell in the cases where our libraries did not provide a standard solution.

8.4 Future Work

With the combination of techniques described in this thesis we have established a firm bridge-head. So what problems are left and how should we proceed from here?

In the first place the organization of the collection of attributes in a linear structure, such as `HList` is costly. It is our experience however that a compiler spends most of its time in the auxiliary code for type-checking and -inferencing and (global) optimization. Thus for a modest language defined by a limited set of attributes we think the approach is not prohibitively costly. For more complicated languages, which use many attributes for their definition, there are several ways to alleviate this problem. Most attributes are not defined in isolation since most aspects are described using a collection of attributes. This is something we can exploit; do not place all attributes in a single linear `HList`, but group them in an tree-like structure [42], thus lowering the nesting depth of the top `HList` products.

Building the complete compiler from scratch as a collection of syntax extensions and fine-grained aspect definitions is probably not always the optimal approach; large parts of the compiler will be shared by all users, and there is no reason to use the relatively expensive techniques enabling extensibility all over the compiler, as long as the core compiler remains extensible. In this way we plan to define an extensible Haskell compiler, where the already existing attribute-grammar based description of `UHC` can be used to generate such an extensible core compiler. Therefore we provide default definitions for all aspects, each of which can be redefined. An additional

benefit of this approach is that we prevent unwanted or illogical combinations of aspects. For example, we may inhibit circumvention of the basic type-checking part of the compiler by simply not exporting that part of the interface.

A second point for improvement is the way attribute evaluation is scheduled. In the description above we use a very straightforward approach which uses Haskell's lazy evaluation; a tree attribution is seen as a single large data flow graph, with attributes in the nodes and semantic functions for defining the values of the nodes [31, 36, 16, 15]. Unfortunately this elegant approach breaks down when large trees are to be attributed; a lazy evaluation scheduling first builds a large dependency graph in memory, and only starts doing some real work when this large graph has been constructed. This resembles the application of function *foldr* to a very long list, usually remedied by using *foldl'* instead. Unfortunately there is no similar simple transformation which alleviates this problem for an arbitrary attribute grammar, since this requires a global flow analysis of the attribute dependencies [34]. However, the `UUAGC` already performs these analyses and can generate strict implementations containing explicitly scheduled code, and thus an efficient version for the sketched core compiler can be generated. Interfacing with this core compiler will be a bit more cumbersome, since the dependencies between the attributes now have become visible. Since these dependencies usually reflect the way the compiler programmer thinks about his attribute grammars [46] we expect this extra burden to be bearable.

A third problem arises from the way we construct our parsers and combine our aspects. With the current Haskell implementations *every time* we use the compiler the complete parser and attribute grammar is reconstructed from scratch; the individual grammar components are constructed first (*gramIni* and *gramExt*), then they are merged into a single large grammar (the calls to `+>>`) and references are resolved (*closeGram*); subsequently this large grammar is analysed and subjected to the Left-Corner Transform, and finally out of this resulting grammar the actual parser is constructed. A similar sequence of steps is done for the aspects. The final parser and evaluator, however, do not depend on the input of the compiler; they are global constant Haskell values; i.e. are in constant applicative form (CAF). Having such values repeatedly being constructed is not a problem of our approach alone, but occurs whenever some form of composition, analysis and transformation is taking place. We expect this to occur more often once the expressiveness of our techniques become more widely known and we think this problem is to be solved at the Haskell level in a generic way, e.g., by making it possible to save evaluated global values just before a program quits (using pragmas), and reading them back when the program is run for the next time; in this way the evaluation of CAFs is memoized over different runs of the program.

One might object that library code used in this paper goes far beyond the normal use of the Haskell type system, and that our type-level programming is not for the everyday Haskell programmer. We agree completely, although some of the complexity is already hidden in the libraries. Moreover, we believe type-level programming is a promising research area, which has broad interest in the (functional) programming languages community. Another possible line of future work is to explore the

implementation of our techniques in a dependently-typed language, such as Agda or Coq.

A Oberon0 Syntax

A.1 Concrete Grammar

A.1.1 L1

```
$ (csLabels ["cs_Module", "cs_Declarations", "cs_Expression", "cs_Factor"
            , "cs_StmtSeq", "cs_Statement", "cs_MaybeElseStmt"
            , "cs_Ident", "cs_IdentL", "cs_Type"])

l1 sf = proc _ → do
  rec
    modul    ← addNT ≺ ⟦ (pModule sf)
                          "MODULE" ident ";"
                          decls
                          (pMaybe (pEmptyStmt sf, id) (⟦ "BEGIN" ss ⟧))
                          "END" ident "." ⟧
    decls    ← addNT ≺ ⟦ (pDeclarations sf)
                          (pMaybe (pDeclL_Nil sf, id) (⟦ "CONST" cstDeclL ⟧))
                          (pMaybe (pDeclL_Nil sf, id) (⟦ "TYPE" typDeclL ⟧))
                          (pMaybe (pDeclL_Nil sf, id) (⟦ "VAR"  varDeclL ⟧)) ⟧
    cstDeclL ← addNT ≺ pFoldr (pDeclL_Cons sf, pDeclL_Nil sf)
                              (⟦ (pCstDecl sf) ident "=" exp ";" ⟧)
    typDeclL ← addNT ≺ pFoldr (pDeclL_Cons sf, pDeclL_Nil sf)
                              (⟦ (pTypDecl sf) ident "=" typ ";" ⟧)
    varDeclL ← addNT ≺ pFoldr (pDeclL_Cons sf, pDeclL_Nil sf)
                              (⟦ (pVarDecl sf) idL ":" typ ";" ⟧)
    idL      ← addNT ≺ ⟦ (pIdentL_Cons sf) ident
                          (pFoldr (pIdentL_Cons sf, pIdentL_Nil sf)
                                  (⟦ "," ident ⟧)) ⟧
    typ      ← addNT ≺ ⟦ (pType sf) ident ⟧
    exp      ← addNT ≺ ⟦ sexp ⟧
                   <|> ⟦ (eExp  sf) exp "="  sexp ⟧
                   <|> ⟦ (neExp sf) exp "#"  sexp ⟧
                   <|> ⟦ (lExp  sf) exp "<"  sexp ⟧
                   <|> ⟦ (leExp sf) exp "<=" sexp ⟧
                   <|> ⟦ (gExp  sf) exp ">"  sexp ⟧
                   <|> ⟦ (geExp sf) exp ">=" sexp ⟧
```

```
    sexp      ← addNT ≺ ⟦ signed ⟧
                    <|> ⟦ (plusExp  sf) sexp "+"  signed ⟧
                    <|> ⟦ (minusExp sf) sexp "-"  signed ⟧
                    <|> ⟦ (orExp    sf) sexp "OR" signed ⟧
    signed    ← addNT ≺ ⟦ term ⟧
                    <|> ⟦ (posExp sf) "+" term ⟧ <|> ⟦ (negExp sf) "-" term ⟧
    term      ← addNT ≺ ⟦ factor ⟧
                    <|> ⟦ (timesExp sf) term "*"   factor ⟧
                    <|> ⟦ (divExp   sf) term "DIV" factor ⟧
                    <|> ⟦ (modExp   sf) term "MOD" factor ⟧
                    <|> ⟦ (andExp   sf) term "&"   factor ⟧
    factor    ← addNT ≺ ⟦ (trueExp  sf) (kw "TRUE")  ⟧
                    <|> ⟦ (falseExp sf) (kw "FALSE") ⟧
                    <|> ⟦ (pParExp  sf) "(" exp ")"  ⟧
                    <|> ⟦ (notExp   sf) "~" factor   ⟧
                    <|> ⟦ (pIdExp sf) ident ⟧ <|> ⟦ (pIntExp sf) int ⟧
    ss        ← addNT ≺ ⟦ (pSeqStmt sf) stmt
                                          (pFoldr (pSeqStmt sf, pEmptyStmt sf)
                                                  (⟦ ";" stmt ⟧)) ⟧
    stmt      ← addNT ≺ ⟦ (pAssigStmt sf) ident ":=" exp ⟧
                    <|> ⟦ (pIfStmt    sf)
                          "IF" cond
                          (pFoldr (pCondStmtL_Cons sf, pCondStmtL_Nil sf)
                                  (⟦ "ELSIF" cond ⟧))
                          mbelse
                          "END" ⟧
                    <|> ⟦ (pWhileStmt sf) "WHILE" exp "DO" ss "END" ⟧
                    <|> ⟦ (pEmptyStmt sf) ⟧
    cond      ← addNT ≺ ⟦ (pCondStmt sf) exp "THEN" ss ⟧
    mbelse    ← addNT ≺ pMaybe ( pMaybeElseStmt_Nothing sf
                               , pMaybeElseStmt_Just sf)
                               (⟦ "ELSE" ss ⟧)
    ident     ← addNT ≺ ⟦ var ⟧ <|> ⟦ con ⟧
  exportNTs ≺ exportList modul $ export cs_Declarations   decls
                               . export cs_Expression     exp
                               . export cs_Factor         factor
                               . export cs_StmtSeq        ss
                               . export cs_Statement      stmt
                               . export cs_Ident          ident
                               . export cs_IdentL         idL
                               . export cs_MaybeElseStmt  mbelse
                               . export cs_Type           typ
```

A.1.2 L2

```
l2 sf = proc imported → do
  let ss     = getNT cs_StmtSeq        imported
  let stmt   = getNT cs_Statement      imported
  let exp    = getNT cs_Expression     imported
  let ident  = getNT cs_Ident          imported
  let mbelse = getNT cs_MaybeElseStmt  imported
  rec
    addProds ≺ (stmt    , ⟦ (pForStmt  sf) "FOR" ident ":=" exp dir exp mbexp
                                           "DO" ss "END" ⟧
                      <|> ⟦ (pCaseStmt sf) "CASE" exp "OF"
                                           c cs mbelse "END" ⟧)
    dir      ← addNT ≺ ⟦ (pTo sf) "TO" ⟧ <|> ⟦ (pDownto sf) "DOWNTO" ⟧
    mbexp    ← addNT ≺ pMaybe (pCst1Exp sf, id) (⟦ "BY" exp ⟧)
    cs       ← addNT ≺ pFoldr (pCaseL_Cons sf, pCaseL_Nil sf) (⟦ "|" c ⟧)
    c        ← addNT ≺ ⟦ (pCase sf) labels ":" ss ⟧
    labels   ← addNT ≺ ⟦ (pLabelL_Cons sf) label
                          (pFoldr (pLabelL_Cons sf, pLabelL_Nil sf)
                                  (⟦ "," label ⟧)) ⟧
    label    ← addNT ≺ ⟦ (pExpreLbl sf) exp ⟧
                   <|> ⟦ (pRangeLbl sf) exp ".." exp ⟧
  exportNTs ≺ imported
```

A.1.3 L3

```
l3 sf = proc imported → do
  let decls = getNT cs_Declarations imported
  let stmt  = getNT cs_Statement    imported
  let ss    = getNT cs_StmtSeq      imported
  let exp   = getNT cs_Expression   imported
  let ident = getNT cs_Ident        imported
  let idl   = getNT cs_IdentL       imported
  let typ   = getNT cs_Type         imported
  rec
    addProds ≺ (stmt,      ⟦ (pProcCStmt sf) ident params ⟧)
    params   ← addNT ≺ ⟦ "(" paraml ")" ⟧ <|> ⟦ (pExpressionL_Nil sf) ⟧
    paraml   ← addNT ≺ ⟦ (pExpressionL_Cons sf) exp
                           (pFoldr (pExpressionL_Cons sf, pExpressionL_Nil sf)
                                   (⟦ "," exp ⟧)) ⟧
                   <|> ⟦ (pExpressionL_Nil sf) ⟧
    updProds ≺ (decls, λdeclarations → ⟦ (pExtDeclarations sf) declarations
```

```
                                                          procDeclL ⟧)
    procDeclL ← addNT  ≺  pFoldr (pDeclL_Cons' sf, pDeclL_Nil' sf)
                                  (⟦ procDecl ⟧)
    procDecl  ← addNT  ≺  ⟦ (pProcDecl sf) "PROCEDURE" ident fparams ";"
                                           decls
                                           (pMaybe (pEmptyStmt' sf, id)
                                                   (⟦ "BEGIN" ss ⟧))
                                           "END" ident ";" ⟧
    fparams   ← addNT  ≺  ⟦ "(" fparaml ")" ⟧ <|> ⟦ (pParamL_Nil sf) ⟧
    fparaml   ← addNT  ≺  ⟦ (pParamL_Cons sf) fparam
                            (pFoldr (pParamL_Cons sf, pParamL_Nil sf)
                                    (⟦ ";" fparam ⟧)) ⟧
                      <|> ⟦ (pParamL_Nil sf) ⟧
    fparam    ← addNT  ≺  ⟦ (fpVar  sf) "VAR" idl ":" typ ⟧
                      <|> ⟦ (fpVal  sf)       idl ":" typ ⟧
  exportNTs ≺ imported
```

A.1.4 L4

```
l4 sf = proc imported → do
  let stmt   = getNT cs_Statement  imported
  let exp    = getNT cs_Expression imported
  let factor = getNT cs_Factor     imported
  let ident  = getNT cs_Ident      imported
  let idl    = getNT cs_IdentL     imported
  let typ    = getNT cs_Type       imported
  rec
    addProds ≺ (typ      ,  ⟦ (pArrayType  sf) "ARRAY" exp "OF" typ ⟧
                       <|> ⟦ (pRecordType sf) "RECORD" fieldl "END" ⟧)
    fieldl     ← addNT  ≺  ⟦ (pFieldL_Cons sf) field
                            (pFoldr  (pFieldL_Cons sf, pFieldL_Nil sf)
                                     (⟦ ";" field ⟧)) ⟧
    field      ← addNT  ≺  ⟦ (pField sf) idl ":" typ ⟧ <|> ⟦ (pEmptyField sf) ⟧
    addProds ≺ (factor,    ⟦ (pSelExp sf) ident selector ⟧)
    selector   ← addNT  ≺  ⟦ (pSelectL_Cons sf) sel
                            (pFoldr  (pSelectL_Cons sf, pSelectL_Nil sf)
                                     (⟦ sel ⟧)) ⟧
    sel        ← addNT  ≺  ⟦ (pSelField sf) "." ident ⟧
                       <|> ⟦ (pSelArray sf)  "[" exp "]" ⟧
    addProds ≺ (stmt,      ⟦ (pAssigSelStmt sf) ident selector ":=" exp ⟧)
  exportNTs ≺ imported
```

A.2 Abstract Syntax

A.2.1 L1

```
data Module = Module { idbgn_Module :: String
                     , decls_Module  :: Declarations
                     , stmts_Module :: Statement
                     , idend_Module :: String }

data Declarations = Declarations { cstdecl_Declarations :: DeclL
                                 , typdecl_Declarations :: DeclL
                                 , vardecl_Declarations :: DeclL }

type DeclL = [Decl]

data Decl = CstDecl { id_CstDecl  :: String, exp_CstDecl :: Expression }
          | TypDecl { id_TypDecl  :: String, typ_TypDecl :: Type }
          | VarDecl { idl_VarDecl :: IdentL, typ_VarDecl :: Type }

data Type = Type { id_Type :: String }

data Statement =   AssigStmt { id_AssigStmt   :: String
                             , exp_AssigStmt  :: Expression }
               |   IfStmt    { if_IfStmt      :: CondStmt
                             , elsif_IfStmt   :: CondStmtL
                             , else_IfStmt    :: MaybeElseStmt }
               |   WhileStmt { exp_WhileStmt  :: Expression
                             , ss_WhileStmt   :: Statement }
               |   SeqStmt   { s1_SeqStmt     :: Statement
                             , s2_SeqStmt     :: Statement }
               |   EmptyStmt

type CondStmtL = [CondStmt]

data CondStmt = CondStmt { exp_CondStmt :: Expression
                         , ss_CondStmt  :: Statement }

type MaybeElseStmt = Maybe Statement

type IdentL = [String]

type GHC_IntCmp = IntCmp
data IntCmp = ECmp | NECmp | LCmp | LECmp | GCmp | GECmp
type GHC_IntBOp = IntBOp
data IntBOp = Plus | Minus | Times | Div | Mod
type GHC_IntUOp = IntUOp
data IntUOp = Ng | Ps

type GHC_BoolBOp = BoolBOp
data BoolBOp = Or | And
type GHC_BoolUOp = BoolUOp
data BoolUOp = Not
```

```
data Expression = IntCmpExp   { op_IntCmpExp   :: GHC_IntCmp
                              , e1_IntCmpExp   :: Expression
                              , e2_IntCmpExp   :: Expression }
                | IntBOpExp   { op_IntBOpExp   :: GHC_IntBOp
                              , e1_IntBOpExp   :: Expression
                              , e2_IntBOpExp   :: Expression }
                | IntUOpExp   { op_IntUOpExp   :: GHC_IntUOp
                              , e_IntUOpExp    :: Expression }
                | BoolBOpExp  { op_BoolBOpExp  :: GHC_BoolBOp
                              , e1_BoolBOpExp  :: Expression
                              , e2_BoolBOpExp  :: Expression }
                | BoolUOpExp  { op_BoolUOpExp  :: GHC_BoolUOp
                              , e_BoolUOpExp   :: Expression }
                | IdExp       { id_IdExp       :: String }
                | IntExp      { int_IntExp     :: Int }
                | BoolExp     { bool_BoolExp   :: Bool }
                | ParExp      { e_ParExp       :: Expression }

$ (deriveAG ''Module)
$ (deriveLang "L1" [''Module, ''Declarations, ''DeclL, ''Decl, ''Type
                   , ''Statement, ''CondStmtL, ''CondStmt, ''MaybeElseStmt
                   , ''Expression, ''IdentL])


eExp  sf = pIntCmpExp sf (sem_Lit ECmp)
neExp sf = pIntCmpExp sf (sem_Lit NECmp)
lExp  sf = pIntCmpExp sf (sem_Lit LCmp)
leExp sf = pIntCmpExp sf (sem_Lit LECmp)
gExp  sf = pIntCmpExp sf (sem_Lit GCmp)
geExp sf = pIntCmpExp sf (sem_Lit GECmp)

plusExp  sf = pIntBOpExp sf (sem_Lit Plus)
minusExp sf = pIntBOpExp sf (sem_Lit Minus)
timesExp sf = pIntBOpExp sf (sem_Lit Times)
divExp   sf = pIntBOpExp sf (sem_Lit Div)
modExp   sf = pIntBOpExp sf (sem_Lit Mod)

posExp sf = pIntUOpExp sf (sem_Lit Ps)
negExp sf = pIntUOpExp sf (sem_Lit Ng)

orExp  sf = pBoolBOpExp sf (sem_Lit Or)
andExp sf = pBoolBOpExp sf (sem_Lit And)

notExp sf = pBoolUOpExp sf (sem_Lit Not)

trueExp  sf t = pBoolExp sf (λr → DTerm (pos (t r)) True)
falseExp sf f = pBoolExp sf (λr → DTerm (pos (f r)) False)
```

A.2.2 L2

```
data EXT_Statement
   = ForStmt   { id_ForStmt :: String, start_ForStmt :: Expression
               , dir_ForStmt :: ForDir, stop_ForStmt :: Expression
               , step_ForStmt :: Expression, ss_ForStmt :: Statement }
   | CaseStmt { exp_CaseStmt :: Expression, case_CaseStmt :: Case
               , cases_CaseStmt :: CaseL, else_CaseStmt :: MaybeElseStmt }
data ForDir = To | Downto
type CaseL = [Case]
data Case = Case { label_Case :: LabelL, ss_Case :: Statement }
type LabelL = [Label]
data Label = ExpreLbl  { exp_ExpreLbl :: Expression }
           | RangeLbl { e1_RangeLbl  :: Expression
                      , e2_RangeLbl  :: Expression }
data EXT_Expression = Cst1Exp


$ (extendAG ''EXT_Statement [''Statement, ''MaybeElseStmt, ''Expression])
$ (extendAG ''EXT_Expression [])
$ (deriveLang "L2" [''EXT_Statement, ''ForDir, ''CaseL, ''Case
                   , ''LabelL, ''Label, ''EXT_Expression])
```

A.2.3 L3

```
type GHC_KindParam = KindParam
data KindParam = VarP | ValP
data Param = Param { kind_Param :: GHC_KindParam
                   , idl_Param   :: IdentL
                   , typ_Param   :: Type }
type ParamL = [Param]
data EXT_Decl = ProcDecl { id_ProcDecl       :: String
                         , params_ProcDecl :: ParamL
                         , decls_ProcDecl   :: Declarations
                         , stmts_ProcDecl   :: Statement
                         , idend_ProcDecl   :: String }
data EXT_Declarations
          = ExtDeclarations    { decls_ExtDeclarations    :: Declarations
                               , prcdecl_ExtDeclarations :: DeclL }
type ExpressionL = [Expression]
data EXT2_Statement = ProcCStmt { id_ProcCStmt        :: String
                                , params_ProcCStmt :: ExpressionL }
```

```
  $ (extendAG ''EXT_Decl [''Declarations, ''Statement, ''IdentL, ''Type])
  $ (extendAG ''EXT_Declarations [''Declarations, ''DeclL])
  $ (extendAG ''EXT2_Statement [''Expression])
  $ (deriveLang "L3" [''EXT_Declarations, ''EXT_Decl, ''Param, ''ParamL
                     , ''EXT2_Statement, ''ExpressionL])
```

A.2.4 L4

```
data EXT_Type = ArrayType  { exp_ArrayType :: Expression
                           , typ_ArrayType :: Type }
              | RecordType { fields_RecordType :: FieldL }
type FieldL = [Field]
data Field = Field { idl_Field :: IdentL, typ_Field :: Type }
           | EmptyField
data EXT2_Expression = SelExp { id_SelExp :: String, sel_SelExp :: SelectL }
type SelectL = [Select]
data Select = SelField { id_SelField    :: String }
            | SelArray { exp_SelArray :: Expression }
data EXT3_Statement = AssigSelStmt { id_AssigSelStmt  :: String
                                   , sel_AssigSelStmt :: SelectL
                                   , exp_AssigSelStmt :: Expression }
  $ (extendAG ''EXT_Type [''Expression, ''IdentL, ''Type])
  $ (extendAG ''EXT2_Expression [''Expression])
  $ (extendAG ''EXT3_Statement [''SelectL, ''Expression])
  $ (deriveLang "L4" [''EXT_Type, ''FieldL, ''Field, ''EXT2_Expression
                     , ''SelectL, ''Select, ''EXT3_Statement])
```

Bibliography

[1] Harvey Abramson. Definite clause translation grammars. Technical report, University of British Columbia, Vancouver, BC, Canada, Canada, 1984.

[2] N. I. Adams, IV, D. H. Bartley, G. Brooks, R. K. Dybvig, D. P. Friedman, R. Halstead, C. Hanson, C. T. Haynes, E. Kohlbecker, D. Oxley, K. M. Pitman, G. J. Rozas, G. L. Steele, Jr., G. J. Sussman, M. Wand, and H. Abelson. Revised5 report on the algorithmic language scheme. *SIGPLAN Not.*, 33(9):26–76, September 1998.

[3] Stephen R. Adams. *Modular Grammars for Programming Language Prototyping*. PhD thesis, University of Southampton, Department of Elec. and Comp. Sci., 1991.

[4] Arthur I. Baars, Andres Löh, and S. Doaitse Swierstra. Parsing permutation phrases. *Journal of Functional Programming*, 14(6):635–646, 2004.

[5] Arthur I. Baars and S. Doaitse Swierstra. Type-safe, self inspecting code. In *Proc. of the 2004 ACM SIGPLAN workshop on Haskell*, pages 69–79, New York, NY, USA, 2004. ACM.

[6] Arthur I. Baars, S. Doaitse Swierstra, and Marcos Viera. Typed Transformations of Typed Abstract Syntax. In *TLDI '09: fourth ACM SIGPLAN Workshop on Types in Language Design and Implementation*, pages 15–26, New York, NY, USA, 2009. ACM.

[7] Arthur I. Baars, S. Doaitse Swierstra, and Marcos Viera. Typed Transformations of Typed Grammars: The Left Corner Transform. In *Proc. of the 9th Workshop on Language Descriptions Tools and Applications*, ENTCS, pages 18–33, 2009.

[8] Richard S. Bird. Using circular programs to eliminate multiple traversals of data. *Acta Inf.*, 21:239–250, 1984.

[9] Eric Bouwers, Martin Bravenboer, and Eelco Visser. Grammar engineering support for precedence rule recovery and compatibility checking. *Electron. Notes Theor. Comput. Sci.*, 203(2):85–101, 2008.

[10] John Boyland. Remote attribute grammars. *Journal of the ACM (JACM*, 52(4), Jul 2005.

[11] Martin Bravenboer. *Exercises in Free Syntax. Syntax Definition, Parsing, and Assimilation of Language Conglomerates*. PhD thesis, Utrecht University, Utrecht, The Netherlands, January 2008.

[12] Martin Bravenboer, Karl Trygve Kalleberg, Rob Vermaas, and Eelco Visser. Stratego/XT 0.17. A language and toolset for program transformation. *Science of Computer Programming*, 72(1-2):52–70, June 2008.

[13] Kasper Brink, Stefan Holdermans, and Andres Löh. Dependently typed grammars. In *Mathematics of Program Construction*, volume 6120 of *Lecture Notes in Computer Science*, pages 58–79. 2010.

[14] M. T. Chakravarty, Manuel, Gabriele Keller, and Simon Peyton Jones. Associated type synonyms. In *Proc. of the tenth International Conference on Functional Programming*, pages 241–253, New York, NY, USA, 2005. ACM.

[15] Oege de Moor, Kevin Backhouse, and S. Doaitse Swierstra. First class attribute grammars. *Informatica: An International Journal of Computing and Informatics*, 24(2):329–341, June 2000. Special Issue: Attribute grammars and Their Applications.

[16] Oege de Moor, L. Peyton Jones, Simon, and Van Wyk, Eric. Aspect-oriented compilers. In *Proc. of the 1st Int. Symposium on Generative and Component-Based Software Engineering*, pages 121–133, London, UK, 2000. Springer-Verlag.

[17] Dominique Devriese and Frank Piessens. Explicitly recursive grammar combinators: a better model for shallow parser DSLs. In *Proceedings of PADL 2011*, pages 84–98, 2011.

[18] Atze Dijkstra, Jeroen Fokker, and S. Doaitse Swierstra. The structure of the essential haskell compiler, or coping with compiler complexity. In *Implementation of Functional Languages*, 2007.

[19] Atze Dijkstra, Jeroen Fokker, and S. Doaitse Swierstra. The architecture of the Utrecht Haskell compiler. In *Proc. of the 2nd Symposium on Haskell*, pages 93–104, New York, NY, USA, 2009. ACM.

[20] Atze Dijkstra and S. Doaitse Swierstra. Typing Haskell with an Attribute Grammar. In *Advanced Functional Programming Summerschool*, number 3622 in LNCS. Springer-Verlag, 2004.

[21] Torbjörn Ekman and Görel Hedin. The JastAdd system - modular extensible compiler construction. *Sci. Comput. Program.*, 69(1-3):14–26, 2007.

[22] David Fisher and Olin Shivers. Static analysis for syntax objects. In *Proc. of the eleventh International Conference on Functional Programming*, pages 111–121, New York, NY, USA, 2006. ACM.

[23] David Fisher and Olin Shivers. Building language towers with ziggurat. *Journal of Functional Programming*, 18(5-6):707–780, September 2008.

[24] Jeroen Fokker and S. Doaitse Swierstra. Abstract interpretation of functional programs using an attribute grammar system. In Adrian Johnstone and Jurgen Vinju, editors, *Language Descriptions, Tools and Applications*, 2008.

[25] Benedict R. Gaster and Mark P. Jones. A polymorphic type system for extensible records and variants. NOTTCS-TR 96-3, Nottingham, 1996.

[26] Thomas Hallgren. Fun with functional dependencies or (draft) types as values in static computations in haskell. In *Proc. of the Joint CS/CE Winter Meeting*, 2001.

[27] Bastiaan Heeren, Jurriaan Hage, and S. Doaitse Swierstra. Scripting the type inference process. In *Eighth ACM Sigplan International Conference on Functional Programming*, pages 3 – 13, New York, 2003. ACM Press.

[28] Ralf Hinze. Fun with phantom types. In Jeremy Gibbons and Oege de Moor, editors, *The Fun of Programming*, pages 245–262. Palgrave Macmillan, 2003.

[29] John Hughes. Generalising monads to arrows. *Sci. Comput. Program.*, 37(1-3):67–111, 2000.

[30] M. Johnson. Finite-state approximation of constraint-based grammars using left-corner grammar transforms. In *COLING-ACL Õ98, Montreal, Quebec, Canada*, pages 619–623. Association for Computational Linguistics, 1998.

[31] Thomas Johnsson. Attribute grammars as a functional programming paradigm. In *Proc. of the Functional Programming Languages and Computer Architecture*, pages 154–173, London, UK, 1987. Springer-Verlag.

[32] Mark P. Jones. Typing Haskell in Haskell. In *Haskell Workshop*, 1999.

[33] P. Jones, Mark. Type classes with functional dependencies. In *Proc. of the 9th European Symposium on Programming Languages and Systems*, pages 230–244, London, UK, 2000. Springer-Verlag.

[34] Uwe Kastens. Ordered Attribute Grammars. *Acta Informatica*, 13:229–256, 1980.

[35] Oleg Kiselyov, Ralf Lämmel, and Keean Schupke. Strongly typed heterogeneous collections. In *Proc. of the 2004 Workshop on Haskell*, pages 96–107. ACM Press, 2004.

[36] M. F. Kuiper and S. Doaitse Swierstra. Using attribute grammars to derive efficient functional programs. RUU-CS 86-16, Department of Computer Science, 1986.

[37] Ralf Lämmel and Simon Peyton Jones. Scrap your boilerplate: a practical design pattern for generic programming. *ACM SIGPLAN Notices*, 38(3):26–37, March 2003. Proc. of the ACM SIGPLAN Workshop on Types in Language Design and Implementation (TLDI 2003).

[38] Ralf Lämmel and Simon Peyton Jones. Scrap more boilerplate: reflection, zips, and generalised casts. In *Proc. of the ACM SIGPLAN International Conference on Functional Programming (ICFP 2004)*, pages 244–255. ACM Press, 2004.

[39] B. M. Leavenworth. Syntax macros and extended translation. *Commun. ACM*, 9(11):790–793, 1966.

[40] William Maddox. Semantically-sensitive macroprocessing. Technical report, Berkeley, CA, USA, 1989.

[41] Geoffrey Mainland. Why it's nice to be quoted: quasiquoting for Haskell. In *Proceedings of Haskell Symposium 2007*, pages 73–82, 2007.

[42] Bruno Martínez, Marcos Viera, and Alberto Pardo. Just Do It While Compiling!: Fast Extensible Records in Haskell. In *Proc. of the ACM SIGPLAN 2013 Workshop on Partial Evaluation and Program Manipulation (PEPM'13)*, 2013.

[43] Conor McBride. Faking it simulating dependent types in haskell. *Journal of Functional Programming*, 12(5):375–392, 2002.

[44] Conor McBride and Ross Paterson. Applicative programming with effects. *Journal of Functional Programming*, 18(01):1–13, 2007.

[45] Marjan Mernik and Viljem Žumer. Incremental programming language development. *Computer languages, Systems and Structures*, 31:1–16, 2005.

[46] Arie Middelkoop, Atze Dijkstra, and S. Doaitse Swierstra. Visit Functions for the Semantics of Programming Languages. In *Workshop on Generative Programming*, 2010.

[47] Ulf Norell. Dependently typed programming in Agda. In *6th International School on Advanced Functional Programming*, 2008.

[48] Nicolas Oury and Wouter Swierstra. The power of Pi. In *Proc. of the Thirteenth International Conference on Functional Programming*, 2008.

[49] Emir Pasalic and Nathan Linger. Meta-programming with typed object-language representations. In *Generative Programming and Component Engineering (GPCE'04)*, volume LNCS 3286, pages 136 – 167, October 2004.

[50] Ross Paterson. A new notation for arrows. In *Proc. of the 6th Int. Conference on Functional Programming*, pages 229–240, New York, USA, 2001. ACM.

[51] JC Petruzza, Koen Claessen, and Simon Peyton Jones. Derived read instances for recursive datatypes with infix constructors are too inefficient. GHC Ticket 1544.

[52] Simon Peyton Jones. *Haskell 98 Language and Libraries: the Revised Report.* 2003.

[53] Simon Peyton Jones, Mark Jones, and Erik Meijer. Type classes: an exploration of the design space. In *Haskell Workshop*, June 1997.

[54] Simon Peyton Jones, Dimitrios Vytiniotis, Stephanie Weirich, and Geoffrey Washburn. Simple unification-based type inference for gadts. *SIGPLAN Not.*, 41(9):50–61, 2006.

[55] W. Reps, Thomas, Carla Marceau, and Tim Teitelbaum. Remote attribute updating for language-based editors. In *Proc. of the 13th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 1–13, New York, NY, USA, 1986. ACM.

[56] J.C. Reynolds. User defined types and procedural data as complementary approaches to data abstraction. In S.A. Schuman, editor, *New Directions in Algorithmic Languages*. INRIA, 1975.

[57] Tom Schrijvers and Bruno C.d.S. Oliveira. Monads, zippers and views: virtualizing the monad stack. In *ICFP 2011*, pages 32–44. ACM, 2011.

[58] Tim Sheard and Simon Peyton Jones. Template meta-programming for Haskell. In *Proc. of the ACM SIGPLAN workshop on Haskell*, pages 1–16. ACM Press, 2002.

[59] Alexandra Silva and Joost Visser. Strong types for relational databases. In *Proc. of the 2006 Workshop on Haskell*, pages 25–36, New York, NY, USA, 2006. ACM.

[60] Anthony M. Sloane, Lennart C. L. Kats, and Eelco Visser. A pure object-oriented embedding of attribute grammars. In *Proc. of the Ninth Workshop on Language Descriptions, Tools, and Applications*, March 2009.

[61] S. Doaitse Swierstra. Linear, online, functional pretty printing (extended and corrected version). Technical Report UU-CS-2004-025a, Inst. of Information and Comp. Science, Utrecht Univ., 2004.

[62] S. Doaitse Swierstra. The Utrecht Parsing Libraries. `http://www.cs.uu.nl/wiki/bin/view/HUT/ParserCombinators`, July 2008.

[63] S. Doaitse Swierstra. Combinator parsers: a short tutorial. In A. Bove, L. Barbosa, A. Pardo, , and J. Sousa Pinto, editors, *Language Engineering and Rigorous Software Development*, volume 5520 of *LNCS*, pages 252–300. Spinger, 2009.

[64] S. Doaitse Swierstra, Pablo R. Azero Alcocer, and João A. Saraiva. Designing and implementing combinator languages. In S. Doaitse Swierstra, Pedro Henriques, and José Oliveira, editors, *Advanced Functional Programming, Third International School, AFP'98*, volume 1608 of *LNCS*, pages 150–206. Springer-Verlag, 1999.

[65] S. Doaitse Swierstra and Olaf Chitil. Linear, bounded, functional pretty-printing. *Journal of Functional Programming*, 19(01):1–16, 2009.

[66] Wouter Swierstra. Data types à la carte. *J. Funct. Program.*, 18(4):423–436, July 2008.

[67] E. Van Wyk, O. de Moor, K. Backhouse, and P. Kwiatkowski. Forwarding in attribute grammars for modular language design. In *Proc. 11th International Conf. on Compiler Construction*, volume 2304 of *LNCS*. Springer-Verlag, 2002.

[68] Eric Van Wyk, Derek Bodin, Jimin Gao, and Lijesh Krishnan. Silver: an extensible attribute grammar system. *Science of Computer Programming*, 75(1–2):39–54, January 2010.

[69] Marcos Viera and S. Doaitse Swierstra. Attribute grammar macros. In *XVI Simpósio Brasileiro de Linguagens de Programação*, LNCS, pages 150–165, 2012.

[70] Marcos Viera, S. Doaitse Swierstra, and Atze Dijkstra. Grammar Fragments Fly First-Class. In *Proc. of the 12th Workshop on Language Descriptions Tools and Applications*, pages 47–60, 2012.

[71] Marcos Viera, S. Doaitse Swierstra, and Eelco Lempsink. Haskell, Do You Read Me?: Constructing and composing efficient top-down parsers at runtime. In *Proc. of the first Symposium on Haskell*, pages 63–74, New York, NY, USA, 2008. ACM.

[72] Marcos Viera, S. Doaitse Swierstra, and Arie Middelkoop. UUAG Meets AspectAG. In *Proc. of the 12th Workshop on Language Descriptions Tools and Applications*, 2012.

[73] Marcos Viera, S. Doaitse Swierstra, and Wouter Swierstra. Attribute Grammars Fly First-Class: How to do aspect oriented programming in Haskell. In *Proc. of the 14th Int. Conf. on Functional Programming*, pages 245–256, New York, USA, 2009. ACM.

[74] Phil Wadler. The Expression Problem. E-mail available online., 1998.

[75] Daniel Weise and Roger Crew. Programmable syntax macros. In *Proc. of the 1993 conference on Programming Language Design and Implementation*, pages 156–165, New York, NY, USA, 1993. ACM.

[76] Niklaus Wirth. *Compiler construction*. International computer science series. Addison-Wesley, 1996.

[77] Alexey Rodriguez Yakushev, Stefan Holdermans, Andres Löh, and Johan Jeuring. Generic programming with fixed points for mutually recursive datatypes. In *Proceedings of ICFP 2009*, pages 233–244, 2009.

[78] Brent A. Yorgey, Stephanie Weirich, Julien Cretin, Simon Peyton Jones, Dimitrios Vytiniotis, and José Pedro Magalhães. Giving haskell a promotion. In *Proceedings of the 8th ACM SIGPLAN workshop on Types in language design and implementation*, TLDI '12, pages 53–66, New York, NY, USA, 2012. ACM.

Samenvatting

Een domeinspecieke taal (Domain Specific Language, DSL) is een programmeertaal die op een specifiek toepassingsgebied is toegesneden, en waarmee domeinexperts gemakkelijk programma's kunnen schrijven die hun probleem oplossen.

Het bieden van ondersteuning bij het gebruik van een dergelijke programmeertaal is geen sine cure: telkens moet een volledige vertaler geschreven worden, die programma's vanuit een dergelijke taal afbeeldt op code die door een computer kan worden verwerkt.

Een manier om dit probleem aan te pakken in om de DSL in te bedden in een bestaande taal, gebruik makend van de taalconstructies die deze gasttaal al biedt, zoals functies, combinatoren of macro's. Op deze manier realiseren we de voordelen van een DSL in de vorm van een biliotheek uitgedrukt in de gasttaal.

Deze oplossing komt echter met zijn eigen problemen: doordat nieuwe constructies eerst worden afgebeeld op bestaande constructies is de uiteindelijke feedback van de compiler, bijvoorbeeld over geconsteerde fouten, uitgedrukt in termen van de gasttaal. Wij stellen een benadering voor waarbij de gasttaal zelf gemakkelijk uitgebreid kan worden met nieuwe taalconstructies.

We ontwikkelen een aantal vertalerbouwtechnieken (Compositional Compiler Construction), die de voordelen van een echte DSL combineren met het implementatiegemak van het gebruik van een gasttaal: vertalers worden samengesteld uit apart vertaalde, statisch getypeerde "taal-definitie-fragmenten"

Onze gereedschapskist bestaat uit een collectie EDSL's uitgedrukt in de moderne functionele programmeertaal Haskell. Zowel syntax als semantiek van onze (uitbreidbare) talen zijn normale Haskell waarden, die kunnen worden gedefinierd, samengesteld, getransformeerd, geparametriseerd en genstantieerd. Het gebruik van Haskell garandeert belangrijke consistentie eigenschappen van aldus geconstrueerde vertalers.

Curriculum Vitae

Marcos Omar Viera Larrea was born September 09 1979 in San José, Uruguay. From 1998 to 2003, he studied Computing Systems Engineering at Universidad de la República, Uruguay. From 2004 to 2007, he did a Master in Computer Science at PEDECIBA, Uruguay. From 2008 to 2013, he was a PhD student at PEDECIBA, and from 2012 also at Utrecht University, under the supervision of Doaitse Swierstra and Alberto Pardo. He works doing teaching activities at Universidad de la República since 2002. Since November 2012 he is an Assistant Professor at Department of Computer Science, Engineering School, Universidad de la República.