Modular and type-safe definition of Attribute Grammars with AspectAG

Juan García-Garland

Programa de Posgrado de Maestría en Informática
PEDECIBA Informática, Facultad de Ingeniería, Instituto de Computación
Universidad de la República - Universidad de la República

Montevideo – Uruguay
Mayo de 2022

Modular and type-safe definition of Attribute Grammars with AspectAG

Juan García-Garland

Tesis de Maestría presentada al Programa de Posgrado de Maestría en Informática, PEDECIBA Informática de la Universidad de la República, como parte de los requisitos necesarios para la obtención del título de Magíster en Maestría en Informática.

Directores:
Dr. Alberto Pardo
Dr. Marcos Viera

Director académico:
Dr. Alberto Pardo

Montevideo – Uruguay
Mayo de 2022

García-Garland, Juan

Modular and type-safe definition of Attribute Grammars with AspectAG / Juan García-Garland. - Montevideo: Universidad de la República, Universidad de la República, PEDECIBA Informática, Facultad de Ingeniería, Instituto de Computación, 2022.

X, 200 p. 29, 7cm.

Directores:

Alberto Pardo

Marcos Viera

Director académico:

Alberto Pardo

Tesis de Maestría – Universidad de la República, Programa de Maestría en Informática, 2022.

Referencias bibliográficas: p. 193 – 200.

1. Programación a nivel de tipos, 2. Lenguajes de dominio específico, 3. Manejo de errores, 4. Gramáticas de atributos, 5. Diseño de compiladores. I. Pardo, Alberto, Viera, Marcos, . II. Universidad de la República, Programa de Posgrado de Maestría en Informática. III. Título.

INTEGRANTES DEL TRIBUNAL DE DEFENSA DE TESIS

Dr. Rodrigo Ribeiro

Dr. Alejandro Gadea

Dr. Daniel Calegari

Montevideo – Uruguay
Mayo de 2022

Science is what we understand
well enough to explain to a computer.
Art is everything else we do.

Donald Knuth.

RESUMEN

`AspectAG` es un lenguage de dominio específico embebido (EDSL) que codifica gramáticas de atributos (AGs) como ciudadanos de primera clase. `AspectAG` garantiza la buena formación de las AGs en tiempo de compilación por medio del uso de registros extensibles y predicados, codificados gracias al uso de características antiguas de programación a nivel de tipos, como clases multiparámetro y dependencias funcionales.

`AspectAG` sufre las desventajas usuales de los EDSLs: cuando ocurren errores de tipado, los mensajes de error reportados no se expresan en términos del dominio, sino del lenguage anfitrión. También es usual que detalles de implementación del EDSL se vean filtrados en estos mensajes. El uso de técnicas de programación a nivel de tipos agrava la situación porque los mecanismos de abstracción a nivel de tipos son pobres. Además, las técnicas de programación a nivel de tipos usadas en `AspectAG` son esencialmente no tipadas, lo que es inconsistente con nuestro enfoque de tipado fuerte.

Usando extensiones modernas al sistema de tipos de Haskell, proponemos una nueva versión de la biblioteca `AspectAG`, abordando los problemas antes mencionados. Las nuevas definiciones de AGs son más seguras tanto a nivel de tipado como a nivel de *kinds* (tipado a nivel de tipos). Además, un conjunto identificado de errores específicos del dominio son reportados con mensajes referentes al mismo. Para lograr esto, definimos y utilizamos un *framework* para manipular errores de tipado, que puede ser aplicado a cualquier EDSL.

Mostramos la pragmática de `AspectAG` definiendo lenguajes y extendiéndoles con nueva sintaxis y con nueva semántica. Utilizamos el lenguaje MateFun, un lenguaje funcional puro utilizado para enseñar matemáticas como caso de estudio.

Palabras claves:
Programación a nivel de tipos, Lenguajes de dominio específico, Manejo de errores, Gramáticas de atributos, Diseño de compiladores.

ABSTRACT

`AspectAG` is a Haskell-embedded domain-specific language (EDSL) that encodes first-class attribute grammars (AGs). `AspectAG` ensures the well-formedness of AGs at compile time by using extensible records and predicates encoded using old-fashioned type-level programming features, such as multi-parameter type classes and functional dependencies.

`AspectAG` suffers the usual drawbacks of EDSLs: when type errors occur they usually do not deliver error messages that refer to domain terms, but to the host language. Often, implementation details of the EDSL are leaked in those messages. The use of type-level programming techniques makes the situation worse since type-level abstraction mechanisms are quite poor. Additionally, old-fashioned type-level programs are untyped at type-level, which is inconsistent with the general approach of strongly-typed functional programming.

By using modern Haskell extensions and techniques we propose a reworked version of `AspectAG` that tackles those weaknesses. New AG definitions are safer, both at the level of types and at the level of kinds. Furthemore, a set of identified domain-specific errors are reported with DSL-oriented messages. To achieve this, we define and use a framework for manipulating type errors that can be used in any EDSL.

We show the pragmatics of `AspectAG` by defining languages and extending them both with new syntax and semantics. We use MateFun, a purely-functional language used to teach mathematics as a case study.

Keywords:
Type-level programming, DSLs, Error Handling, Attribute Grammars, Compiler Design.

Contents

Chapter 1

Introduction

Compiler construction is a central topic in computer science. Compilers are translators from one formal language to another, according to a given semantic specification. Parsers build abstract syntax trees (ASTs), which are tree representations of the syntactic structure of words in the language. Typical stages of the compiler front-end as object binding, type checking, or the generation of code in an intermediate representation are tree-processing tasks.

Functional programmers have available higher-order functions such as *map* and *fold* as powerful abstraction tools to manipulate terms [Gibbons, 2003], and therefore, to build compilers. Fold captures the structural recursion associated to a data type requiring a function for each constructor that combines contained data and recursive calls. Altogether these functions form an *algebra*, capturing semantics for the grammar -data type- and the fold builds the computation.

In practice, however, when constructing real-world compilers some problems arise. ASTs tend to have a lot of constructors, meaning huge algebras. Some information such as symbol tables must flow top-down (while this can be implemented with folds in a higher-order language as Haskell, perhaps it is not so intuitive). But a more notorious issue overshadows the previous ones: adding a constructor in an AST means breaking every implemented semantics. Note that this is not an infrequent scenario: it is usual for programming languages to evolve over time.

More generally, given any functional program, extending data (e.g. if a data type is extended with a new case construct) is not easy. Each case expression where a value of the extended type is matched has to be inspected and modified accordingly. However, it is trivial to extended the program by defining new functions that process the data type.

On the other side, object-oriented programing languages excels when defining new data: users can implement algebraic data types with a composite design pattern, and by simply adding a new class the job is done. However, to define a new function for a data type, all the existing subclasses must be modified to add a new method.

In some sense, those programming paradigms are dual: functional programming simplifies extending semantics, while object-oriented programming simplifies extending syntax. This duality was first noted by Reynolds [Reynolds, 1975] and later referred by Wadler [Wadler, 1998]. Wadler coined the name "The expression problem" for the challenge of finding a formalism where we can easily extend syntax and semantics (while maintaining type safety).

Attribute grammars (AGs, for short) offer an approach to tackle the expression problem. AGs were originally introduced to describe semantics for context-free languages [Knuth, 1968]. Given a context-free grammar, attributes are associated to each of its productions. Attribute values are computed in every node of the AST, according to semantic rules that are expressed in terms of the attribute values of the children and the parent. Attributes are classified in at least two sets: synthesized attributes (where information flows bottom-up) and inherited attibutes (where it flows top-down). AGs have not only proven useful to implement programming language semantics, but as a general purpose programming paradigm.

AGs can be thought of a particular case of a domain-specific language (DSL). DSLs are a useful abstraction tool to solve problems using specialized domain terms. DSLs can be implemented as a standalone language, introducing a full compiler toolchain, or embedded as a library in a host language (embedded DSLs, EDSLs for short). EDSLs have some advantages. For instance, all constructs of the host language and its libraries are available to the users. Furthemore, the amount of work required compared to the standalone approach is minimal. In higher-order functional programming languages such as Haskell, the embedded approach is widely used and successful [Hudak, 1998, Gibbons, 2013].

An important drawback of EDSLs is that they are simply embedded libraries, thus, when type errors occur they usually do not deliver error messages that refer to domain terms, leaking in addition implementation details in those messages. This breaks all abstraction mechanisms that may have been taken into account in building the library. The problem is even worse if we use type-level programming techniques to implement the DSL.

`AspectAG` is a Haskell EDSL introduced by Viera [Viera et al., 2009], that implements first-class AGs. It uses extensible polymorphic records and predicates encoded using old-fashioned type-level programming features, such as multi-parameter type classes [Peyton Jones et al., 1997] and functional dependencies [Jones, 2000], to ensure the well-formedness of AGs at compile time.

Type errors were of course a weakness, aggravated by the fact that an AG is a structure that can be easily ill-formed. For instance, for the grammar implementer it is a common mistake to try to use attributes that are not defined in some production. In the specific case of the original `AspectAG` library, the type-level programming techniques that were used were really ad-hoc, exploiting extensions originally introduced for other uses. In particular, at type level, programming was essentially untyped.

More recent versions of GHC provide extensions to the type system to support a more robust and trustworthy programming at the type level. By using such extensions, in this thesis we propose a reworked version of `AspectAG`[1] that tackles some of its most important weaknesses. The used type-level programming techniques allowed us to program in a strongly-typed fashion at type level (we say, strongly kinded). We also define a framework to manipulate type errors, keeping track of the context of the possible sources of errors, to show precise (DSL oriented) messages when they occur.

The contributions of this thesis are the following:

- We introduce a reworked version of the `AspectAG` library using modern type-level programming techniques. In particular, this results in a system with strongly-kinded types, having a strongly typed discipline both at the level of terms and at the level of types. We also explore new features such as polymorphism.
- We develop the library `poly-rec` to implement the main data structure used in `AspectAG`, the *extensible record*. We use type-level programming techniques to have a very general structure that can be specialized in many concrete instances.
- We develop `require`, a framework to manage user-defined type errors

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

in EDSLs. We use the framework to implement both `AspectAG` and `poly-rec`.

- We develop a case study consisting of a modular compiler for a functional programming programming language called MateFun.

In Chapter 2, we introduce preliminary concepts of our topic of study. In Chapter 3, we show an overview of the reimplementaion of the `AspectAG` library, while using a simple expression language as an example. In Chapter 4 we define a set of errors that AG coders might make when building an AG, and we show how the GHC compiler reports compile-time domain-specific error messages when those mistakes are commited writting code in the `AspectAG` EDSL. In Chapter 5 we show the implementation details of the library to explain how everything was achieved: from the definition of a framework to handle errors to the data structures in which `AspectAG` is built. In Chapter 6 we show a case study where we implement a compiler for the MateFun language [Carboni et al., 2018]. Finally, in Chapter 7 we discuss our conclusions and we outline possible lines of future work.

Chapter 2

Preliminaries.

In this chapter, we present the main concepts that are used as a foundation to build this thesis. In Section 2.1 a brief presentation of attribute grammars is given. In Section 2.2 we review the main techniques of type-level programming in Haskell. In Section 2.3 we present a well-known problem in programming language design: "the expression problem". Finally, in Section 2.4 we briefly discuss the concept of embedded domain-specific languages, focusing in `AspectAG`.

2.1 Attribute grammars.

Attribute grammars (AGs, for short) were introduced by Knuth [Knuth, 1968, Knuth, 1990] to define semantics for context-free languages. Context-free languages are formal languages defined by context-free grammars. In this section, we formally define this domain of discourse.

2.1.1 Formal languages.

Given an *alphabet* (a set of symbols) Σ, we denote as Σ^* the set of strings (lists of elements, words) over Σ. Given $v, w \in \Sigma^*$ we denote the concatenation of v and w as vw. Given $a \in \Sigma$ we overload the notation and also denote as a the singleton list with the character a. For instance, if $\Sigma = \{a, b\}$ then $u := a$, $v := ab$, $w := bba$ are all elements of Σ^* and so is $uvw = aabbba$.

A language $\mathcal{L}$ over the alphabet Σ is any subset of Σ^*.

2.1.2 Context-free grammars.

Some languages, like those usually used in computer science, can be described from formal grammars. Context-free grammars are a particular class of formal grammars.

A context-free grammar is defined as a tuple (T, N, S, P) where:

- T is a set of symbols, called *terminal* symbols.
- N is a finite set of symbols, called *non-terminal* symbols. N and T satisfy $N \cap T = \emptyset$. The set $V := N \cup T$ is the set of *grammar symbols.*
- $S \in N$ is called the *start symbol.*
- P is a set of productions. A production is a pair (l, r) where $l \in N$ and $r \in (V \cup \mathfrak{P}(T))^*$, where $\mathfrak{P}(T)$ is the powerset of T. We usually write $l \to r$ instead of (l, r). We refer to l and r as the left and right hand side, respectively, of the production. More explicitly, productions have the shape $X \to X_1 X_2 \ldots X_n$ where $X \in N$ and $X_i \in V \cup \mathfrak{P}(T)$.

In the literature, T is usually restricted to be finite, and $r \in V^*$. We generalize the definition in order to be able to put sets of terminals (such as the set of integers, or the set of strings) in the right hand side of rules (see Example 1).

Given a grammar $G = (T, N, S, P)$ and $u, v \in V^*$, we say that $u \in V^*$ *yields* v, if for $a, b \in V^*$, $l \to r \in P$ we have that $u = alb$ and $v = ar'b$, where $r' \in V^*$ is equal to r except that each occurence of a set is swapped by an element belonging to that set. We say that v is the result of applying the rule $l \to r$ to u. This definition induces a relation $R \subseteq V^* \times V^*$. We write $u \Rightarrow v$ if $(u, v) \in R$. Rules can be applied many times. We write $u \Rightarrow^* v$ if v can be obtained from u applying a finite number of rules (possibly zero). In other words the $\Rightarrow^*$ relation is defined as the reflexive-transitive closure of $\Rightarrow$.

Now, we can specify the language $\mathcal{L}_G$ defined from a grammar G as:

$$\mathcal{L}_G = \{w \in T^* : S \Rightarrow^* w\}$$

Example 1. *Given sets $\mathbb{N}$, $\mathbb{V}$ representing natural numbers and strings. The following tuple is a grammar:*

$$(\mathbb{N} \cup \mathbb{V} \cup \{\texttt{+}\}, \{S, E\}, S, \{(S, E), (E, E\texttt{+}E), (E, \mathbb{N}), (E, \mathbb{V})\})$$

In EBNF notation it would be written as:

$$
\begin{aligned}
S &\rightarrow E\\
E &\rightarrow E\texttt{+}E\\
E &\rightarrow \mathbb{N}\\
E &\rightarrow \mathbb{V}
\end{aligned}
$$

Note that in the last two rules we use sets of terminals in the right hand side, hence the condition $r \in (V \cup \mathfrak{P}(T))^*$ *in the definition of a grammar. Note that* $\mathbb{N}$ *and* $\mathbb{V}$ *can be alternatively defined as context-free languages from a finite set of symbols. As long as we use context-free languages in the right hand side of rules, our approach of taking a potentially infinite number of terminal symbols and using sets in the right hand side of rules does not add power to the formalism with respect to the traditional definition.*

In this example, considering $0, 1 \in \mathbb{N}$, $\texttt{"x"} \in \mathbb{V}$, *we can assert that* $S \Rightarrow^* 0 + 1 + \texttt{"x"}$. *To verify it, note that* $S \Rightarrow E \Rightarrow E + E \Rightarrow E + E + E \Rightarrow 0 + E + E \Rightarrow 0 + 1 + E \Rightarrow 0 + 1 + \texttt{"x"}$.

A *derivation tree* of a string $v \in \mathcal{L}_G$ generated by $G = (T, N, S, P)$ is a tree where:

1. Each node is labelled by a symbol of V (recall $V = N \cup T$).
2. The label of the root is S.
3. If a node labelled X has children labelled $X_1 \dots X_n$ then there is a production in P of the shape $X \rightarrow Y_1 \dots Y_n$ where $Y_i = X_i$ if $X_i \in V$ or $Y_i \in X_i$ if $X_i \in \mathfrak{P}(T)$.
4. The leaves of the tree concatenated left to right form v.

Example 2. *Consider the grammar defined in Example 1. In Figure 2.1 we show two different derivation trees for* $0 + 1 + \texttt{"x"}$*:*

There can be many ways in which we can apply rules to prove a symbol belongs to a language. However, some chains are inherently different: when the corresponding derivation trees are different. When at least one word of a language has more than one derivation tree we say the language is *ambiguous*. Precedence rules are used to disambiguate the meaning of concrete words. Building the "correct" derivation tree for a grammar specification and a set

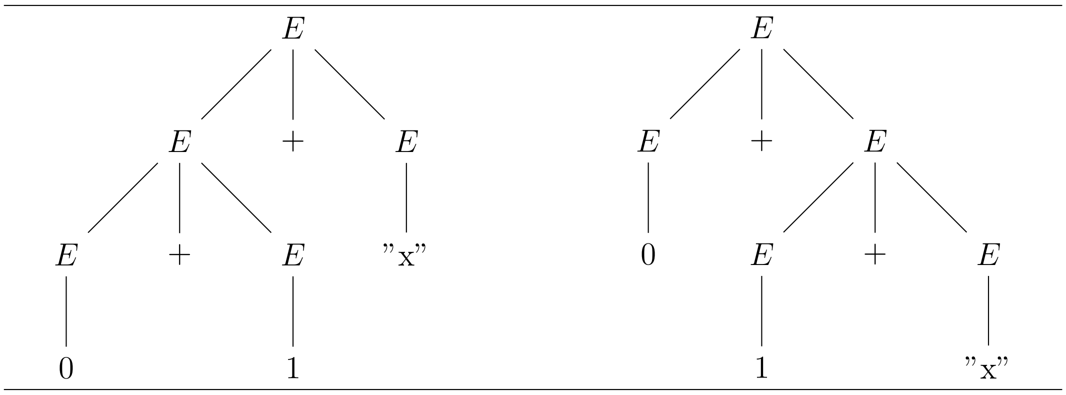


Figure 2.1: Two possible derivation trees for the expression $0 + 1 +$ `"x"`.

of precedence rules is a parsing problem. We will consider derivation trees are given, so discussions about how to handle operator precedence are out of our scope.

We must note that context-free grammars are not general enough to describe all useful languages. Examples of such languages can be given, such as the set of prime numbers, but this is also true for what interests us as compiler designers: programming languages. Many programming languages have context-dependent constructs. For instance, in C-like languages, every variable that occurs in a program should be declared before.

So, why are context-free grammars important? Dealing with context-dependent grammars is cumbersome. A widely used approach that works well when building a compiler is to recognize a context-free language and check context dependencies later on. This makes context-free grammars interesting because they are a compromise, being useful and manageable at the same time.

2.1.3 Attribute Grammars.

Attribute grammars [Knuth, 1968, Knuth, 1990] were introduced to describe semantics for context-free languages. Given a context-free grammar, attributes are associated to each of its productions. Attribute values are computed in every node of the abstract syntax tree, according to semantic rules that are expressed in terms of the attribute values of the children and the parent.

An AG is a tuple (G, A, R) where:

- G is a context-free grammar.

- A is a finite set of *attributes*. Each symbol has a set of attributes $A(X)$, so $A = \cup_{X \in V} A(X)$.
- R is a finite set of *semantic rules*. We will specify its nature later.

Usually each symbol $X \in V$ has its finite set of attributes $A(X)$ partitioned in disjoint sets $I(X), S(X)$ of inherited and synthesized attributes, respectively. Terminal symbols do not have inherited attributes. Some presentations [Paakki, 1995] forbid the start symbol to have inherited attributes also.

We say a production $p \equiv X_0 \to X_1 \ldots X_n \in P$, has an *attribute occurence* $X_i.a$ if $a \in A(X_i), (0 \leqslant i \leqslant n)$.

The *semantic rules* of the set R are formal constructions of the form $X_i.a = f(\gamma_1, \ldots, \gamma_k), k \geqslant 0$, where either $i = 0$ and $a \in S(X_i)$ or $1 \leqslant i \leqslant n$ and $a \in I(X_i)$, and each γ_j is an attribute occurence in p.

Intuitively, rules specify how to compute attribute values. Synthesized attributes are output of LHS symbols (*parents*), and inherited attributes are output of RHS symbols (*children*). Inherited attributes of parents and synthesized attributes of children are inputs of rules.

An *attributed tree* for a word v is a derivation tree where each node labelled by X is attached with *attribute instances* that correspond to the attributes of X.

The *attribute evaluation* is a process for computing the values of all attribute instances according to the semantic rules of the grammar.

An AG is *well defined* if for any word v and any derivation tree T of v the values of the attribute instances can be unambiguously computed by the attribute evaluation process.

This means:

1. For each production $p \in P$ there are rules to compute each attribute occurence.
2. The rules to compute attributes are *acyclic*.

Note that given an attributed tree, rules introduce dependencies over attributes, when some value is needed to compute another one. We can think in a graph with each attribute instance and an edge for each dependency. An acyclic set of rules means the graph is acyclic for any attribute tree. We will not formalize this concept further. For our purposes, the condition (2) will be lifted since with non-strict semantics circular programs can be productive.

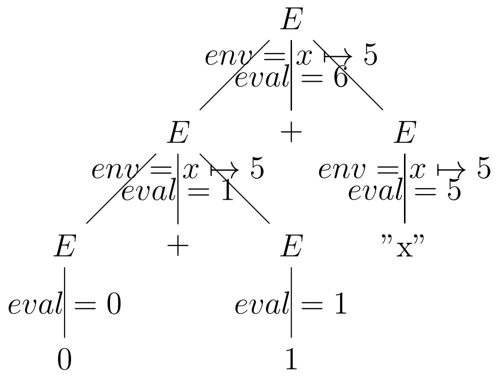


Figure 2.2: Attributed tree for an expression.

The condition (1) is the one we wish to control with the `AspectAG` EDSL, as we will see.

Example 3. *Consider the grammar G defined in the Example 1. Consider $A = \{eval, env\}$ and $R = \{S.eval = E.eval, E.eval = E_1.eval + E_2.eval, E.eval = v(\mathbb{N}), E.eval = (E.env)(v(\mathbb{V})), E_1.env = E.env, E_2.env = E.env\}$*[1]*.*

Then, the tuple $\Delta = (G, A, R)$ is a well-formed AG with $I(E) = \{env\}$, $S(E) = \{eval\}$, $S(S) = \{eval\}$. Note that each rule is related to an attribute occurence, hence to a production. In this case we do not write the relations explicitly since they can be inferred from the attribute occurences used in each rule.

Example 4. *In Figure 2.2 the attributed tree for the expression $(0 + 1 +$ `"x"`$)$ for the AG Δ is given. We write $x \mapsto 5$ to denote the mapping with the singleton domain $\{"x"\}$ that maps $"x"$ to 5.*

2.2 Type-level programming in Haskell.

Type-level programming is a term widely used in the strongly-typed functional programming community, especially in the Haskell community. The term refers to the practice of programming within the type system (to the

[1] In the rules we consider some notational devices: we named E_1 and E_2 the ocurrences of E in the right-hand side of the production representing the addition. $v(\mathbb{N})/v(\mathbb{V})$ denotes the natural number/string denoted by the symbol in the right-hand side. The value of the attribute *env* is a function from strings to integers, so it make sense to apply it as we did.

extent that it allows it) code that will "run" at compile time. This allows the encoding of expressive properties at the type level that must be satisfied by the corresponding code at the value level. Type-level programming techniques allow programmers to simulate dependent types in a type system where types and terms mantain their level independence. This independence is important because type inference can be mantained at some extend.

Haskell was standardized in the 1998 report [Jones, 2002], with no support to do type-level programming at first. On the other hand, compiler developers kept experimenting with extensions. This was true for GHC, the nowadays de facto standard Haskell implementation, as for more compilers at that time.

Extensions such as `MultiParamTypeClasses` [Peyton Jones et al., 1997] and `FunctionalDependencies` [Jones, 2000] are worth to be mentioned. Haskell was designed with type classes restricted to be parametrized by only one argument. This decision was motivated because type classes was a very experimental feature in the early '90s and language designers were cautious [Peyton Jones et al., 1997]. `MultiParamTypeClasses` extended the language to lift this restriction, allowing programmers to define relations on types. This introduces a technical subtlety: when a class method had no occurrence of a class type parameter in its type, occurrences of the method will have ambiguous types. The `FunctionalDependencies` extension was introduced to make type class relations functional, which helped the compiler to pick class instances. Other extensions such as `FlexibleContexts`, `FlexibleInstances`, `UndecidableInstances` existed to lift restrictions to type class definitions that existed to ensure the termination of the type checker.

The interaction between those extensions results to be very powerful: the possibility to express functional relations means programmers can write type functions. Haskell programmers can take advantage of this expressiveness. A demonstration of these techniques are given in [McBride, 2002], in the original incarnation of `AspectAG` [Viera et al., 2009], or in case studies such as [Silva and Visser, 2006].

Once type-level programming was born, further extensions emerged to make the job more pleasant. `Typefamilies` [Chakravarty et al., 2005a, Chakravarty et al., 2005b,Eisenberg et al., 2014] allowed to define proper type-level functions. The `DataKinds` extension introduces *data promotion*. With this extension enabled, the kind system is enriched with type-level data types (named data kinds). Each time the programmer defines a data type, it is defined at both levels. This also motivates the introduction of kind polymorphism. `KindSignatures` provides the possibility to do type-level annotations just as with types. Generalized algebraic data types are enabled by the `GADTs` extension. This extension allows programmers to define type-indexed families. `TypeOperators` and `TypeApplications` are not groundbreaking but help with notation.

With all this machinery the expressiveness of types is enhanced and a great portion of what programmers can express in a dependently typed language can now be written in Haskell. For instance, the following code is an implementation of heterogeneous lists in modern Haskell:

```
data HList (l :: [Type]) :: Type where
  HNil :: HList '[]
  (:::)  :: a → HList l → HList (a : l)
infixr 5 :::
```

The type is defined as a GADT indexed by a -promoted- list with elements of kind *Type* (which is an alias for the nowadays deprecated kind $*$). Data promotion implies some notation overloading. Most of the time, whether a constructor occurrence is a term or a type it is clear from the context. If not, promoted constructors can be prepended with a backslash. This is what we do in the first constructor; otherwise, the compiler cannot deduce if [ ] is the promoted empty list or the list constructor. For the type operator (:) -the promoted cons- in the return type of the second constructor this is not necessary.

With this definition, the term:

```
l = 'a' ::: False ::: "" ::: HNil
```

is an heterogeneous list of type *HList* '[*Char*, *Bool*, *String*].

Now we can define functions as *safeHead*:

```
safeHead :: HList (t : ts) → t
safeHead (x ::: _) = x
```

Contrary to the *head* function defined in the Haskell prelude over standard lists, *safeHead* is total, since it cannot be applied to an empty list. Moreover, the pattern matching used to define the function is exhaustive, since the constructor *HNil* has a type that would mismatch with the declared type.

Using type families programmers can define functions over types. The following type family defines the type-level append:

```
type family (l :: [Type]) :++: (m :: [Type]) :: [Type]
type instance '[]        :++: l = l
type instance (x ': xs) :++: l = x ': (xs :++: l)
```

Alternatively, programmers can define the same type-level function using a *closed* [Eisenberg et al., 2014] type family notation:

```
type family (l :: [Type]) :++: (m :: [Type]) :: [Type] where
  '[]        :++: l = l
  (x ': xs) :++: l = x ': (xs :++: l)
```

With any of the previous definitions the type $'[\mathit{Char}]$:++: $'[\mathit{Bool}, \mathit{String}]$ reduces to $'[\mathit{Char}, \mathit{Bool}, \mathit{String}]$.

Both the open and the closed styles are useful and most of the time interchangeable. The difference is that open type families can be extended with new cases. This is useful considering that the kind *Type* is extensible. We can add inhabitants of the kind *Type* by declaring new types and extend functions over types of kind *Type* adding type instances for the new type. On the other hand, the drawback of open type families is that overlapping equations are not permitted (otherwise the compiler won't know which instance to pick). Closed type families are usual non-extensible functions, and overlapping equations are permitted. As in term-level pattern matching, reduction tries the equations in order.

A downside is that we must write the same functions twice (at term and type levels). For instance, the append algorithm is already defined in the Haskell's *Prelude* for value-level lists. This issue can be tackled promoting functions to type families by using Template Haskell [Eisenberg and Stolarek, 2014]. We will not use the technique in this thesis.

Using the type-level append we can define a term-level append function for heterogeneous lists:

```
(+++) :: HList l → HList m → HList (l :++: m)
HNil       +++ l = l
(x ::: xs) +++ l = x ::: (xs +++ l)
```

Pattern matching is performed in the first argument. When matching is done over a GADT, the type constraints each GADT constructor introduces are available in the corresponding body definition.

After matching with the *HNil* constructor in the first equation the compiler knows the first argument of $(+\!\!+\!\!+)$, in this case, has type $\mathit{HList}\ '[]$. This implies the result of the function type must be of type $\mathit{HList}\ ('[]\ {:}\!\!+\!\!+\!\!{:}\ m)$, which is the same as $\mathit{HList}\ m$ after applying a type reduction according to the type family definition. For that reason, the compiler accepts the occurrence of l on the right hand side.

Analogously, after matching the first argument with the pattern $(x\ {:}{:}{:}\ xs)$ in the second clause, the compiler learns that $l\ \sim\ (t\,'{:}\ ts)$ [2] and therefore the result of the function must be of type $\mathit{HList}\ ((t\,'{:}\ ts)\ {:}\!\!+\!\!+\!\!{:}\ m)$. This expression can be reduced into $\mathit{HList}\ (t\,'{:}\ (ts\ {:}\!\!+\!\!+\!\!{:}\ m))$, that is, the type of the right hand side expression $x\ {:}{:}{:}\ (xs +\!\!+\!\!+ l)$.

Readers with knlowledge of dependently typed programming should notice these definitions look just identical to the ones given in a dependently typed language, although formally we never mixed types and terms. Also, just as in dependently typed programming languages, the definition of $(+\!\!+\!\!+)$ type checks because it is defined over the first argument just as the $({:}\!\!+\!\!+\!\!{:})$ family.

Suppose alternatively we want to encode the (rather inefficient) version of append where we use the first argument as an accumulator. We perform the recursion over the second list, putting each element of the second list at the end of the accumulator. Assuming we have available the *snoc* function one would think that the following is enough:

$$\begin{aligned}
&(+\!\!+\!\!+) :: \mathit{HList}\ l \to \mathit{HList}\ m \to \mathit{HList}\ (l\ {:}\!\!+\!\!+\!\!{:}\ m)\\
&l +\!\!+\!\!+ \mathit{HNil} \qquad\quad = l\\
&l +\!\!+\!\!+ (x\ {:}{:}{:}\ xs) = \mathit{snoc}\ l\ x +\!\!+\!\!+ xs
\end{aligned}$$

Unfortunately, that code does not compile. Looking at the first clause of the pattern matching, where the second argument is *HNil* the compiler learns that $m\ \sim\ '[]$, so the result should be of type $\mathit{HList}\ (l\ {:}\!\!+\!\!+\!\!{:}'\ [])$ while l is of type $\mathit{HList}\ l$. While it is true that $(l\ {:}\!\!+\!\!+\!\!{:}'\ [])\ \sim\ l$ for all instances of l this does not follow from the computation of the $({:}\!\!+\!\!+\!\!{:})$ family. For the second clause, a similar problems arise.

The solution is well-known for the dependently typed programmer. One needs to provide evidence to the compiler that those types are equal using propositional equality. The same we will do so.

[2] The notation $(t\ \sim\ u)$ stands for an *equality constraint* [The GHC team, 2020], and means that the types t and u need to be the same.

The module `Data.Type.Equality` of the current GHC `base` library defines the type:

$$\begin{array}{l}\textbf{data}\ a \cong b\ \textbf{where}\\ \quad \mathit{Refl} :: a \cong a\end{array}$$

The goal is to build an inhabitant of $l \cong l\ {:}{+\!\!+}{:}'\ []$ for each l, just as the dependently typed programmer would try to build a term of type $\Pi l \in [\mathit{Type}]\ (l \cong l\ {:}{+\!\!+}{:}'\ [])$. Then the equality proof can be used to "rewrite" types. Figure 2.3 shows the complete implementation of the alternative version of $(+\!\!+\!\!+)$. The function *lem_neutApp* builds a proof of $l \cong l\ {:}{+\!\!+}{:}'\ []$ given a term of type *HList* l. The proof is built recursively in the list argument, and corresponds to prove by induction over l. Then, the lemma is used in the implementation of $(+\!\!+\!\!+)$ with the *castWith* function. Analogously the lemma *lem_snocApp* is used to type the recursive case. Figure 2.4 shows the types of some combinators to manipulate equality.

Of course, all this "fake" dependently typed programming has a big hole: GHC does not run a termination checker, which means that the whole logic is inconsistent because we can write loops as proofs, like the following:

$$\begin{array}{l}\mathit{bottom} :: \forall\ a.a\\ \mathit{bottom} = \mathit{bottom}\end{array}$$

This does not mean proving over Haskell is not useful at all, but programmers should be careful to dodge unproductive proofs.

2.2.1 Type-level programming design patterns.

In this section we discuss a set of specific techniques used by Haskell programmers to emulate dependently-typed features.

2.2.1.1 Singletons.

In a full spectrum dependently typed language such as Agda [Bove et al., 2009], Coq [Bertot and Castéran, 2013] or Idris [Brady, 2013], the distinction between types and terms is blurred. To simulate dependent types in Haskell, where types and terms are still separated we need to find some way to relate the type and value representations of data types.

To show that, let us define a less noisy data type example, as the natural numbers:

```
lem_neutApp
  :: HList l → l ≅ l :⧺:' []
lem_neutApp HNil
  = Refl
lem_neutApp (x ::: xs)
  = let r = lem_neutApp xs
    in apply Refl r


th_snocApp
  :: HList l → a → HList m →
       HList ((l :⧺:' [a]) :⧺: m)
     ≅ HList (l :⧺: (a : m))
th_snocApp HNil a l = Refl
th_snocApp (x ::: xs) a l
  = let k = th_snocApp xs a l
        s = inner k
    in  apply Refl (apply Refl s)
```

```
snoc :: HList l → a
     → HList (l :⧺:' [a])
snoc HNil a
     = a ::: HNil
snoc (x ::: xs) a
     = x ::: snoc xs a


(⧻) :: HList l → HList m
  → HList (l :⧺: m)
l ⧻ HNil
  = castWith (apply Refl
               (lem_neutApp l))
  $ l
l ⧻ (x ::: xs)
  = castWith (th_snocApp l x xs)
  $ snoc l x ⧻ xs
```

Figure 2.3: Alternative implementation of *append*.

```
sym      :: (a ≅ b) → b ≅ a
trans    :: (a ≅ b) → (b ≅ c) → a ≅ c
castWith :: (a ≅ b) → a → b
apply    :: (f ≅ g) → (a ≅ b) → f a ≅ g b
inner    :: (f a ≅ g b) → a ≅ b
outer    :: (f a ≅ g b) → f ≅ g
```

Figure 2.4: Propositional equality manipulation.

```
data Nat = Z | S Nat
```

We assume we have the `DataKinds` extension enabled and therefore we have available both the type *Nat* and the kind *Nat*. Let us define type-level addition:

```
type family m :+ n where
  Z     :+ n = n
  (S m) :+ n = S (m :+ n)
```

Now we want to prove that Z is the right neutral element of (:+). In other words, let us try to build an inhabitant of the type m :+ $Z \cong m$ for any m. This is done using structural induction, but how do we apply induction? We need a way to pattern match over m, but m is a type, and pattern matching is done over terms. The solution is to build *singleton* types. A singleton type is a GADT indexed by a type we are trying to represent dynamically. There is exactly one inhabitant for each index[3]. The definition is given as follows:

```
data SNat (n :: Nat) where
  SZ :: SNat Z
  SS :: SNat n → SNat (S n)
```

The constructor *SZ* is the value representation of the type Z. Given a term representation of the type n of kind *Nat*, *SS* builds the term representation for the type $(S\ n)$.

Using this machinery, when matching over terms of type *SNat* n we do it over the type n at the same time. The following proof can then be written:

```
th_plusNeut :: SNat n → n :+ Z ≅ n
th_plusNeut SZ     = Refl
th_plusNeut (SS n) = apply Refl (th_plusNeut n)
```

Singletons can be used any time we need dynamic information over *Nat*s. For instance, the following is an implementation of *take* for *HList*s:

[3] This is true if we do not consider bottom, of course.

```
type family Take (n :: Nat) (l :: [Type]) where
  Take Z      _        = []
  Take _      '[]      = []
  Take (S n) (t ': ts) = t ': Take n ts
hTake :: SNat n → HList l → HList (Take n l)
hTake SZ _              = HNil
hTake _  HNil           = HNil
hTake (SS n) (x ::: xs) = x ::: hTake n xs
```

We defined the type-level function *Take*. The value-level function *hTake* takes a natural number represented with a singleton, and a list. Note that the first argument cannot be just of type *Nat*, otherwise there is no way to refer to that argument in the return type.

Sometimes we do not need to define singletons at all, but just apply the idea. For instance consider the type of a function we defined before:

```
lem_neutApp :: HList l → l ≅ l :++:' []
```

We used a full heterogeneous list as an argument, while we just needed the type l. While *HList* is not a singleton type it plays the same role here: we use the list to pattern match over the type l.

2.2.1.2 Proxies.

In some contexts we do not need all the runtime information of a data type dynamically, but just a value that carries a type to use it in compile time. A proxy is a value isomorphic to () (the unit value) at the level of values, but indexed by a certain type. We define it as follows:

```
data Proxy (a :: k) :: Type where
  Proxy :: Proxy a
```

The *Proxy* type is kind polymorphic; if the k variable is instantiated then the a variable can be instantiated only to a type of that kind. We can, for instance, instantiate k to *Nat* and a to $S\ Z$.

```
proxy1 = Proxy @ (S Z)
```

where the operator (@) denotes type application. Then the value *proxy1* can be used to fix types at any place. We can for instance implement the following:

```
class TakeI (n :: Nat) (l :: [Type]) where
  takeI :: Proxy n → HList l → HList (Take n l)
instance TakeI Z l where
  takeI _ _ = HNil
instance TakeI n '[] where
  takeI _ _ = HNil
instance TakeI n l
   ⇒ TakeI (S n) (t': l) where
  takeI _ (x ::: xs)
     = x ::: takeI (Proxy @ n) xs
```

This is another way to write a dependent function. In some way, this is the version of *take* using an implicit argument, hence its name. The expression (*takeI proxy1 l*) has type (*HList* '[*Char*]), just as (*hTake* (*SS SZ*) *l*). They are similar in complexity: *hTake* does recursion over the structure of the natural number in runtime, while *takeI* does it over the dictionaries used in the implementation of type classes.

Singletons and proxies are not always interchangeable. Singletons are stronger, allowing programmers to manipulate dynamic data. They have the disadvantage that they need to be built, while a proxy can be trivially constructed if its index is in scope. In `AspectAG`, we use proxies extensively, which makes sense since the library users specify a static structure.

Proxies are useful in other idioms. For instance, they can also act as an explicit application of a type to a polymorphic expression.

2.2.1.3 Class-directed programming.

The definition of *takeI* shows a technique to define dependently-typed Haskell functions. With no singleton GADT for natural numbers to pattern match against, the way to pattern match against the arguments is to define a type class declaration. This should not be a surprise: type classes are the way of implementing ad-hoc polymorphism, and *takeI* is just a polymorphic function with parameters n and l.

This idiom is sometimes used even when we have the GADT available. In Figure 2.5 another definition of append is shown.

There are advantages of doing so. One is to define the related type families as indexed type families as we do with (:++:), keeping everything in the same

```
class Append (l :: [Type]) (m :: [Type]) where
  type l :++: m :: [Type]
  (++) :: HList l → HList m → HList (l :++: m)
instance Append '[] m where
  type '[] :++: m = m
  HNil ++ m = m
instance
  Append l m ⇒ Append (t ': l) m where
  type (t ': l) :++: m = t ': l :++: m
  (x ::: xs) ++ m = x ::: xs ++ m
```

Figure 2.5: Definition of *append* using type classes.

place. Another reason is that when we declare the instances, all the information of the refined types is in scope with names that can be used (if the extension `ScopeTypeVariables` is enabled), as we do in the recursive case of *takeI* with the parameter n. The main disadvantage, on the other hand, is verbosity with a somewhat heavy notation.

2.2.1.4 Advanced overlap.

Type classes simulate pattern matching over type-level data, but this idiom has some disadvantages. Overlapping patterns are handy, but GHC does not allow overlapping instances (and this is good for many reasons out of our scope). In Figure 2.6 a function extracting all booleans from an *HList* is defined. Unfortunately, it does not compile since the last two instances do overlap when the first type of the heterogeneous list is *Bool*. One could think of adding a constraint such as $(a == \mathit{Bool} \sim {}'\mathit{False})$ in the last declaration, but the problem would persist. Type class resolution does not backtrack. The type checker only decides upon head declarations. There is no way to decide instances depending on their context. The solution is to put that information in the head, as an extra parameter.

In Figure 2.7 we implement a working solution. The *GetBoolsR* class has an extra parameter indicating if the head type of a non-empty heterogeneous list is *Bool*. The proper class implementing the function, *GetBools* is a wrapper, depending on *GetBoolsR* with the suitable instance. The $(==)$ type family

```
class GetBools (l :: [Type]) where
  getBools :: HList l → [Bool]
instance GetBools '[] where
  getBools _ = []
instance GetBools l ⇒ GetBools (Bool ': l) where
  getBools (x ::: xs) = x : getBools xs
instance GetBools l ⇒ GetBools (a ': l) where
  getBools (x ::: xs) = getBools xs
```

Figure 2.6: Definition of *getBools* not compiling due to overlapping instances.

```
class GetBools (l :: [Type]) where
  getBools :: HList l → [Bool]
instance GetBools '[] where
  getBools _ = []
instance (GetBoolsR (t == Bool) (t ': ts)) ⇒ GetBools (t ': ts) where
  getBools = getBoolsR (Proxy @ (t == Bool))


class GetBoolsR (headIsBool :: Bool) (l :: [Type]) where
  getBoolsR :: Proxy headIsBool → HList l → [Bool]
instance GetBools l ⇒ GetBoolsR 'True (Bool ': l) where
  getBoolsR _ (b ::: xs) = b : getBools xs
instance GetBools l ⇒ GetBoolsR 'False (t ': l) where
  getBoolsR _ (b ::: xs) = getBools xs
```

Figure 2.7: Definition of *getBools*.

is exported by the *Data.Type.Equality* and works just as the term-level (==) function.

2.3 The expression problem.

The expression problem is a challenge to the expressiveness of programming languages. The name was given by Wadler [Wadler, 1998] in a famous mailing list post, though the concept was already known at that time, discussed, for instance, by Reynolds [Reynolds, 1978].

In Wadler's words: *"The Expression Problem is a new name for an old problem. The goal is to define a data type by cases, where one can add new cases to the data type and new functions over the data type, without recompiling existing code, and while retaining static type safety (e.g., no casts)."*

The expression problem is stated in terms of data types, but that is just another way to talk about languages: algebraic data types and context-free grammars are in some sense equivalent formalisms, where a data type term is equivalent to a corresponding parse tree.

There are two orthogonal ways a language can be extended: new syntax constructs (new cases in a data type) and new semantics (new functions consuming it). The expression problem consists in finding a way to define a formalism in which languages can be extended easily in both directions.

Traditionally object-oriented programming and functional programming shine solving extensionality in one direction, failing the other way.

Given an algebraic data type in a functional programming language, it is easy to define new functions over it, but if we wish to add a new construct, we must modify the proper data type declaration and every function consuming it must be potentially refactored to handle the extra case.

In object-oriented programming, algebraic data types can be implemented using a class (for instance using a composite pattern [Gamma et al., 1994]). New classes can be easily extended to the hierarchy in new modules. But to add a new method means that we need to potentially modify and recompile every class.

AGs are one of the proposals to tackle the expression problem. Given an AG defining a language, it is natural to add new constructs by adding productions (and appropriate rules to compute the already defined attributes). It is also natural to define new semantics by adding new attributes.

There is a significant number of AG implementations. Some implementations are standalone compilers or generators like LRC [Saraiva, 2002], UUAGC [Swierstra et al., 1999], LISA [Mernik and Žumer, 2005], JastAdd [Ekman and Hedin, 2007] or Silver [Van Wyk et al., 2010]. Others are embbeded in languages like Scala [Sloane et al., 2009] or Haskell [de Moor et al., 1999, de Moor et al., 2000, Viera et al., 2009, Viera et al., 2018, Martins et al., 2013, Balestrieri, 2015].

2.4 EDSLs and the `AspectAG` library.

Domain-specific Languages (DSLs, for short) [Fowler, 2010] are computer languages specialized in a particular domain. In contrast, usual general-purpose programming languages aim to cover a broad domain set. When using a DSL, solutions are expressed in domain terms, making them easier to understand for any programmer and approachable even for domain experts that are non-programmers.

DSLs can be implemented as a standalone language, introducing a complete compiler toolchain, or embedded as a library in a host language (what is usually called an embedded DSL, EDSL for short). The embedded approach has its advantages. One of them is that all constructs of the host language and its libraries are available to users. Also, the amount of work required compared to the standalone approach is minimal. In higher-order functional programming languages such as Haskell, the embedded approach is commonly used and successful.

Type-level programming in Haskell offers convenient features to support EDSLs since complex properties can be encoded in types. It allows the EDSL developer to precisely specify the domain modeled, checking properties of the client code at compile time.

The embedded approach has drawbacks as well. Error handling is perhaps one of the main ones. When type errors are raised, they do not deliver error messages that refer to domain terms, but to host language aspects. This makes errors hard to understand, especially for users that are not experts in the host programming language.

Additionally, it is usual that error messages leak implementation details, breaking all abstraction mechanisms that may have been considered when building the library. The leaking problem is even worse if we use type-level programming techniques to implement the EDSL. Type-level programming historically evolved in a somewhat unplanned way. For that reason, type-level Haskell lacks some desirable abstraction features. For instance, types are normalized before they are printed in an error message, showing unfolded definitions that perhaps the library designer wants to hide.

`AspectAG` is a Haskell EDSL, introduced by Viera *et al* [Viera et al., 2009], that implements first-class AGs. By first-class, we mean that all pieces of the AG are Haskell values (and types) and can be manipulated as such.

Initially, `AspectAG` used extensible polymorphic records and predicates encoded using old fashioned type-level programming features, such as Multi-parameter type classes [Peyton Jones et al., 1997] and functional dependencies [Jones, 2000], to ensure well-formedness of AGs at compile time.

In `AspectAG` users define types for every building block an attribute grammar has, such as productions, children, and attributes. Collections of attributes, called *attributions* are defined as extensible records mapping attribute names to values. Semantic rules are implemented as functions from a so-called *input family* (inherited attributes of the father and synthesized ones from children) to an *output family* (synthesized attributes of the father and inherited ones to children). This definition is more general than the one we give in Section 2.1. In `AspectAG` there is exactly one rule per production. However, note that this is just an implementation detail. Rules to compute each individual attribute are a particular case and they are still the ones defined in practice when using `AspectAG`. Then they are combined to form those bigger rules.

Sets of rules are combined into *aspects*, which are implemented as mappings from productions to rules. *Semantic functions* take aspects and some representation of parse trees to compute the global output family, and implicitly the full attribute evaluation.

The use of extensible records means every attribute used in every rule is known at compile-time, so the type system checks if a grammar is well defined. Users do not explicitly define attribute occurrences. Instead, users define computation rules, and the semantic function checks that all dependencies are fulfilled. This approach helps modularity since dependencies are lazily inferred from the set of rules used in each case.

`AspectAG` was tested in concrete case studies such as a modular compiler for a dialect of the Oberon language [Viera and Swierstra, 2015]. While it proved to be useful, type errors were the main weakness. The fact that an

AG can be easily ill-formed due to its entangled structure, and the use of an embedded language using type-level programming explains this.

Chapter 3

Kind safe `AspectAG` - overview.

In this chapter, we explain the main features of the `AspectAG` library using a running example. We start with a simple expression language consisting of integer values, variables, and the addition operation. We endow the language with semantics, implementing an evaluator. After that, to show how `AspectAG` tackles the expression problem we extend the language in two ways: semantically, defining new ways to compute from expressions; and syntactically, adding a new construct: function calls. Finally, we add a polymorphism mechanism, which allows us to define a family of languages parametrized by their terminal symbols.

This chapter is partially based on the overview given in [García-Garland et al., 2019], though the library has evolved. The most notable additions from then on are what we call the "deforested" approach (already used in [Viera and Swierstra, 2015]) and the addition of polymorphism.

This chapter is a literate program [Knuth, 1984].[1] To show how `AspectAG` supports compositional compiler construction each of the sections 3.1, 3.2, 3.3, and 3.5 compiles as an independent module. Section 3.4 is divided among two modules.

As in every Haskell module part of a bigger system, we must declare each module giving it a name, and a list of module imports (at least the `AspectAG` library modules in this case). Usually, the user also declares a set of pragmas, like the set of extensions to use. We omit import headers and pragma declarations for brevity. In Figure 3.1 the dependencies among modules are represented (an arrow from A to B means A imports B).

[1] The source code can be found at `https://gitlab.fing.edu.uy/jpgarcia/tesis` in the directory `./AAGExample`

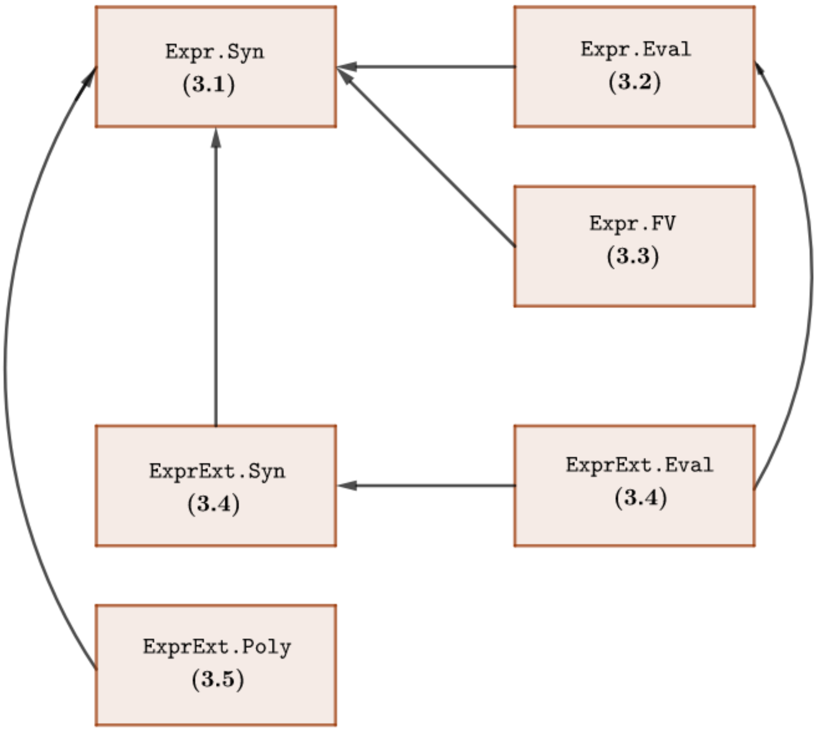


Figure 3.1: Module dependencies for the language of expressions.

In Section 3.1 we show how to declare a grammar for the expression language in terms of `AspectAG`. In Section 3.2 we specify a semantics for the language. In Section 3.3 we orthogonally define a new semantics. In Section 3.4 we show how we can add new syntax in a modular way, also defining semantics for it. Finally, in Section 3.5 we explore how polymorphism interacts with `AspectAG` definitions.

3.1 Grammar specification.

The abstract syntax of our expression language is given by the following grammar:

$$expr \rightarrow Int_{val}$$
$$expr \rightarrow String_{var}$$
$$expr \rightarrow expr_l + expr_r$$

where *Int* and *String* are terminals corresponding to integer values and variable names, respectively. In the third production, there are two non-terminal symbols. The indexes (l, r, var, val) are names for each symbol occurrence (children) so that we can refer to each one unambiguously.

We declare the syntax of the expression language in the `Expr.Syn` module. The `Language.Grammars.AspectAG` module contains all the constructs of the `AspectAG` EDSL. The `Language.Grammars.AspectAG.TH` module contains some useful Template Haskell [Sheard and Jones, 2002] utilities to generate some boilerplate for us. We import both, but it is relevant to note that `AspectAG` can be used alone with no Template Haskell requirement. `AspectAG` modules can be plain Haskell modules with no preprocessing.

To declare a grammar in `AspectAG` we need to declare all the ingredients: terminal and nonterminal symbols, productions, and children. Since our goal is to use all this information at compile-time, the information will be explicit at the type level. At the value level, all these objects are trivial, used only to be passed as arguments. In `AspectAG` all these objects are of type *Label*, which is defined exactly as the *Proxy* type. We do not use *Proxy* directly just to use a more mnemonic name.

In `AspectAG`, the specified grammar is declared as follows. First, we declare the non-terminal for expressions:

$$
\begin{aligned}
&\textbf{type}\ Nt_{Expr} = \text{`}NT\ \texttt{"Expr"} \\
&nt_{Expr} = Label\ @\ Nt_{Expr}
\end{aligned}
$$

Non-terminals are defined by introducing a name (like `"Expr"`) at the type level using a promoted string literal (which has kind *Symbol*, a special built-in kind for type-level strings). We use the `TypeApplications` extension of GHC to associate type information to a label. The @ character denotes type application.

Productions are also identified by a name and they are related to a non-terminal, the one on the left-hand side of the production:

$$
\begin{aligned}
&\textbf{type}\ P_{Val} &&= \text{`}Prd\ \texttt{"Val"}\ Nt_{Expr} \\
&p_{Val} &&= Label\ @\ P_{Val} \\
&\textbf{type}\ P_{Var} &&= \text{`}Prd\ \texttt{"Var"}\ Nt_{Expr} \\
&p_{Var} &&= Label\ @\ P_{Var} \\
&\textbf{type}\ P_{Add} &&= \text{`}Prd\ \texttt{"Add"}\ Nt_{Expr} \\
&p_{Add} &&= Label\ @\ P_{Add}
\end{aligned}
$$

The last ingredient of the grammar declaration is given by the introduction of the children that occur in the productions:

ch_{Add_l} $= Label$ @ ('Chi "Add_l" P_{Add} ($NonTerminal$ Nt_{Expr}))
ch_{Add_r} $= Label$ @ ('Chi "Add_r" P_{Add} ($NonTerminal$ Nt_{Expr}))
$ch_{Val_{val}}$ $= Label$ @ ('Chi "Val_val" P_{Val} ($Terminal$ $Integer$))
$ch_{Var_{var}}$ $= Label$ @ ('Chi "Var_var" P_{Var} ($Terminal$ [$Char$]))

Each child has a name and it is related to a production. Children can be either non-terminals or terminals. In the latter case, the type of values they denote must be provided. Note that any type can be used as a terminal. In this case, we use types with an infinite number of inhabitants. This is natural in an implementation but not necessary as an abstract definition of languages. We incidentally reduced the gap when we defined grammars such as in Chapter 2.

Summing up the provided information, we can see that our grammar declaration indirectly contains the same information that would be given in the Haskell data type representation of its abstract syntax tree:

```
data Expr
  = Add {l :: Expr, r :: Expr}
  | Var {var :: String}
  | Val {val :: Integer}
```

By defining this data type we have a way to embed terms of the expression language in Haskell. Then, we can consume those terms according to a semantic specification. So far there is no relationship defined among the labels previously introduced and the data type *Expr*. The relationship between `AspectAG` definitions and a data type representation is given by the semantic functions. The following code defines the semantic function for the expression language:

```
sem_Expr asp (Add l r)
  = knitAspect p_Add asp $ ch_Add_l .=. sem_Expr asp l
                         .*. ch_Add_r .=. sem_Expr asp r
                         .*. emptyGenRec
sem_Expr asp (Var var)
  = knitAspect p_Var asp $ ch_Var_var .=. sem_Lit var
                         .*. emptyGenRec
sem_Expr asp (Val val)
  = knitAspect p_Val asp $ ch_Val_val .=. sem_Lit val
                         .*. emptyGenRec
```

The semantic function sem_{Expr} takes the argument *asp*, an *aspect* (a semantics definition) and an expression represented by a tree of type *Expr*, to build a function from Attributions (inherited attributes, an environment) to Attributions (synthesized attributes, which correspond to the computation).

It is not even explicit from the definition that a function is built. To understand it, consider that *knitAspect* takes four arguments. Details of how *knitAspect* is implemented are given in Section 5.3.6.

What should be clear from this definition is that the relationship between labels defined with `AspectAG` constructs and the specified grammar (and the *Expr* data type) is now explicit. For instance, in the first clause, where we are taking an AST representing an addition, we are stating the following:

- The constructor *Add* corresponds to the production p_{Add}.
- The subexpressions l and r in the data type correspond to non-terminals ch_{Add_l} and ch_{Add_r} in the `AspectAG` definition.
- Subexpressions will be processed recursively with the same semantic function.

The definition of the grammar is now complete. If the user prefers, there is a way to define it in a form more similar to an EBNF notation. All the previous code as it is can be generated by Template Haskell splices which are special expressions that evaluate at compile time to generate Haskell source code at that point of the program they occur.

The following splices generate the labels:

```
$ (addNont "Expr")
$ (addProd "Val" Nt_Expr [("val", Ter     "Integer)])
$ (addProd "Var" Nt_Expr [("var", Ter     "String)])
$ (addProd "Add" Nt_Expr [("l",   NonTer Nt_Expr),
                          ("r",   NonTer Nt_Expr)])
```

The following splices generate the data type *Expr*, and the semantic function sem_{Expr}, respectively:

```
$ (closeNT Nt_Expr)
$ (mkSemFunc Nt_Expr)
```

To extend the definition of the grammar in `AspectAG` by adding new productions we can add suitable labels. This can be done even in a new module as we will do in Section 3.4 in the module `ExprExt.Syn`. However, this approach has an issue: the grammar as a collection of labels is open, but when we build a Haskell data type such as *Expr*, since Haskell data types are closed, there is no way to extend it.

The solution we present in [García-Garland et al., 2019] is to make an updated "copy" of *Expr* and sem_{Expr} in the new module with the new constructors either by hand or by calling the *closeNT* and *mkSemFunc* splices there. We avoid ambiguity by using qualified names: *AST.Expr* is the data type for the original grammar, while *ASTExt.Expr* is the extended data type.

This duplication is not nice. Someone could argue that our approach to tackling the expression problem is avoiding recompilation by actually rewriting the compiled code. This is partially true, but still, this approach has something to give us. Firstly, semantic definitions will be reusable for both versions of the grammar. As we shall see later in Section 3.2 rules to compute aspects are tied (by their type) to a production. They can be used in any grammar where that production exists. Secondly, using Template Haskell splices this duplication is hidden to the programmer, offering a solution similar to libraries like uuagc [Swierstra et al., 1999]. Still, we can push in this direction, as we will see in Section 3.6.

Note that this grammar is similar to the one declared in Chapter 2, but there is no start symbol declared. `AspectAG` handles the start symbol and the attribute occurrences implicitly.

3.2 Semantics definition.

In this section we will show how to define concrete instances of semantics in `AspectAG`.

With the aim to provide semantics, AGs decorate the productions of context-free grammars with attribute computations. In an expression language as the one defined, the usual semantics is the evaluation semantics. This can be defined by using two attributes: *eval*, to represent the result of the evaluation, and *env*, to distribute the environment containing the values for variables. In this section we explain how the evaluation semantics can be implemented using our library. All the source code of this section is part of the `MF.Sem` module.

Attributes in `AspectAG` are also defined as *Label*s. They have attached the type of the computation that they represent.

$eval = Label\ @\ (\text{'}Att\ \texttt{"eval"}\ Integer)$
$env\ \ = Label\ @\ (\text{'}Att\ \texttt{"env"}\ \ Env)$

The attribute *eval* computes a value of type *Integer*, while the attribute *env* computes an environment of type *Env*.

The environment contains the value of the variables in a scope. We use the *Map* datatype provided by the GHC's `base` library (qualified as *M*) to represent environments:

type $Env = M.Map\ String\ Integer$

There is no information specifying that *eval* and *env* are respecively a synthesized and inherited attributes. In `AspectAG` attributes are just declared giving a name and a type. They can be used in any role.

Now we can build the proper rules. To compute the attribute *eval* in the production p_{Add} we take the values of *eval* at each child, which are the integers denotated by the subexpressions, and sum them up. This is written as follows:

$$\begin{aligned} add_{eval} = syndefM\ eval\ p_{Add}\ \ &\$\ \ (+)\ @\ Integer \\ &<\$>\ at\ ch_{Add_l}\ eval \\ &\circledast\ \ at\ ch_{Add_r}\ eval \end{aligned}$$

The name add_{eval} is just a Haskell identifier. We adopt this name convention for rules: the uncapitalized name of the production followed by the name of the attribute.

The rule is defined using the function *syndefM*, which is the library operation to define synthesized attributes. It takes an attribute label (the one for which the semantics is being defined), a production label (where it is being defined), and the respective computation rule for the attribute (in a Reader monad).

Using an applicative interface, the computation means that we take the values of the *eval* attibute at children ch_{Add_l} and ch_{Add_r}, and combine them with the operator $(+)$. In the expression $at\ ch_{Add_l}\ eval / at\ ch_{Add_r}\ eval$ we pick up the attribute *eval* from the collection of synthesized attributes of the child ch_{Add_l}/ch_{Add_r}. We refer to these collections of attributes as *attributions*.

This definition expects that the *eval* attribute is defined in those subexpressions. This is explicit on its type as a constraint. Rules keep at type level all the information about attributes and children referred. The whole structure of the grammar is known at compile time, which is used to compute precise errors, for instance if a subexpression has no rule to compute *eval*, a suitable type error will be raised, as we shall see later.

In the case of literals, semantics is given by the integer that the literal denotes. This is defined as follows:

$$val_{eval} = syndefM\ eval\ p_{Val}\ (ter\ ch_{Val_{val}})$$

At the p_{Val} production the value of the terminal corresponds to the semantics of the expression. In terms of our implementation the attribute *eval* is defined as the value of the terminal $ch_{Val_{val}}$. The combinator *ter* is simply a reserved keyword in our EDSL used to access the terminal value.

Finally, with variables, the *eval* attribute is computed by looking up the value of the variable in the environment.

```
var_eval = syndefM eval p_Var (do
                env ← at lhs env
                x   ← ter ch_Var_var
                case M.lookup x env of
                   Just e → return e)
```

We take the inherited environment and the variable name. Then we lookup for the value associated with that name. Here we use do notation, i.e. a monadic interface. The name *lhs* is a reserved word like *ter* indicating that we receive the *env* attribute from the parent[2]. Note that the use of the monadic notation in the case of var_{eval} is just a matter of convenience. Users can use any style (monadic or applicative) they want to.

The reader may be concerned with the pattern matching we do in the result of the lookup as it is not exhaustive. In fact, if the environment does not contain the variable we are looking for this would generate a runtime failure. For brevity, we will not handle name binding in this example language, but it can be handled with another attribute in an orthogonal way (for instance, in a new, independent module). We will see more details in chapter 6.

We can combine all these rules in an *aspect*:

```
asp_eval = traceAspect (Proxy @ ('Text "eval")) $
                add_eval ◁ val_eval ◁ var_eval ◁ emptyAspect
```

[2] The LHS names an idiom widely used in AG systems. Note that parents are on the left-hand side of the production in the EBNF notation

The operator ($\triangleleft$) is a combinator that adds a rule to an aspect (($\triangleleft$) associates to the right). An *aspect* is a collection of rules. Here we build an aspect with all the rules for a given attribute, but the users can combine them in the way they want (for example, by production). Aspects can be orthogonal among them, but this is not required.

The function *traceAspect* and also -implicitly- each application of *syndefM* tag definitions to show more information on type errors. This is useful to have a hint where the error was actually introduced. For instance, note that asp_{eval} clearly depends on an attribute *env* with no rules attached to it at this point, so it is not -yet- useful at all. We cannot decide locally if the definition is wrong since the rules for *env* could be defined later (as we will do), or perhaps in another module that the compiler does not even know about yet! If we use asp_{eval} calling it on a semantic function there will be a type error but it will be raised on the semantic function application, once some of the constraints about attribute and children dependencies fail to be solved. Programmers not used to library internals may find this misleading. Showing the trace is helpful as we will see in Chapter 4. A function *traceRule* is also provided so the user can apply it to rules instead of aspects. Actually, *traceAspect* is a sort of mapping of *traceRule* to every rule contained in the aspect.

To define the inherited attribute *env* we use de *inhdefM* combinator. It takes three labels: an attribute, a production where the rule is being defined, and a child to which the information is being distributed. Then, the rule is defined. In our example we define rules add_{env_l} and add_{env_r} to copy the environment to children in the case of additions:

$$add_{env_l} = inhdefM\ env\ p_{Add}\ ch_{Add_l}\ (at\ lhs\ env)$$

$$add_{env_r} = inhdefM\ env\ p_{Add}\ ch_{Add_r}\ (at\ lhs\ env)$$

Then, we can combine those rules in an aspect. We add trace information again.

$$asp_{env} = traceAspect\ (Proxy\ @\ (\text{'}Text\ \texttt{"env"}))\ \$\ add_{env_l} \triangleleft add_{env_r} \triangleleft emptyAspect$$

Finally we can combine aspects asp_{eval} and asp_{env}. For that puropose `AspectAG` provides the ($\bowtie$) operator.

$$asp_{sem} = asp_{eval} \bowtie asp_{env}$$

Note that in this case no new tag was added to the trace, but we could do it if we want to.

Finally we can implement the evaluation function. Let us work with the forestated approach, taking an *Expr* tree as an argument:

$$\begin{array}{l} \mathit{evaluator} :: \mathit{Expr} \rightarrow \mathit{Env} \rightarrow \mathit{Integer} \\ \mathit{evaluator}\ \mathit{exp}\ \mathit{envi} = \\ \quad \mathit{sem}_{\mathit{Expr}}\ \mathit{asp}_{\mathit{sem}}\ \mathit{exp}\ (\mathit{env}\ \ =.\ \mathit{envi} *.\ \mathit{emptyAtt})\ \#.\ \mathit{eval} \end{array}$$

The $\mathit{sem}_{\mathit{Expr}}$ function takes the semantic definition and an expression to build a semantic function. Semantic functions take an attribution (inherited attributes for the expression) and return an attribution (synthesized attributes). The expression $\mathit{env}\ \ =.\ \mathit{envi} *.\ \mathit{emptyAtt}$ is the inherited attribution at the root, consisting in only one attribute, *env* with the value of *envi*. The expression $\mathit{env}\ \ =.\ \mathit{envi}$ is building an attribute while $(*.)$ is a cons-like operator. So, the expression $\mathit{sem}_{\mathit{Expr}}\ \mathit{asp}_{\mathit{sem}}\ \mathit{exp}\ (\mathit{env}\ \ =.\ \mathit{envi} *.\ \mathit{emptyAtt})$ computes the synthesized attributes of the root. In this case, it is an attribution with only one attribute *eval* containing the result of the computation. The operator $(\#.)$ extracts it.

3.3 Semantics extension and modification.

Our approach allows the users to define alternative semantics or extending already defined ones in a simple and modular way. For instance, suppose that we want to collect the free variables in an expression (this means all variables, since there are no binders in our expression language). We collect them in the order they occur, if they can occur many times, they will appear many times in the result.

Let us define this new interpretation in a new module called `Expr.FV`. It depends on the `Expr.Syn` module but it is completely independent of the `Expr.Eval` module.

Now, let us define an attribute to collect the variables:

$$\mathit{fv} = \mathit{Label}\ @\ (\text{`}\mathit{Att}\ \texttt{"fv"}\ [\mathit{String}])$$

We define the aspect with the rules to compute the new attribute:

$$\begin{aligned}
asp_{fv} = {} & syndefM\ fv\ p_{Add}\ ((+\!\!+)\ \text{<\$>}\ at\ ch_{Add_l}\ fv \circledast at\ ch_{Add_r}\ fv) \\
& \lhd\ syndefM\ fv\ p_{Var}\ ((:[\,])\ \text{<\$>}\ ter\ ch_{Var_{var}}) \\
& \lhd\ syndefM\ fv\ p_{Val}\ (pure\ [\,]) \\
& \lhd\ emptyAspect
\end{aligned}$$

In this case we defined and combined all rules in the same expression. In the p_{Add} production we concatenate variables occuring in each child. In the p_{Var} expression we obtain the terminal representing the variable and wrap it in a singleton list. In the case of the p_{Val} production we simply return the empty list.

Again, we can implement a traversal to collect all free variables of an expression by applying the defined aspect to the semantic funcion:

$$freeVars\ e = sem_{Expr}\ asp_{fv}\ e\ emptyAtt\ \#.\ fv$$

We can also modify already defined semantics definitions in a modular way. Suppose that, instead of collecting all variable occurences, we only want to collect the first one (i.e. get all occuring variables once, in their occuring order). We can take the asp_{fv} aspect and modify the rule for the p_{Add} expression as follows:

$$\begin{aligned}
asp'_{fv} = {} & synmodM\ fv\ p_{Add}\ (Data.List.union\ \text{<\$>}\ at\ ch_l\ fv \circledast at\ ch_r\ fv) \\
& \lhd\ asp_{fv}
\end{aligned}$$

This redefinition of the semantics can be defined in a new module (in this case depending on `Expr.Sem` since asp_{fv} must be imported) with no need to recompile the original one.

3.4 Syntactic Extension.

In the previous section we showed how to extend a language with new semantics in `AspectAG`, but in fact, that is not a great accomplishment since the host language is already purely functional, and purely functional languages solve well that axis of the expression problem.

In this section, we show how users of the library can extend the syntax of the language. Suppose that we want to add a new syntactic construct: function calls. In other words, we add a new production to the grammar defining the expression language. Using the EBNF notation this new production can be written as follows:

$expr \rightarrow fname\ (expr)$

An expression can be a function application, consisting in a function name and an argument expression. We implement this in a new module `ExprExt.Syn`.

We declare the required labels as follows:

```
type P_Call      = 'Prd "Call" Nt_Expr
p_Call           = Label @ P_Call
ch_Call_fun      = Label @ ('Chi "Call_fun" P_Call (Terminal     String))
ch_Call_arg      = Label @ ('Chi "Call_arg" P_Call (NonTerminal Nt_Expr))
```

We add a production label p_{Call}. The `"Call"` production is a rewrite rule for the non terminal Nt_{Expr}. It has two children. The $ch_{Call_{fun}}$ child is a terminal of type *String*, representing the name of the applyed function. The $ch_{Call_{arg}}$ child is a non terminal, the argument expression.

Recall that we can add the production using the Template Haskell splices instead of explicitly defining the labels:

```
$ (addProd "Call" ''Nt_Expr [("fun", Ter      ''String),
                             ("arg", NonTer ''Nt_Expr)])
```

We can also reify the grammar in scope as a Haskell data type using a splice with *closeNT*, and generate the semantic functions with a splice with *mkSemFunc*.

```
$ (closeNT      Nt_Expr)
$ (mkSemFunc Nt_Expr)
```

The derived data type is a new copy of the *Expr.Syn.Expr* data type, with the new constructor.

```
data Expr
  = Add  {l :: Expr, r :: Expr}
  | Var  {var :: String}
  | Val  {val :: Integer}
  | Call {fun :: String, arg :: Expr}
    deriving (Show, Eq, Read)
```

Now, it is time to add semantics to the extended language. The following code is part of a new module `ExprExt.Eval`. We extend the evaluation semantics with function calls. In our expression language there is no way to define functions yet, so let us assume that there is a set of known built-in functions. It makes sense to make them available in an environment at a given scope, so it could be extended with new definitions in a future extension.

To minimize the complexity of the running example we keep it simple here. Consider an environment of functions named by Strings. We use a call-by-value regime.

Let us define an inherited attribute for this environment:

$$\mathit{fenv} = \mathit{Label}\ @\ (\text{`}\mathit{Att}\ \texttt{"fenv"}\ (\mathit{M.Map\ String}\ (\mathit{Integer} \to \mathit{Integer})))$$

We must pass down this environment from parents to children in each non-terminal. For instance, we can implement the rule for the left argument of the sum as:

$$\mathit{add}_{\mathit{fenv}_l} = \mathit{inhdefM}\ \mathit{fenv}\ p_{\mathit{Add}}\ \mathit{ch}_{\mathit{Add}_l}\ (\mathit{at}\ \mathit{lhs}\ \mathit{fenv})$$

Note that this is almost the same definition as in $\mathit{add}_{\mathit{env}_l}$ in Section 3.2. We are only passing the attribute untouched from the parent to the children. One big advantage of using the AG System as an EDSL is that we can abstract this pattern easily as a Haskell function, and then build a rule such as for any argument given. We can, for instance, build a function that given an attribute, builds an aspect with rules that copy that attribute in each recursive definition. Moreover, we can for instance define a copy rule to all recursive children and use *inhmodM* exactly in those special productions where the environment is modified. We will see examples in Chapter 6.

We must also note that this pattern -passing information from top to down, untouched- is very common when using AGs: they are called *copy rules* in the AG jargon. The `AspectAG` library has some builtin constructs to build this type of rule, such as *copyAtChi* that builds the copy rule given an attribute and a child.

Then, we can define the rules to copy the environment in the following way:

$$\mathit{add}_{\mathit{fenv}_l} = \mathit{copyAtChi}\ \mathit{fenv}\ \mathit{ch}_{\mathit{Add}_l}$$
$$\mathit{add}_{\mathit{fenv}_r} = \mathit{copyAtChi}\ \mathit{fenv}\ \mathit{ch}_{\mathit{Add}_r}$$

For the new production p_{Call}, we also need to copy the environment in the function argument child.

```
add_fenv_arg = copyAtChi fenv ch_Call_arg
```

We also need to define rules for the previously defined attributes *eval* and *env* in the new production. Let us combine both rules at once with the (⋄) operator, that combines a pair of rules for a given production.

```
call_evalenv_arg = copyAtChi env ch_Call_arg
                 ⋄ (syndefM eval p_Call (do
                     fe   ← at lhs fenv
                     fn   ← ter ch_Call_fun
                     argv ← at ch_Call_arg eval
                     case M.lookup fn fe of
                       Just f → return (f argv)))
```

For the *env* attribute we use a copy rule. To compute the value of the *eval* attribute we get the inherited function environment, and the function name (that is a terminal). We look at the synthesized *eval* attribute at the argument child and apply the function. Again we do not care about undefined functions for now.

Finally we can write the new evaluator for the expression language as:

```
type FEnv = M.Map String (Integer → Integer)
evaluator :: Expr → (Env, FEnv) → Integer
evaluator exp (envi, fenvi) =
  sem_Expr asp'_sem exp (env  =. envi *.
                         fenv =. fenvi *.
                         emptyAtt) #. eval
  where
    asp'_sem = add_fenv_l
             ◁ add_fenv_r
             ◁ add_fenv_arg
             ◁ call_evalenv_arg
             ◁ asp_sem
```

Given an expression, and the two environments, the evaluator computes the denotation of the expression.

3.5 Polymorphism in `AspectAG`.

In the previous sections, we showed how the expression problem can be tackled using `AspectAG`. How well the expression problem is tackled is a good measurement of modularity, but it is not the only one we are interested in. Functional programming provides tools to achieve modularity such as higher-order functions and lazy evaluation [Hughes, 1989]. Polymorphism is another useful feature to modularize programs since it adds the possibility to code a function once to manipulate many data structures. As a DSL embedded in Haskell `AspectAG` can exploit the benefits of polymorphic, higher-order functional programming. We briefly discussed in the previous sections a couple of examples of how to use lazy evaluation (partial functions in attribute definitions that will not crash) and polymorphism (the *copyAtChi* template). But since `AspectAG` is a language used to create other languages it is natural to wonder if the generated language can be itself polymorphic or if we can produce higher-order functions from the application of semantic functions.

For instance, consider the *Data.List* module in GHC's base library. It is an EDSL used to manipulate lists, and most functions in it are polymorphic and higher-order. Can we program functions in *Data.List* using `AspectAG`? In fact, an arbitrarily chosen subset of them is included in an example program in the `AspectAG` package distribution.

When using `AspectAG` to build abstract syntax for compilers (i.e. not embedded languages such as the expression language, but to implement the backend for a full compiler toolchain) there are also good reasons to make the generated abstract syntax polymorphic. One could parametrize the abstract syntax tree to add type information. One could also build extensible data types using trees that grow [Najd and Jones, 2017]. There is another useful use that we will extensively exploit in the case study presented later. It is to parametrize the language over its terminals. This is the same we do in the *List* DSL over the type of the contained elements.

To show the benefits of parametrization over terminals let us come back to the expression language. Suppose that we want to add the possibility to manipulate real numbers instead of integers. We can drop the p_{Val} production as it is and introduce an alternative, say p_{ValR}, with the correct type. Unfortunately our semantics for the *eval* and *env* attributes will no longer be reusable since they have the *Integer* type attached. We can define a new pair of attributes, and define rules to compute them, but note that those rules will be written the same way as for *eval* and *env*. For *eval* we would take the terminal on p_{Val}, sum the recursive denotations in p_{Add}, and lookup the environment in p_{Var}. For *env* we use copy rules.

Moreover, if instead of swapping integers by real numbers we want to manipulate both real numbers and integers in the expression language, things will get more complicated. p_{Val} and p_{ValR} productions can coexist, but the rule to compute *eval* on the p_{Add} production will be ill typed (even using the polymorphic addition). The issue can be solved by computing the evaluation as a sum type and combining it properly in the case of addition. Nevertheless, this does not scale well if adding more productions.

All these patterns can be factored and repetition avoided. We can think of the expression language as a way to build variables, additions, and values of some abstract type. We can delegate the responsibility of how values are handled to the abstract type.

Let us implement it to see how it works. The Nt_{Expr} non-terminal and the p_{Add} and p_{Var} productions remain as they were already defined. We build an alternative production for values, p_{ValP} that contains a polymorphic terminal:

$$
\begin{aligned}
&\textbf{type}\ P_{ValP} = (\text{‘}Prd\ \texttt{"ValP"}\ Nt_{Expr})\\
&p_{ValP} = Label\ @\ P_{ValP}\\
&ch_{ValP_{val}} :: \forall\ v.Label\ (\text{‘}Chi\ \texttt{"val"}\ P_{ValP}\ (Terminal\ v))\\
&ch_{ValP_{val}} = Label
\end{aligned}
$$

Note that actually the p_{Val} label does not change, the children definition is the one that differs from the definition in Section 3.1. We make the type of the value contained in the child polymorphic.

Now, we define a polymorphic attribute *evalP*, abstracting the type in the same way.

$$
\begin{aligned}
&evalP :: \forall\ v.Label\ (\text{‘}Att\ \texttt{"evalP"}\ v)\\
&evalP = Label
\end{aligned}
$$

Now we define an interface for the abstract type of our values. We call it *Number* and whatever it is, the type implementing the interface must support addition. We do it with a type class.

$$
\begin{aligned}
&\textbf{class}\ Number\ v\ \textbf{where}\\
&\quad add :: v \to v \to v
\end{aligned}
$$

To document types better we can have this class as a constraint in the polymorphic type v of $evalP$ and $ch_{ValP_{val}}$, but it is not necessary at all. Now we define the semantics for the $evalP$ attribute.

$$
\begin{aligned}
&add\_evalP = \lambda(Proxy :: Proxy\ v) \rightarrow \\
&\quad syndefM\ (evalP\ @\ v)\ p_{Add} \\
&\qquad\quad \$ \quad add \\
&\qquad <\$> at\ ch_{Add_l}\ (evalP\ @\ v) \\
&\qquad\quad \circledast \quad at\ ch_{Add_r}\ (evalP\ @\ v)
\end{aligned}
$$

Those definitions are very similar to the ones in Section 3.2. The difference is that we added a *Proxy* parameter with the type of values and applied that type to all polymorphic labels. That *Proxy* argument solves a subtle technical issue. As we will see in Chapter 5 `AspectAG` structures are implemented with extensible records. When a record contains polymorphic values and we look up fields (either with polymorphic or monomorphic values) variables may not unify. Even if we could deduce they do, GHC inference algorithm may not, and type inference may get stuck. The solution is to pass the type explicitly and force unification early.

With this approach, there are no relevant differences in the definition of the semantic function or with data type definitions. As an example we can show the type that we would generate with the forested approach:

$$
\begin{aligned}
&\mathbf{data}\ Expr\ a \\
&\quad = Add\ \{l :: Expr\ a, r :: Expr\ a\} \\
&\quad\ \mid\ Var\ \{var :: String\} \\
&\quad\ \mid\ Val\ \ \{val :: a\}
\end{aligned}
$$

We can show that we can get what we had before. If we implement the instance of *Number* for the type *Integer* we get the same language defined in Section 3.1. The evaluator looks like:

$$
\begin{aligned}
&evaluator :: Expr\ Integer \rightarrow Env\ Integer \rightarrow Integer \\
&evaluator\ exp\ envi = \\
&\quad sem_{Expr}\ (asp_{sem}\ (Proxy\ @\ Integer))\ exp \\
&\qquad\qquad\quad ((envP\ @\ Integer)\ \ =.\ \ envi\ *.\ emptyAtt)\ \#.\ (eval\ @\ Integer)
\end{aligned}
$$

where asp_{sem} is built by a combinator similar to $(\bowtie)$ (actually the same, since in the current `AspectAG` implementation it is polymorphic) that combines polymorphic rule definitions (it passes the proxy to every rule). We omitted the definition of the attribute *envP* and the rules to compute it, since it is straightforward and does not offer any new insight.

The news is that we also got a family of languages, each type implementing the *Number* interface. If we want to combine *Integer*s, real numbers (say, represented as *Double*) and unary naturals, we can:

```
data MyNum = I Integer | R Double | N Nat
data Nat = Z | S Nat

instance Number MyNum where
  add = ...
```

How they are combined when added up will depend on the implementation of the *add* method.

Finally, we must note that we do not lose any expressivity with polymorphic attributes. The language can still be extended with new productions. If we need more functions to combine elements of the type of terminals (suppose, for instance, if we add a production to model other operation than addition) we can implement a subclass of *Number* and proper instances for each implementation of terminals.

3.6 A deforested approach.

In Section 3.1 we discussed how to define context-free grammars in `AspectAG`. Pieces of a grammar are defined by labels, and then the full grammar is reified in a data type. Semantic functions take the rol of relating the labels and the reified data type.

When a grammar is extended, new data types are defined, possibly 'sharing' many constructors, and new semantic functions are defined, possibly sharing many clauses in its definitions. While namespaces in modules offer a solution to handle them unambiguously, this duplication of the source code violates one condition of the statement of the expression problem as stated in Section 2.3.

To avoid the code duplication, there are many possible solutions. One possibility is to use actual extensible algebraic data types. Vanilla Haskell does not support them, but there are ways to mimic them [Swierstra, 2008, Najd and Jones, 2017,Henry, 2020]. Unfortunately, these encodings are heavyweight, generating complex types. `AspectAG` is already implemented using advanced type-level programming, therefore, adding more complexity to types does not seem to be wise.

The alternative is not to use data types at all, building terms of the expression language using a *shallow embedding* [Fowler, 2010]. Instead of defining the semantic function sem_{Expr} as a traversal over a data type, we define expressions directly as a composition of semantic functions. Let us see how it works.

First, consider the following alternative definition of the semantic function:

$$\begin{array}{l}
sem\_Expr\_Add'\ asp\ sl\ sr = \\
\quad knitAspect\ p_{Add}\ asp\ \$ \\
\quad ch_{Add_l}\ \ .{=}.\ \ sl\ \ .{*}. \\
\quad ch_{Add_r}\ \ .{=}.\ \ sr\ \ .{*}.\ \ emptyGenRec
\end{array}$$

$$\begin{array}{l}
sem\_Expr\_Val'\ asp\ val = \\
\quad knitAspect\ p_{Val}\ asp\ \$ \\
\quad ch_{Val_{val}}\ \ .{=}.\ \ sem_{Lit}\ @\ Integer\ val\ \ .{*}.\ \ emptyGenRec
\end{array}$$

$$\begin{array}{l}
sem\_Expr\_Var'\ asp\ var = \\
\quad knitAspect\ p_{Var}\ asp\ \$ \\
\quad ch_{Var_{var}}\ \ .{=}.\ \ sem_{Lit}\ @\ String\ var\ \ .{*}.\ \ emptyGenRec
\end{array}$$

Instead of pattern matching over the data type $Expr$, we define a function for each production. The semantics of nonterminal subexpressions like in the sem_Expr_Add' function are given themselves in terms of semantic functions.

For example, the expression `4 + x` was expressed as $(Add\ (Val\ 4)\ (Var\ \texttt{"x"}))$ before. Now it can be implemented as:

$$\begin{array}{l}
e = \lambda asp \rightarrow sem\_Expr\_Add'\ asp\ (sem\_Expr\_Val'\ asp\ 4) \\
\qquad\qquad\qquad\qquad\qquad\qquad\quad (sem\_Expr\_Var'\ asp\ \texttt{"x"})
\end{array}$$

Now, to add a new production to our expression language we can add suitable labels and the new semantic function. Even if e was precompiled and the new production is defined in another module, e becomes an expression of the extended language, for free.

Perhaps the reader is worried about the verbosity of the new representation for terms. We can hide the argument asp with a reader monad [Jones, 1995], or use a fixed one when we are working with a concrete semantics. We can then build smart constructors to actually write the term in a readable way such as *add* (*vl* 4) (*vr* `"x"`).

There is a subtle flaw with this approach as we presented it. As we will see in the following sections, when building aspects by adding rules, we build a polymorphic expression with constraints defining the types of the rules for each production. In the forested approach, semantic functions instantiate those expressions further by applying the function *knitAspect* (or its variants, such as the more primitive function *knit*), as we will see in Section 5.3.6. The function *knit* does indireclty a lot of work, in particular it is where we decide the set of children in each production. Note that while children are tied to productions by their type, productions are not tied to children anywhere else (and this is good, because we can extend productions easily).

In the deforested approach we can build expressions that do not use every production with rules defined in the aspect, which means some type variables are not instantiated by any application of *knit*. Being them 'too polymorphic', a type error will be raised.

Fortunately, this can be solved in a relatively easy way, combining those too polymorphic aspects with empty rules that have their arguments instantiated enough.

There is a solution of compromise: to avoid grouping rules of different productions in aspects at all, just grouping semantics in a production basis. The following is yet another possible definition of the semantic functions:

```
    $sem\_Expr\_Add''\ asp\_add\ sl\ sr =$
      $knit\ Proxy\ asp\_add\ \$$
      $ch_{Add_l}\ .{=}.\ sl\ .{*}.$
      $ch_{Add_r}\ .{=}.\ sr\ .{*}.\ emptyGenRec$


    $sem\_Expr\_Val''\ asp\_val\ val =$
      $knit\ Proxy\ asp\_aval\ \$$
      $ch_{Val_{val}}\ .{=}.\ sem_{Lit}\ @\ Integer\ val\ .{*}.\ emptyGenRec$


    $sem\_Expr\_Var''\ asp\_var\ var =$
      $knit\ Proxy\ asp\_var\ \$$
      $ch_{Var_{var}}\ .{=}.\ sem_{Lit}\ @\ String\ var\ .{*}.\ emptyGenRec$
```

The alternative definition for the expression e in this context would be e' defined such as:

$$\begin{aligned} e' = &\ \mathit{sem\_Expr\_Add}''\ \mathit{asp\_add}\ (\mathit{sem\_Expr\_Val}''\ \mathit{asp\_val}\ 4) \\ &\qquad\qquad\qquad\qquad (\mathit{sem\_Expr\_Var}''\ \mathit{asp\_var}\ \texttt{"x"}) \end{aligned}$$

where *asp_add*, *asp_val* and *asp_var* are rules for each production, instead of aspects. We gain the hability to use e' without issues if the language grows syntactically, but at the same time we tied e' to a concrete definition of semantics. However, when implementing compilers such as in the case study of Chapter 6, at the time of defining a tangible function that process a program, we know all the semantics defined at that point, so definitions such as e' are adequate. Definitions of expressions like e make sense if we want to use `AspectAG` to implement an *embedded* DSL.

What should be clear from this section is that, `AspectAG` is powerful enough to solve the expression problem avoiding recompilation.

Chapter 4

Error Messages.

In an EDSL implemented using type-level programming type error messages are usually hard to understand and they often leak implementation details. The `AspectAG` library is designed to provide good, domain-specific error messages for the typical errors the programmer may make.

In this chapter, we identify which domain-specific type errors should be captured and we show examples of how they are printed to the user. How this is achieved will be shown in Chapter 5.

4.1 Identifying domain-specific type errors.

To capture type errors in a domain-specific manner, we need to understand how the domain is encoded and how the errors that users can make are materialized.

The `AspectAG` library is implemented using extensible records. Each production has a record of children. Synthesized and inherited attributions are implemented as records of attributes. The `AspectAG` library user defines attribute computations (1) using references to attributes (2) and children (3). We analyze the point of failures in those.

1. When defining attribute computations, since attributes are typed, the computation defined must be of the same type as the defined attribute (4.2.1). Another thing we must control is that attributes are unique. There should be only one definition to compute an attribute in a given production (4.2.2). This is a design decision: we could admit more than one definition where the latter one overrides the previous ones. For this kind of use case the *synmodM* and *inhmodM* combinators exist,

where a new opportunity to make mistakes rises: we can modify rules for attributes only if they already have rules attached (4.2.3).

2. When referencing attributes, the programmer could make two kinds of mistakes: refer to attributes using the incorrect type (4.2.4), and refer to attributes for which there is no defined behavior (4.2.5). The latter one is tricky since this information is nonlocal, as we will explain later.
3. When referencing children, they must be valid ones (i.e. to have in their type the correct production). This is true when accessing to a child, and when defining an inherited attribute (4.2.6).

4.2 Examples of error messages.

To illustrate examples of error messages consider the expression language defined in the previous chapter. In Figure 4.1 we condense the definitions of the `Expr.Eval` module defined in Section 3.2. We made some small refactoring, adding calls to *traceAspect* to better show how adding traces will help.

In this section we show how the error messages are printed to the user when introducing errors in the example.

4.2.1 Type mismatch in attribute definition.

Attributes are typed. When defining an attribute, the type of the expression that computes its value should match the type of the attribute.

For instance, if in Line 4 of Figure 4.1 we use an attribute with a type different from the one expected, writing *env* instead of *eval*:

$$\begin{aligned} add_{eval} = syndefM\ env\ p_{Add}\ ((+)\ &@\ Integer \\ &<\$>\ at\ ch_{Add_l}\ eval \\ &\circledast\ \ at\ ch_{Add_r}\ eval) \end{aligned}$$

we obtain a type error telling us there is a mismatch between the declared type of the attribute and the type of the expression we used to define its semantics. The error points out to Line 4. The message is the following:

```
• Error: Map String Integer /= Integer
         type mismatch in attribute definition
         attribute type does not match with the computation that defines it
  trace: - syndef: definition of Attribute (env:Env)
```

```
type Env = M.Map String Integer

eval = Label @ ('Att "eval" Integer)
env  = Label @ ('Att "env"  Env)

add_eval = syndefM eval p_Add ((+)   @ Integer
                               <$>  at ch_Add_l eval
                               ⊛    at ch_Add_r eval)
val_eval  = syndefM eval p_Val (ter ch_Val_val)
var_eval  = syndefM eval p_Var (do env ← at lhs env
                                  x   ← ter ch_Var_var
                                  case M.lookup x env of
                                    Just e → return e)

asp_eval = traceAspect (Proxy @ ('Text "combination of rules for Eval"))
             $ add_eval ◁ val_eval ◁ var_eval ◁ emptyAspect

add_env_l = inhdefM env p_Add ch_Add_l (at lhs env)
add_env_r = inhdefM env p_Add ch_Add_r (at lhs env)

asp_env = traceAspect (Proxy @ ('Text "combination of rules for Env"))
            $ add_env_l ◁ add_env_r ◁ emptyAspect

asp_sem = traceAspect (Proxy @ ('Text "combination of aspects Eval and Env"))
            $ asp_eval ⋈ asp_env

evaluator :: Expr → Env → Integer
evaluator exp envi =
  sem_Expr asp_sem exp (env  =. envi ∗. emptyAtt) #. eval
```

Figure 4.1: Evaluation Semantics for expressions.

```
            in Production Add of Non-Terminal Expr
• In the expression:
    syndefM env p_Add ((+) <$> at ch_Add_l eval <*> at ch_Add_r eval)
  In an equation for ‘add_eval’:
      add_eval
        = syndefM
            env p_Add ((+) <$> at ch_Add_l eval <*> at ch_Add_r eval)

 > add_eval   = syndefM env p_Add  (  (+)  at Integer
                  ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^...
```

Probably a more realistic example of this type of error, is, instead of grossly confusing attribute labels, to make the mistake when writing the rule computation. For example, if we drop line 8 of Figure 4.1 and define var_{eval} with the following declaration:

$$var_{eval} = syndefM\ \ eval\ var\ \$\ lookup <\$> ter\ ch\_var\_var \circledast at\ lhs\ env$$

the defined computation has type *Maybe Integer*, while the declared type of the attribute *eval* is *Integer*. Hence, the following type error is raised:

```
• Error: Integer /= Maybe Integer
         type mismatch in attribute definition
         attribute type does not match with the computation that defines it
  trace: - syndef: definition of attribute (eval:Integer)
            in production (Var of Non-Terminal Expr)
• In the expression: syndefM eval var
  In the expression:
    syndefM eval var $ M.lookup <$> ter ch_Var_var <*> at lhs env
  In an equation for ‘var_eval’:
      var_eval
        = syndefM eval var $ M.lookup <$> ter ch_Var_var <*> at lhs env

  > var_eval   =  syndefM eval var
                  ^^^^^^^^^^^^^^^^
```

In both cases there is also a `trace` information, showing the context where the error appears. In these examples traces do not offer new insight, they show information that the programmer can deduce by looking at the expression where the error is detected. In these cases, is the place where the error was actually made. We will see later in this chapter some examples where this information will be more helpful.

4.2.2 Duplication of Attributes.

Attributes must have a rule to compute them in every place they are used. Rules to compute attributes at a given production must be unique. As we said before we could opt for a different design decision and make rules overridable. We consider this would not be a good decision. In fact, by avoiding the possibility to override rules, our implementation is safer since programmers know at every time which rules are defined. There is no way to accidentally override a rule, which is a threat in medium to big-sized developments. Also, the current implementation satisfies the property that the combination of rules and aspects is commutative if we avoid the use of rule modifications (like *synmodM*), which is a nice property if users want to reason about the AG definition.

If two or more rules are defined to compute an attribute in a production, an error is raised. Suppose that we have the following instead of the definition in Line 12:

$$\begin{aligned} asp_{eval} = {} & traceAspect\ (Proxy\ @\ (\text{'}Text\ \texttt{"combination of rules for Eval"})) \\ & \$\ add_{env_l} \lhd add_{eval} \lhd val_{eval} \lhd var_{eval} \lhd emptyAspect \end{aligned}$$

Then, we get the following error:

```
• Error: cannot extend Attribution
         collision in attribute (env:Map String Integer)
  trace: - inhdef: definition of attribute (env:Env)
            in production (Add of Non-Terminal Expr)
         - traceAspect: combination of rules for Eval
         - traceAspect: combination of aspects Eval and Env
• In the first argument of ‘sem_Expr’, namely ‘asp_sem’
  In the first argument of ‘(#.)’, namely
    ‘sem_Expr asp_sem exp (env =. envi *. emptyAtt)’
  In the expression:
    sem_Expr asp_sem exp (env =. envi *. emptyAtt) #. eval

 >  sem_Expr asp_sem exp (env =. envi *. emptyAtt) #. eval
             ^^^^^^^
```

pointing to the Line 21. Why does it point to the line where *evaluator* is defined when the error was actually done in Line 12? To allow easy composability of rules, the type of expressions such as asp_{eval} is not a ground type. It is polymorphic and required record fields translate to type class constraints. As a result incompatible constraints are not detected until they are solved. This is performed when the aspect asp_{eval} is actually used as an argument for a semantic function in the evaluator definition.

This is an example of how EDSL type errors are hard to handle. While the error is caught by the Haskell compiler, with the help of the domain knowledge one can diagnose it with more precision by analyzing the program. We cannot modify where the error is raised since we are "trapped" in GHC, but what we do in `AspectAG` is to provide the *trace*. Trace contents are actually pointers to the source code. The previous trace gives the user the following information:

- The error was raised because we combined the rule defined by *inhdef* for the attribute *env* in the production *Add* of non-terminal *Expr*. This first entry of the trace is a pointer to Line 14.
- This ill-typed definition appears when we combine the rules for *eval*. This is the second entry in the trace, a pointer to the tag added in Line 12. Note that the message printed is the one we put in the *traceAspect* application.
- Then we combined the aspects *Eval* and *Env*. This is the third entry in the trace, a pointer to the tag added in Line 19.

Following the trace information bottom-up, the user can infer the problematic occurrence of the rule add_{env_l} that was introduced. Are traces enough to always get this information unambiguously? This depends on the tags added by the programmer. Let us see two extreme cases:

- If we drop all tags (*traceAspect* calls in the code are erased), the error only shows the first trace entry. We know the collision occurs because we added a rule for *env* defined by *inhdef*. We do not know which rule is the problematic one. This is actually enough information for this example! The rule add_{env_l} was used twice and removing any occurrence is enough.
- We could add a tag for every rule manipulation. If we add a tag entry at every use of operators ($\triangleleft$) and ($\bowtie$) we always get the exact trace from the semantic function call to the rule expresssion where the error

is raised. Of course, traces will be huge, and their interpretation could be cumbersome.

In our experience, the approach followed in the example -tagging in combinations of aspects- works well in practice.

4.2.3 Modifying undefined attributes.

The previous error arises when we combine two definitions of the same attribute. If the intention is to override a definition, we can use the combinators (e.g. *inhmodM* or *synmodM*) to do that. To override a rule definition, there must be already some rule for the given attribute. Suppose that instead of Line 7 of Figure 4.1 where the rule val_{eval} is defined, we put:

$$val_{eval} = synmodM\ eval\ p_{Val}\ (ter\ ch_{Val_{val}})$$

Then we get an error pointing to Line 21.

```
• Error: field not Found on Attribution
         updating the attribute(eval:Integer)
  trace: - synmod: redefinition of attribute (eval:Integer)
            in production (Val of Non-Terminal Expr)
         - traceAspect: combination of rules for Eval
         - traceAspect: combination of aspects Eval and Env

• In the first argument of ‘sem_Expr’, namely ‘asp_sem’
  In the first argument of ‘(#.)’, namely
   ‘sem_Expr asp_sem exp (env =. envi *. emptyAtt)’
  In the expression:
   sem_Expr asp_sem exp (env =. envi *. emptyAtt) #. eval

  >  sem_Expr asp_sem exp (env =. envi *. emptyAtt) #. eval
              ^^^^^^^
```

The type error reports that we tried to update an attribute that was not found in the attribution.

4.2.4 Type mismatch in attribute reference.

This kind of error occurs when we refer to an attribute value in a rule definition that has type different from the one expected. In order to not abuse the swap of *eval* and *env* in examples, suppose we have an attribute *pp* of type *String*, that collects a pretty printed representation of the expression.

$$pp = Label\ @\ (\text{'}Att\ \texttt{"pp"}\ String)$$

Now, suppose that instead of the definition for the rule add_{eval} given in Line 4 of Figure 4.1, we give the following:

$$\begin{aligned} add_{eval} = syndefM\ eval\ p_{Add}\ ((+)\quad & @\ Integer \\ <\$>\ & at\ ch_{Add_l}\ pp \\ \circledast\quad & at\ ch_{Add_r}\ eval) \end{aligned}$$

We get a type error, pointing to that line:

```
• Error:  [Char] /= Integer
          ill-typed attribute computation.
  trace: - syndef: definition of attribute (eval:Integer)
            in production (Add of Non-Terminal Expr)
 • In the second argument of ‘(<>)’, namely ‘at ch_Add_l pp’
   In the first argument of ‘(<*>)’, namely
    ‘(+) Integer <> at ch_Add_l pp’
  In the third argument of ‘syndefM’, namely
    ‘((+) Integer <> at ch_Add_l pp <*> at ch_Add_r eval)’


                                    <>  at ch_Add_l pp
                                           ^^^^^^^^^^^^^^
```

Note that while the custom type error does not show all information (like the attribute/child where the error arises) this is not necessary since the exact line is pointed. But we could print more information with little effort if we want to.

4.2.5 Not defined attribute reference.

When the value of an attribute (at some child, or at the lhs) is used when defining a rule, but there is no defined rule to compute that attribute in that position, an error occurs.

Let us suppose we ignore the definition of the rule add_{env_r} given in Line 15 of Figure 4.1, and, of course, we do not combine this nonexistent rule in Line 17.

Then, we would get the following type errors:

```
• Error: field not Found on Attribution
         looking up the attribute (env:Map String Integer)
  trace: - syndef: definition of attribute (eval:Integer)
             in production (Var of Non-Terminal Expr)
         - traceAspect: combination of rules for Eval
         - traceAspect: combination of aspects Eval and Env

• In the first argument of ‘sem_Expr’, namely ‘asp_sem’
  In the first argument of ‘(#.)’, namely
    ‘sem_Expr asp_sem exp (env =. envi *. emptyAtt)’
  In the expression:
    sem_Expr asp_sem exp (env =. envi *. emptyAtt) #. eval

    sem_Expr asp_sem exp (env =. envi *. emptyAtt) #. eval
             ^^^^^^^

• Error: field not Found on Attribution
         looking up the attribute (env:Map String Integer)
  trace: - inhdef: definition of attribute (env:Map String Integer)
             in production (Add of Non-Terminal Expr)
         - traceAspect: combination of rules for Env
         - traceAspect: combination of aspects Eval and Env

• In the first argument of ‘sem_Expr’, namely ‘asp_sem’
  In the first argument of ‘(#.)’, namely
    ‘sem_Expr asp_sem exp (env =. envi *. emptyAtt)’
  In the expression:
    sem_Expr asp_sem exp (env =. envi *. emptyAtt) #. eval

    sem_Expr asp_sem exp (env =. envi *. emptyAtt) #. eval
             ^^^^^^^
```

Both type errors point again to the Line 21. This happens for the same reason we explained before: in that position the full attribution is analyzed and constraints are solved.

In this case it should be clear that we can hardly do a better effort to catch the error before, because there exists no error before!. While asp_{sem}

is incomplete[1], an incomplete aspect could be extended later. If we consider asp_{sem} ill-typed, so it is, for instance, asp_{eval}, because rules in asp_{eval} depend on the attribte *env*.

Allowing users to manipulate aspects that do not represent well-behaved semantics in their own is necessary to allow modular construction. When we apply an incomplete aspect to a semantic function, that is the place where the error is really made.

If the user wants to check if an aspect is 'complete' (i.e. all dependencies are fulfilled for the rules, so it can be used as the argument of a semantic function), then the user can test if the application type checks. `AspectAG` provides a function called *checkAspect* that performs the check given a semantic function and an aspect. It can be used like an `assert` with no runtime overhead since it operates purely on types.

In the example, the errors obtained are due to the use of the incomplete aspect asp_{sem} with a semantic function.

The trace of the first type error points out to the Line 8 of Figure 4.1 where we are using the attribute *env* at the lhs to compute the denotation of variables. This error is what we should expect since the attribute *env* is no longer distributed top-down.

The second type error points out that in the definition of add_{env_l} in Line 14 of Figure 4.1 the attribute *env* was missed. This makes perfect sense since we are no longer copying *env* to the right subexpressions, so in any node not in the left spine of an expression the attribute is missed.

We used an inherited attribute but in the same way this kind of error can be found with synthesized attributes.

Suppose in Line 13 of Figure 4.1 we forget to add the add_{eval} rule, so the asp_{eval} definition would be the following:

$$
\begin{aligned}
asp_{eval} = {} & traceAspect\ (Proxy\ @\ (\text{'}Text\ \texttt{"eval"})) \\
& \$\ val_{eval} \lhd var_{eval} \lhd emptyAspect
\end{aligned}
$$

We get the following error when applying the semantic function:

```
• Error: field not Found on Attribution
         looking up the attribute (eval:Integer)
```

[1] In other words, the defined grammar with the set of rules given in asp_{sem} is not well-defined according to the definition given in Section 2.1. In terms of the `AspectAG` EDSL, there are rules with undefined dependencies.

```
  trace: looking up attribute (eval:Integer)

• In the expression:
    sem_Expr asp_sem exp (env =. envi *. emptyAtt) #. eval
  In an equation for ‘evaluator’:
      evaluator exp envi
        = sem_Expr asp_sem exp (env =. envi *. emptyAtt) #. eval

   sem_Expr asp_sem exp (env =. envi *. emptyAtt) #. eval
   ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
```

Suppose, that we forget to combine var_{eval} instead of asp_{eval}, thus Line 12 of Figure 4.1 is redefined as:

$$\begin{aligned} asp_{eval} = {} & traceAspect\ (Proxy\ @\ (\text{'}Text\ \texttt{"eval"})) \\ & \$\ add_{eval} \triangleleft val_{eval} \triangleleft emptyAspect \end{aligned}$$

In that case we get the following error:

```
• Error: field not Found on Aspect
         looking up the production named  (Var of Non-Terminal Expr)
  trace: knit(Var of Non-Terminal Expr)

 • In the first argument of ‘(#.)’, namely
    ‘sem_Expr asp_sem exp (env =. envi *. emptyAtt)’
  In the expression:
    sem_Expr asp_sem exp (env =. envi *. emptyAtt) #. eval
  In an equation for ‘evaluator’:
      evaluator exp envi
        = sem_Expr asp_sem exp (env =. envi *. emptyAtt) #. eval

  sem_Expr asp_sem exp (env =. envi *. emptyAtt) #. eval
  ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
```

Note that this error is different from the previous one. The reason is that since the rule var_{eval} was the only one defined for the *Var* production, the modified version of the asp_{eval} aspect does not have a rule for the production *Var*. This is the error reported here.

4.2.6 Children reference mismatch.

When defining a rule to compute an attribute the user provides an aspect for which the computation rule is being defined, and a production where it is defined. To compute that attribute the value of inherited attributes can be used, as well as the values of synthesized ones at some children. Those children should actually belong to the production for which we are defining the rule.

Suppose that we change the Line 7 of Figure 4.1 for the following one:

$$val_{eval} = syndefM\ eval\ p_{Val}\ (ter\ ch_{Var_{var}})$$

We get the following type error:

```
    • Error:
   Child Var_var of producion (Var of Non-Terminal Expr)
/= Child Var_var of producion (Val of Non-Terminal Expr)
      trace: - syndef: definition of attribute (eval:Integer)
                 in production (Val of Non-Terminal Expr)
             Data.Type.Require.ShowCTX ctx
    • In the third argument of ‘syndefM’, namely ‘(ter ch_Var_var)’
      In the expression: syndefM eval p_Val (ter ch_Var_var)
      In an equation for ‘val_eval’:
          val_eval = syndefM eval p_Val (ter ch_Var_var)

 val_eval   = syndefM eval p_Val (ter ch_Var_var)
                                  ^^^^^^^^^^^^^^
```

When defining a rule to compute an inherited attribute with the *inhdef* family of combinators, users must provide a production, and a child. Children are associated to productions, and that information is visible on types. `AspectAG` does the check, and in case of a mismatch, an error is reported.

Finally, suppose instead of Line 14 of Figure 4.1 we define add_{env_l} as:

$$add_{env_l} = inhdefM\ env\ p_{Add}\ ch_{Var_{var}}\ (at\ lhs\ env)$$

We get the error:

```
    • Error: (Add of Non-Terminal Expr) /= (Var of Non-Terminal Expr)
             production and child mismatch in inherited attribute definition
      trace: - inhdef: definition of attribute (env:Env)
```

```
              in production (Add of Non-Terminal Expr)
           Data.Type.Require.ShowCTX ctx
    • In the expression: inhdefM env p_Add ch_Var_var (at lhs env)
      In an equation for ‘add_env_l’:
          add_env_l = inhdefM env p_Add ch_Var_var (at lhs env)

add_env_l  =  inhdefM env p_Add ch_Var_var (at lhs env)
                ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
```

Chapter 5

Implementation details

In this chapter, we show the details of how the `AspectAG` library is implemented. This includes all the machinery developed to make it possible to get domain-specific type errors like the ones shown in Chapter 4.

In Section 5.1, we present a framework to manage user-defined type errors. While initially conceived as an abstraction layer for `AspectAG`, the framework works well in many scenarios, so it evolved into a standalone library: the `requirements` library.

In Section 5.2 we present the `poly-rec` library, implementing extensible, polymorphic-kinded (from now on, polykinded) records, the basic building block for `AspectAG` data structures.

Finally, in Section 5.3 we present the design and implementation of `AspectAG`'s data structures and combinators. The general structure of the implementation resembles the presentation given in [de Moor et al., 2000,de Moor et al., 1999]. This same architecture was already implemented in Haskell [Viera et al., 2009]. The new contribution is to do it in a kind-safe manner, combined with the `requirements` library to get good type errors, and the addition of polymorphism.

5.1 The `requirements` library.

We introduce the `requirements` library. This tool is used to program user-defined type errors uniformly. Firstly, in Section 5.1.1 we introduce the main technique we developed: the '`Require`' pattern. Afterwards, in section 5.1.2 we abstract over one particular case that will be used extensively in practice.

5.1.1 The `Require` pattern.

In order to show the generality of the tackled problem we avoid using records (the structure we will extensively use in the `AspectAG` implementation) in the running examples on this section.

Let us revisit the example of type-level programming in Haskell given in Section 2.2: the EDSL of heterogeneous lists. In Figure 5.1 we refresh the most relevant definitions and define some new utilities. The definitions given are the following:

- The data type *Nat* (and the promoted kind).
- The singleton type *SNat*.
- The *HList* data structure.
- A type family *Length* that computes the length of a promoted list.
- A type family ($:<$) that computes the "less than or equal" relation over naturals (as a boolean result).
- A type family (:!:) that computes the lookup in the list given a natural number index.

Now, we aim to program the function (!)[1]. Given a natural number m and a heterogeneous list of length n, by applying the function (!) we get the m-th position of the list (positions start at zero, as usual). To do this in a type-safe way we should ensure that $m < n$. There are many well-known ways to do this. One of them is to use a bounded natural as argument (the *Fin* type, well known in the literature) so the user cannot ask for an element outside the limit n. Another solution is requiring a proof of $m < n$.

With those "explicit" approaches only function calls that satisfy the conditions will be allowed. While it is nice to forbid programmers to even write ill-typed expressions, this approach has some drawbacks. For instance, it leads to data duplication (using *Fin* means adding yet another way to represent natural numbers). Another drawback is that to produce a well-typed term, programmers must use the correct arguments, meaning they need to be aware of more information.

When the domain is complex, writing legal programs can be difficult (for instance, building a well-formed AG!). We thing a good strategy is to let pro-

[1] Note that in the Haskell terminology '!' is an operator. In most situations Haskell users would refer to "the operator '!' ", instead of "the function '(!)' ". Since the term "operator" already has a meaning in the context of the `requirements` library, we deliberately avoid it.

```
data Nat = Z | S Nat
data SNat (n :: Nat) where
  SZ :: SNat Z
  SS :: SNat n → SNat (S n)
data HList (l :: [Type]) :: Type where
  HNil :: HList '[]
  (:::)  :: a → HList l → HList (a : l)
infixr 5 :::
type family Length (l :: [Type]) :: Nat where
  Length '[]      = Z
  Length (_': l) = S (Length l)
type family (:<) m n where
  Z   :< S _ = True
  Z   :< Z   = False
  S _ :< Z   = False
  S n :< S k = n :< k
type family (:!:) l n where
  [t] :!: Z = t
  (t : ts) :!: (S n) = ts :!: n
```

Figure 5.1: Summary of definitions for heterogeneous lists.

```
class Get (l :: [Type]) (n :: Nat) where
  (!) :: HList l → SNat n → l :!: n
instance Get '[t] Z where
  (HCons a _) ! SZ = a
instance (Get ts n) ⇒ Get '( t': ts) (S n) where
  (HCons _ as) ! (SS n) = as ! n
```

Figure 5.2: Conventional implementation of a dependent function in Haskell.

grammers to write their programs and then analyze them in order to precisely report inconsistencies in case they exist.

Going back to the running example, once we have singleton naturals defined, our goal is to have a function with a type similar to the following:

```
(!) :: HList l → SNat n → l :!: n
```

Using (!) whenever ($n \geqslant$ *Length* l) should raise a type error. This can be easily accomplished in Haskell by defining a function (!) in a type class, such that we only define instances for $n \leq m$. Figure 5.2 shows an implementation. Writing an expression accessing indexes out of bounds generates a vanilla type error saying that there is no instance for the type class.

To get domain-specific error reports one can write class instances for the ill-typed cases to trigger the evaluation of the *TypeError* type family provided by GHC. That function is exported by the module `GHC.TypeLits` of the base library. It takes a type-level string as an argument, and prints it as a compile error whenever it is evaluated at compile time. It is the type-level counterpart of the *error* function.

The problem that arises, is that coding each possible error scenario is cumbersome. Instead, what the `requirements` framework does is to provide a unified, standarized way to do this.

Requirements are type-level conditions that are demanded by functions that we call *operations*. Since operations can be of any arity, to manage all operations uniformly we uncurry them, grouping all arguments in a data type. We call those data types *operators*.

For instance, to implement the (!) operation we must build an operator, defining a data type such as the following:

```
instance (TypeError
            (Text "Error: " :◇: m :$$:
             Text "trace: " :◇: ShowCTX ctx))
  ⇒
  Require (OpError m) ctx where
  type ReqR (OpError m) = ()
  req _ _ = error "unreachable"
```

Figure 5.3: Implementation of an error requirement.

```
data OpGet (n :: Nat) (l :: [Type]) :: Type where
  OpGet :: SNat n → HList l → OpGet n l
```

The `requirements` library provides the *Require* class, that is used to implement operations.

```
class
  Require (op :: Type) (ctx :: [ErrorMessage]) where
  type ReqR op :: Type
  req :: Proxy ctx → op → ReqR op
```

The argument *op* is an operator (with its arguments applied, e.g *OpGet n l*). The indexed type family *ReqR* computes the resulting type of the operation. The function *req* implements the semantics of the operation at the value level. Given an operator (i.e. all the operation arguments) *req* computes a value of the result type (*ReqR op*). The extra *Proxy* argument exists to help the type checker to carry a *context ctx*. A context is a list of messages. Users of the library can push messages that will be printed to the users in case an error occurs (as type-level strings). We will see later how contexts are used to implement the traces we showed in Chapter 4.

The `requirements` library exports an especial operator, *OpError*, with no constructors carrying only information at type-level.

```
data OpError (m :: ErrorMessage) :: Type where { }
```

The instance of *Require* for the operator (*OpError m*) prints the error *m* and the contents of the context at that time.

The type family *ReqR* and the function *req* are defined with dummy values, since they are never used.[2] We use the type family *TypeError* as a constraint (this is valid since it has a polymorphic return kind), since in the context of the class *ctx* is in scope. We could have used *TypeError* in the right-hand side of the *ReqR* type family definition instead. In that case type errors would be raised in the same scenarios, but only m is in scope to produce the error message. Passing the context in operators is possible, but we prefer the implemented solution, where operators carry just information related to the algorithm they implement. The type family *ShowCTX* has kind $([\mathit{ErrorMessage}] \to \mathit{ErrorMessage})$. It combines a list of messages in the order they appear.

When a *requirement* is not fulfilled, the programmer can add a constraint with an instance of *Require* for the operator *OpError*. To see an example of how this is done, let us implement all cases for the operator *OpGet*.

First, note that we must decide the fulfillment of a requirement for $(\mathit{OpGet}\ n\ l)$ according to the condition $(n < \mathit{length}\ l)$. We could program the instances based on an induction principle for that relation, but it is more convenient to delegate the work of deciding the relation to the type family $(\lessdot)$ and then pattern match on the result. To pattern match on a boolean in a type class instance declaration, the boolean must occur in the head of the class definition. We need to use the "advanced overlap" design pattern we introduced in Section 2.2.1.4. To perform that, we define an auxiliary operator *OpGet′*, as follows:

```
data OpGet′ (b :: Bool) (n :: Nat) (l :: [Type]) :: Type where
  OpGet′ :: Proxy b → SNat n → HList l → OpGet′ b n l
```

The implementation of the instance of *Require* for the operator *OptGet* is done just calling the instance for *OptGet′*:

[2] Definitions can be omitted. In that case, the compiler shows a warning that can be lifted with the `-Wno-missing-method` flag.

```
instance
  Require (OpGet' (n :< Length l) n l) ctx
   ⇒
  Require (OpGet n l) ctx where
  type ReqR (OpGet n l)
     = ReqR (OpGet' (n :< Length l) n l)
  req proxy (OpGet n l)
     = req proxy (OpGet' (Proxy @ (n :< Length l)) n l)
```

When the boolean condition is met, the proper algorithm for lookup is implemented in the following instances of *Require* for *OptGet'*.

```
instance
  Require (OpGet' True Z (t : ts)) ctx where
  type ReqR (OpGet' True Z (t : ts)) = t
  req proxy (OpGet' _ _ (a ::: _)) = a
instance
  Require (OpGet' True n ts) ctx
   ⇒
  Require (OpGet' True (S n) (t : ts)) ctx where
  type ReqR (OpGet' True (S n) (t : ts)) = ReqR (OpGet' True n ts)
  req proxy (OpGet' p (SS n) (_ ::: as)) = req proxy (OpGet' p n as)
```

We do pattern-matching on the natural number. We focus on the well-kinded cases where $(b \sim \mathit{True})$ (and therefore, the type-level list l is non-empty).

The first instance is the base-case. When looking up a heterogeneous list in the index Z we just return the head value, of type t (since the heterogeneous list had index type $(t : ts)$).

The second instance is the recursive case. When the given position is non-zero, we perform the lookup recursively in the tail list.

The ill-typed cases are captured by requiring the *OpError* operation, as defined in Figure 5.4:

Whenever $(b \sim \mathit{False})$ it means $(\mathit{Length}\ l \geqslant n)$. All the information in scope (the types n and l) is used to produce an informative error message.

Then, it is possible to define the function (!) as:

```
l ! n = req emptyCtx (OpGet n l)
```

```
instance
  Require (OpError (Text "Type error!" :$$:
    Text "Index " :<>: ShowType n
    :<>:  Text " out of bounds."
    :$$:  Text "List of type `" :<>: ShowType (HList l)
    :<>:  Text "', of length: " :<>: ShowType (Length l)
    )) ctx
  =>
  Require (OpGet' False n l) ctx where { }
```

Figure 5.4: Definition of the *OpError* instance.

where *emptyCtx* is a *Proxy* containing an empty list.

The following type error will be produced if we try to get the third index ($SS\ (SS\ (SS\ SZ))$) from a list of type $HList\ '[Char, Bool, [Char]]$.

```
• Error: Type error!
         Index 'S ('S ('S 'Z)) out of bounds.
         List of type `HList '[Char, Bool, [Char]]',
           of length: 'S ('S ('S 'Z))
```

The core idea behind the `requirements` library is just what we have presented. As simple as it is, this design pattern proved to be very useful in many use-cases, justifying to factor it out as the standalone library.

The `requirements` library also provides extra functionalities, like a set of functions to combine error messages, and a set of use-cases to document the library to new users. For instance, one type family exported by `requirements` is the -open- family *ShowTE*, of kind ($k \to ErrorMessage$). It works as the *show* function at type level building error messages. In the Figure 5.4, when defining the error message for *OptGet* we used the more primitive type function *ShowType*, exported by the module `GHC.TypeLits`. While *ShowType* prints the types as they are written syntactically, the type family *ShowTE* can be used to implement a custom way to print types in error messages.

5.1.2 Equality requirements.

A common pattern when using requirements is to require two types to be equal. We will show that it can be performed with the presented technique, but as it is a common pattern, then we abstract a way to handle it.

Consider an EDSL to represent and manipulate figures of different dimensions (e.g. 2-dimensional objects, a 3-dimensional bodies, etc). In a real implementation, it should probably have more attributes, but dimension is enough for the example. Two figures can be combined, whenever they have the same dimension. An idiomatic way to implement this in Haskell using modern type-level programming features is by a data type indexed by a type of a kind representing dimensions.

```
data Dim = D2 | D3
data family Figure (d :: Dim)
```

The use of a data family construction makes it possible to define a data type for each instance of *Dim*. For instance, for *D2* we can define the following data type:

```
data instance Figure D2 = Rectangle Float Float
                        | Triangle Float Float
```

Now, we aim to program a function that takes two figures of the same dimension and builds a new figure. This is easy for a Haskell programmer, and it is natural to use the type system to ensure the desired property about dimensions. The type of the function can be given as:

```
combine :: Figure d → Figure d → Figure d
```

If we try to compile an expression (*combine f g*) where (*f* :: *Figure D2*) and (*g* :: *Figure D3*) we get the following error message:

```
. Couldn't match type ''D3' with ''D2';
  Expected type: Figure 'D2
  Actual type: Figure 'D3
```

This is perhaps acceptable for a Haskell programmer but it is not idiomatic for the user of an EDSL implemented in Haskell. Moreover, the error is clear just because the data type is simple, but in real-world DSLs users will face more complex cases. We aim to get a custom, more expressive type error.

```
data OpCombine d d′ where
  OpCombine :: Figure d → Figure d′ → OpCombine d d′
data OpCombine′ (b :: Bool) d d′ where
  OpCombine′ :: Proxy b → Figure d → Figure d′ → OpCombine′ b d d′


instance Require (OpCombine′ (d == d′) d d′) ctx
  ⇒
  Require (OpCombine d d′) ctx where
  type ReqR (OpCombine d d′) = ReqR (OpCombine′ (d == d′) d d′)
  req proxy (OpCombine f g)
    = req proxy (OpCombine′ (Proxy @ (d == d′)) f g)


instance
  Require (OpCombine′ True d d) ctx where
  type ReqR (OpCombine′ True d d) = Figure d
  req proxy (OpCombine′ _ f g)
    = combine f g


  -- the well behaved implementation:
combine′ = req (Proxy @ ′[])


instance
  Require (OpError (
    Text "Cannot combine figures. Dimension mismatch"
    :$$: Text "The first argument is of dimension " :◇: ShowTE d
    :$$: Text "The second argument is of dimension " :◇: ShowTE d′
  )) ctx
  ⇒
  Require (OpCombine′ False d d′) ctx where { }
```

Figure 5.5: Combination implementation.

In Figure 5.5 an implementation is given using the technique presented in Section 5.1. First, we define an operator *OpCombine* that takes the two arguments needed to implement the combination operation. We use the advanced overlap pattern building an operator with the explicit requirement condition (equality of dimensions, in this case) visible. Then we write the wrapper instance (the instance for *OpCombine* that uses the instance for *OpCombine'*), the well-typed case, and the ill-typed case requiring an error operator. The type error that is printed if users try to combine images of different dimensions is implemented in the last instance definition. Let us assume an implementation of the family *ShowTE* for *Dim* that prints integer literals (e.g `"2"` instead of `"D2"`) is given, we would get the following error:

```
. Cannot combine figures. Dimension mismatch.
  The first argument is of dimension 2
  The second argument is of dimension 3
```

Note that in the well-typed case we call the function *combine* of type $(\mathit{Figure}\ d \to \mathit{Figure}\ d \to \mathit{Figure}\ d)$. There should not be a surprise that this is accepted by the compiler, since in this instance the dimensions of both figures were unified.

While this is an acceptable solution, such a common pattern as equality is worth systematizing. To do so, we introduce a type family *RequireEq* with kind $(k \to k \to [\mathit{ErrorMessage}] \to \mathit{Constraint})$. We use it to require that two types (of kind k) are equal. The third argument is the usual context that can be given to print more information with the error case.

The following is a valid implementation for combination of figures:

$$\mathit{combine''} :: \mathit{RequireEq}\ d\ d'\ \mathit{ctx} \Rightarrow \mathit{Figure}\ d \to \mathit{Figure}\ d' \to \mathit{Figure}\ d$$
$$\mathit{combine''} = \mathit{combine}$$

The type family *RequireEq*, exported in the `requirements` library is defined as follows:

$$\textbf{type}\ \mathit{RequireEq}\ (t :: k)\ (t' :: k)\ (\mathit{ctx} :: [\mathit{ErrorMessage}])$$
$$\quad = (\mathit{AssertEq}\ t\ t'\ \mathit{ctx}, t\ \sim\ t')$$

$$\textbf{type family}\ \mathit{AssertEq}\ (t :: k)\ (t' :: k)\ \mathit{ctx} :: \mathit{Constraint}\ \textbf{where}$$
$$\quad \mathit{AssertEq}\ a\ a\ \mathit{ctx} = ()$$
$$\quad \mathit{AssertEq}\ a\ b\ \mathit{ctx} = \mathit{Require}\ (\mathit{OpError}\ (\mathit{Text}\ \texttt{"\textbackslash n \ \ \ \ "} :\diamond: \mathit{ShowTE}\ a$$
$$\qquad :\diamond: \mathit{Text}\ \texttt{"\textbackslash n/= "} :\diamond: \mathit{ShowTE}\ b))\ \mathit{ctx}$$

To be more clear, let us analyze it using the type variables that occur in the type of $combine''$. The constraint $(RequireEq\ d\ d'\ ctx)$ reduces into a pair of constraints $(AssertEq\ d\ d'\ ctx, d \sim d')$. The first constraint, $(AssertEq\ d\ d'\ ctx)$ is used to compute the type error in the cases where $d \nsim d'$. If the arguments are equal it reduces into the empty constraint $()$. On the other hand, if the arguments are not equal, it reduces into an instance of *Require* applied to the operator *OpError*, that will further reduce to an application of *TypeError*, which means a type error will be raised. A peculiarity in this code is the use of non-linear pattern matching in the definition of *AssertEq*. This feature is not available at the value level in Haskell but it can be used in type family definitions.

The second constraint, $(d \sim d')$ is perhaps surprising. At a first sigh the reader could think that the constraint $(t \sim t')$ in the type family definition is redundant once we have $(AssertEq\ t\ t'\ ctx)$, since the constraint *AssertEq* cannot be satisfied whenever $t \nsim t'$. However, the compiler cannot perform this sophisticated reasoning over the semantics of *AssertEq*. Moreover, note that the function $combine'$ cannot compile if we do not state somewhere that d and d' are equal, since it calls the function *combine*, that has arguments of the same type.

In general, in presence of constraints, GHC typechecks analyzing at first the *shape* of types, while collecting constraints, and then it solves the constraints. We cannot use the syntactical equality in the type of $combine''$ (writing it as $Figure\ d \rightarrow Figure\ d \rightarrow Figure\ d$) as we did in the function *combine* since the compiler will report the domain-agnostic type error before doing anything else. What we do is to de-unify equal type variables in the type of the function, and push the equality into constraints, to “defer” checking the equality. If the types d and d' are equal the compiler will accept the definition since the call to *combine* is legal. If d and d' are not equal, an application of *TypeError* will appear in the constraints and an error will be raised.

Somehow, this is the dual to the “advanced overlap” technique. Instead of pulling information from the constraints to force the compiler to decide earlier, we push information to constraints to force the compiler to decide later enough to trigger a *TypeError* application. This is the spirit behind the `requirements` library pragmatics: pushing the properties we want to check to produce domain-specific errors to context.

In case we apply arguments of different dimensions to the function *combine''*, the type error printed is:

```
. Error:
  R2 /= R3
```

That is still very general, we do not win almost anything in comparison with the default error that GHC shows in the original *combine* implementation. The reason is that we used a very general *RequireEq* class that knows anything about how to precisely report the error. But using the ideas presented we can improve the result.

Since type families are not parametric (we can decide depending on the kinds in the polymorphic arguments) the type family *RequireEq* could decide how to report the errors depending on the kind of d and d', but that means the user must implement the specific type family instance. As seen before, technicalities of the type system arise, and we want to avoid burdening users of the EDSL with that responsibility.

For an ordinary functional programmer, who is used to work at the value level, the solution should be obvious: abstracting the algorithm that produces the error message and passing it as an argument to *RequireEq*.

However, at type level, type functions are not first-class citizens. In particular, an expression involving type families should be always saturated when passed as an argument. Though there is research to add the feature to GHC [Kiss et al., 2019], this is, not implemented in the main compiler branch.

Fortunately, this issue can be tackled by using the defunctionalization technique [Reynolds, 1972, Danvy and Nielsen, 2001]. Originally presented as a compile-time technique to eliminate higher-order functions, it can be used as a programming technique as well. In Figure 5.6 a solution is provided. A similar idea is already implemented in the First-class families library [Xia, 2018], and presented in [Eisenberg and Stolarek, 2014].

We give a type family *RequireEqWithMsg* of kind $(k \to k \to (k \to k \to \mathit{Type}) \to [\mathit{ErrorMessage}] \to \mathit{Constraint})$. The third argument is a data constructor that plays the role of a *function symbol*. It represents the function that prints a type error reporting that the arguments given are not equal types.

An open type family *Eval*, of kind $(\mathit{Type} \to k)$ takes a function symbol and the arguments of the function it represents. Users can extend it with new semantics for each function symbol they introduce.

```
type family RequireEqWithMsg (t :: k) (t′ :: k) (mkmsg :: k → k →
Type)(ctx :: [ErrorMessage]) :: Constraint
type instance RequireEqWithMsg t t′ mkmsg ctx =
  (AssertEq t t′ mkmsg ctx, t ∼ t′)
type family AssertEq (t :: k) (t′ :: k)
                     (mkmsg :: k → k → Type) ctx :: Constraint
  where
  AssertEq t t  mkmsg ctx = ()
  AssertEq t t′ mkmsg ctx = Require (OpError (Eval (mkmsg t t′))
                               ) ctx
```

Figure 5.6: *RequireEqWithMsg* implementation.

```
data DimEq (d :: Dim) (d′ :: Dim)
type instance Eval (DimEq d d′) =
  Text "Cannot combine figures. Dimension mismatch"
  :$$: Text "The first argument is of dimension " :◇: ShowTE d
  :$$: Text "The second argument is of dimension " :◇: ShowTE d′
combine‴ :: RequireEqWithMsg d d′ DimEq ′[] ⇒
  Figure d → Figure d′ → Figure d
combine‴ = combine
```

Figure 5.7: Shorter implementation of combination.

In Figure 5.7, we provide an improved solution. The symbol *DimEq* is introduced, and the corresponding instance of the type family *Eval*. This solution is shorter, and generally lighter than the more general approach of defining an operator, while as general in the kind of errors users can program.

Users get the following error if they try to combine figures of dimensions 2 and 3:

```
. Error: Cannot combine figures. Dimension mismatch
         The first argument is of dimension 2
         The second argument is of dimension 3
```

```
type family IsEmptyCtx (ms :: [ErrorMessage]) :: Bool where
  IsEmptyCtx '[]          = True
  IsEmptyCtx (m ': ms)    = IsEmptyMsg m ∧ IsEmptyCtx ms
  IsEmptyCtx _            = True
type family IsEmptyMsg (m :: ErrorMessage) :: Bool where
  IsEmptyMsg (Text "")    = True
  IsEmptyMsg (l :◇: r)    = IsEmptyMsg l ∧ IsEmptyMsg r
  IsEmptyMsg other        = False
type family ShowCTX (ctx :: k) :: ErrorMessage where
  ShowCTX ('[] :: [ErrorMessage])
    = Text ""
  ShowCTX ((m :: ErrorMessage) ': (ms :: [ErrorMessage]))
    = m :$$: ShowCTX ms
```

Figure 5.8: Manipulation of context messages.

5.1.3 Printing context in type errors.

As we have shown, the class *Require* has a parameter *ctx* of kind [*ErrorMessage*]. Figure 5.3 shows how the context is appended to the generated error message. In Figure 5.8 we show the implementation of the type families involved, which consists of a simple traversal to generate one error message.

The context can be instantiated the way the users want to. One way to provide contexts is to give a proxy argument to the *req* function, as we have seen. In Figure 5.5 we used an empty context, but any message can be defined. For instance, adding the information that the function *combine* was used.

A good way to take advantage of contexts is to have in mind that when using EDSLs the compiler can detect errors that were introduced in different places of the source code. In fact, the main motivation to introduce contexts was to use them as traces. In Chapter 4 examples of outputs were given, where type errors raised in an application of a semantic function traced us to the rule definitions introducing the bug.

The main idea is to make types of our EDSL carry a context argument. When applying functions, the contexts in the arguments and in the return type can be combined. For instance, instead of using the type (*Figure d*) we can use a type (*CFigure d ctx*). To have a simple way to manipulate contexts without complex type annotation it is handy to have available a proxy argument with the context. The following is a possible definition for *CFigure*:

$$\textbf{newtype}\ \mathit{CFigure}\ d\ (\mathit{ctx} :: [\mathit{ErrorMessage}])$$
$$\quad = \mathit{CFigure}\ \{\mathit{mkCFigure} :: \mathit{Proxy}\ \mathit{ctx} \to \mathit{Figure}\ d\}$$

We present this pattern here because we will use it later. We are using a standard idea of putting contexts in an argument, and just packing everything into a data type. This can be abstracted using a reader monad [Jones, 1995]. In Section 5.3.7 we will study a concrete case.

5.2 Polykinded Extensible Records

In computer science, a record is a basic data structure consisting of a collection of *fields* storing values of -possibly- different types, indexed by *names*. Conceptually, they can be thought of as *partial mappings* from names to values. For that reason, we will refer to the set of labels of a record as its *domain*.

Extensible records, are records that can be extended with new fields. Extensible records are a well-known feature in the strongly typed functional programming community. There is a theoretical basis to add extensible records in type systems based on the Hindley-Milner type system such as the Haskell's one [Gaster and Jones, 1996, Leijen, 2004, Leijen, 2005]. In the Hugs Haskell compiler they were implemented as a non-standard extension [Gaster and Jones, 1996]. In Purescript [PureScript developers, 2017], a modern close descendant of Haskell, the feature is implemented.

Although not implemented in GHC Haskell, extensible records can be encoded using type-level programming. There are libraries solving the problem already. The HList library [Kiselyov et al., 2004] popularized extensible records. Other implementations such as Vinyl [Sterling, 2012] or CTRex [van der Ploeg, 2013] are available.

One usual way of implementing a record is by using a GADT indexed by the list of types of the values stored in its fields. These types are usually of kind *Type*, which makes sense since *Type* is the kind of all inhabited types, and records store values. However, in `AspectAG` we have a complex use-case: we will consider children records in productions where we store attributions that are also implemented by records. We would like to be able to reflect some of this structural information at the indexes of the record. This can be achieved if records are polymorphic in the kind of the types corresponding to the record fields. This is a feature the record data structure we use must provide. To our best knowledge, no available record library provides that feature. For that reason, we implement the `poly-rec` library, from scratch.

We will use the `requirements` library to manage type errors of `poly-rec`. But note, that we could build an interface combining `requirements` with any already available record library. The main motivation to build `poly-rec` is having kind-polymorphic records.

The achieved data structure will have the following features:

- It implements extensible records.
- Records implemented are intended to be general, so many record-like data structures can be implemented specializing the general definition. We call concrete specializations *record instances*.
- The data structure is type-safe, but also kind-safe. In particular, records are polymorphic in the kind of the contained values to enforce properties over the structure of the stored values.
- Records use first-class labels, meaning labels can be manipulated as any Haskell value.
- Type errors in the interface of the library will produce domain-specific error messages.

5.2.1 Record definition.

Records in `poly-rec` are implemented as a GADT, as follows:

$$
\begin{array}{lll}
\multicolumn{3}{l}{\textbf{data}\ \mathit{Rec}\ (c :: k)\ (r :: [(k', k'')]) :: \mathit{Type}\ \textbf{where}} \\
\quad \mathit{EmptyRec} & :: & \mathit{Rec}\ c\ '[] \\
\quad \mathit{ConsRec} & :: & \mathit{LabelSet}\ ('(l, v)\ ': r) \\
 & & \Rightarrow \mathit{TagField}\ c\ l\ v \to \mathit{Rec}\ c\ r \to \mathit{Rec}\ c\ ('(l, v)\ ': r)
\end{array}
$$

The GADT is indexed by two types: c and r. The index c is a tag for the type of record we are using. Each specific record instance implemented using *Rec* will have a different tag c. The index r represents the mapping from labels (of kind k') to values (indexed by k'').

There are two constructors: *EmptyRec* builds an empty Record, where the index r is instantiated as the empty type-level list. The constructor *ConsRec* extends a record given a value of type *TagField*. The data type *TagField* is defined as follows:

$$\textbf{data}\ \mathit{TagField}\ (c :: k)\ (l :: k')\ (v :: k'')\ \textbf{where}$$
$$\quad \mathit{TagField} :: \mathit{Label}\ c \to \mathit{Label}\ l \to \mathit{WrapField}\ c\ v \to \mathit{TagField}\ c\ l\ v$$

A tagged field is a value, tagged with the phantom types c and l.

The constructor *TagField* takes a pair of label arguments to instantiate c and l easily. Labels are isomorphic to proxies, defined with the following data type:

$$\textbf{data}\ \mathit{Label}\ (t :: k) = \mathit{Label}$$

The data type *Label* is polymorphic in the type and the kind of its argument, and is computational irrelevant at the value level. Note that labels have a value-level representation, which means they can be manipulated as any Haskell value; they can be passed as arguments, generated programmatically, etc. This, in the jargon, means labels are first-class citizens, which is a feature that not necessarily all record implementations have.

The third argument of *TagField* represents the proper value contained in the field. If the type v is any type of kind *Type* (i.e. the polymorphic kind k'' is instantiated as *Type*) it would make sense to have a constructor of type ($\mathit{Label}\ c \to \mathit{Label}\ l \to v \to \mathit{TagField}\ c\ l\ v$). However, in `poly-rec`, the type v can have a rich structure. For instance, if the fields of the record are records themselves, v can be a list of pairs.

Those rich type-level structures do not have kind *Type*, so they are not inhabited. The type ($\mathit{Label}\ c \to \mathit{Label}\ l \to v \to \mathit{TagField}\ c\ l\ v$) is ill-kinded if v is instantiated to any type of kind different from *Type*.

That is the reason why we use the open type family *WrapField*, defined as follows:

$$\textbf{type family}\ \mathit{WrapField}\ (c :: k')\ (v :: k) :: \mathit{Type}$$

Since it is open, we can add new instances when we add new record instances.

Example 5. *The motivation to use kind polymorphism and the WrapField family will be more clear with an example. Consider a record r with a field named "x" of type Bool, and a field named "y" of type Char. Its type would be:*

$r :: Rec\ Dim1\ [(\texttt{"x"}, Bool), (\texttt{"y"}, Char)]$

where Dim1 is the type used as a tag indicating that the record is one-dimensional. Defined simply as:

data *Dim1*

The value r is built by applying two times the constructor ConsRec to glue two values t1 of type (TagField Dim1 "x" *Bool) and t2 of type (TagField Dim1* "y" *Char). To build t1 we apply TagField to two suitable labels, and to a value of type Bool. Then, (WrapField Dim1 Bool) should reduce to Bool, and is implemented as such:*

type instance *WrapField Dim1 b* $= b$

Example 6. *Now suppose we want to build a 2-dimensional record (i.e. a record having records in its fields), and we want to reflect this in types and kinds. Suppose the record m has fields named* "a" *and* "b" *and each one contains a copy of r.*

The type would be as:

$$\begin{aligned} m :: Rec\ Dim2\ [&(\texttt{"a"}, [(\texttt{"x"}, Bool), (\texttt{"y"}, Char)]),\\ &(\texttt{"b"}, [(\texttt{"x"}, Bool), (\texttt{"y"}, Char)])] \end{aligned}$$

We can write this because we used kind polymorphism in the definition. In this case, fields are build by applying TagField to suitable labels, and the record r. Note that the type $([(\texttt{"x"}, Bool), (\texttt{"y"}, Char)])$ *is not inhabited, we need a value of type* $Rec\ Dim1\ [(\texttt{"x"}, Bool), (\texttt{"y"}, Char)]$*. So (WrapField Dim2 v) should reduce to (Rec Dim1 v), and the type family instance is implemented as such:*

type instance *WrapField Dim2 v* $=$ *Rec Dim1 v*

Actually, we could avoid all this complexity and design *Tagfield* so it always take a type of kind *Type*. A matrix such as *m* of Example 6 can still be built, of type:

$$\begin{aligned} m' :: Rec\ Dim2\ [&(\texttt{"a"}, Rec\ Dim1\ [(\texttt{"x"}, Bool), (\texttt{"y"}, Char)]),\\ &(\texttt{"b"}, Rec\ Dim1\ [(\texttt{"x"}, Bool), (\texttt{"y"}, Char)])] \end{aligned}$$

Type constructors can work as a way of "cheating" on the kind system because they hide any kind information building a *Type*. However, in this case, at kind-level nothing prevents extending the matrix with a field that is not a record.

In the value m the variable r of the *Rec* definition has its kind instantiated to $[(\mathit{Symbol}, [(\mathit{Symbol}, \mathit{Type})])]$. On the other hand, in m' to $[(\mathit{Symbol}, \mathit{Type})]$, where those variables of kind *Type* hide a record structure.

We gain some guarantees in the definition using polykinded records. At the same time, the implementation still can admit a mix of records and non-records as fields with a suitable *WrapField* instance, so we just built a safer system without losing any flexibility.

Going back to the definition of the data type *Rec*, note that the constructor *ConsRec* definition, has a *LabelSet* constraint. The definition of this family is given in as follows:.

```
type family LabelSetF (r :: [(k, k')]) :: Bool where
  LabelSetF []                   = True
  LabelSetF [(l, v)]             = True
  LabelSetF ((l, v)': (l', v') : r) = Cmp l l' == LT ∧ LabelSetF ('(l', v') : r)
type family LabelSet r = RequireEq (LabelSetF r) True
```

Note that we use the constraint *RequireEq* of the `requirements` library. The constraint *LabelSet* ensures that labels of the record are unique, and ordered. In some iterations during the development of the library, we did not care about how labels were ordered [García-Garland et al., 2019]. Nevertheless, ordering the labels simplifies all record operations except insertion[3]. A drawback is that to be ordered, labels must implement a comparison operation, while without ordering labels, equality is enough. We choose to pay that cost. For this reason, the kind of the labels must implement the type family *Cmp*:

```
type family Cmp (a :: k) (b :: k) :: Ordering
```

Since records have their labels ordered, the constructor *ConsRec* cannot be used to build arbitrary records (we can use *ConsRec* only with a new key smaller than the ones in the domain). A smart constructor (.∗.) is provided to extend a record with any label (raising a type error in case of a label clash).

[3] The use of ordered labels makes the *LabelSet* constraint irrelevant. Records will be built by extending them with a smart constructor that raises type errors if we try to duplicate labels. It still works as a sanity check.

Three more type family instances are required to be implemented to properly define a record instance if we want pretty error messages. Depending on the c index, we decide how to call records with the *ShowRec* type family, and how to call its fields with the *ShowField* type family. How labels are printed, is handled by the *ShowLabel* type family.

```
type family ShowRec   c :: Symbol
type family ShowField c :: Symbol
type family ShowLabel c :: Symbol
```

How do we use those families will be clear in Section 5.2.3.

5.2.2 Examples of record instances.

In the previous section, we defined the data structure *Rec* and discussed briefly the motivation to use polymorphic kinds in fields. We drafted what is necessary to implement a record instance. In this section, we properly implement two record instances, and we show how to use some Haskell features to implement records while hiding the complexity of `poly-rec`.

5.2.2.1 "Vanilla" extensible records

Firstly, we present a "classic" extensible record: mappings from names to types, where the keys consist of type-level strings (types of kind *Symbol*).

We introduce an index to name this kind of record:

```
data Reco
```

Then, we implement the type families *WrapField*, *ShowRec*, *ShowField*, and *ShowLabel*.

```
type instance WrapField Reco (v :: Type)       = v
type instance ShowRec   Reco                   = "Record"
type instance ShowField Reco                   = "field named"
type instance ShowLabel s                      = s
type instance Cmp (a :: Symbol) (b :: Symbol) = CmpSymbol a b
```

The `poly-rec` library already exports an implementation of the *Cmp* family for *Symbol*s. Just for completeness, we show the implementation here, calling the type family *CmpSymbol* exported by the module `GHC.TypeLits`.

Now, we can implement the *Record* type, by instantiating *Rec*:

$$\textbf{type}\ \mathit{Record} = \mathit{Rec}\ \mathit{Reco} :: [(\mathit{Symbol}, \mathit{Type})] \to \mathit{Type}$$

The type *Record* is, of course *Rec* applied to *Reco*, and by annotating its kind we instantiate the kinds k', k''. In the same manner, tagged fields can be specialized:

$$\textbf{type}\ \mathit{Tagged}\ (l :: \mathit{Symbol})\ (v :: \mathit{Type}) = \mathit{TagField}\ \mathit{Reco}\ l\ v$$

Note that by exporting the types *Record* and *Tagged*, while hiding the types *Rec* and *TagField*, clients are not aware that we used the general polykinded definition behind the scenes. By hiding the constructors we lose the ability to perform pattern matching, but we can use pattern synonyms [Pickering et al., 2016] to recover pattern matching:

$$\textbf{pattern}\ \mathit{Tagged} :: v \to \mathit{Tagged}\ l\ v$$
$$\textbf{pattern}\ \mathit{Tagged}\ v = \mathit{TagField}\ \mathit{Label}\ \mathit{Label}\ v$$

The name of the data type *Tagged* was chosen because it behaves exaclty as the type defined in the *Data.Tagged* module of the well-known `tagged` library [Kmett, 2009].

We can define a pretty operator to build tagged fields:

$$\textbf{infix}\ 4 .{=}{=}.$$
$$(.{=}{=}.) :: \mathit{Label}\ l \to v \to \mathit{Tagged}\ l\ v$$
$$l .{=}{=}. v = \mathit{Tagged}\ v$$

In Section 5.2.3 we will show some operations on generic records. For instance, the operator $(.{*}.)$ is a smart constructor for record extension. It has type $\mathit{LabelSet}\ ('(l, v)\text{'}{:}\ r) \Rightarrow \mathit{TagField}\ c\ l\ v \to \mathit{Rec}\ c\ r \to \mathit{Rec}\ c\ ('(l, v)\text{'}{:}\ r)$, the same type as the constructor *ConsRec*, but inserting the field in its correct position to mantain labels ordered. We can also specialize this operation:

$$\textbf{infixr}\ 2\ .{*}{*}.$$
$$(lv :: \mathit{Tagged}\ l\ v)\ .{*}{*}.\ r = lv\ .{*}.\ r$$

The operator $(.{*}{*}.)$ has type
$\mathit{LabelSet}\ ('(l, v)\text{'}{:}\ r) \Rightarrow \mathit{Tagged}\ l\ v \to \mathit{Record}\ r \to \mathit{Record}\ ('(l, v)\text{'}{:}\ r)$
which is more concrete than the type of $(.{*}.)$. In the same way we can define a smart constructor for the empty record:

$\mathit{emptyRecord} :: \mathit{Record}\ ('[] :: [(\mathit{Symbol}, \mathit{Type})])$
$\mathit{emptyRecord} = \mathit{EmptyRec}$

Finally, we have all the machinery to define records such as:

$$\begin{aligned} r = \quad & (\mathit{Label}\ @\ \texttt{"integer"} \mathrel{.{=}{=}.} 3) \\ .{*}{*}. \quad & (\mathit{Label}\ @\ \texttt{"boolean"} \mathrel{.{=}{=}.} \mathit{True}) \\ .{*}{*}. \quad & \mathit{emptyRecord} \end{aligned}$$

The record r has type $(\mathit{Record}\ [(\texttt{"boolean"}, \mathit{Bool}), (\texttt{"integer"}, \mathit{Integer})])$, as one would expect. Note that in the type the order of the fields was inverted, because field names are ordered in lexicographic order.

Using $(\ .{*}{*}.\ )$, $(.{=}{=}.)$ and $\mathit{emptyRecord}$ we can hide the implementation of Rec. However, at the same time note that if the primitive constructors of polykinded records are visible to clients, generic operations over the type Rec can be used for Record. For instance, the following definition is equivalent to r:

$$\begin{aligned} r' = \quad & (\mathit{Label}\ @\ \texttt{"integer"} \mathrel{.{=}{=}.} 3) \\ .{*}. \quad & (\mathit{Label}\ @\ \texttt{"boolean"} \mathrel{.{=}{=}.} \mathit{True}) \\ .{*}. \quad & \mathit{emptyRecord} \end{aligned}$$

While $(\ .{*}.\ )$ is more general than $(\ .{*}{*}.\ )$, its type in this occurence is instantiated to the type of $(\ .{*}{*}.\ )$ since we have enough context to instantiate the variable c of the generic record definition, by the use of $\mathit{emptyRecord}$ or $(.{=}{=}.)$.

5.2.2.2 Two Dimensional Records

Now we implement two dimensional records. We call them matrices [4]. To illustrate how a different type of label can be used, labels will be type-level naturals.

We build an index:

data Mat

then, we define the Matrix type:

type $\mathit{Matrix} = \mathit{Rec}\ \mathit{Mat} :: [(\mathit{Nat}, [(\mathit{Symbol}, \mathit{Type})])] \to \mathit{Type}$

[4] Actually, we define ragged matrices with labelled columns, -dependently- labelled rows, and heterogeneous values.

Note the double list. Each natural label (say, the columns) maps to fields that are a full map from names to values. Those fields have an index that is a type-level list of pairs, not inhabited. Actual field values have a type, built by a type constructor that wraps the index:

$$\textbf{type instance}\ \mathit{WrapField}\ \mathit{Mat}\ (r :: [(\mathit{Symbol}, \mathit{Type})]) = \mathit{Record}\ r$$

Recall we must define information to print errors:

$$\begin{aligned}
&\textbf{type instance}\ \mathit{ShowRec}\ \mathit{Mat} &&= \texttt{"Matrix"}\\
&\textbf{type instance}\ \mathit{ShowField}\ \mathit{Mat} &&= \texttt{"record named"}\\
&\textbf{type instance}\ \mathit{ShowLabel}\ (l :: \mathit{Nat}) &&= \mathit{ShowNat}\ l
\end{aligned}$$

where the type family *ShowNat* computes the string representation of a number. The last type family to implement is *Cmp*, to give a total order for labels. We use the type-level naturals comparison *GHC.TypeNats.CmpNat*:

$$\textbf{type instance}\ \mathit{Cmp}\ (m :: \mathit{Nat})\ (n :: \mathit{Nat}) = \mathit{CmpNat}\ m\ n$$

To avoid extending the length of this example, we can now use the functions over *GenRec* to build a matrix example. The following definition consists of a matrix with the vector r defined in Section 5.2.2.1 in the column tagged by 1 and the empty vector in the column tagged by 2:

$$\begin{aligned}
m = \textbf{let}\ & \mathit{tf} = (\mathit{TagField} :: \forall\ l\ r.\\
& \qquad \mathit{Label}\ \mathit{Mat} \to \mathit{Label}\ l \to \mathit{Record}\ r \to \mathit{TagField}\ \mathit{Mat}\ l\ r)\\
\textbf{in}\ & \quad \mathit{tf}\ \mathit{Label}\ (\mathit{Label}\ @\ 1)\ r\\
& .{*}.\ \mathit{tf}\ \mathit{Label}\ (\mathit{Label}\ @\ 2)\ \mathit{emptyRecord}\\
& .{*}.\ \mathit{EmptyRec}
\end{aligned}$$

We must use an annotated instance of *TagField* to help the type checker. Perhaps the reader would expect the following -cleaner- definition to be enough:

$$\begin{aligned}
m = &\quad \mathit{TagField}\ (\mathit{Label}\ @\ \mathit{Mat})\ (\mathit{Label}\ @\ 1)\ r\\
&.{*}.\ \mathit{TagField}\ (\mathit{Label}\ @\ \mathit{Mat})\ (\mathit{Label}\ @\ 2)\ \mathit{emptyRecord}\\
&.{*}.\ \mathit{EmptyRec}
\end{aligned}$$

Unfortunately, this definition does not type check. The first *TagField* constructor expects a value of type $(\mathit{WrapField}\ \mathit{Mat}\ r)$ for some r. The record r has type $(\mathit{Record}\ [(\texttt{"boolean"}, \mathit{Bool}), (\texttt{"integer"}, \mathit{Integer})])$. The compiler cannot deduce that $(r \sim [(\texttt{"boolean"}, \mathit{Bool}), (\texttt{"integer"}, \mathit{Integer})])$, because

WrapField is not injective, which means that (*Record* r) does not need to be necessarily a result of reducing (*WrapField Mat* r). Many other instances for *WrapField* could be defined reducing to a *Record*! In fact the *WrapField* type family does not need to be injective, in general.

These technical difficulties arise only if we use generic operations eagerly without giving further typing context, and can be avoided by using specialized definitions, though.

5.2.3 Operations over records.

In this section we show how record operations over the polykinded definition are implemented. In this section we will use the term "record" to refer to the general, polykinded definition.

Record lookup given a label, update at a label, and extension are the minimum set of operations needed by `AspectAG`. However, as it evolved to a general-purpose library, more operations were implemented in `poly-rec` such as right and left joins, traversals and so on. To implement record operations we use the standard type-level programming techniques, in combination with the use of the `requirements` library to generate domain-specific type errors. We use the programming patterns presented in Section 5.1. For each operation over records, we build a `requirements` operation. This means, we define an operator, where all arguments are grouped.

We just give one full example: record lookup. Other operations does not offer new insight, they are just applications of the techniques we have developed.

5.2.3.1 Example: Lookup

We build the *OpLookup* operator as an algebraic data type in the following way:

$$\begin{array}{l}\textbf{data}\ \mathit{OpLookup}\ (c :: k)\ (l :: k')\ (r :: [(k', k'')]) :: \mathit{Type}\ \textbf{where}\\ \quad \mathit{OpLookup} :: \mathit{Label}\ l \rightarrow \mathit{Rec}\ c\ r \rightarrow \mathit{OpLookup}\ c\ l\ r\end{array}$$

The lookup operation takes as arguments a label to be looked up, and a record to look up. Lookup is defined recursively. Since labels are ordered, by inspecting the label of the first field of the record we can decide either to return the head value, to perform a recursive call or to fail depending on how it orders respect to the looked up label. That means we must decide the type of the result based on this comparison, so we use the *advanced overlap* pattern to put it in the head of the class definition. For that reason, we define the *OpLookup′* operator, that adds a new index of kind *Ordering*.

```
data OpLookup′ (b :: Ordering) (c :: Type) (l :: k) (r :: [(k, k′)]) :: Type
  where
    OpLookup′ :: Proxy b → Label l → Rec c r → OpLookup′ b c l r
```

To define instances of *Require* we will also pattern match over r, since when $(r \sim {}'[])$ an error must be reported. The instance of *Require* for *OpLookup* for nonempty records is implemented as follows:

```
instance
  Require (OpLookup′ (Cmp l l′) c l ('(l′, v) ': r)) ctx
   ⇒
  Require (OpLookup c l ('(l′, v) ': r)) ctx where
  type ReqR (OpLookup c l ('(l′, v) ': r)) =
    ReqR (OpLookup′ (Cmp l l′) c l ('(l′, v) ': r))
  req ctx (OpLookup l r) =
    req ctx (OpLookup′ (Proxy @ (Cmp l l′)) l r)
```

We simply compute the type $(Cmp\ l\ l')$ to put it explicitly on the head of the class we require as context. At the value level we copy the arguments and add the annotated proxy.

Now we implement instances of *Require* for the auxiliar operator. If the argument label is equal to the head of the record, we have found the field:

```
instance
  Require (OpLookup′ 'EQ c l ('(l, v) ': r)) ctx where
  type ReqR (OpLookup′ 'EQ c l ('(l, v) ': r)) =
    WrapField c v
  req Proxy (OpLookup′ Proxy Label (ConsRec (TagField _ _ v) _)) =
    v
```

the type of the value returned is of course *WrapField c v*.

If the label looked up is greater than the one in the head, that means we should keep traversing the record seeking for the label. We implement the recursive case, calling a lookup with the tail of the record:

```
instance (Require (OpLookup c l r) ctx)
   ⇒
  Require (OpLookup' 'GT c l ('(l', v) ': r)) ctx where
  type ReqR (OpLookup' 'GT c l ('(l', v) ': r)) =
    ReqR (OpLookup c l r)
  req ctx (OpLookup' Proxy l (ConsRec _ r)) =
    req ctx (OpLookup l r)
```

Note that this time, in the instance for *OpLookup′* we call the instance for *OpLookup*. In constrat, in Section 5.1.1 we stuck to *OpGet′* after the first call. The reason is that here we need to update the ordering at each step.

Now we can write the ill-typed cases. If the label we are looking up is smaller than the one in the head of the record, we know the looked up label is not in the record domain, since labels are ordered. We require an instance of *OpError*, as follows:

```
instance Require (OpError (LookupError c l)) ctx
   ⇒
  Require (OpLookup' 'LT c l ('(l', v) ': r)) ctx where { }
```

The case of looking up in an empty record is similar:

```
instance
  Require (OpError (LookupError c l) ctx
   ⇒
  Require (OpLookup c l ('[] :: [(k, k')])) ctx where { }
```

In both error cases, the *OpError* operator takes an error message depending on *l* and *c*. *LookupError* is a type family that computes the string showed to the user.

```
type family LookupError c l
type instance LookupError c l =
  Text "field not Found on " :◇: Text (ShowRec c)
    :$$: Text "looking up the " :◇: Text (ShowField c)
      :◇: Text " " :◇: ShowLabel l)
```

Now it should be clear how *ShowRec*, *ShowField*, and *ShowLabel* play their role here. Ill-typed lookups for the type *Record* (Section 5.2.2.1) or *Matrix* (Section 5.2.2.2) raise type errors such as:

```
Error: field not Found on Record
looking up the field named z
```

or

```
Error: field not Found on Matrix
looking up the record named 3
```

respectively, by evaluating the same type family.

Errors such as the one showed in Section 4.2.5 are also the result of printing this error. Note that the domain-specific type errors only show information about the looked up label and the record instance, since the family *LookupError* depends only on arguments l and c. In particular, information about the full type of the original looked up record is not showed. This was a decision we made to keep errors simple and small, but it is important to note that the developed techniques allow us to do that if desired. Note that, in this case it wouldn't be enough to pass the record index r as an argument to the type family *LookupError*, because the record being looked up when we decide to raise an error is a 'suffix' of the original one. One way to solve this problem is to add the type of the original record as an argument of *OpLookup* and *OpLookup′* (as a phantom type, at value level we do not require anything). Another way is to simply push that information to the context (that, by the way, is untouched in the previous implementation) the first time we call the lookup operation.

Finally, we can write a pretty operator (#) for lookup. This is useful for cosmetic reasons and to hide the use of the `requirements` framework to users. The implementation is the following:

$$r \mathbin{\#} l = \mathit{req}\ \mathit{emptyCtx}\ (\mathit{OpLookup}\ l\ r)$$

Here we used *emptyCtx*, a proxy of type ($\mathit{Proxy}\ '[]$), but as discussed we could use any context (in particular we have the indexes c, l, and r in scope if we want to use them).

In a similar way, we programmed the operations *OpExtend* and *OpUpdate*. The implementation is public in the `AspectAG` repository.

5.3 Implementation of `AspectAG`.

In this section we show how `AspectAG` internal structures are implemented using the techniques presented in the previous sections. A more condensed presentation can be found in [García-Garland et al., 2019]. In `AspectAG` users define building blocks of attribute grammars (productions, children, attributes) as labels. How to define `AspectAG` labels is shown in Section 5.3.1.

Collections of attributes, called *attributions*, are defined as extensible records mapping attribute names to values, using the `poly-rec` library. The details are given in Section 5.3.2.

Semantic rules are implemented as functions from *input families* to *output families*. Data structures for families, rules and collections of rules (*aspects*) are defined in Section 5.3.3.

The interface to define concrete rules is provided in Sections 5.3.4 and 5.3.5.

In Section 5.3.6 we define the *knit* function. The *knit* function does the job of computing the values of the synthesized attributes of a node in a syntax tree, given an aspect and the values of the inherited attributes at the node. The function *knit* is the main engine behind the AG system, building a tangible function to compute the attributed tree from the semantics definitions.

In Section 5.3.7 we show how to use the contexts given in `requirements` to implement traces.

In Section 5.3.8 we disuss how polymorphic attributes and terminals can be implemented in a simple way exploiting the type system of the host language, without affecting the domain-specific type errors.

In Section 5.3.9 we discuss a performance issue that we found when using the library, and provide an optimization that solves it.

In Section 5.3.10 we give an overview of what have we changed with respect to the original implementation of the `AspectAG` library.

5.3.1 Data structures for label types.

Building blocks of `AspectAG` are defined with values of type *Label* instantiated with the types defined in Figure 5.9. We already introduced the type *Label* in the `poly-rec` library. The five defined data types can only be used in its promoted form as a data kind, since all of them have at least a field of type *Symbol* or *Type*, that have no term-level inhabitants.

```
data Att   = Att Symbol Type
data Prod  = Prd Symbol NT
data Child = Chi Symbol Prod (Either NT T)
data NT    = NT Symbol
data T     = T  Type
```

Figure 5.9: Definition of data kinds for labels.

Attribute names are implemented with the type *Att*. Attributes have a name of kind *Symbol* and an associated type. Productions are implemented with the type *Prod*. A production has a name and it is related to a non-terminal whereas it belongs. Non-terminals are tagged with a name, implemented in the *NT* type. Children are implemented with the type *Child*. They have a name and they belong to a production. Each child represents a position in the right-hand side of a production, so a child is related to either a non-terminal or a terminal, depending to what kind of position it holds in the production.

5.3.2 Attributes and Attributions.

Once we have labels to name attributes, we can implement a data structure for attributes. Moreover, we are interested in implementing *attributions*. Attributions are collections of attribute values, i.e. mappings from attributes to values. Thus, attributions are implemented as a record instance, using the type *Att* as the type of labels. In Figure 5.10 the implementation is given.

We define *AttReco*, the tag to identify the record instance, and the record type is called *Attribution*. For the kind *Att* to represent fields of this record instance, it must implement an order among its values. This is given in the extension of the open type family *Cmp*, that compares attributes by their names.

To correctly print type errors, type instances for *ShowField*, *ShowRec*, and *ShowLabel* are given. In the latter one we use the operator (:++), the concatenation over symbols. This record instance is called "Attribution" and each field is called "attribute". When printing an attribute of name "foo" it is ok to refer it with the string "attribute named foo", which explains the definition of *ShowField*.

```
data AttReco

type Attribution (attr :: [(Att, Type)]) = Rec AttReco attr
type instance Cmp ('Att a _) ('Att b _) = CmpSymbol a b

type instance ShowRec AttReco     = "Attribution"
type instance ShowField AttReco   = "attribute named"
type instance ShowLabel ('Att l t) = "(" :++ l :++ ":" :++ ShowType t :++ ")"

type Attribute (l :: Att) (v :: Type) = TagField AttReco l v
pattern Attribute :: v → TagField AttReco l v
pattern Attribute v = TagField Label Label v

infixr 4 =.
( =. ) :: Label l → v → Attribute l v
Label =. v = Attribute v

infixr 2 *.
(l :: Attribute att val) *. (r :: Attribution atts) = l .*. r

emptyAtt = EmptyRec :: Attribution [ ]
```

Figure 5.10: Attribution definition.

```
type ChAttsRec prd (chs :: [(Child, [(Att, Type)])])
  = Rec (ChiReco prd) chs
data ChiReco (prd :: Prod)
type instance WrapField (ChiReco prd) v
  = Attribution v
type instance ShowRec (ChiReco prd)   = "Children Map"
type instance ShowField (ChiReco prd) = "child labelled"
type instance ShowLabel (Chi l p s)
  = "Child " :++ l :++ " of producion " :++ ShowLabel p
```

Figure 5.11: Children record definition.

A set of specialized combinators is also defined in Figure 5.10 such as the *Attribute* constructor, useful for pattern matching; the constructor (=.) to build attributes; the operator (∗.) for attribution extension; and the value *emptyAtt*, the empty attribution.

As a final clarification, note that when defining the instance for the type family *Cmp* for the type *Att*, we did not consider the second argument of *Att*, neither we consider it later. It is arguably that, since a label of kind $(Att\ s\ t)$ represents an attribute named s ot type t, we should forbid to build values of type $(Attribute\ (Att\ s\ t)\ t')$ if $t \not\sim t'$ (using the `requirements` library, of course!). However, just by an implementation decision, instead of controlling this here, we will take care of it in the `AspectAG` combinators as we shall see later.

5.3.3 Families, rules and aspects

A family represents inputs and outputs of computations ocurring in a node of the attribute grammar. A family contains a single attribution for the father and a collection of attributions for children. The type *Fam* is a GADT indexed by all its components, defined as follows:

```
data
  Fam (prd :: Prod) (c :: [(Child, [(Att, Type)])]) (p :: [(Att, Type)]) :: Type
where
  Fam :: ChAttsRec prd c → Attribution p → Fam prd c p
```

The collection of children attributions *ChAttsRec* is another record instance, where labels are children and fields are a full attribution. In Figure 5.11 the most relevant definitions of the instance are given.

A novelty is that we introduce the tag *ChiReco* with an arity. Children records belong to a production, so its type, *ChAttsRec* is indexed by a *prd*. In the type *Rec* we use the argument c to store that phantom type. Also note the instance for *WrapField*, this time wrapping the *Attribution* constructor.

Just as with attributions the full interface is redefined, for instance $(.*)$ and $(.\#)$ are functions to extend and access children records, and $(.=)$ makes a field, associating a child and an attribution.

Rules are functions from the input family (inherited attributes from the parent, and synthesized from the children) to the output family (synthesized attributes of the parent, inherited to the children). To make families composable an extra arity is added, a technique already defined in [de Moor et al., 2000]. The type is defined as follows:

```
type Rule
  (prd :: Prod)
  (sc  :: [(Child, [(Att, Type)])])
  (ip  :: [(Att, Type)])
  (ic  :: [(Child, [(Att, Type)])])
  (sp  :: [(Att, Type)])
  (ic' :: [(Child, [(Att, Type)])])
  (sp' :: [(Att, Type)])
   = Fam prd sc ip → Fam prd ic sp → Fam prd ic' sp'
```

One can interpret this as a function that given an input family, it builds a function that updates the output family constructed thus far.

To print domain-specific type errors using the `requirements` library, each rule can save some context information to be shown in case the rule triggers it, so we will really use rules with attached context, as follows:

```
newtype CRule (ctx :: [ErrorMessage]) prd sc ip ic sp ic' sp'
   = CRule {mkRule :: (Proxy ctx → Rule prd sc ip ic sp ic' sp')}
```

Rules of the same production can be combined. Given an input family, to combine two rules is to compose the resulting applied rules:

```
data PrdReco
type instance WrapField PrdReco (rule :: Type)
  = rule
type Aspect (asp :: [(Prod, Type)]) = Rec PrdReco asp
type instance ShowRec PrdReco   = "Aspect"
type instance ShowField PrdReco = "production named"
type instance ShowLabel ('Prd l nt)
  = "(" :++ l :++ " of " :++ ShowTE nt :++ ")"
```

Figure 5.12: Aspect definition.

```
ext' :: CRule ctx prd sc ip ic sp ic' sp'
  -> CRule ctx prd sc ip a b ic sp
  -> CRule ctx prd sc ip a b ic' sp'
(CRule f) `ext'` (CRule g)
  = CRule $ λctx input -> f ctx input ∘ g ctx input
```

The function *ext'* is auxiliar. The real combination controls if the combined rules belong to the same production:

```
ext :: RequireEq prd prd' (Text "ext" : ctx)
  => CRule ctx prd sc ip ic sp ic' sp'
  -> CRule ctx prd' sc ip a b ic sp
  -> CRule ctx prd sc ip a b ic' sp'
ext = ext'
```

The function *ext* is also exported in `AspectAG` as the operator $(\diamond)$. We chose to use the *RequireEq* constraint here for brevity, but the actual implementation uses *RequireEqWithMsg* with a suitable error message format.

An *aspect* is a set of rules. Aspects are implemented as a mapping from productions to rules, again, as a record instance, as given in Figure 5.12.

Like rules, user defined aspects are decorated with context, so we define the proper data structure used as:

```
newtype CAspect (ctx :: [ErrorMessage]) (asp :: [(Prod, Type)])
  = CAspect {mkAspect :: Proxy ctx -> Aspect asp}
```

The interface for aspects is a bit different to a set of specialized record operations. Aspects can be built by a set of combinators. The value *emptyAspect*

Left argument	Right argument	operation	exported operator
rule	rule	none (*ext*)	$(\diamond)$
rule	aspect	*OpComRA*	$(\triangleleft)$
aspect	rule	none (*flip* $(\triangleleft)$)	$(\triangleright)$
aspect	aspect	*OpComAsp*	$(\bowtie)$

Figure 5.13: Semantic combinators.

represents the empty aspect, the function *singAsp* creates an aspect from a single rule.

$$\begin{aligned} &\mathit{emptyAspect} &&= \mathit{CAspect}\ \$\ \mathit{const}\ \mathit{EmptyRec} \\ &\mathit{singAsp}\ r &&= r \triangleleft \mathit{emptyAspect} \end{aligned}$$

Then, aspects can be combined by adding new rules to existing aspects or by merging already defined aspects. In Figure 5.13 we show the main combinators that work over rules and aspects.

To combine rules we used the already defined function *ext*, also defined as the operator $(\diamond)$.

To combine a rule and an aspect the operator $(\triangleleft)$ can be used, and for convenience there is a mirror operator $(\triangleright)$. The former associates to the right, while the latter does to the left, which is useful to write combinations minimizing brackets.

To implement those functions we use the already presented techniques. In particular, we define an operator *OpRuleAsp* to do the job using the `requirements` library. There are two cases: if the rule is related to a production not belonging to the aspect domain, we just extend the mapping (using record extension). On the contrary, if there is already a rule for that production, we combine those rules, updating the record. Then the operator $(\triangleleft)$ is implemented as a call to *req*, and $(\triangleright)$ is the flipped version.

Finally, we can combine two aspects by merging them, using the operator $(\bowtie)$. By merging, we mean that when both arguments have rules for a given production, the rules are combined. This is implemented again using the techniques of the `requirements` library, with the operator *OpComAsp*.

The implementation of these functions does not offer new insights. They can be found in the repository.

5.3.4 Defining concrete rules and aspects.

In this section we show how to implement concrete rules. We need to have available at least a way to define rules to compute either a single synthesized attribute or a single inherited attributte, given a production. Then, we have available the combinators defined in the previous section to build more complex semantics.

5.3.4.1 Synthesized attributes.

The function *syndef* builds rules to compute a synthesized attribute. It takes an attribute label *att*, a production label *prd* and a function f. The function f computes the value of the attribute from the input family. The extra arity is used to carry the context.

$$
\begin{aligned}
&\mathit{syndef}\\
&\quad ::\ \mathit{Syndef}\ t\ t'\ \mathit{ctx}\ \mathit{att}\ \mathit{sp}\ \mathit{sp}'\ \mathit{prd}\ \mathit{prd}'\\
&\quad \Rightarrow \mathit{Label}\ (\text{`}\mathit{Att}\ \mathit{att}\ t)\\
&\quad \to \mathit{Label}\ \mathit{prd}\\
&\quad \to (\mathit{Proxy}\ \mathit{ctx} \to \mathit{Fam}\ \mathit{prd}'\ \mathit{sc}\ \mathit{ip} \to t')\\
&\quad \to \mathit{CRule}\ \mathit{ctx}\ \mathit{prd}\ \mathit{sc}\ \mathit{ip}\ \mathit{ic}\ \mathit{sp}\ \mathit{ic}\ \mathit{sp}'\\
&\mathit{syndef}\ \mathit{att}\ \mathit{prd}\ f\\
&\quad = \mathit{CRule}\ \$\ \lambda \mathit{ctx}\ \mathit{inp}\ (\mathit{Fam}\ \mathit{ic}\ \mathit{sp})\\
&\qquad \to \mathit{Fam}\ \mathit{ic}\ \$\ \mathit{req}\ \mathit{ctx}\ (\mathit{OpExtend}\ \mathit{att}\ (f\ \mathit{Proxy}\ \mathit{inp})\ \mathit{sp})
\end{aligned}
$$

The constraint ($\mathit{Syndef}\ t\ t'\ \mathit{ctx}\ \mathit{att}\ \mathit{sp}\ \mathit{sp}'\ \mathit{prd}\ \mathit{prd}'$) is implemented as a type family that reduces to the set of constraints we need to be fullfilled. Each constraint is a requirement of the `requirements` library and works to catch a particular type error of those presented in Chapter 4. In this case in particular, the type of the attribute according to the label must be equal to the type of the one computed in f. The type of the label of the production argument must be equal to the one of the family appearing in the computation f. The resulting record sp' (synthesized attributes of the parent after the rule is applied) must be the result of updating *sp* (synthesized attributes of the parent before the rule was computed). Moreover, the update consists on extending the record with a field of name ($\mathit{Att}\ \mathit{att}\ t$) with type t.

```
type family Syndef t t′ ctx att sp sp′ prd prd′ :: Constraint where
  Syndef t t′ ctx att sp sp′ prd prd′ =
    (RequireEqWithMsg t t′ AttTypeMatch ctx
    , RequireEqWithMsg prd prd′ PrdTypeMatch ctx
    , RequireR (OpExtend AttReco (‘Att att t) t′ sp) ctx (Attribution sp′)
    )
```

The constraint *RequireR* is an alias for two constraints. When we pattern match on the result of an application of *ReqR*, we give an equality constraint of shape *ReqR op* ∼ *res*. This constraint is always paired with one constraint of shape *Require op ctx* since *ReqR* is an associated type in the class *Require*. The type family *RequireR* is defined as follows:

```
type RequireR op (ctx :: [ErrorMessage]) res =
  (Require op ctx, ReqR op ∼ res) :: Constraint
```

As seen in Chapter 3 we used a monadic interface to define rules. Instead of the function *syndef*, we used the function *syndefM*. We hide the context arguments of *f* using a reader monad, putting both the proxy carrying context about the type error and the input family in the environment. So, the monadic definition can be given as:

```
syndefM
  :: Syndef t t′ ctx att sp sp′ prd prd′
  ⇒ Label (‘Att att t)
  → Label prd
  → Reader (Proxy ctx,
    Fam prd sc ip) t′
  → CRule ctx prd sc ip ic sp ic sp′
syndefM att prd f
  = syndef att prd ∘ curry ∘ runReader $ f
```

The trick is simply to use the usual *Reader* type to abstract an uncurried version of a function, in this case *f*. Then we can apply *curry* to get an argument for *syndef*.

Moreover, recall that we saw in Chapter 4 that *syndefM* implicitly added information to the trace. So, the actual definition of *syndefM* is given as:

```
syndefM
  :: Syndef t t' (MkMsg SyndefMsg att t prd nt': ctx) att sp sp' prd prd'
  ⇒ Label ('Att att t)
  → Label ('Prd prd nt)
  → Reader (Proxy (MkMsg SyndefMsg att t prd nt': ctx),
     Fam ('Prd prd' nt) sc ip) t'
  → CRule ctx ('Prd prd nt) sc ip ic sp ic sp'
syndefM att prd f
  = mapCRule (mkMsg (Proxy @ SyndefMsg) att prd `consErr`)
    $ syndef att prd ∘ curry ∘ runReader $ f
```

The difference with respect to the previous definition is that a message is 'pushed' into the constraints. The type family *MkMsg* of kind (*Type* → *Symbol* → *Type* → *Symbol* → *NT* → *ErrorMessage*) simply computes a pretty printed error message. The function *mapCRule* has no value-level computational content, it is just a coercion that adds context information. We define it in Section 5.3.7. Note that we pattern match over the type *prd* just because we need its field *nt* to build the error message.

5.3.4.2 Inherited Attributes.

The function *inhdef* builds rules to compute an inherited attribute. It is defined with the same techniques as *syndef*, though it is slightly more complex since a child comes into the equation. It takes an attribute label *att*, a production label *prd*, a child *chi*, and a function *f*. The function *inhdef* is defined as follows:

```
inhdef
  :: Inhdef t t' ctx att r r2 prd prd' nt nt' chi ntch ic ic' n
  ⇒ Label ('Att att t)
  → Label ('Prd prd nt)
  → Label ('Chi chi ('Prd prd' nt') ntch)
  → (Proxy ctx → Fam ('Prd prd nt) sc ip → t')
  → CRule ctx ('Prd prd nt) sc ip ic sp ic' sp
inhdef att prd chi f =
  CRule $ λctx inp (Fam ic sp) →
    let ic'    = req ctx (OpUpdate chi catts' ic)
        catts  = req ctx (OpLookup chi ic)
        catts' = req ctx (OpExtend att (f Proxy inp) catts)
      in Fam ic' sp
```

Given the input family (*Fam ic sp*), what the algorithm does is to lookup the child *chi* in the inherited attributes of the children *ic*, to obtain its attribution *catts*. That attribution is extended with the new field *att* with the value returned by *f*, resulting in the attribution *catts'*. Then, the record *ic* is updated with the new attribution, resulting in *ic'*. Finally the updated family (*Fam ic' sp*) is returned. Again, the constraint (*Inhdef t t' ctx att r r2 prd prd' nt nt' chi ntch ic ic' n*) computes all requirements:

```
type family Inhdef t t' ctx att r r2 prd prd' nt nt' chi ntch ic ic' n
  where
  Inhdef t t' ctx att r r2 prd prd' nt nt' chi ntch ic ic' n
    = (RequireEqWithMsg t t' AttTypeMatch ctx
      , RequireEqWithMsg ('Prd prd nt) ('Prd prd' nt') ChiPrdMatch ctx
      , RequireEqWithMsg nt nt' NTMatch ctx
      , RequireR (OpExtend AttReco ('Att att t) t' r) ctx (Attribution r2)
      , RequireR (OpUpdate (ChiReco ('Prd prd nt))
         ('Chi chi ('Prd prd' nt') ntch) r2 ic) ctx
           (ChAttsRec ('Prd prd' nt') ic')
      , RequireR (OpLookup (ChiReco ('Prd prd nt))
         ('Chi chi ('Prd prd' nt') ntch) ic) ctx (Attribution r)
      )
```

Then, we can define the monadic version *inhdefM*:

```
inhdefM
   :: Inhdef t t′ (MkMsg InhdefMsg att t prd nt'′: ctx)
     att r r2 prd prd′ nt nt′ chi ntch ic ic′ n
   ⇒ Label ('Att att t)
   → Label ('Prd prd nt)
   → Label ('Chi chi ('Prd prd′ nt′) ntch)
   → Reader (Proxy (MkMsg InhdefMsg att t prd nt'′: ctx), Fam ('Prd prd nt) sc ip) t′
   → CRule ctx ('Prd prd nt) sc ip ic sp ic′ sp
inhdefM att prd chi f = mapCRule (mkMsg (Proxy @ InhdefMsg) att prd ‘consErr‘)
   $ (inhdef att prd chi ∘ def) f
```

In a similar manner, more functions such as *synmod*, *inhmod*, *synmodM*, and *inhmodM* are defined, modifying attributes instead of defining new ones.

5.3.5 Monadic selectors.

As we have shown in the running example of Chapter 3, users can use the monadic interface (or the more general applicative interface) to define the last argument of functions such as *syndefM* or *inhdefM*. Though we could, we never access to the input family with the *ask* function the reader monad interface gives us. Instead, we use a particular interface we provide, with the *at* keyword.

The function *at* takes what we call a 'position' and an attribute. A position can be a child, or the special value *lhs*. The function *at* is defined in a typeclass (since it is ad-hoc polymorphic). The class *At* is defined as follows:

```
class (Monad m) ⇒ At pos att m where
type ResAt pos att m
at :: Label pos → Label att → m (ResAt pos att m)
```

We implement one instance for each position.

5.3.5.1 Selecting the parent.

To use the *at* function with the left hand side, first we define a type to be used as an instance of *pos*, and the value-level label:

```
data Lhs
lhs :: Label Lhs
lhs = Label
```

```
instance
  (RequireR (OpLookup @ Att @ Type AttReco ('Att att t) par) ctx t
  , RequireEqWithMsg t t' AttTypeMatch ctx)
  ⇒
  At Lhs ('Att att t) (Reader (Proxy ctx, Fam prd chi par)) where
  type ResAt Lhs ('Att att t) (Reader (Proxy ctx, Fam prd chi par))
    = ReqR (OpLookup @ Att @ Type AttReco ('Att att t)
      (UnWrap @ Att @ Type (Rec AttReco par)))
  at lhs att
    = liftM (λ(ctx, Fam _ par) → req ctx (OpLookup att par)) ask
```

Figure 5.14: Parent selector.

In Figure 5.14 we implement the semantics for the use of the idiom (*at lhs att*). What we do is to lift the function that gets the attribute from the input family into the Reader monad. The function is defined using pattern matching over the family, to get the inherited attribution. Since the class *At* is kind-polymorphic in its first two arguments, we had to convince the compiler that our definition makes sense annotating kinds in some places.

5.3.5.2 Selecting children.

In the same way we can implement *at* for children. The code given in Figure 5.15 perhaps looks complicated, but it is actually straightforward. Note that the formal parameter *pos* is instantiated with a type of kind *Chi*, while in the previous case *Lhs* had kind *Type*. This is a valid way to use kind polymorphism, though one could arguably prefer to use a fixed data kind that considers all posible type of positions. We preferred the presented approach to make *At* extensible for the case that users want to extend the library. For instance, to implement a way to refer to more distant ancestors or local attributes.

```
instance
    (RequireR (OpLookup (ChiReco prd') ('Chi ch prd nt) chi) ctx
              (Attribution r)
    , RequireR (OpLookup AttReco ('Att att t) r) ctx t'
    , RequireEqWithMsg prd prd' PrdTypeMatch ctx
    , RequireEqWithMsg t t' GetAttTypeMatch ctx
    , ReqR (OpLookup @ Att @ Type AttReco ('Att att t')
      (UnWrap @ Att @ Type (Rec AttReco r)))
        ~ t'
    , r ~ UnWrap (Attribution r)
    )
  ⇒ At ('Chi ch prd nt) ('Att att t) (Reader (Proxy ctx, Fam prd' chi par)) where
  type ResAt ('Chi ch prd nt) ('Att att t) (Reader (Proxy ctx, Fam prd' chi par))
    = ReqR (OpLookup AttReco ('Att att t)
      (UnWrap @ Att @ Type (ReqR (OpLookup (ChiReco prd) ('Chi ch prd nt) chi))))
  at ch att
    = liftM (λ(ctx, Fam chi _) → let atts = req ctx (OpLookup ch chi)
                                 in  req ctx (OpLookup att atts)) ask
```

Figure 5.15: Children selector.

5.3.6 Semantic functions and the knit function.

In this section, we explain how to build semantic functions and what the function *knit* does.

5.3.6.1 An algorithm to derive semantic functions.

Semantic functions compute the synthesized attributes of a node given its inherited attributes.

Semantic functions are parametrized over an aspect that represents concrete semantics. In the forested approach, where we define an AST for the language, semantic functions take an AST. In the deforested approach they take semantic functions for subexpressions.

All semantic functions can be derived from the grammar structure. When consuming a node of the abstract suntax tree, we call some variant of a *knit* function with the rule to compute attributes at the current production and a record of semantic functions for recursively calculate on children.

To understand it we use the running example given in Chapter 3. Recall that we had an expression language where one production represented addition of subexpressions. The production was represented with a label p_{Add}. [5] Children were identified with the values ch_{Add_r} and ch_{Add_l}. So, the clause that defines the semantic function is:

$$\begin{aligned}
&sem_{Expr}\ asp\ (Add\ l\ r)\\
&\quad = knitAspect\ p_{Add}\ asp\ \$\ ch_{Add_l}\ .=.\ sem_{Expr}\ asp\ l\\
&\qquad\qquad\qquad\qquad\quad .*.\ ch_{Add_r}\ .=.\ sem_{Expr}\ asp\ r\\
&\qquad\qquad\qquad\qquad\quad .*.\ emptyGenRec
\end{aligned}$$

The function *knitAspect* is an utility that actually calls *knit* with the rule in *asp* indexed by the production p_{Add}. So, the following definition would be equivalent (apart of some context manipulation as we will see later):

$$\begin{aligned}
&sem_{Expr}\ asp\ (Add\ l\ r)\\
&\quad = knit\ ctx\ (asp\ \#.\ p_{Add})\ \$\ ch_{Add_l}\ .=.\ sem_{Expr}\ asp\ l\\
&\qquad\qquad\qquad\qquad\qquad .*.\ ch_{Add_r}\ .=.\ sem_{Expr}\ asp\ r\\
&\qquad\qquad\qquad\qquad\qquad .*.\ emptyGenRec
\end{aligned}$$

What we do to define a semantic function is to apply *knit* to the corresponding rule and a record with all semantic functions of the children. Each semantic function of the children knows how to compute the synthesized attribution from the inherited one.

5.3.6.2 The function *knit*.

The function *knit* does the hard work. It takes the rule for a node and the record with the semantic functions of the children, and builds a function from the inherited attributes of the parent to its synthesized attributes.

Recall that a rule is implemented as a function from the input family to the output family. However, to make rules composable we use an extra arity and implement rules as a function that takes the input family to build a function that updates the output family constructed thus far. When we combine rules we compose those update functions. What we need to start computing at a node is an initial input family with empty attributions in the parent and in each children position. We can create this kind of structure at any place where we have the type information (labels) of the full production, but actually, in the semantic function definition is where we have it by the first time.

[5] Remember that the type of p_{Add} is *Label* (*Prd* `"Add"` (*NT* `"Expr"`)). That type information is what defines it, as with the other labels. For brevity we just identify the values here.

This was designed this way to make the library extensible. We define non-terminals without fixing the productions to rewrite them, and the producions without defining the children they have. When we define rules we introduce some dependencies (a rule for a production expects the children it uses), but it never defines that set of children.

The function *empties* builds an empty children record from a collection of semantic functions:

```
class Empties (fc :: [(Child, Type)]) (prd :: Prod) where
  type EmptiesR fc :: [(Child, [(Att, Type)])]
  empties :: Record fc → ChAttsRec prd (EmptiesR fc)
```

The implementation it is a routine application of type-level programming techniques, where we define the children record inductively using the labels of the semantic function record. Note that here is the place where the contents of children records in productions is settled down, and therefore the structure of attributed trees. The types of the labels determine to which production each child belongs, but do not determine the set of children a production has, until this point.

The function *kn* takes the semantic functions of the children, and the record with their inputs, to compute a record with the results for each children.

```
class Kn (fcr :: [(Child, Type)]) (prd :: Prod) where
  type ICh fcr :: [(Child, [(Att, Type)])]
  type SCh fcr :: [(Child, [(Att, Type)])]
  kn :: Record fcr → ChAttsRec prd (ICh fcr) → ChAttsRec prd (SCh fcr)
```

This dependent function is defined by induction over the first argument. From the type of semantic functions all other types can be derived, in particular we easily compute the indexed families *ICh* and *SCh* (inherited and synthesized attributes of the children).

Finally, we can define the proper *knit* function:

```
knit (ctx :: Proxy ctx)
  (rule :: CRule ctx prd (SCh fc) ip (EmptiesR fc) '[] (ICh fc) sp)
  (fc    :: Record fc)
  (ip    :: Attribution ip)
   = let Fam ic sp = mkRule rule ctx (Fam sc ip) initFam
         sc         = kn fc ic
         ec         = empties fc
         initFam    = Fam ec emptyAtt
     in  sp
```

What we do, is to create an empty starting family (*initFam*) with the empty children record created with *empties* and an empty attribution for the parent. The function *mkrule* has type ($CRule\ ctx\ prd\ sc\ ip\ ic\ sp\ ic'\ sp' \rightarrow Proxy\ ctx \rightarrow Rule\ prd\ sc\ ip\ ic\ sp\ ic'\ sp'$), context matters for printing type errors, but the important information to understand this algorithm is that the expression (*mkRule rule ctx*) is a rule, i.e it has type with shape ($Fam\ prd\ sc\ ip \rightarrow Fam\ prd\ ic\ sp \rightarrow Fam\ prd\ ic'\ sp'$). Moreover, some of those type variables are instantiated, and the expression further applied. The next arguments are the input family and the initial family. The input family is given by the expression (*Fam sc ip*). The inherited attributes of the parent *ip* were of course given as a parameter. The synthesized of the children *sc* are calculated applying *kn*. But note, that to apply *kn* we need the inherited attributes for the children, wich are actually taken from the output family, pattern matching on the result. The circularity of the definition should not surprise any lazy functional programmer. Of course, this could lead to non-terminating (and worse, non-productive) code. This will depend on how the rules are built, and we do not consider it a misfeature, but a necessity if we want a general system.

5.3.6.3 Implementation of terminals.

We implement the terminals of a grammar having a child that always contains an unique inherited attribute of name `"Ter"`. Note, for instance in the running example of Chapter 3 we had the following clause to define the semantic function:

$$
\begin{aligned}
&sem_{Expr}\ asp\ (Val\ val)\\
&\quad = knitAspect\ p_{Val}\ asp\ \$\ ch_{Val_{val}}\ .{=}.\ sem_{Lit}\ val\\
&\qquad\qquad\qquad\qquad\qquad\quad .{*}.\ emptyGenRec
\end{aligned}
$$

Recall $ch_{Val_{val}}$ was the child name for the terminal. The sem_{Lit} function builds the semantic function "pushing up" the value *val* (the terminal in the AST). So, sem_{Lit} is applied to a value to build a function from the empty attribution (since terminals have no synthesized attributes) to the attribution with the unique attribute `"Ter"` mapped to the value *val*.

We implement this with a typeclass with an unique instance:

$$
\begin{aligned}
&\textbf{class}\ SemLit\ a\ \textbf{where}\\
&\quad sem_{Lit} :: a \to Attribution\ ('[] :: [(Att, Type)])\\
&\qquad \to Attribution\ [(\text{`}Att\ \texttt{"term"}\ a, a)]\\
&\quad lit :: Label\ (\text{`}Att\ \texttt{"term"}\ a)\\
&\textbf{instance}\ SemLit\ a\ \textbf{where}\\
&\quad sem_{Lit}\ a\ \_ = (Label\ \ .{=}.\ a)\ {*}.\ emptyAtt\\
&\quad lit = Label\ @\ (\text{`}Att\ \texttt{"term"}\ a)
\end{aligned}
$$

The *lit* value is the label that references the terminal label.

5.3.7 Trace manipulation.

In this chapter we have already shown that types such as *CRule* or *CAspect* carry context information that it is used in *Require* constraints to print domain-specific type error messages. In Chapter 4 we shown how the functions *traceAspect* and *traceRule* were provided to users of `AspectAG` to assist them in debugging their definitions. Both functions *traceRule* and *traceAspect* are implemented using the more general *mapCRule* (already used in Section 5.3.4 to define *syndefM* and *inhdefM*) and *mapCAspect*. Let us see how we implement them.

The function *mapCRule* transforms the context of a rule using a function that transforms contexts.

$$
\begin{aligned}
&mapCRule :: (Proxy\ ctx \to Proxy\ ctx')\\
&\qquad\qquad\quad \to CRule\ ctx'\ prd\ sc\ ip\ ic\ sp\ ic'\ sp'\\
&\qquad\qquad\quad \to CRule\ ctx\ prd\ sc\ ip\ ic\ sp\ ic'\ sp'\\
&mapCRule\ fctx\ (CRule\ frule) = CRule\ \$\ frule \circ fctx
\end{aligned}
$$

Then, we can implement *traceRule*. It takes a proxy containing an error message and a rule, returning a rule with the appended message.

$$\begin{aligned}&\mathit{traceRule}\ (\_ :: \mathit{Proxy}\ (e :: \mathit{ErrorMessage}))\\ &\quad = \mathit{mapCRule}\ \$\ \lambda(\_ :: \mathit{Proxy}\ \mathit{ctx}) \rightarrow\\ &\qquad \mathit{Proxy}\ @\ (\mathit{Text}\ \texttt{"- traceRule: "} :\!\diamond\!:\ e : \mathit{ctx})\end{aligned}$$

Note that no manipulation is done at the value level.

Perhaps surprisingly, note that the order of the contexts is inverted in the rules with respect to the proxy arguments. This is not a mistake, in fact, the type of the function *traceRule* is the following:

$$\begin{aligned}\mathit{traceRule}\ ::\ &\mathit{Proxy}\ e\\ &\rightarrow \mathit{CRule}\ ((\text{'}\mathit{Text}\ \texttt{"- traceRule: "}\ ' :\!<>\!:\ e) : \mathit{ctx})\ \mathit{prd}\ \mathit{sc}\ \mathit{ip}\ \mathit{ic}\ \mathit{sp}\ \mathit{ic}'\ \mathit{sp}'\\ &\rightarrow \mathit{CRule}\ \mathit{ctx}\ \mathit{prd}\ \mathit{sc}\ \mathit{ip}\ \mathit{ic}\ \mathit{sp}\ \mathit{ic}'\ \mathit{sp}'\end{aligned}$$

What is going on here? When we apply *traceRule* we are actually pushing type information into the type of the argument, instead of the return type. This can be counterintuitive because one tends to think that information of functions flows from the arguments to the computed values; and it is true. However, here we use a type in the right of an $(\rightarrow)$ to compute the type in the left, which is a different thing.

Why *traceRule* makes sense as it is can be more clear with an example. Consider the following expression:

$$\begin{aligned}r = &(\mathit{traceRule}\ (\mathit{Proxy}\ @\ Msg_a)\ \$\ r_a)\\ &\diamond\ (\mathit{traceRule}\ (\mathit{Proxy}\ @\ Msg_b)\ \$\ r_b)\end{aligned}$$

where r_a, r_b are suitable rules; and Msg_a, Msg_b different error messages. Applications of *traceRule* add the messages Msg_a/Msg_b to the context of r_a/r_b, respectively, but the combined rules have the same context (remember that the operator $(\diamond)$, defined in Section 5.3.3 combines rules with the same context).

Moreover, if we define the following rule:

$$r' = \mathit{traceRule}\ (\mathit{Proxy}\ @\ Msg_r)\ \$\ r$$

we are instantiating further the context of r, and so the contexts of rules r_a and r_b! Suppose we use r' to build an aspect and use it as the argument of a semantic function. If an error is raised due to a *Require* constraint introduced by the rule r_a, then we will see Msg_r and Msg_a in the trace, while if an error is raised due to a *Require* constraint introduced by the rule r_b, then we will see Msg_r and Msg_b in the trace.

Moreover, the type of the context argument of r' will be such as:

$$\mathit{Text}\ \texttt{"- traceRule: "}\ {:}\diamond{:}\ Msg_r\ \text{'}{:}\ \mathit{ctx}$$

which means *ctx* can be still instantiated further by new messages, in that case adding information to the trace of r', and to the traces of r_a and r_b.

The function *mapCAspect* transforms the context of an aspect using a function that transforms contexts. However, aspects are implemented as a record with rules, each one having a context attached. To show context information added in applications of *mapCAspect* when a type error is raised by the use of a rule in the aspect, we must add the type information to each rule. So, just as with the function *mapCRule*, the function *mapCAspect* resembles a usual *map* function in the functional programming jargon, though this time being a recursive traversal.

In Figure 5.16 we implement the auxiliar function *MapCtxAsp*, and then we can implement *mapCAspect*:

$$\mathit{mapCAspect}\ \mathit{fctx}\ (\mathit{CAspect}\ \mathit{fasp}) = \mathit{CAspect}\ \$\ \mathit{mapCtxRec}\ \mathit{fctx} \circ \mathit{fasp} \circ \mathit{fctx}$$

Finally, *traceAspect* can be given as:

$$\begin{aligned}&\mathit{traceAspect}\ (\_ :: \mathit{Proxy}\ (e :: \mathit{ErrorMessage}))\\ &\quad = \mathit{mapCAspect}\ \$\ \lambda(\_ :: \mathit{Proxy}\ \mathit{ctx}) \rightarrow\\ &\qquad \mathit{Proxy}\ @\ ((\mathit{Text}\ \texttt{"- traceAspect: "}\ {:}\diamond{:}\ e) : \mathit{ctx})\end{aligned}$$

5.3.8 Polymorphism.

In Section 3.5 we discussed how to implement polymorphic languages using `AspectAG`. We can define polymorphic attributes and polymorphic non-terminals. When using them, we endow semantics (rules and aspects) with a proxy argument holding in its type the polymorphic variables we are considering. Then, we apply that type variable to occurrences of polymorphic attributes and non-terminals. Both attributes and non-terminals could be polymorphic in more than one type variable, in that case, the used proxy could store a tuple with all the variables on discourse, or we could use more than one proxy argument. However, an idiom we are going to use is having

```
class MapCtxAsp (r :: [(Prod, Type)]) (ctx :: [ErrorMessage])
                                      (ctx' :: [ErrorMessage]) where
  type ResMapCtx r ctx ctx' :: [(Prod, Type)]
  mapCtxRec :: (Proxy ctx → Proxy ctx')
            → Aspect r → Aspect (ResMapCtx r ctx ctx')
instance MapCtxAsp ('[] :: [(Prod, Type)]) ctx ctx' where
  type ResMapCtx ('[] :: [(Prod, Type)]) ctx ctx' = '[]
  mapCtxRec _ EmptyRec = EmptyRec
instance
  (MapCtxAsp r ctx ctx')
   ⇒
  MapCtxAsp ('(l, CRule ctx' prd sc ip ic sp ic' sp')': r) ctx ctx' where
  type ResMapCtx ('(l, CRule ctx' prd sc ip ic sp ic' sp')': r) ctx ctx'
    = '( l, CRule ctx prd sc ip ic sp ic' sp')': ResMapCtx r ctx ctx'
  mapCtxRec fctx (ConsRec (TagField c l r) rs)
    = (ConsRec (TagField c l (mapCRule fctx r)) (mapCtxRec fctx rs))
```

Figure 5.16: Context mapping over an Aspect.

only one polymorphic argument and making all other types dependent on it using functional dependencies. This simplifies the task though leads to some code duplication. We will see this in detail in Chapter 6.

The use of polymorphism within `AspectAG` does not require to implement substantial new source code. The introduction of polymorphism is a matter of the *pragmatics* of the EDSL and how do we take advantage of the features of the host language. Anyway, some commonly used combinators are provided. For instance, in Chapter 3 we used '*extP*', defined as follows:

$$extP\ l\ r = \lambda(p :: Proxy\ p) \rightarrow l\ p \diamond r\ p$$

The function '*extP*' is the counterpart of $(\diamond)$ (or *ext*, hence its name). Similarly, we can define the corresponding counterparts for $(\triangleleft)$, $(\bowtie)$, or $(\triangleright)$.

What must concern us is whether what we have achieved in terms of error messages, as shown in Chapter 4 is not broken by the presence of polymorphic variables. We will show that everything keeps working, as long as we stick to the pragmatics.

Two things could go wrong in the presence of polymorphic values. The first is that error messages could be displayed with 'noise' because GHC might not know how to print them. The second, arguably more serious, would be that those type families related with the machinery of the `requirements` library get stuck, leading to unsolved constraints that would be displayed instead of producing the readable error messages we programmed.

For instance, consider the following incorrect implementation of a rule for the expression language we introduced in Chapter 3. We mistakenly referred to the production p_{Val} (that was a production of the non-polymorphic version of the language, different from p_{ValP} that would be the correct one to use):

$$\begin{array}{l} \mathit{val\_evalP'} = \lambda(\mathit{Proxy} :: \mathit{Proxy}\ v) \rightarrow \\ \quad \mathit{syndefM}\ (\mathit{evalP}\ @\ v)\ p_{Val}\ (\mathit{ter}\ (\mathit{ch}_{ValP_{val}}\ @\ v)) \end{array}$$

The following error message is printed:

```
    . Error:
   Child val of producion (ValP of Non-Terminal Expr)
/= Child val of producion (Val of Non-Terminal Expr)
      trace: - syndef: definition of attribute (evalP:v)
                in production (Val of Non-Terminal Expr)
```

Note how the polymorphic variable was printed. That behavior of the type families used to show types such as *ShowType* prevents that garbage is displayed. The expression *ShowType v* is printed as "v". So our first concern is not an issue at all.

Now let us analyze our second concern. There are two types of constraints introduced by *syndefM*. On the one hand, those constraints controlling whether the rule is legal (the type of the resulting computation must be equal to the type of the attribute, and children referred must belong to the declared production). On the other hand, those constraints referring to the properties of the AG (requirements about records). The former ones report errors in the proper definition of the rule, while the latter ones generate a laying constraint that will be solved when more context is known, such as in applications of semantic functions.

The constraints of the first type are equality constraints (see Section 5.3.4.1 where the type family *Syndef* was defined). Type equality in Haskell works well with polymorphic values, in the sense that constraints such as $T\ a \sim T\ a$ are satisfied, as expected. In our experience, most of the time polymorphic rules do not leave those constraints unsolved (whether if the rule compiles or if a type error is raised).

There are some corner cases where we can achieve unsolved constraints in those requirements that were solved in the monomorphic definitions. For instance given a type family F we can write the following rule:

$$
\begin{array}{l}
\mathit{add\_evalP} = \lambda(\mathit{Proxy} :: \mathit{Proxy}\ v) \rightarrow \\
\quad \mathit{syndefM}\ (\mathit{evalP}\ @\ v)\ p_{\mathit{Add}} \\
\qquad\quad \$\quad \mathit{add} \\
\qquad\ \ \mathbin{<\!\$\!>}\ \mathit{at}\ \mathit{ch}_{\mathit{Add}_l}\ (\mathit{evalP}\ @\ (F\ v)) \\
\qquad\quad \circledast\ \ \mathit{at}\ \mathit{ch}_{\mathit{Add}_r}\ (\mathit{evalP}\ @\ v)
\end{array}
$$

The introduced constraint, $(\mathit{RequireEqWithMsg}\ v\ (F\ v)\ \mathit{AttTypeMatch}\ \mathit{ctx})$ reduces to $(v\ \sim\ F\ v)$, that is unsolved until we know more about v (at least in the most general case, since $(F\ v)$ could actually reduce to v!).

So, $\mathit{add\_evalP}$ type checks, even when it could fail for any concrete instance of v, while a non-polymorphic version of the rule would not. Either way, in this worst-case the type errors are just being deferred. Once we apply a semantic function, polymorphic variables are instantiated. Following the example of the expression language, recall the following definition:

$$
\begin{array}{l}
\mathit{evaluator}\ \mathit{exp}\ \mathit{envi} = \\
\quad \mathit{sem}_{\mathit{Expr}}\ (\mathit{asp}_{\mathit{sem}}\ (\mathit{Proxy}\ @\ \mathit{Integer}))\ \mathit{exp} \\
\qquad\qquad ((\mathit{envP}\ @\ \mathit{Integer})\ \ =.\ \ \mathit{envi}\ *.\ \mathit{emptyAtt})\ \#.\ (\mathit{eval}\ @\ \mathit{Integer})
\end{array}
$$

At this point v is instantiated (with *Integer*) so stucked constraints will reduce.

For constraints of the second type (those about records), they will also be solved in this semantic function application. Note that all polymorphic values are applied to concrete types, so the types in the constraints of $\mathit{asp}_{\mathit{sem}}$ will be instantiated.

However, we could write an evaluator polymorphic in v. In that case, it will have a type with laying unsolved constraints. We can even apply it to polymorphic ASTs. However, eventually, if we want to use the defined semantics, polymorphic variables must be instantiated and then constraints will be solved, and existing type errors will be raised.

In conclusion, in the worst cases, the type errors we programmed are just deferred. This means they could be thrown in a different location of the source code, but in that case, the traces come into play to help users.

This is true *as long as users stick to the pragmatics we defined*, this is, instantiating polymorphic labels. For instance, the following function:

$$\begin{array}{l} \textit{evaluator exp envi} = \\ \quad \textit{sem}_{\textit{Expr}}\ (\textit{asp}_{\textit{sem}}\ \textit{Proxy})\ \textit{exp} \\ \qquad\qquad (\textit{envP} \ =.\ \textit{envi} \ast.\ \textit{emptyAtt})\ \#.\ \textit{eval} \end{array}$$

generates a 'bad' type error (402 lines of size) reporting unsatisfied constraints. It starts as follows:

```
. Could not deduce: Data.Type.Require.ReqR
 (Data.GenRec.OpLookup'
 'EQ
 (Language.Grammars.AspectAG.RecordInstances.ChiReco
 ('Prd "Val" Nt_Expr))
 ('Chi "Val_val" P_Val ('Right ('T v1)))
 '[ '( 'Chi "Val_val" P_Val ('Right ('T v3)),
 '[ '( 'Att "term" v3, v3)])])
 ~ Language.Grammars.AspectAG.RecordInstances.Attribution r0
   .......
```

The problem is that the compiler has no way to relate some of the many occurrences of polymorphic variables. In this case, the type variable of a looked-up polymorphic terminal cannot be unified to the type variable of the child in the children's record. The application of *OpLookup* is stuck, so is *ReqR*, so we never get near to evaluating the logic implemented in the `requirements` library.

5.3.9 Performance issues and Optimizations.

`AspectAG` behaved fine in the tests we performed during development. However, when implementing a bigger example (see Chapter 6) we had a blow-up in compilation time and space, apparently exponential (though we cannot assert it rigorously from a limited amount of empirical tests). Time and space grew in terms of the number of productions considered in the language.

More precisely, performance issues happen when combining big aspects. Our conjecture is that the constraint solver of GHC is the bottleneck since algorithms manipulating records are trivially polynomial. Recall that polymorphic variables appear everywhere, for instance in the type of families.

Unfortunately, at least to our knowledge, there are no good tools to perform profiling of compilation, so understanding this issue better is left as future work. But the good news is that we can drastically improve the performance by doing a simple optimization.

Combining all rules of a grammar in an aspect, while handy, is completely unnecessary. The function *knit* uses only the combined rule for the corresponding production. In other words, when building aspects we build a record of rules just to extract the individual rules later.

The optimization we propose is to use a non-extensible Haskell record (basically a sum type, a tuple) to store rules instead of an extensible record. This kind of optimization was already discussed within old versions of the library [Viera et al., 2012].

For instance, consider a language like the one shown in Chapter 3. There are three productions represented by labels p_{Add}, p_{Val} and p_{Var}. We had rules add_{eval}, var_{eval}, val_{eval}, add_{env_l} and add_{env_r} (their names suggest for which producions they encode semantics). The following expression denotes the combined aspect:

$$asp_{eval} = add_{eval} \triangleleft var_{eval} \triangleleft val_{eval} \triangleleft add_{env_l} \triangleleft add_{env_r} \triangleleft emptyAspect$$

Instead of defining asp_{eval}, we could define the following data type:

$$\textbf{data}\ Sem\ add\ var\ val = Sem\ add\ var\ val$$

and then the following value:

$$\begin{aligned} asp\_eval' = Sem\ & (add_{eval} \diamond add_{env_l}.{+}, add_{env_r}) \\ & (var_{eval}) \\ & (val_{eval}) \end{aligned}$$

The expression asp_eval' works as an alternative to the aspects we defined as a record instance, once we use the following alternative semantic function:

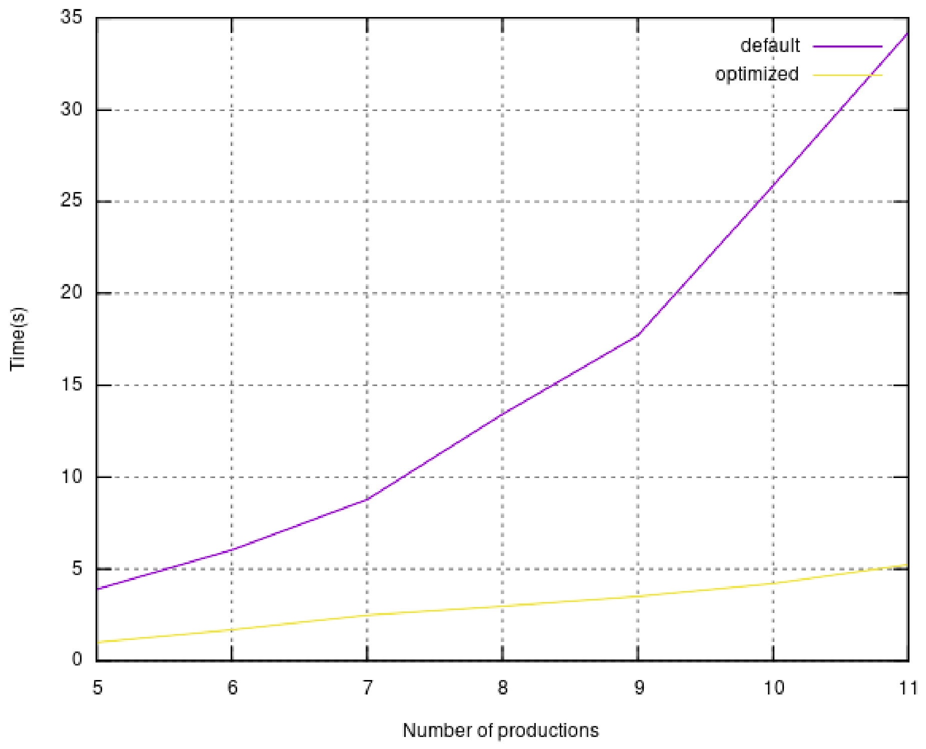


Figure 5.17: Time performance comparison among default and optimized aspects.

$$
\begin{aligned}
&\mathit{sem}_{\mathit{Expr}}\ \mathit{asp}\ @\ (\mathit{Sem}\ \mathit{add}\ \mathit{var}\ \mathit{val})\ (\mathit{Add}\ l\ r)\\
&\quad = \mathit{knit}\ \mathit{Proxy}\ p_{\mathit{Add}}\ \mathit{add}\ \ \$\ \ \mathit{ch}_{\mathit{Add}_l}\ .\!=\!.\ \mathit{sem}_{\mathit{Expr}}\ \mathit{asp}\ l\\
&\qquad\qquad\qquad\qquad\qquad\quad .\!*\!.\ \mathit{ch}_{\mathit{Add}_r}\ .\!=\!.\ \mathit{sem}_{\mathit{Expr}}\ \mathit{asp}\ r\\
&\qquad\qquad\qquad\qquad\qquad\quad .\!*\!.\ \mathit{emptyGenRec}\\
&\mathit{sem}_{\mathit{Expr}}\ \mathit{asp}\ @\ (\mathit{Sem}\ \mathit{add}\ \mathit{var}\ \mathit{val})\ (\mathit{Var}\ \mathit{var})\\
&\quad = \mathit{knit}\ \mathit{Proxy}\ p_{\mathit{Var}}\ \mathit{var}\ \ \$\ \ \mathit{ch}_{\mathit{Var}_{\mathit{var}}}\ .\!=\!.\ \mathit{sem}_{\mathit{Lit}}\ \mathit{var}\\
&\qquad\qquad\qquad\qquad\qquad\quad .\!*\!.\ \mathit{emptyGenRec}\\
&\mathit{sem}_{\mathit{Expr}}\ \mathit{asp}\ @\ (\mathit{Sem}\ \mathit{add}\ \mathit{var}\ \mathit{val})\ (\mathit{Val}\ \mathit{val})\\
&\quad = \mathit{knit}\ \mathit{Proxy}\ p_{\mathit{Val}}\ \mathit{val}\ \ \$\ \ \mathit{ch}_{\mathit{Val}_{\mathit{val}}}\ .\!=\!.\ \mathit{sem}_{\mathit{Lit}}\ \mathit{val}\\
&\qquad\qquad\qquad\qquad\qquad\quad .\!*\!.\ \mathit{emptyGenRec}
\end{aligned}
$$

In figures 5.17 and 5.18 we compare the performance of the implementations. The language used is a variation of a lambda calculus (The Core language defined in Chapter 6) where we added different recursive productions to make bigger and bigger versions of the language. With the extensible records, it was impossible to handle more than 11 productions without running out of

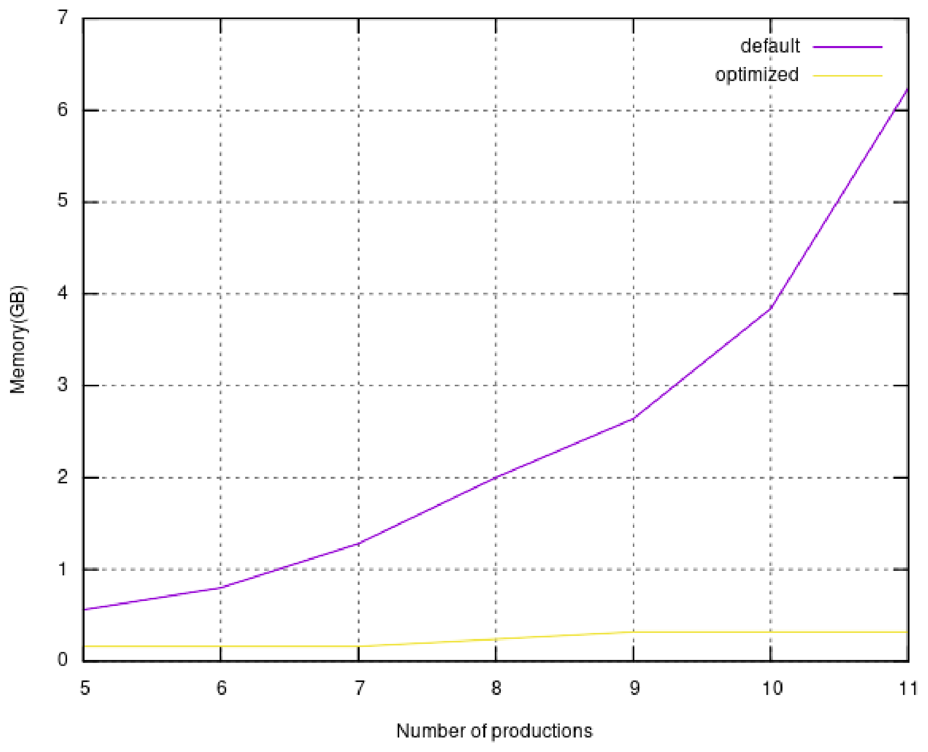


Figure 5.18: Memory performance comparison among default and optimized aspects.

memory [6]. With the optimized implementation, the memory usage is minimal, while times grow acceptably.

The use of a fixed data type to store semantics is honestly a step backwards in terms of modularity. For instance, if we extend the language with a new production we must define a new data type, for instance as follows:

$$\textbf{data}\ \mathit{SemExt}\ \mathit{add}\ \mathit{var}\ \mathit{val}\ \mathit{call} = \mathit{Sem}\ \mathit{add}\ \mathit{var}\ \mathit{val}\ \mathit{call}$$

Still, semantics defined with the data type *Sem* can be extended with simple functions such as the following:

$$\mathit{extSem}\ (\mathit{Sem}\ \mathit{add}\ \mathit{var}\ \mathit{val})\ \mathit{call} = \mathit{SemExt}\ \mathit{add}\ \mathit{var}\ \mathit{val}\ \mathit{call}$$

Moreover, we can use a similar set of combinatios as those we used with aspects:

$$\begin{aligned}&(\mathit{Sem}\ \mathit{add}\ \mathit{var}\ \mathit{val}) \bowtie\!\!\bowtie (\mathit{Sem}\ \mathit{add}'\ \mathit{var}'\ \mathit{val}') = \\ &\quad \mathit{Sem}\ (\mathit{add} \diamond \mathit{add}')\ (\mathit{var} \diamond \mathit{var}')\ (\mathit{val} \diamond \mathit{val}')\end{aligned}$$

We could even reuse the symbol ($\bowtie$) with a suitable abstraction using a typeclass. But the bad thing is that to define *extSem* and ($\bowtie\!\!\bowtie$) we need to provide new code for each new defined language, while before we had the general combinators working in all cases. However, the process could be automatized. If the forested approach was used we were already doing the same thing for semantic functions and reification of data types, so we did not lose that much.

Another issue is that *Sem* and *SemExt*, as defined, do not enforce that their fields are actually rules, abeit this can be easily fixed. We did not care too much because the type of the semantic function will assert that.

5.3.10 Comparison with previous implementations.

There are many AG implementations available. Some of them are implemented as standalone compilers or generators. For instance LRC [Saraiva, 2002], UUAGC [Swierstra et al., 1999], LISA [Mernik and Žumer, 2005], JastAdd [Ekman and Hedin, 2007] or Silver [Van Wyk et al., 2010].

Other systems are embedded in languages like Haskell ([de Moor et al., 1999, de Moor et al., 2000, Viera et al., 2009, Viera et al., 2018, Martins et al., 2013, Balestrieri, 2015]).

[6] Tests were performed in a laptop computer equipped with an 8-core CPU Intel® Core^{TM} i7-1065G7 at 1.30GHz, with 8 GB of RAM memory.

The original implementation of `AspectAG` was introduced more than a decade ago [Viera et al., 2009]. The previous attempts at incorporating AGs as an EDSL in higher-order lazy functional languages always used some form of extensible records. The way that `AspectAG` represents grammars (using records, families, rules, semantic functions based on the *knit* function and so) was already implemented in [de Moor et al., 2000]. Their approach lacked of many static checks. In [de Moor et al., 1999] a safer, similar idea is implemented. However, they used a host language with built-in extensible records, and implemented a set of combinators to encode record positions explicitly, leading to a first-class, less flexible system.

`AspectAG` removed the need for built-in extensible records (as long as some GHC extensions were available), and achieved at the same time an extensible, first-class system while ensuring static properties.

There were two big drawbacks that we tackled in the reimplementation: kind safety and error messages. In the state of the art of GHC extension ecosystem when the original implementation was released, the type system used at kind level had only the kind '$*$' (latter referred to as *Type*), and arrows. This makes the system almost untyped at type level.

In Figure 5.19 we show an implementation of a simple AG (binary trees and rules to compute the addition of all values) in the old system. There are many differences with respect to what we have introduced in the new implementation, but the definition should be familiar to the reader.

The introduced labels are just brand new types. There is no way to discriminate between labels representing non-terminals, productions, children (we only know the type is a pair) or attributes. Moreover, there is no way to separate them from other types such as *Int* or *Char*. This is also true for the implementation of records the library use. Heterogeneous collections like records were essentially nested pairs.

While all that freedom on the kinds can lead to somewhat questionable implementations, for instance, by using labels with controversial criteria[7], some static checks are performed. The occurrences of functions such as (#) introduce constraints over the argument record that will need to be solved when applying the semantic functions to aspects as in the function *sum*. The issue is

[7] For instance, consider a well-defined program implemented using an AG. Then, swap all occurrences of two labels, even if they are different types of labels like an attribute and a production. The program will still work.

```
data Nt_Tree

data P_Leaf; p_Leaf  = proxy :: Proxy P_Leaf
data P_Node; p_Node = proxy :: Proxy P_Node

data Ch_l; ch_l   = proxy :: Proxy (Ch_l, Nt_Tree)
data Ch_r; ch_r   = proxy :: Proxy (Ch_r, Nt_Tree)
data Ch_i; ch_i = proxy :: Proxy (Ch_i, Int)

data Tree = Node Tree Tree | Leaf Int

sem_Tree asp (Node l r)
   = knit (asp # p_Node)  $  ch_l .=. sem_Tree asp l
                          .*. ch_r .=. sem_Tree asp r
                          .*. emptyRecord
sem_Tree asp (Leaf i)
   = knit (asp # p_Leaf)  $  ch_i .=. sem_Lit i
                          .*. emptyRecord

data Att_sres; sres = proxy :: Proxy Att_sres

node_sres (Fam chi par)
   = syndef sres (Node ((chi # ch_l) # sres) ((chi # ch_r) # sres))

leaf_sres (Fam chi par)
   = syndef sres (Leaf (chi # ch_i))

asp_sres = p_Node .=. node_sres .*. p_Leaf .=. leaf_sres .*. emptyRecord

sum t = sem_Tree asp_sres t emptyRecord #. sres
```

Figure 5.19: Grammar implementation in old versions of `AspectAG`.

that when constraints do not solve, we get huge type errors reporting missing instances that can have thousands of lines of length, instead of a domain-specific type error.

In contrast, all data structures of the reimplementation of the library have kind information that defines their intended use and restricts how they can be combined. Thanks to data promotion information is just as expressive as what we can have in usual types. We used labels as an example before but this is true for any type of `AspectAG`. For instance, recall the following definition for the type *Fam*:

```
data Fam (prd :: Prod) (c :: [(Child, [(Att, Type)])]) (p :: [(Att, Type)]) :: Type
  where
    Fam :: ChAttsRec prd c → Attribution p → Fam prd c p
```

The kind of the *Fam* type constructor is the following:

```
Fam :: Prod → [(Child, [(Att, Type)])] → [(Att, Type)] → Type
```

That means types of families must be built satisfying it, ensuring many properties. In comparison, the *Fam* type in the old implementation is just a pair. The type (*Fam Bool Char*) was legal. Of course, if we managed to create a value type (*Fam Bool Char*) and used it in a grammar, a type error would occur somewhere. But now, instead of generating a huge error somewhere else, we just forbid creating those types.

Once we have forbidden the construction of nonsense combinations of types by the kind system, we were able to concentrate a relatively limited variety of possible structural errors of the domain. We caught them using the `requirements` library.

The new information in types allows us to perform new checks related to the structure of the domain. For instance, note that to define a rule in the modern version of `AspectAG` we provided a production label. Its type information was checked against the information of productions of children in the rule definition. While that would be possible to implement in the old version, it would only lead to another confusing type error reporting a missing instance.

Unfortunately, we also found a disadvantage. The new version of the library introduces a performance penalty in runtime. In Figure 5.20 we show a comparison between versions `0.3.1` (a release from 2012) and the current implementation of `AspectAG` for solutions of the repmin [Bird, 1984] problem.

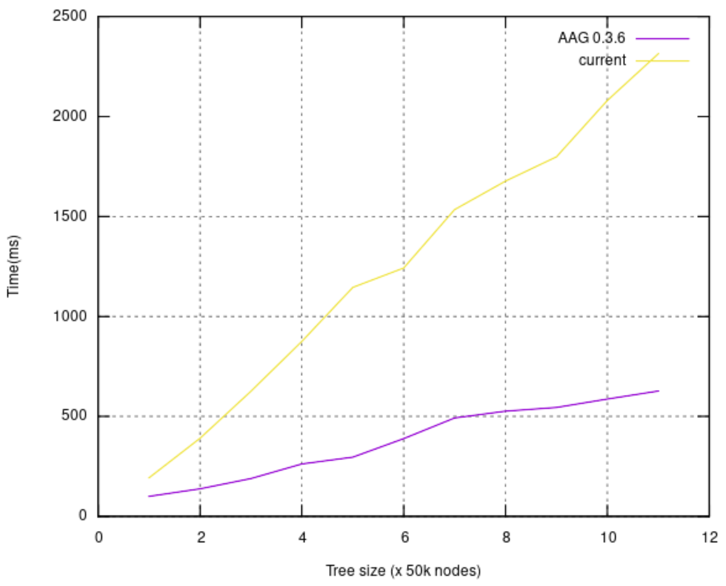


Figure 5.20: Runtime performance comparison.

We detect a big linear overhead. This overhead is not surprising considering all the type constructors the pieces of the library have. Moreover, in the implementation of terminals (see Section 5.3.6.3) we use a child that has a trivial attribution with the attribute named `"Ter"`. In old versions of the library, since everything was unkinded, instead of having an attribution, children were just paired with the terminal value. So, in the reimplementation, to get a terminal we have a linear overhead in both time and space. To understand how much impacts this, note that in a binary tree the number of leaves is linear with respect to the tree size, so this means a linear overhead just to traverse the tree.

The good news is that the issue is not fundamental. For instance, to improve performance we could transform all the constructs of an AG to a simpler (less safe) implementation, and then use it to do the actual computation. If the transformation is 'correct' and users cannot manipulate the generated artifact, we keep the type safety. This is left as future work.

Chapter 6

Case Study: The Matefun Language.

In this chapter, we show how to use the introduced techniques to implement a prototype implementation of a real-world programming language.

The goal of this project is twofold:

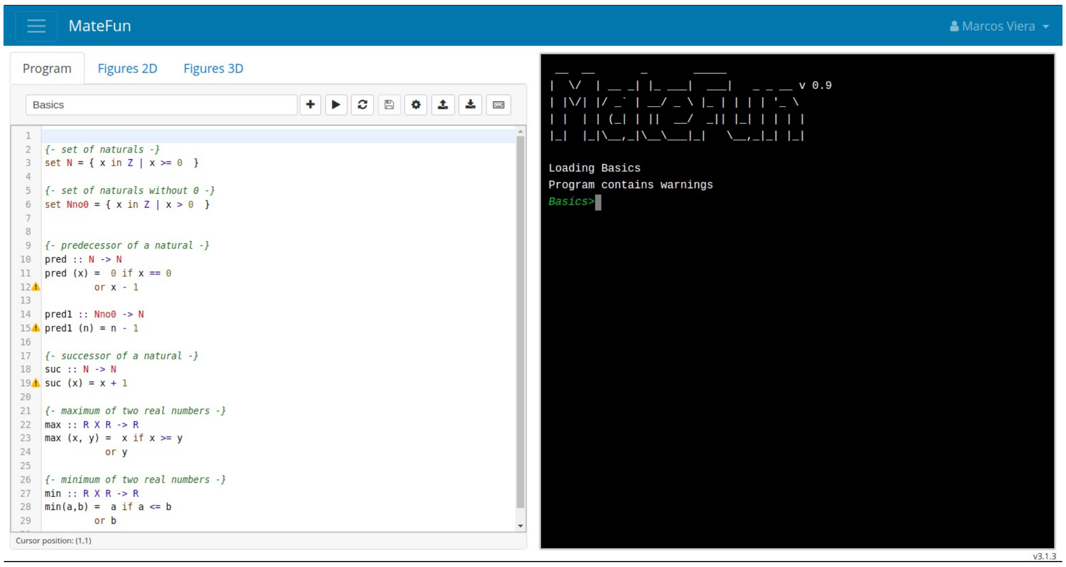
1. To show how the `AspectAG` library (both the DSL combinators and its pragmatics) is applied to solve the problems that arise when building a full compiler implementation.
2. To get a proper modular implementation of MateFun to develop new features later.

The goal of this chapter is not to show the full codebase of the compiler, but to show how the techniques are applied to solve the problems that arise when building a full implementation. So, for instance, we will try not to get bogged down in the full implementation of some interfaces.

As done during this thesis, we will concentrate in the backend of compilation, manipulating ASTs that we suppose already been given. In particular, we do not discuss parsing, or concrete syntax in general, except when it is strictly necessary.

6.1 Introduction to MateFun.

MateFun [Carboni et al., 2018] is a purely functional language. Syntax and semantics of MateFun were thought to be a tool to express mathematics. The



```

{- set of naturals -}
set N = { x in Z | x >= 0  }

{- set of naturals without 0 -}
set Nno0 = { x in Z | x > 0  }


{- predecessor of a natural -}
pred :: N -> N
pred (x) =  0 if x == 0
         or x - 1

pred1 :: Nno0 -> N
pred1 (n) = n - 1

{- successor of a natural -}
suc :: N -> N
suc (x) = x + 1

{- maximum of two real numbers -}
max :: R X R -> R
max (x, y) =  x if x >= y
           or y

{- minimum of two real numbers -}
min :: R X R -> R
min(a,b) =  a if a <= b
         or b
```

```
 __  __       _       _____
|  \/  | __ _| |_ ___|  ___|   _ _ __  v 0.9
| |\/| |/ _` | __/ _ \ |_ | | | | '_ \
| |  | | (_| | ||  __/  _|| |_| | | | |
|_|  |_|\__,_|\__\___|_|   \__,_|_| |_|

Loading Basics
Program contains warnings
Basics>
```

Figure 6.1: MateFun web interface.

current MateFun implementation has been used as a didactic tool in previous projects [da Rosa et al., 2020a, da Rosa et al., 2020b, da Rosa et al., 2021]. MateFun ended up being used in ways that exceed the original expectations of teaching the concept of function to secondary school students, and has been a tool to introduce programming concepts to a broad mixture of audiences. MateFun is constantly evolving considering the feedback given by its users, such as high school or college students; math, physics, or informatics teachers.

MateFun has currently a reference implementation (written in Haskell) and it is mainly used in a web integrated development environment (IDE)[1]. Figure 6.1 shows a screenshot of the IDE. Users can write and manage programs files in the IDE, and then they evaluate expressions in a read-eval-print-loop (REPL) console. Writing expressions and viewing the displayed result is the only way to interact. No side effects can be performed in MateFun, including input/output.

The syntax of MateFun is minimal and close to the usual mathematical notation. Functions in MateFun are explicitly annotated with types. Types in MateFun are called *sets*.

A MateFun script is a list of definitions of *sets* and *functions* over such sets. Predefined sets such as `R` (representing real numbers) or `Z` (representing integer numbers) are available as built-in constructs.

[1] `https://www.fing.edu.uy/proyectos/matefun`

Users can define new sets either by set comprehension or by extension just as usually presented in mathematics courses. In the following example we define the sets of natural numbers (`N`), non-zero real numbers (`Rno0`) and days of the week (`Day`):

```
set N    = { x in Z | x >= 0 }
set Rno0 = { x in R | x /= 0 }
set Day  = { Mon, Tue, Wed, Thu, Fri, Sat, Sun }
```

Sets such as `Rno0`, defined by comprehension, take an existing set (`R` in this case) and refine it with a predicate. Predicates can be built by relational operators and from other predicates by conjunctions.

In type theory, types such as `Rno0`, that are endowed with a predicate over its inhabitants are called *refinement types* [Freeman and Pfenning, 1991].

Sets defined by extension introduce a set of constructors, which are always 0-ary. These are the only ways to define data types, in particular, note that there is no way to define recursive data types in MateFun.

Functions are defined by giving a signature and a definition. For instance, one could define the inverse function over the non-zero real numbers:

```
inv :: Rno0 -> R
inv (x) = 1/x
```

MateFun supports some of the idioms used to define functions in mathematics. For instance, piecewise-defined functions can be given[2], while, unlike most functional languages, general pattern matching or conditional expressions are not supported. The following MateFun definition specifies the absolute value function over the real numbers:

```
abs :: R -> R
abs (x) =   x if x >= 0
        or -x
```

This program resembles the definition in the usual mathematical notation given

[2] `https://en.wikipedia.org/wiki/Piecewise`.

by:

$$abs : \mathbb{R} \rightarrow \mathbb{R}$$
$$abs(x) = \begin{cases} x & \text{if } x \geqslant 0 \\ -x & \text{otherwise} \end{cases}$$

Functions with multiple variables can be defined using *n-tuples* (Cartesian products). Pattern matching is only allowed in tuple arguments. No other form of pattern matching exists in MateFun. A projection operator (!) is provided to destruct tuples (for instance, $t \,!\, 1$ denotes the first component of t). The following function computes the area of a rectangle, given its width and height:

```
rectArea :: R X R -> R
rectArea (h, w)  = h * w
```

Alternatively, it can be written as:

```
rectArea :: R X R -> R
rectArea (r)  = r!1 * r!2
```

MateFun implements a form of subtyping and overloading. Integer numbers can be implicitly used as real numbers, while arithmetic operations can be used (and are closed) in both base sets.

In addition to reals, integers, enumerations and tuples, MateFun provides *sequences*. The sequence set `A*` is defined for any set `A`. The empty sequence is given by the value `[]`, while `a:as` is a sequence whenever `a∈A` and `as∈A*`. Built-in functions such as `head` and `tail` are given to destruct sequences. Subtyping is extended to those sets. Whenever `A` is a subtype of `B`, `A*` is a subtype of `B*`.

The language also includes the primitive sets `Fig` to represent *figures* and `Color` to represent *colors*. Primitive functions to create and transform figures and colors is given. For example, the following function returns a red-coloured circle of a given radius, centred in the $(0, 0)$ point of a Cartesian plane.

```
redCirc :: R -> Fig
redCirc (r) = color(circ(r), Red)
```

In MateFun, *animations* are sequences of figures. The following function takes a figure and a number n of steps, and returns an animation in which the figure is moved n times one unit to the right on the x-axis:

```
moveRight :: Fig X Z -> Fig*
moveRight (f, n)
  =  [] if n == 0
  or f : moveRight(move(f, (1, 0)), n - 1)
```

Note how the definition is recursively defined.

When values of type `Fig`/`Fig*` are evaluated in the console, figures/animations are displayed in a tab of the web IDE.

MateFun has an ambitious property: it has functions in the mathematical sense (total), and a type system based in mathematical sets, with no restriction on how to build functions. Of course, an ideal type system ensuring those properties statically is undecidable.

In order to handle this, the philosophy of MateFun is making an effort to to statically typecheck what is possible, and leaving what it cannot for dynamic typing.

A first approach is to perform statica type checking considering only the base types, where MateFun is simple-typed, and then dynamically check the predicates introduced in the refinement types. In the reference implementation, there have been experiments extending the static typechecking with heuristics using SMT solvers [de Moura and Bjørner, 2008, Biere et al., 2009]. A long-term goal of MateFun developers is to formalize the type system and push the boundaries of what is statically checked.

6.2 Designing an implementation.

From our experience using MateFun in practice, we discovered a desirable property of the implementation is modularity. Inspired by GHC extensions, our goal is to have a set of language features that can be added or removed depending on the target user (for instance, by using flags). In some sense, in this approach, we do not implement one language, but a family of languages.

We define the language incrementally, starting from simpler subsets. Let us refer as the "full-featured" MateFun to the language described in Section

6.1. Without being exhaustive, we consider some reasons why we think this kind of modularity is needed:

- The full-featured MateFun language uses real numbers. Any approach implementing them has its drawbacks. The reference implementation has used fixed-point and floating-point numbers in different iterations of its development. There are, of course, representation issues in both approaches. While it can be considered interesting to introduce representation issues to users, some school teachers using the language in their courses requested a "more precise" computation. While the request sometimes involved things impossible to accomplish, we think it is a good idea to have available, for instance, symbolic computation, so values such as 4π can be manipulated without introducing errors.
- The full-featured MateFun requires types to be annotated in every function declaration. While this was desirable when the language was designed to teach mathematical functions, where the domain and the codomain are an explicit part of the definition, this condition could be lifted, for instance to use MateFun as a programable calculator without too much boilerplate.
- The full-featured MateFun language implements refinement types, with unconstrained predicates. A long-term goal of the implementation project is to try to check as many static properties as possible. We can take more conservative approaches or more eager ones (for instance using SMT solvers). We consider it is sane that users can control if they want to mess up with more eager and experimental approaches that might imply non-termination or slow type checking, or if they prefer a more conservative approach using dynamic checking of predicates.
- The implementation using the web interface uses figures and function graphs. This feature could be disabled in a CLI-based implementation.
- Also related to the web interface, non-terminating programs are an issue when using a client-server architecture. The reference implementation bounds the number of reductions that can be performed by an evaluation. The ability to configure this feature is desirable.

The flexibility in the possible variants of syntax and semantics of real numbers and the possibility to add or remove terminals such as figures tune in well with the parametrization over terminals we have shown it is possible to implement with `AspectAG`. Terminals will satisfy interfaces that each concrete implementation handles in a particular way. The implementation of that interface is at the same time implementing operational semantics of MateFun. In algebraic terms, terminals are specified by an algebraic structure. Each concrete implementation of terminals is an algebra.

Let us set these ideas with a concrete example. Consider a MateFun version with real numbers and integers. A concrete implementation could use floating-point numbers and 32-bit integers, while other implementation could use some implementation of symbolic reals and arbitrary precision integers.

Syntactically equal MateFun programs have different semantics in each case. To model those we can think that the language is parametrized over an algebraic structure $\mathcal{M}$ with a shape like the following:

$$\begin{aligned}\mathcal{M} = (&R, Z,\\ &rz : R \to Z, zr : Z \to R,\\ &+_Z : Z \to Z \to Z,\\ &+_R : R \to R \to R, \cdots)\end{aligned}$$

There we define the sets that represent reals and integers, and some constants and operations. We can interpret a MateFun expression such as (1+1.5), over an instance of $\mathcal{M}$, interpreting constants as members of R and Z and syntax symbols with its corresponding functions in the algebra. So with a suitable definition we can achieve the interpretation of that expression to be something like "$+_R(zr([\![1]\!]), [\![1.5]\!])$".

Richer sets of terminals could require richer structures. For instance, we add lists to the surface language, we probably want a richer structure $\mathcal{M}'$ with lists.

6.3 Architecture of the compiler.

As we stated before the goal of this implementation is twofold. We show how `AspectAG` solves the expression problem in a real-world programming language while we get the proper modular implementation of the MateFun language to develop new features in the future.

As a way to organize this developing and having a clear benchmark of how modularity is gotten, we inspired us in the LDTA 2011 tool challenge[3]. The challenge proposed a set of incremental sub-problems organized in a two-dimensional space. One dimension defined a series of language levels, each one adding new features. The other dimension consisted of language processing tasks, such as parsing, pretty-printing, static analysis, optimizations and code generation. Though we perfectly could, we do not translate exactly the same levels and tasks to this case study, but adapt the idea to a set of tasks and language levels we are interested in implementing as developers in the MateFun project.

The original challenge used Oberon0, a subset of Oberon [Wirth, 1988] as the language to implement. The original implementation of `AspectAG` was used to solve the challenge [Viera and Swierstra, 2015].

In this chapter, we show that the same ideas can be applied to a functional language and at the same time we show the power of the parametrization over terminals to solve some subproblems without even touching the grammar definition.

We define a set of language levels from L1 to L6, and a set of processing tasks from T1 to T4. In Figure 6.2 they are summarized. Let us explain the sub-problems further.

For language levels, we have:

(L1) An expression language. This is the language users can input in the REPL, and also used as a construct to write functions. We take modularity seriously here, and reuse some of the constructs we used in the running example of Chapter 3.

(L2) Untyped MateFun programs. They are lists of function definitions that can be compiled individualy into environments to evaluate programs of L1. L1 and L2 implement, in principle real numbers and naturals as the data types to manipulate, but they will be abstract over terminals, so we could use richer and poorer sets of terminal values.

(L3) Adds lists and enumerations. We do not need to touch `AspectAG` definitions in this step, but just give an extended implementation of terminals. We consider this an achievement of our system: it is so modular that we

[3] `http://ldta.info/tool.html`

L1	Expression language
L2	Programs
L3	Lists and enumerations
L4	Tuples
L5	Simple types
L6	Type definitions and refinements

T1	Pretty-printing
T2	Name binding
T3	Code generation
T4	Type checking

Figure 6.2: Organization of the implementation.

can extend languages without defining new syntax[4].

(L4) Adds tuples (just pairs, to keep it simple). Apart from giving an extended implementation of terminals as in L3, here we extend the syntax of expressions.

(L5) Adds simple types. Here we add the possibility -and obligation- to declare the domain and codomain of functions.

(L6) Adds type definitions and refinements types. Here we provide the posibility of declaring precise non-base types.

The processing tasks are the following:

(T1) Pretty-printing.

(T2) Name binding/analysis. Basically we need to control that no variable occurs free.

(T3) Code generation to "Core", an intermediate language we will define in Section 6.5.

(T4) Static type checking[5].

The combination of a language level and a processing task is called an "artifact". Any combination makes sense, though they vary in how useful they are. For instance, the combination L1-T4 means we implement type checking to an expression language that had, in principle no types. Type checking for the expression language will make more sense once expressions are a subset of richer languages L5 and L6. Still, once types are defined (semantically, independently of how they are declared in the syntax) some aspects related to typing can be computed for expressions independently from any further language extension, like the inference of a least-general simple type

[4] We refer to abstract syntax, of course. Concrete syntax is extended. [5] Dynamic checking of refinements will be considered in T3.

(like deciding if an expressions denotes an integer, or a real). So, just to help the reader and having a simple organization in this document we will walk from L1-T1 to L6-T4 lexicographically in the following chapters.

The implementation is available online[6]. Figure 6.3 shows the set of modules in charge of implementing each combination. Essentially, what we have is a set of modules with qualified names of shape `MF`.L_i.T_j (with more descriptive names). Modules at the same height in the chart implement the same task, while modules in the same column implement the same language. An extra "task" we consider apart from L1-L4 is the definition of syntax (modules named `MF`.L_i.`Syn`). There are more modules not shown in the Figure 6.3. The modules `MF.Terminals.Values` and `MF.Terminals.Types` define the interfaces (as type classes) for terminal values. The modules `MF.Core.Syn` and `MF.Core.Eval` define the syntax of the target language and an evaluator for it, respectively. `MF.Expr.Syn` imports syntax defined in `MF.Expr` and `MF.ExprExt.Syn`, modules used in the running example of Chapter 3. There is yet another family of modules named `MF`.L_i.`Close`, where we define functions actually computing over ASTs (i.e. semantic function applications).

In general, modules depend on the syntax definition in their column, but are otherwise independent among them. In the actual implementation we introduced some dependencies to reduce the number of modules. For instance, as we will see later with more detail, the pretty-printed program is computed with a synthesized attribute *spp*. We define *spp* in `MF.Expr.Syn` (in the smaller language, that is included in all other language levels), so then we include this module in every module with the `PP` suffix. But note that attributes are just names, so those dependencies could be lifted easily[7].

The modules with `Close` suffix put all pieces together. We implement all tasks in each one, so each one depends (directly or indirectly) in its corresponding column and all ones at its left.

[6] `https://gitlab.fing.edu.uy/jpgarcia/tesis/AAGExample/MF`

[7] Remember that this is also true for syntax. We could implement modules so they do not depend in syntax definitions (this would be cumbersome, and error-prone, though). Moreover, we could implement every module independently from each other and just importing all definitions when putting everything together in the application of semantic functions.

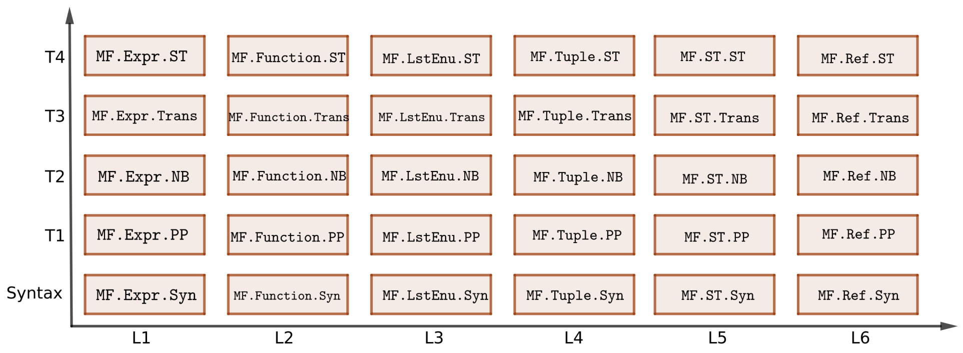


Figure 6.3: Modules implementing language level/task combinations.

6.4 Terminals.

Terminal values are very important in `AspectAG` because by parametrizing over them we can concentrate semantics in the implementation of algebraic structures, abstracting us from language constructs. Note that in this context, values model computation, not just represent concrete syntax. They could also have more information, for example, information about the source position given by the parser that could be used to produce better error messages.

There are two types of terminal values in the simpler languages that we will implement: operators and values. There will be also terminals for types in the latter iterations. We show in this section how the interfaces for terminals data types are defined.

We can implement all terminals in a single data type, and the compiler (for instance, when parsing) could control if we are using operators or values in each context. We prefer stronger types and implement one data type for each type of terminal. In principle, this implies that abstract syntax trees will be parametrized by two types, or by a pair of types. While it is possible to implement this in `AspectAG` we decided to keep it simple and use only one polymorphic parameter. To perform this, we relate each implementation of operators and types by a functional dependency. The following is the interface we use for implementations of MateFun's terminals:

```
class Values op v | op → v, v → op where
  type Val2Op v :: *
  ap  :: op → v → v → v
  cmp :: Ordering → v → v → Bool
  mki :: Integer → v
```

While data types are trees, this double dependency encapsulates the idea of a forest. The obvious disadvantage of this approach is that we cannot reuse implementations of one data type for operators with a different counterpart for values, or the converse. For instance, to extend our language with a new operator, without adding any new value we would need to duplicate the data type used for values. However, this is easily accomplished using a **newtype** definition, and a common pattern even with one-parameter classes when we want to implement more than one class instance (the typical example being the two natural ways in which integers are a monoid).[8].

The interface *Values* is simple. The method *ap* means that there is a way to combine two values with an operator. Operators in our ASTs will be always binary. A notable exception in the concrete syntax of MateFun is the minus symbol used to write negative numbers. To avoid an extra production we will not represent this unary operator in the ASTs of expressions. This can be handled by the parser, for instance, desugaring expressions such as $(-x)$ to $(0 - x)$. The method *cmp* implements comparison of values. The method *mki* provides a way to create an integer value. This interface is an arbitrary minimum we decided to require for a simple language of our family (a version of the MateFun language with just integers). Note that this is more general than the structure $\mathcal{M}$ we drafted in the previous section, since it does not discriminate reals and integers. If programmers want to specify more details of an algebra for terminals, we can always define a more specific class as a subclass of *Values*.

Implementations of methods, notably *op* and *cmp* are not necessarily total. For instance, in the full-featured MateFun language, we would have figure values and the product operator, and those cannot be combined. The type checker has the job of avoiding those operations in runtime. However, we also implement untyped languages in this chapter; in that case, the semantics of those operations are confined in the implementation of the instance of *Values*.

[8] GHC language extensions such as `GeneralizedNewtypeDeriving` are useful in such cases, to help programmers to reduce the boilerplate while defining new types

We think this is a good design decision.

Finally, let us refer to the indexed type *Val2Op*. It works as a way to get the corresponding operator type given a value type. So, the type of values is the one we will pass in proxies, and we can get the data type for operators whenever we need it.

One possible implementation for values and operators, for a language with integers and reals (as we implement in L1) is the following:

```
data Op = Add | Times | Div | Minus
data Val = I Integer | R Double
```

The class instance has the following definition:

```
instance Values Op Val where
  type Val2Op Val = Op
  ap Add (I a) (I b)  = I $ a + b
  ap Add (R a) (R b) = R $ a + b
  ap Add (R a) (I b)  = R $ a + fromInteger b
  ap Add (I a) (R b)  = R $ fromInteger a + b
  ...
```

The implementation of the methods is simple. We ommit most here since it does not offer any new insight. Arithmetic operators are overloaded so we can combine reals and integers, wich leads to many patters in the implementation. We decided to make the semantics of operators closed when we can. For instance, the result of adding integers is an integer. This is the way the reference implementation works.

Typed versions of MateFun are introduced as language levels L5 and L6. Base types[9] are not terminal symbols of the languages implemented in L1-L4. Still, it makes sense to talk about types in any language level. In the MateFun's type system, given the types of free variables and functions occuring in an expression we can infer if the expression is well-typed and in case it is, its type. In L5 we just introduce a way to write the types of functions, but given suitable environments with the type of the functions occurring withing an expression we can implement a type checker already for L1.

The following type class implement the interface used for types:

[9] Recall we used the term *set* in the MateFun jargon to refer to types. Here, in the implementation we just use the term *type*.

```
class (Values op v) ⇒ Types op v t | v → op t, t → v op, op → v t where
  type Val2Type v :: *
  typeOf    :: v → Either String t
  tyComb    :: t → op → t → Either String t
  join      :: t → t → Either String t
  isSubtype :: t → t → Bool
```

Given a pair of types *op* and *v*, the type *t* completes a triple. The return types *Either String t* are used to model failures. If everything goes well a type *t* is returned under the *Right* constructor, otherwise an error message under the *Left* constructor. Of course, errors could be represented using more information than just a string, but as said at the start of this chapter, the goal of this implementation is to show techniques rather than losing us under details.

The function *typeOf* computes the least-general base type of a value. For instance, given a natural number value, *typeOf* returns that it is an integer, instead of a real number. The function *tyComb* computes the resulting least-general type of combining two types by using an operator. The function *join* computes the union of two types (considering their semantics as sets, for instance the union of the set of naturals and the set of reals numbers will be the set of real numbers). The name "*join*" is inspired by the fact that the structure of MateFun sets is a partially ordered set (using the inclusion relation as the corresponding order). The function *isSubtype* is a predicate deciding whether if one type is a subtype of another (in the semantics, whether if a MateFun set is a subset of another). Again we have an indexed family, *Val2Type* that is used to get a type for types from a type for values, so we can use the latter always as the proxy we pass in polymorphic grammars.

An initial implementation of a data type for types, representing just integers and reals is the following:

```
data Ty =
  TyR | TyZ
```

Then, we can give an implementation of the type class *Types*[10]:

[10] Note that in this implementation we make all operators closed. This means that, to make sense, the operator *Div* should denote an integer division when two integer arguments are applied.

```
instance Types Op Val Ty where
  type Val2Type Val  = Ty
  typeOf  (I _)        = Right TyZ
  typeOf  (R _)        = Right TyR
  tyComb TyZ _ TyZ = Right TyZ
  tyComb _     _ _     = Right TyR
  join      TyZ TyZ   = Right TyZ
  join      _     _        = Right TyR
  isSubtype TyZ _ = True
  isSubtype _     _ = False
```

6.5 A Core Language.

The backend of compilation (such as generating machine code) is out of the scope of this thesis. Before dealing with the modular implementation of the surface language, we design an intermediate language to use as the target of our compiler. Operational semantics for evaluation is implemented in the intermediate language using Haskell as host. We call this language "Core MateFun" or just "Core". The "Core" name is inspired by the name of GHC's intermediate language.

The main idea behind this design decision is making available a fixed and closed language to use as the interface between the family of MateFun versions and low-level manipulation of compiled programs. We can, in the future, design different implementations of the core evaluation (for instance, pursuing efficiency). We can perform optimizations, or store precompiled programs, for instance. All this could be performed by using `AspectAG`, or with other approaches.

The Core language must to be simple, otherwise we just would manipulate the source language. At the same time, the Core language must be powerful enough to encode any feature we can develop in the surface language levels. We decided that the Core language is parametrized over the terminals, since they encode the semantics of values, and untyped, since the most general versions of MateFun are such.

The implemented Core language is an enriched version of a lambda calculus [Barendregt, 1985]. The lambda calculus is Turing complete, so any feature

```
$ (addNont "Core")

$ (addProd "Val" ''Nt_Core [("val", Ter ''Poly)])

$ (addProd "Var" ''Nt_Core [("var", Ter ''String)])

$ (addProd "Lam" ''Nt_Core [("binder", Ter     ''String),
                            ("body",   NonTer ''Nt_Core)])

$ (addProd "App" ''Nt_Core [("l", NonTer ''Nt_Core),
                            ("r", NonTer ''Nt_Core)])

$ (addProd "Op" ''Nt_Core [("l",  NonTer ''Nt_Core),
                           ("op", Ter    ''Poly),
                           ("r",  NonTer ''Nt_Core)])

$ (addProd "Comp" ''Nt_Core [("l",  NonTer ''Nt_Core),
                             ("op", Ter    ''Ordering),
                             ("r",  NonTer ''Nt_Core)])

$ (addProd "CError" ''Nt_Core [("err", Ter ''String)]
```

Figure 6.4: Core definition in `AspectAG` using splices.

can be encoded, while it is simple.

In Figures 6.4 and 6.5 we present two alternatives to implement in `AspectAG` the grammar of the Core language. Both definitions are equivalent, in fact the version written with splices in Figure 6.4, generates exactly the code in Figure 6.5[11]. From now on, in this chapter we will define grammars by using splices.

The defined language can be synthesized in the following data type:

[11] Modulo α-conversion of the variable names used in the polymorphic attributes.

type Nt_{Core} $=$ *'NT* `"Core"`
nt_Core $=$ *Label* @ Nt_{Core}

type P_{COp} $=$ *'Prd* `"COp"` Nt_{Core}
p_{COp} $=$ *Label* @ P_{COp}
ch_{COp_r} $=$ *Label* @ (*'Chi* `"COp_l"` P_{COp} (*NonTerminal* Nt_{Core}))
ch_{COp_l} $=$ *Label* @ (*'Chi* `"COp_r"` P_{COp} (*NonTerminal* Nt_{Core}))
$ch_{COp_{op}}$ $=$ *Label* :: $\forall$ *op*.*Label* (*'Chi* `"COp_op"` P_{COp} (*Terminal op*))

type P_{CVal} $=$ *'Prd* `"CVal"` Nt_{Core}
p_{CVal} $=$ *Label* @ P_{CVal}
$ch_{CVal_{val}}$ $=$ *Label* :: $\forall$ *v*.*Label* (*'Chi* `"CVal_val"` P_{CVal} (*Terminal v*))

type P_{CVar} $=$ *'Prd* `"CVar"` Nt_{Core}
p_{CVar} $=$ *Label* @ P_{CVar}
$ch_{CVar_{var}}$ $=$ *Label* @ (*'Chi* `"CVar_var"` P_{CVar} (*Terminal String*))

type P_{CComp} $=$ *'Prd* `"CComp"` Nt_{Core}
p_{CComp} $=$ *Label* @ P_{CComp}
ch_{CComp_r} $=$ *Label* @ (*'Chi* `"CComp_l"` P_{CComp} (*NonTerminal* Nt_{Core}))
ch_{CComp_l} $=$ *Label* @ (*'Chi* `"CComp_r"` P_{CComp} (*NonTerminal* Nt_{Core}))
$ch_{CComp_{op}}$ $=$ *Label* @ (*'Chi* `"CComp_op"` P_{CComp} (*Terminal Ordering*))

type P_{CLam} $=$ *'Prd* `"CLam"` Nt_{Core}
p_{CLam} $=$ *Label* @ P_{CLam}
$ch_{CLam_{binder}}$ $=$ *Label* @ (*'Chi* `"CLam_binder"` P_{CLam} (*Terminal String*))
$ch_{CLam_{body}}$ $=$ *Label* @ (*'Chi* `"CLam_body"` P_{CLam} (*NonTerminal* Nt_{Core}))

type P_{CApp} $=$ *'Prd* `"CApp"` Nt_{Core}
p_{CApp} $=$ *Label* @ P_{CApp}
ch_{CApp_l} $=$ *Label* @ (*'Chi* `"CApp_l"` P_{CApp} (*NonTerminal* Nt_{Core}))
ch_{CApp_r} $=$ *Label* @ (*'Chi* `"CApp_r"` P_{CApp} (*NonTerminal* Nt_{Core}))

type P_{CError} $=$ *'Prd* `"CError"` Nt_{Core}
p_{CError} $=$ *Label* @ P_{CError}
$ch_{CError_{err}}$ $=$ *Label* @ (*'Chi* `"CError_err"` P_{CError} (*Terminal String*))

Figure 6.5: Core definition in `AspectAG`.

```
data CTerm op v = CVal    v
                | CVar    String
                | CLam    String (CTerm op v)
                | CApp    (CTerm op v) (CTerm op v)
                | COp     (CTerm op v) op (CTerm op v)
                | CComp   (CTerm op v) Ordering (CTerm op v)
                | CError  String
```

While the Core language is parameterized over two types of terminals, operators, and values, we can use one parameter since *op* and *v* determine each other. We will use one parameter in MateFun syntax, so both styles will be shown in this document.

We introduced the constructors *CVal* for values, *CVar* for variables, *CLam* for abstraction and *CApp* for application, following up a lambda calculus. *COp* is natural as the way we have to combine terminals. *CComp* is introduced since piecewise definitions of the surface language can have comparations in conditions. In general, MateFun values are comparable at least by equality, but we decided to not implement this in the *Values* interface. Finally, the constructor *CError* represents runtime errors.

Note that there is no way to define function calls. We will use applications and variables for that purpose, combined with an environment mapping variable names to terms with function definitions in scope.

Expressions introduced in the REPL will be compiled to Core, and evaluated using a context with the program already compiled to Core. Core expressions reduce into core expressions. We should, in principle always get a value to print (under the *CVal* constructor), or an error, though this will depend in the correctness of the compilation of the surface MateFun language.

Note that we can define the Core language formally using an EBNF notation:

$$
\begin{aligned}
C &\to v\\
C &\to x\\
C &\to \lambda x.C\\
C &\to C\ C\\
C &\to C \,\square\, C\\
C &\to C \,\triangle\, C\\
C &\to e
\end{aligned}
$$

where $\mathbb{V}$ and $\mathbb{O}$ represent terminals (values and operators), $\mathfrak{S}$ strings and $\mathfrak{S}'$ error messages[12], and $\triangle \in \{<, =, >\}$, $\square \in \mathbb{O}$, $x \in \mathfrak{S}$, $e \in \mathfrak{S}'$, $v \in \mathbb{V}$. We use a usual notational device based on the names introduced in this formal definition: if t and u are any Core terms, we assume the following are Core terms: v, x, $\lambda x.t$, tu, $t\square u$, $t\triangle u$, and e; without specifying the domains of x, $\square$, $triangle$, e.

The induction principle over the Core language ensures that we can define functions over the full language defining them for those cases. For instance, the following definition is the identity function over Core terms:

$$
\begin{aligned}
id(v) &:= v\\
id(x) &:= x\\
id(\lambda x.t) &:= \lambda x.id(t)\\
id(tu) &:= id(t)id(u)\\
id(t\square u) &:= id(t)\square id(u)\\
id(t\triangle u) &:= id(t)\triangle id(u)\\
id(e) &:= e
\end{aligned}
$$

Coming again into the implementation, we also define *sem_CTerm*, the semantic function for the Core in the standard way (or we can get it with a one-liner Template Haskell splice). Now that the grammar is encoded in

[12] Also strings, but distinguishable from variable names.

`AspectAG` we can start defining semantics for the Core language, for instance the *self*[13] attribute and an aspect that computes it:

$$
\begin{array}{llll}
\multicolumn{4}{l}{self :: \forall\ v\ op. Values\ op\ v \Rightarrow Label\ (\text{‘}Att\ \texttt{"self"}\ (CTerm\ op\ v))} \\
\multicolumn{4}{l}{self = Label} \\
\multicolumn{4}{l}{asp\_Core\_id = \lambda(p :: Proxy\ v) \rightarrow} \\
\quad syndefM\ (self\ @\ v)\ p_{COp} & (COp & <\$> & at\ ch_{COp_r}\ (self\ @\ v) \\
 & & \circledast & ter\ ch_{COp_{op}} \\
 & & \circledast & at\ ch_{COp_l}\ (self\ @\ v)) \\
\quad \lhd\ syndefM\ (self\ @\ v)\ p_{CComp} & (CComp & <\$> & at\ ch_{CComp_r}\ (self\ @\ v) \\
 & & \circledast & ter\ ch_{CComp_{op}} \\
 & & \circledast & at\ ch_{CComp_l}\ (self\ @\ v)) \\
\quad \lhd\ syndefM\ (self\ @\ v)\ p_{CVal} & (CVal & <\$> & ter\ ch_{CVal_{val}}) \\
\quad \lhd\ syndefM\ (self\ @\ v)\ p_{CVar} & (CVar & <\$> & ter\ ch_{CVar_{var}}) \\
\quad \lhd\ syndefM\ (self\ @\ v)\ p_{CLam} & (CLam & <\$> & ter\ ch_{CLam_{binder}} \\
 & & \circledast & at\ ch_{CLam_{body}}\ (self\ @\ v)) \\
\quad \lhd\ syndefM\ (self\ @\ v)\ p_{CApp} & (CApp & <\$> & at\ ch_{CApp_l}\ (self\ @\ v) \\
 & & \circledast & at\ ch_{CApp_r}\ (self\ @\ v)) \\
\quad \lhd\ syndefM\ (self\ @\ v)\ p_{CError} & (CError & <\$> & ter\ ch_{CError_{err}}) \\
\quad \lhd\ emptyAspect & & &
\end{array}
$$

The following is a fancy way to implement the identity function for the Core language:

$$
\begin{array}{l}
cterm\_id :: Values\ op\ v \Rightarrow CTerm\ op\ v \rightarrow CTerm\ op\ v \\
cterm\_id\ (e :: CTerm\ op\ v) \\
\quad = sem\_CTerm\ (asp\_Core\_id\ (Proxy\ @\ v))\ e\ emptyAtt\ \#.\ (self\ @\ v)
\end{array}
$$

Note that this implements exactly what we wrote in the definition of *id*. This is actually a very inefficient way to implement the identity since we are consuming a tree to generate a copy of it, but the semantics defined in *asp_Core_id* are much more useful than that, as we will see later.

Another useful aspect we can encode is the one to pass an environment top-down, the copy rule. We can use the advantages of `AspectAG` being an embedded DSL to parametrize this pattern over any attribute, as follows:

[13] In the AGs jargon the *self* attribute has the value of the subterm. In other words, it computes the identity.

$$
\begin{aligned}
&asp\_Core\_env\ att = \\
&\quad\ \ inhdefM\ att\ p_{COp}\ \ \ ch_{COp_r}\ (at\ lhs\ att) \\
&\quad \triangleleft\ inhdefM\ att\ p_{COp}\ \ \ ch_{COp_l}\ (at\ lhs\ att) \\
&\quad \triangleleft\ inhdefM\ att\ p_{CComp}\ ch_{CComp_r}\ (at\ lhs\ att) \\
&\quad \triangleleft\ inhdefM\ att\ p_{CComp}\ ch_{CComp_l}\ (at\ lhs\ att) \\
&\quad \triangleleft\ inhdefM\ att\ p_{CLam}\ ch_{CLam_{body}}\ (at\ lhs\ att) \\
&\quad \triangleleft\ inhdefM\ att\ p_{CApp}\ ch_{CApp_l}\ (at\ lhs\ att) \\
&\quad \triangleleft\ inhdefM\ att\ p_{CApp}\ ch_{CApp_r}\ (at\ lhs\ att) \\
&\quad \triangleleft\ emptyAspect
\end{aligned}
$$

6.5.1 Core Evaluation.

While the main goal of the Core language is to detach us from the low-level evaluation we will implement an evaluator for completeness.

There are many ways to efficiently compile a lambda calculus, like using De Bruijn indices [de Bruijn, 1972] or higher-order abstract syntax [Pfenning and Elliott, 1988]. We will not analyze any of them since it is out of our scope. We implement an evaluator based on explicit substitutions. Usually, this is, besides inefficient, considered complicated since capture-avoiding substitution is non-trivial to implement. However, the terms of the Core language generated from MateFun code should never have free variables (an error in the surface language should be thrown before, for instance, in T2). This fact simplifies the job, avoiding tedious tasks such as generating fresh variables for renaming.

6.5.1.1 Substitution.

We denote as $t[x/u]$ to the substitution of the -free- occurences of x in t by the term u. It can be defined as:

$$
\begin{aligned}
v[x/u] &:= v\\
y[x/u] &:= y \text{ (when } y \neq x)\\
x[x/u] &:= u\\
(\lambda x.t)[x/u] &:= \lambda x.t\\
(\lambda x.t)[y/u] &:= \lambda x.t[y/u] \text{(when } y \neq x)\\
(tu)[x/u] &:= (t[x/u])(u[x := u])\\
(t\Box u)[x/u] &:= (t[x/u])\Box(u[x/u])\\
(t\triangle u)[x/u] &:= (t[x/u])\triangle(u[x/u])\\
e[x/u] &:= e
\end{aligned}
$$

To implement this function using `AspectAG` the natural approach is to pass the context (the variable being substituted and the substitutee term) using inherited attributes, and synthesizing the resulting expression.

We define the *ivar* inherited attribute to make the variable available:

```
ivar = Label @ ('Att "ivar" String)
asp_ivar = asp_Core_env ivar
```

To distribute the term we substitute by this variable, note that once the substituted variable appears under the binder, no further transformation is applied in the subexpression. So, we use an inherited attribute with a value of type (*Maybe* (*CTerm op v*)). A *Nothing* means that we do not make further substitutions[14].

We define the attribute *isubst*:

```
isubst :: ∀ v op. Values op v
   ⇒ Label ('Att "isubst" (Maybe (CTerm op v)))
isubst = Label
```

Note that its type is, of course polymorphic. For that reason, we will use a *Proxy* argument when defining rules to have the type available. The semantics for *isubst* are trivial in most productions: we just use the copy rule and modify

[14] An alternative is to use a boolean attribute to mark if we must make further substitutions. The "trick" saves us the need to use an extra attribute.

the semantics at p_{CLam}. In this production, if there is an abstraction binding the variable we are substituting, we pass down a *Nothing* value. Otherwise we keep what we had (perhaps already a *Nothing* when there is an outer binding of the variable):

```
asp_isubst = λ(Proxy :: Proxy v) →
  inhmodM (isubst @ v) $p_{CLam}$ $ch_{CLam_{body}}$ (
  do t  ← at lhs (isubst @ v)
     x  ← ter $ch_{CLam_{binder}}$
     x' ← at lhs ivar
     if x == x' then return Nothing else return t
    )
    ◁ asp_Core_env (isubst @ v)
```

Finally, we need to compute the substituted term. As we can see in the definition of $t[x/u]$ we gave before, it is similar to the identity function in most cases, except in the case of a variable and in the case of an abstraction. The abstraction case was actually handled with the inherited attribute *isubst*, we just need to handle variables.

We code te attribute *asp_subst* modifying the aspect (*asp_Core_id*) that we wrote for the attribute *self*, as follows:

```
asp_subst = λ(Proxy :: Proxy v) → (
  synmodM (self @ v) $p_{CVar}$ (
    do x  ← at lhs ivar
       x' ← ter $ch_{CVar_{var}}$
       t  ← at lhs (isubst @ v)
       case t of
         Nothing → return (CVar x')
         Just t'  → if x == x' then return t' else return (CVar x')
     )
    ◁ asp_Core_id (Proxy @ v))
```

And we are done.

The following function computes substitutions:

```
$subst :: Values\ op\ v \Rightarrow$
    $Proxy\ v \rightarrow String \rightarrow CTerm\ op\ v \rightarrow CTerm\ op\ v \rightarrow CTerm\ op\ v$
$subst\ (p :: Proxy\ v)\ x\ t\ e = sem\_CTerm\ (asp\_ivar \bowtie asp\_subst\ p$
    $\bowtie asp\_isubst\ p)$
    $e\ (ivar\ .=.\ x\ .*.\ isubst\ .=.\ Just\ t\ .*.\ emptyAtt)\ \#.\ (self\ @\ v)$
```

In this case we decided to require a *Proxy* argument to instantiate v easily (and use as the argument of aspects *asp_subst* and *asp_isubst*), but we can easily extract the proxy from a term of type *CTerm op v*.

6.5.1.2 β-reduction.

β-reduction is a relation on terms. We define it informally here for the Core language, but it is extensively studied and formalized [Barendregt, 1985]. When we apply an abstraction (i.e we have a term of shape $(\lambda x.t)u$) it can be reduced to $t[x/u]$. Moreover, any term can have subterms of this shape (what is called a *redex* -reductible expression-), and we can swap a redex by its reduced form in any subterm. This idea can be formalized by *evaluation contexts*, but we will not do it here. A term with no redexes is in *normal form*.

The reduction relation for the Core language can be given as a calculus like in Figure 6.6. We do not consider the error cases here, but we will in the implementation.

We assume functions *ap* and *cmp* over sets $\mathbb{V}$ and $\mathbb{O}$ as their counterparts in the class *Values*. We use some abuse of notation overloading v and w as terms and members of $\mathbb{V}$ and $\mathbb{O}$. In all cases each symbol belongs to the same domain as in the Core definition given before. In the rule *Call* the symbol f is a string. The context Γ is a function from names to Core terms. In the rules Cmp_t and Cmp_f we reduce into $\#t$ and $\#f$. Those are Church booleans [Jansen, 2013] defined as $\#t := \lambda$"T".λ"F"."T" and $\#f := \lambda$"T".λ"F"."F". The Core language has no builtin booleans and this is a nice example of how it is flexible enough to implement new constructs. We will use uppercase variables when we need binders to encode new constructs in the Core. Variables from compiled MateFun are allways lowercase. This will avoid name clashes.

The reduction defined by this set of rules is called parallel reduction in the literature. It can potentially reduce many redexes at the same time. It can be proved that the transitive closure of that relation is equal to the transitive closure of β-reduction with one redex at a time. Since for evaluation we are

$$\frac{\Gamma \vdash t \succ_\beta t' \qquad \Gamma \vdash u \succ_\beta u'}{\Gamma \vdash tu \succ_\beta t'u'}\ \textsc{App} \qquad \frac{}{\Gamma \vdash (\lambda x.t)u \succ_\beta t[x/u]}\ \beta$$

$$\frac{\Gamma \vdash t \succ_\beta t' \qquad \Gamma \vdash u \succ_\beta u'}{\Gamma \vdash t\Box u \succ_\beta t'\Box u'}\ \textsc{Op} \qquad \frac{f \in \Gamma \qquad \Gamma(f) = t}{\Gamma \vdash fu \succ_\beta tu}\ \textsc{Call}$$

$$\frac{\Gamma \vdash t \succ_\beta t' \qquad \Gamma \vdash u \succ_\beta u'}{\Gamma \vdash t\triangle u \succ_\beta t'\triangle u'}\ \textsc{Cmp}$$

$$\frac{\Gamma \vdash t \succ_\beta t'}{\Gamma \vdash \lambda x.t \succ_\beta \lambda x.t'}\ \textsc{Lam} \qquad \frac{v \in \mathbb{V} \qquad w \in \mathbb{V}}{\Gamma \vdash v\Box w \succ_\beta ap(\Box, v, w)}\ \textsc{Op}_2$$

$$\frac{v \in \mathbb{V} \qquad w \in \mathbb{V} \qquad cmp(\triangle, v, w) \equiv \texttt{True}}{\Gamma \vdash v\triangle w \succ_\beta \#t}\ \textsc{Cmp}_t$$

$$\frac{v \in \mathbb{V} \qquad w \in \mathbb{V} \qquad cmp(\triangle, v, w) \equiv \texttt{False}}{\Gamma \vdash v\triangle w \succ_\beta \#f}\ \textsc{Cmp}_f$$

Figure 6.6: Rules for Core reduction.

interested in getting a normal form, both definitions are equivalent for our purposes.

Now, let us implement this reduction relation in `AspectAG`. Let us define the attribute for the function environment and an aspect that encapsulates its semantics:

$fenv :: \forall\ v\ op.Label\ (\text{'}Att\ \texttt{"fenv"}\ (M.Map\ String\ (CTerm\ op\ v)))$
$fenv = Label$

$asp\_Core\_fenv = \lambda(Proxy :: Proxy\ v) \rightarrow asp\_Core\_env\ (fenv\ @\ v)$

And an attribute for reduction:

$redu :: \forall\ v\ op. Values\ op\ v \Rightarrow Label\ (\text{'}Att\ \texttt{"redu"}\ (CTerm\ op\ v))$
$redu = Label$

The definition for its semantics is given in Figure 6.7 with the definition of *asp_redu*, being a direct translation of what we defined in Figure 6.6. The main difference is that we consider error cases in all types of applications (proper applications, operators and comparissons), propagating the error messages.

```
asp_redu = λ(p :: Proxy v) →
     syndefM (redu @ v) p_CVal  (CVal  <$> ter ch_CVal_val)
   ◃ syndefM (redu @ v) p_CVar  (CVar  <$> ter ch_CVar_var)
   ◃ syndefM (redu @ v) p_CLam  (CLam <$> ter ch_CLam_binder
                                       ⊛ at  ch_CLam_body (redu @ v))
   ◃ syndefM (redu @ v) p_CApp  (do
     l   ← at ch_CApp_l (self @ v)
     l′  ← at ch_CApp_l (redu @ v)
     r   ← at ch_CApp_r (self @ v)
     r′  ← at ch_CApp_r (redu @ v)
     fe  ← at lhs (fenv @ v)
     case l of
          CLam x t → return $ subst p x r t
          CVar x   → case M.lookup x fe of
                       Just f → return $ CApp f r
                       Nothing → return $ CApp l r
          CError e → return $ CError e
          _        → return $ CApp l′ r′
     )
   ◃ syndefM (redu @ v) p_COp   (do
     l   ← at ch_COp_r (redu @ v)
     r   ← at ch_COp_l (redu @ v)
     op ← ter ch_COp_op
     case (l, r) of
        (CVal l′,  CVal r′) → return (CVal $ ap op l′ r′)
        (CError e, _)       → return $ CError e
        (_, CError e)       → return $ CError e
        (l, r)              → return $ COp l op r
     )
   ◃ syndefM (redu @ v) p_CComp (do
     l   ← at ch_CComp_r (redu @ v)
     r   ← at ch_CComp_l (redu @ v)
     op ← ter ch_CComp_op
     case (l, r) of
        (CVal l′, CVal r′) →
           case cmp op l′ r′ of
              True  → return (CLam "T" (CLam "F" (CVar "T")))
              False → return (CLam "T" (CLam "F" (CVar "F")))
        (CError e, _)      → return $ CError e
        (_, CError e)      → return $ CError e
        _                  → return (CComp l op r)
     )
   ◃ syndefM (redu @ v) p_CError  (CError <$> ter ch_CError_err)
   ◃ emptyAspect
```

Figure 6.7: Semantics for parallell reduction in `AspectAG`.

Finally, we can define functions such as *step* to perform a parallel reduction, and *reduce* to compute a normal form:

```
step :: ∀ op v. Values op v ⇒
  M.Map String (CTerm op v) → CTerm op v → CTerm op v
step funs expr = sem_CTerm  (  asp_redu (Proxy @ v)
                             ⋈ asp_Core_fenv (Proxy @ v)
                             ⋈ asp_Core_id (Proxy @ v))
                   expr (fenv .=. funs .*. emptyAtt) #. (redu @ v)

reduce :: (Eq op, Eq v, Values op v) ⇒
  M.Map String (CTerm op v) → CTerm op v → CTerm op v
reduce e ex =
  let ex' = step e ex
  in if ex == ex' then ex else reduce e ex'
```

We were sloppy in the definition of the function *reduce*, comparing the resulting expressions to detect if we got a normal form. This can be easily improved, for instance, by tracking if the expression changed when computing *asp_redu*. However, remember that the goal of this evaluaor is to use it to test our development. For these purposes this is more than enough.

6.6 L1 - MateFun expressions.

It is time to go through the implementation of language levels. L1 is the expression language of MateFun. In MateFun, expressions are used as the input in the REPL and in the definition of functions.

Since there is no way to globally bind variables in MateFun programs[15], variables are not used in the language of the REPL. So we could define two different language levels for expressions. However, this does not offer new insight according to the goals of this document. Also note that once defined L1 including variables, we can just implement the parser of REPL expressions to just not recognize variables. Also, a natural future extension of MateFun is to add the ability to define variables, so it makes sense also as a design decision to define the language expression once.

[15] A common idiom in MateFun programs is to define a unit type (it can be done with an enumeration, or latter versions introduce it built-in) and model constants with functions with the unit domain.

```
$ (addNont "Expr")
$ (addProd "Var"  'Nt_Expr [("var", Ter     'String)])
$ (addProd "Val"  'Nt_Expr [("val", Ter     'Poly)])
$ (addProd "Op"   'Nt_Expr [("l",   NonTer 'Nt_Expr),
                            ("op",  Ter     'Poly),
                            ("r",   NonTer 'Nt_Expr)])
$ (addProd "Call" 'Nt_Expr [("fun", Ter     'String),
                            ("arg", NonTer 'Nt_Expr)])
```

Figure 6.8: Expression language syntax.

6.6.1 Syntax definition.

MateFun expressions can be variables, values, applications of operators and function calls. The following EBNF grammar defines this language:

$$
\begin{aligned}
Expr &\rightarrow v\\
Expr &\rightarrow x\\
Expr &\rightarrow Expr \,\square\, Expr\\
Expr &\rightarrow f\,(Expr)
\end{aligned}
$$

where $\mathbb{V}$ denotes the set of values, $\mathbb{O}$ the set of operators, and $\mathfrak{S}$ the set of identifiers, and $v \in \mathbb{V}$, $x, f \in \mathfrak{S}$, and $\square \in \mathbb{O}$.

As a demonstration of how modular `AspectAG` is, in the implementation we reused parts of the code of the running example of Chapter 3. We used the non-terminal Nt_{Expr} and the productions for variables defined in `Expr.Syn`, and function calls implemented in `ExprExt.Syn`. Just to have this chapter self-contained we show all definitions in Figure 6.8.

We also define sem_{Expr}, the semantic function. While the expression language has only four productions, it will be part of a bigger language, so to avoid performance issues we will pack semantics in a non-extensible record. We just define a polymorphic tuple *SemExpr* as follows.

```
data SemExpr var val op call
   = SemExpr { gvar :: var,
               gval  :: val,
               gop   :: op,
               gcall :: call }
```

We will keep a convention we have used previously: when defining semantics for an attribute *att* at the production *prd*, we will call the defined rule as '*att_prd*', uncapitalized.

As always, we can reify the syntax definition to build an ordinary Haskell data to represent expressions, as follows:

```
data Expr v = Var String
            | Val v
            | Op (Expr v) (Val2Op v) (Expr v)
            | Call String (Expr v)
```

We used the indexed family *Val2Op*. Doing things this way the data type *Expr* is parametrized by only one polymorphic argument. As we have seen in the Core implementation we can put any number of arguments in this datatype, but having many makes it difficult to automatize the generation using Template Haskell.

6.6.2 T1 - Pretty-printing.

A naïve pretty-printer is simple to implement, while it gets challenging if we take seriously the quality of the printed code, for instance, taking indentation into account.

At least, it is important that the printed code is semantically equivalent to the abstract syntax being printed, which means adding brackets when they are necessary. The simplest approach is to add brackets everywhere, but in that case the quality of the generated code is not good.

There are two factors to take into account to avoid redundant brackets in an expression language: operator precedence and associativity [Ramsey, 1998]. The former is obvious while the latter applies in successive applications of operators with the same precedence value.

We advisedly took precedence out of the *Values* interface to avoid dealing with concrete syntax there, giving the interface a semantic imprint. However, type classes are flexible to solve this issue since we can extend them. If we want to pretty print values, we can use the following interface:

```
class (Values op v) ⇒ PPValues op v | v → op, op → v where
  opProperties :: op → (TAssoc, Int)


data TAssoc = RightAssoc | LeftAssoc | NonAssoc
```

An operator has associativity and precedence values. The higher the precedence value, the tighter the operator "binds". When printing an expression we decide if we put brackets in subexpressions depending on the precedence of the subexpressions last operators and their precedence.

We implement a synthesized attribute *slastop* that returns the properties of the operator applied if there is such, or *Nothing* when there is no operator at all (in variables, values, and applications).

```
slastop :: Label ('Att "lastop" (Maybe (TAssoc, Int)))
slastop = Label
```

The printed string is computed within a synthesized attribute *spp*:

```
spp :: Label ('Att "pp" String)
spp = Label
```

The most interesting piece of semantics appears of course in the *P_Op* production:

```
spp_op = λ(p :: Proxy v) →
  syndefM spp p_Op (do
    lp ← at ch_Op_r spp
    op ← ter (ch_Op_op @ (Val2Op v))
    rp ← at ch_Op_r spp
    llop ← at ch_Op_r slastop
    lrop ← at ch_Op_l slastop
    return $ handleParen lp op rp llop lrop
  )
```

We just collect all the interesting information and use the ordinary Haskel function *handleParen* to produce the string. The reader can refer to the codebase to see the details of this function.

All other rules are simple:

$$\begin{array}{ll}
\mathit{spp\_var} & = \mathit{const}\ \$\ \mathit{syndefM}\ \mathit{spp}\ p_{\mathit{Var}}\ (\mathit{ter}\ \mathit{ch}_{\mathit{Var}_{\mathit{var}}}) \\
\mathit{spp\_val} & = \lambda(p :: \mathit{Proxy}\ v) \rightarrow \\
\quad \mathit{syndefM}\ \mathit{spp}\ p_{\mathit{Val}}\ (\mathit{show}\ @\ v\ \mathbin{<\$>}\ \mathit{ter}\ (\mathit{ch}_{\mathit{Val}_{\mathit{val}}}\ @\ v)) & \\
\mathit{spp\_call} & = \mathit{const}\ \$\ \mathit{syndefM}\ \mathit{spp}\ p_{\mathit{Call}}\ (\mathbf{do} \\
\quad \mathit{fname} \leftarrow \mathit{ter}\ \mathit{ch}_{\mathit{Call}_{\mathit{fun}}} & \\
\quad \mathit{argp}\ \ \ \leftarrow \mathit{at}\ \mathit{ch}_{\mathit{Call}_{\mathit{arg}}}\ \mathit{spp} & \\
\quad \mathit{return}\ \$\ \mathit{fname} \mathbin{+\!\!+} \texttt{"("} \mathbin{+\!\!+} \mathit{argp} \mathbin{+\!\!+} \texttt{")"} & \\
\quad) & \\
\mathit{slastop\_var} & = \mathit{const}\ \$\ \mathit{syndefM}\ \mathit{slastop}\ p_{\mathit{Var}}\ \ (\mathit{return}\ \mathit{Nothing}) \\
\mathit{slastop\_val} & = \mathit{const}\ \$\ \mathit{syndefM}\ \mathit{slastop}\ p_{\mathit{Val}}\ \ (\mathit{return}\ \mathit{Nothing}) \\
\mathit{slastop\_call} & = \mathit{const}\ \$\ \mathit{syndefM}\ \mathit{slastop}\ p_{\mathit{Call}}\ (\mathit{return}\ \mathit{Nothing}) \\
\mathit{slastop\_op} & = \lambda(p :: \mathit{Proxy}\ v) \rightarrow \\
\quad \mathit{syndefM}\ \mathit{slastop}\ p_{\mathit{Op}}\ (\mathit{Just} \mathbin{<\$>} \mathit{opProperties}\ @_{\_}\ @\ v & \\
\qquad\qquad\qquad\qquad \mathbin{<\$>} \mathit{ter}\ (\mathit{ch}_{\mathit{Op}_{\mathit{op}}}\ @\ (\mathit{Val2Op}\ v))) &
\end{array}$$

6.6.3 T2 - Name Binding.

Name binding will make more sense once the expression language is a subset of a language with binders. However, there are things we can take into account at this point. We can collect the free variables to use the aspect later. We could control if names are legal since MateFun variables and functions are capitalized alphanumeric strings, but we consider this a parsing problem.

Let us define the synthesized attributes *sfv* and *sffv*, to collect the free variables, and free function names, respectively.

$$\begin{array}{l}
\mathit{sfv} :: \mathit{Label}\ (\text{'}\mathit{Att}\ \texttt{"fv"}\ [\mathit{String}]) \\
\mathit{sfv} = \mathit{Label}
\end{array}$$

$$\begin{array}{l}
\mathit{sffv} :: \mathit{Label}\ (\text{'}\mathit{Att}\ \texttt{"ffv"}\ [\mathit{String}]) \\
\mathit{sffv} = \mathit{Label}
\end{array}$$

Controlling that variables and function names occurring in an expression are in scope are similar tasks. However, while variables can be bound just in equation definitions, functions can be defined anywhere in the program. This implies that variables require less effort, so we take different approaches

```
var_env att =
  SemExpr (const emptyRule)
          (const emptyRule)
          (const $ inhdefM att p_Op ch_Op_r (at lhs att)
                 ◇ inhdefM att p_Op ch_Op_r (at lhs att))
          (const $ inhdefM att p_Call ch_Call_arg (at lhs att))
```

Figure 6.9: Copy rules in expressions.

accordingly. On the one hand, for variables, we just collect all of them. Later, when using expressions in a richer language with binders, in the production where the binder occurs, we will erase bound variables from the value of *sfv*. On the other hand, for functions, we consider we have available an environment with the defined function names in scope, and when a function name occurs, deciding whether it is free depends on the contents of that environment.

The attribute *ifnames* contains function names in scope:

```
ifnames :: Label ('Att "fnames" [String])
ifnames = Label
```

The semantics are trivial, just copy rules. We will use this pattern many times so it is nice to have a macro. The following would be the definition using extensible records:

```
var_env att = inhdefM att p_Op ch_Op_r (at lhs att)
            ◁ inhdefM att p_Op ch_Op_l (at lhs att)
            ◁ inhdefM att p_Call ch_Call_arg (at lhs att)
            ◁ emptyAspect
```

but remember we are using non-extensible records to store semantics. In Figure 6.9 the macro we actually use is given. Applying the function *var_env* to the attribute *ifnames* we get an aspect with the semantics for it.

Finally, Figure 6.10 shows the definition of all rules.

For instance, the following function traverses an expression and collects *all* occurring variables.

```
getVars e = let iatts = sem_Expr asp_FreeVars (proxyFrom e) e
                          (ifnames =. [] *. emptyAtt)
            in (iatts #. sffv, iatts #. sfv)
```

$$
\begin{array}{l}
\mathit{sfv\_var} \;= \mathit{syndefM}\ \mathit{sfv}\ p_{\mathsf{Var}}\ ((:[\,])\ \mathbin{<\$>}\ \mathit{ter}\ \mathit{ch}_{\mathit{Var}_{\mathit{var}}}) \\
\mathit{sfv\_val} \;\;= \mathit{syndefM}\ \mathit{sfv}\ p_{\mathsf{Val}}\ \$\ \mathit{return}\ [\,] \\
\mathit{sfv\_op} \;\;\,= \mathit{syndefM}\ \mathit{sfv}\ p_{\mathsf{Op}}\ ((+\!\!+)\ \mathbin{<\$>}\ \mathit{at}\ \mathit{ch}_{\mathit{Op}_r}\ \mathit{sfv} \circledast \mathit{at}\ \mathit{ch}_{\mathit{Op}_l}\ \mathit{sfv}) \\
\mathit{sfv\_call} = \mathit{syndefM}\ \mathit{sfv}\ p_{\mathsf{Call}}\ (\mathit{at}\ \mathit{ch}_{\mathit{Call}_{\mathit{arg}}}\ \mathit{sfv})
\end{array}
$$

$$
\begin{array}{l}
\mathit{sffv\_var} \;= \mathit{syndefM}\ \mathit{sffv}\ p_{\mathsf{Var}}\ \$\ \mathit{return}\ [\,] \\
\mathit{sffv\_val} \;\;= \mathit{syndefM}\ \mathit{sffv}\ p_{\mathsf{Val}}\ \$\ \mathit{return}\ [\,] \\
\mathit{sffv\_op} \;\;\,= \mathit{syndefM}\ \mathit{sffv}\ p_{\mathsf{Op}}\ ((+\!\!+)\ \mathbin{<\$>}\ \mathit{at}\ \mathit{ch}_{\mathit{Op}_r}\ \mathit{sffv} \circledast \mathit{at}\ \mathit{ch}_{\mathit{Op}_l}\ \mathit{sffv}) \\
\mathit{sffv\_call} = \mathit{syndefM}\ \mathit{sffv}\ p_{\mathsf{Call}}\ ((\mathbf{do} \\
\quad \mathit{fnames} \leftarrow \mathit{at}\ \mathit{lhs}\ \mathit{ifnames} \\
\quad \mathit{fname} \leftarrow \mathit{ter}\ \mathit{ch}_{\mathit{Call}_{\mathit{fun}}} \\
\quad \mathit{return}\ \$\ \mathbf{if}\ \mathit{elem}\ \mathit{fname}\ \mathit{fnames}\ \mathbf{then}\ [\,]\ \mathbf{else}\ [\mathit{fname}] \\
\quad))
\end{array}
$$

Figure 6.10: Semantics for free names.

Since the initial inherited attribute *ifnames* is defined empty, all function names are collected in the first member of the resulting pair. All variable names are collected in the second. We can, for instance, process this information further to check if variables satisfy syntactic rules.

6.6.4 T3 - Code generation.

Translation of expressions to the Core language is perhaps surprisingly easy since the Core language is actually more complex than the expression language itself.

We define the polymorphic attribute *scomp* that will contain the resulting compiled term:

$$
\begin{array}{l}
\mathit{scomp} :: \forall\ v\ \mathit{op}.\mathit{Values}\ \mathit{op}\ v \Rightarrow \mathit{Label}\ (\text{'}\mathit{Att}\ \texttt{"compile"}\ (\mathit{CTerm}\ \mathit{op}\ v)) \\
\mathit{scomp} = \mathit{Label}
\end{array}
$$

In Figure 6.11 we show the rules, the code is relatively simple, we build the corresponding Core terms from expression terms. We do not control any property about expressions, this is the job of other aspects. Ill-formed expressions will produce ill-formed Core.

$scomp\_var = \lambda(Proxy :: Proxy\ v) \rightarrow$
$\quad syndefM\ (scomp\ @\ v)\ p_{Var}\ (CVar <\$> ter\ ch_{Var_{var}})$

$scomp\_val = \lambda(Proxy :: Proxy\ v) \rightarrow$
$\quad syndefM\ (scomp\ @\ v)\ p_{Val}\ (CVal <\$> ter\ (ch_{Val_{val}}\ @\ v))$

$scomp\_op = \lambda(Proxy :: Proxy\ v) \rightarrow$
$\quad syndefM\ (scomp\ @\ v)\ p_{Op}\ (COp <\$> at\ ch_{Op_r}\ (scomp\ @\ v)$
$\qquad\qquad\qquad\qquad \circledast\ ter\ (ch_{Op_{op}}\ @\ (Val2Op\ v)))$
$\qquad\qquad\qquad\qquad \circledast\ at\ ch_{Op_l}\ (scomp\ @\ v))$

$scomp\_call = \lambda(Proxy :: Proxy\ v) \rightarrow$
$\quad syndefM\ (scomp\ @\ v)\ p_{Call}\ (\mathbf{do}\ f \quad \leftarrow ter\ ch_{Call_{fun}}$
$\qquad\qquad\qquad\qquad carg \leftarrow at\ ch_{Call_{arg}}\ (scomp\ @\ v)$
$\qquad\qquad\qquad\qquad return\ \$\ CApp\ (CVar\ f)\ carg$
$\quad)$

Figure 6.11: Rules for compilation of the expression language (T1-L3).

6.6.5 T4 - Type Checking.

Given an expression, if we know the types of their ocurring variables and functions, we can compute the least-general base type of the resulting expression.

We define the inherited attributes *itypes* and *ftypes* to model the environments with the types of variables and functions in scope. With the synthesized attribute *stype* we compute the result. The attributes are defined as follows:

$itypes :: \forall\ v\ op\ t.Types\ op\ v\ t \Rightarrow Label\ (\text{'}Att\ \texttt{"types"}\ (M.Map\ String\ t))$
$itypes = Label$

$iftypes :: \forall\ v\ op\ t.Types\ op\ v\ t \Rightarrow Label\ (\text{'}Att\ \texttt{"ftypes"}\ (M.Map\ String\ (t, t)))$
$iftypes = Label$

$stype :: \forall\ v\ op\ t.Types\ op\ v\ t \Rightarrow Label\ (\text{'}Att\ \texttt{"type"}\ (Either\ String\ t))$
$stype = Label$

The type of the attribute *stype* can contain a string, denoting a type error. In that case, the string contains the error message to print.

We introduced a new idiom in these definitions. While the three attributes are polymorphic in the type t, we use an explicit quantifier to put v as the first argument, and use the constraint *Types op v* to relate all type variables. By doing this, we can always apply the variable v in the polymorphic attributes.

Semantics for *itypes* and *iftypes* are simple, we use the macro defined in Figure 6.9. Rules for *stype* are defined in Figure 6.12.

We use extensively the interface given by the type class *Types*. In the case of variables we lookup the environment. When there is a literal we apply *typeOf*. To infer the type of an expression defined from an operator application, we use the function *tyCombM*, which is defined from *tyComb* but handling arguments that could fail, as follows:

```
tyCombM :: Types op v t ⇒
  Either String t → op → Either String t → Either String t
tyCombM lt (op :: op) (rt :: Either String t)
   = do l ← lt
        r ← rt
        tyComb l op r
```

Finally, in the case of function application, we control that the type of the argument is a subset of the declared domain. If everything goes well the resulting type is the codomain of the function. Note that we defined a sloppy message `"type error"` as an example, but there is a lot of information in scope to build a richer message.

6.7 L2 - Programs.

In this section, we extend the expression language to define MateFun programs. A MateFun program is a list of function definitions. MateFun programs are compiled to mappings from function names to Core terms, used as environments to evaluate Core expressions.

6.7.1 Syntax definition.

The following EBNF notation defines the grammar to represent MateFun programs:

```
$stype\_var = \lambda(Proxy :: Proxy\ v) \rightarrow$
  $syndefM\ (stype\ @\ v)\ p_{Var}$ (do
    $x \leftarrow ter\ ch_{Var_{var}}$
    $gamma \leftarrow at\ lhs\ (itypes\ @\ v)$
    $return\ \$\ Right\ \$\ (fromJust\ \$\ M.lookup\ x\ gamma)$

$stype\_val = \lambda(Proxy :: Proxy\ v) \rightarrow$
  $syndefM\ (stype\ @\ v)\ p_{Val}\ (typeOf\ <\$>\ ter\ (ch_{Val_{val}}\ @\ v)$

$stype\_op = \lambda(Proxy :: Proxy\ v) \rightarrow$
  $syndefM\ (stype\ @\ v)\ p_{Op}$ (do
    $lt\ \leftarrow at\ ch_{Op_r}\ (stype\ @\ v)$
    $rt\ \leftarrow at\ ch_{Op_l}\ (stype\ @\ v)$
    $op \leftarrow ter\ (ch_{Op_{op}}\ @\ (Val2Op\ v))$
    $return\ \$\ tyCombM\ lt\ op\ rt$
  )

$stype\_call = \lambda(Proxy :: Proxy\ v) \rightarrow$
  $syndefM\ (stype\ @\ v)\ p_{Call}$ (do
    $argty\ \ \ \leftarrow at\ ch_{Call_{arg}}\ (stype\ @\ v)$
    $f\ \ \ \ \ \ \ \ \ \leftarrow ter\ ch_{Call_{fun}}$
    $gamma \leftarrow at\ lhs\ (iftypes\ @\ v)$
    case $argty$ of
      $Left\ error \rightarrow return\ argty$
      $\_\ \ \ \ \ \ \ \ \ \ \rightarrow$
        case $M.lookup\ f\ gamma$ of
          $Just\ (dom, cod) \rightarrow$
            $return\ \$$ if $isSubtype'\ argty\ dom$
                      then $Right\ (cod :: Val2Type\ v)$
                      else $Left$ "type error"
          $Nothing\ \ \ \ \ \ \ \ \rightarrow error$ "impossible!"
  ) where
    $isSubtype'\ argty\ dom =$ case $argty$ of
      $Right\ ty \rightarrow isSubtype\ ty\ dom$
      $\_\ \ \ \ \ \ \ \ \rightarrow False$
```

Figure 6.12: Rules for typing expressions.

$$\begin{aligned}
Program &\rightarrow [FDef] \\
FDef &\rightarrow f\, Ecu \\
Ecu &\rightarrow (x_1, \ldots, x_n) = ExprG \\
ExprG &\rightarrow Expr \\
ExprG &\rightarrow Expr \textbf{ if } Cond \textbf{ or } ExprG \\
Cond &\rightarrow Expr \triangle Expr \\
Cond &\rightarrow \top \\
Cond &\rightarrow Cond \wedge Cond \\
Cond &\rightarrow \neg Cond
\end{aligned}$$

where $x_i \in \mathfrak{S}$, and $\triangle \in \{<, >, =\}$.

Programs are lists of function definitions. Functions have a name and an equation. Equations are a list of variables and a *guarded* expression[16][17]. Guarded expressions are used to represent piecewise definitions. They are just an expression or a conditional. Conditionals rewrite to further guarded expressions in the 'else' case so we can nest conditions. Conditions are built from a comparison of expressions or by combining them using boolean operators. We defined an arbitrary set of boolean operators to be complete. Further operators such as disjunction could be desugared to what we have here, (or we could extend this language with new productions, of course!).

The reader should be convinced that the following program can be represented easily with this syntax[18]:

```
abs (x) =   x if x >= 0
        or -x
```

In Figure 6.14 we give the definitions in `AspectAG`. It is a direct translation of the formal definition that was given before. Lists of functions are implemented with two productions $p_{ProgramNil}$ and $p_{ProgramSnoc}$. We use lists

[16] This name is taken from the reference implementation.
[17] We used some notational abuse here to represent a variable-length list of variables (the correct way using EBNF would be to define another non-terminal, but it is more clear if written this way [18] The expression "$-x$" must be represented as "$0 - x$", though.

that are extended at the tail just because we tend to think in programs built top-down in the source code, but the decision is not important at all. Equations have a terminal of type *Vars* that contains the variables bound by the pattern-matching. Of course, we also define semantic functions for all new productions, notably sem_{Expr}. We also define a datatype to store semantic functions, as follows:

```
data SemFunc eif eor top neg and comp ecu fdef pnil psnoc var val op call =
  SemFunc { geif    :: eif,
            geor    :: eor,
            gtop    :: top,
            gneg    :: neg,
            gand    :: and,
            gcomp   :: comp,
            gfdef   :: fdef,
            gecu    :: ecu,
            gpnil   :: pnil,
            gpsnoc  :: psnoc,
            gexpr   :: (SemExpr var val op call)
     }
```

Note that semantics for expressions are a field of this record, so by defining an instance of *SemFunc* we are extending an instance of *SemExpr*.

In Figure 6.13 we show the reification of L2 as an ordinary Haskell datatype.

6.7.2 T1 - Pretty-printing.

To implement pretty-printing we just need to implement semantics for the already defined attributes *spp* and *slastop* at the new productions. Note that conditions in guarded expressions are themselves a language similar to the expression language of L1 in the sense that we have a set of operators (the boolean operators), so we should minimize brackets. Since conjunctions are associative and with lighter precedence than negation, adding brackets in all negations generates good results.

However, the language could be extended later, so we will do the job with the same technique as before, implementing semantics to compute the attribute *slastop*. The following rules do the job:

```
data FDef v = FDef String (Ecu v)

data Ecu v = Ecu Vars (ExprG v)

data ExprG v = Or (Expr v)
             | If (Expr v) (Cond v) (ExprG v)

data Cond v = Top
            | Comp (Expr v) Ordering (Expr v)
            | And (Cond v) (Cond v)
            | Neg (Cond v)
```

Figure 6.13: Data types for programs of L2.

```
slastop_top  = syndefM slastop p_Top  (return Nothing)
slastop_neg  = syndefM slastop p_Neg  (return Nothing)
slastop_and  = syndefM slastop p_And  (return $ Just (RightAssoc, 3))
slastop_comp = syndefM slastop p_Comp (return $ Just (NonAssoc, 4))
```

We use the same associativity values of Haskell, which is consistent with respect to the reference implementation of MateFun.

To print function definitions we must take indentation into account. We take a simple approach: We define function bodies one line after the assignation symbol, so we can indent always the same number of whitespaces. The most relevant rules are the following:

```
spp_eor = const $ syndefM spp p_EOr (at ch_EOr_or spp)
spp_eif = const $ syndefM spp p_EIf (do
  ife ← at ch_EIf_if spp
  ce  ← at ch_EIf_cond spp
  ee  ← at ch_EIf_else spp
  return $ ife ++ " if " ++ ce ++ "\n     or " ++ ee
  )
```

```
$ (addNont "ExprG")
$ (addProd "EOr" "Nt_ExprG [("or",    NonTer "Nt_Expr)])
$ (addProd "EIf" "Nt_ExprG [("if",    NonTer "Nt_Expr),
                            ("cond", NonTer "Nt_Cond)
                            ("else", NonTer "Nt_ExprG)])


$ (addNont "Cond")
$ (addProd "Top"  "Nt_Cond [])
$ (addProd "Neg"  "Nt_Cond [("e",  NonTer "Nt_Cond)])
$ (addProd "And"  "Nt_Cond [("l",  NonTer "Nt_Cond),
                            ("r",  NonTer "Nt_Cond)])
$ (addProd "Comp" "Nt_Cond [("l",  NonTer "Nt_Expr),
                            ("op", Ter    "Ordering),
                            ("r",  NonTer "Nt_Expr)])


$ (addNont "Ecu")
type Vars = [String]
$ (addProd "Ecu" "Nt_Ecu [("vars",  Ter    "Vars),
                          ("body", NonTer "Nt_ExprG)])


$ (addNont "FDef")
$ (addProd "FDef" "Nt_FDef [("name", Ter "String),
                             ("ecu", NonTer "Nt_Ecu)])


$ (addNont "Program")
$ (addProd "ProgramNil"  "Nt_Program [])
$ (addProd "ProgramSnoc" "Nt_Program
                          [("init", NonTer "Nt_FDef),
                           ("last", NonTer "Nt_Program)])
```

Figure 6.14: Syntax definition for MateFun programs.

```
$spp\_ecu = const\ \$\ syndefM\ spp\ p_{Ecu}$ (do
  $vars \leftarrow ter\ ch\_Ecu\_vars$
  $body \leftarrow at\ ch\_Ecu\_body\ spp$
  $return\ \$\ wrapBrackets\ (intercalate$ ", " $vars)$
        $+\!\!+$ " =\n          " $+\!\!+\ body$
  )
```

```
$spp\_fdef = const\ \$\ syndefM\ spp\ p_{FDef}\ ((+\!\!+) <\$> ter\ ch\_FDef\_name$
                                              $\circledast\ \ at\ ch\_FDef\_ecu\ spp)$
```

More complex handling for indentation could be considered, for instance we could set a custom width of indentation levels by computing the length of indentation and passing it down using an inherited attribute.

With these implementation, the function `abs` would be printed as:

```
abs (x) =
       x if x >= 0
    or 0 - x
```

Note that the expression $-x$ is printed as `"0 - x"` since it was represented this way in the AST. Pretty-printers are not neccesary an inverse of parsers, but the important thing to note is that semantics were preserved. An improved pretty-printer can handle such cases with no great effort. However, in this case we think the issue is more a responsability of how the AST is implemented. Wether if users wrote `"0-x"` of `"-x"`, both expressions are parsed as the same AST. Thus, a good solution is to extend the expression language to handle negations instead of desugaring negations as an instance of the binary operator. We do not do it here since it does not offer new insights.

6.7.3 T2 - Name binding.

Name binding is extended easily. We keep collecting variable occurences with the attributes *sffv* and *sfv*, and passing down the environment *ifnames*. The main task to perform is to handle the binder introduced in function definitions. This is performed in the production p_{Ecu}. There, we must erase the bound variables from the list of free variables of the guarded expression.

```
sfv_ecu = syndefM sfv p_Ecu (do
    vars  ← ter ch_Ecu_vars
    bvars ← at ch_Ecu_body sfv
    return $ foldl (λr a → erase a r) bvars vars
    )

erase x [ ] = [ ]
erase x (y : ys)
    | x == y     = erase x ys
    | otherwise = y : erase x ys
```

In well-formed programs there should not be free variables in the synthesized attribute *sfv* in each p_{Ecu} production. If there are, we want to generate error messages that provide at least the information of in which functions they occur. To do that, in functions and lists of functions we use a more expressive attribute instead of *sfv*, defined as follows:

```
sfunfv :: Label ('Att "funfv" ([(String, [String])]))
sfunfv = Label
```

The attribute *sfunfv* collects a list of function names and free variable occurences. Rules can be defined as follows:

```
sfunfv_fdef = syndefM sfunfv p_FDef (do
    fname ← ter ch_FDef_name
    fvars  ← at ch_FDef_ecu sfv
    return [(fname, fvars)]
    )

sfunfv_programnil   = syndefM sfunfv p_ProgramNil   (return [ ])
sfunfv_programsnoc = syndefM sfunfv p_ProgramSnoc (do
    funcs ← at ch_ProgramSnoc_init sfunfv
    func  ← at ch_ProgramSnoc_last sfunfv
    return $ funcs ++ func
    )
```

With the value of *sfunfv* at the root of the AST we can control if there are free variables in functions, if all functions have different names, and so on.

Although we do not do this in this prototype implementation, a sanity check that we can perform in this task is controlling that the list of variables in a pattern matching does not repeat names.

```
sfcomp :: ∀ v op. Values op v
   ⇒ Label ('Att "fcompile" (String, CTerm op v))
sfcomp = Label


sfcomp_fdef = λ(Proxy :: Proxy v) →
   syndefM (sfcomp @ v) p_FDef
      $ (, ) <$> ter ch_FDef_name ⊛ at ch_FDef_ecu (scomp @ v)


spcomp :: ∀ v op. Values op v
   ⇒ Label ('Att "pcompile" (M.Map String (CTerm op v)))
spcomp = Label


spcomp_nil = λ(Proxy :: Proxy v) →
   syndefM (spcomp @ v) p_ProgramNil (return M.empty)


spcomp_snoc = λ(Proxy :: Proxy v) →
   syndefM (spcomp @ v) p_ProgramSnoc (do
      fenv     ← at ch_ProgramSnoc_init (spcomp @ v)
      (f, ecu) ← at ch_ProgramSnoc_last (sfcomp @ v)
      return $ M.insert f ecu fenv
   )
```

Figure 6.15: Semantics for function compilation.

6.7.4 T3 - Code generation.

Functions are compiled to what we will use as environments to evaluate compiled expressions. In Section 6.5.1.2 we defined the attribute *fenv*, containing values of type $(Map\ String\ (CTerm\ op\ v))$. It stores all the compiled functions in scope when reducing Core terms.

The synthesized attribute *spcomp* is used to compute such an environment from a program. In Figure 6.15 we show how it is defined and the corresponding rules. We use *spcomp* in the productions of *nt_Program* and *sfcom* in the production of nt_{FDef} (though we could use *spcom* in both by returning a singleton map).

Note that in the definition of *sfcomp_fdef* we return a pair, composed by

```
tt = CLam "T" $ CLam "F" $ CVar "T"
ff = CLam "T" $ CLam "F" $ CVar "F"

ite = CLam "C" $ CLam "T" $ CLam "E" $
  CApp (CApp (CVar "C") (CVar "T")) (CVar "E")

neg = CLam "E" $ CLam "T" $ CLam "F"
  $ ap2 (CVar "E") (CVar "F") (CVar "T")

conj = CLam "L" $ CLam "R" $
  ap3 ite (CVar "L") (CVar "R") ff

ap3 f a b c = CApp (CApp (CApp f a) b) c
ap2 f a b   = CApp (CApp f a) b

pair l r = CLam "P" (CApp (CApp (CVar "P") l) r)
```

Figure 6.16: Encodings in the Core language.

the name of the function and the value of the already defined (see Section 6.6.4) attribute *scomp*.

What we have left to do is to define the semantics for *spcomp* at the productions p_{Ecu}, p_{ExprG} and p_{Cond}.

Since Core terms only have lambda abstractions with one variable, we curry the compiled functions. Semantics for p_{Ecu} are defined by the following rule:

```
scomp_ecu = λ(Proxy :: Proxy v) →
  syndefM (scomp @ v) p_Ecu (do
    vars ← ter ch_Ecu_vars
    body ← at ch_Ecu_body (scomp @ v)
    return $ foldr CLam body vars
  )
```

For instance, a term like (*Ecu* ["x", "y"] *e*) is compiled into (*CLam* "x" (*CLam* "y" e')) where e' is the result of compiling e.

Note that there is no way to define guarded expressions and conditions primitively in Core. We can encode them using Church encodings (or any other encoding) [Jansen, 2013,Barendregt, 1985]. In Figure 6.16 we give some encodings we use. The terms *tt* and *ff* are the Church booleans. The term *ite* is the conditional. *neg* and *conj* are negation and conjunction. The reader can check that under the semantics for Core reduction defined in Section 6.5.1.2 that applications reduce as intended.

Using those constructs we can define the following rules:

```
scomp_eif = λ(Proxy :: Proxy v) →
  syndefM (scomp @ v) p_EIf (ap3 <$> pure ite
                                  ⊛  at ch_EIf_cond (scomp @ v)
                                  ⊛  at ch_EIf_if   (scomp @ v)
                                  ⊛  at ch_EIf_else (scomp @ v)
  )


scomp_eor = λ(Proxy :: Proxy v) →
  syndefM (scomp @ v) p_EOr (at ch_EOr_or (scomp @ v))


scomp_top = λ(Proxy :: Proxy v) →
  syndefM (scomp @ v) p_Top (return tt)


scomp_neg = λ(Proxy :: Proxy v) →
  syndefM (scomp @ v) p_Neg (CApp neg <$> at ch_Neg_e (scomp @ v))


scomp_and = λ(Proxy :: Proxy v) →
  syndefM (scomp @ v) p_And (ap2 <$> pure conj
                                  ⊛  at ch_And_l (scomp @ v)
                                  ⊛  at ch_And_r (scomp @ v))


scomp_comp = λ(Proxy :: Proxy v) →
  syndefM (scomp @ v) p_Comp (CComp <$> at ch_Comp_l (scomp @ v)
                                     ⊛  ter ch_Comp_op
                                     ⊛  at ch_Comp_r (scomp @ v))
```

6.7.5 T4 - Type checking.

L2-T4 is a weird combination. Function definitions do not have declared types, so we cannot guess the type of the binders in equations to extend the environments used to perform type cheching in expressions.

Still, here it makes sense to extend the semantics defined for expressions to guarded expressions. Environments are just passed down to the recursive children. For the attribute *stype* (that computed the least-general type of an expression) we have work to do.

In the production *p_Or* we just get the type of the child expression:

```
stype_eor = λ(Proxy :: Proxy v) →
  syndefM (stype @ v) p_EOr (at ch_EOr_or (stype @ v))
```

In conditionals, all the possible returning expressions should have a compatible type, in the sense that we must be able to compute the union. The least-general type of the guarded expression is the union among all types. Another thing to take into account is that expressions in conditionals should not be ill-typed.

This can be implemented as follows:

```
stype_eif = λ(Proxy :: Proxy v) →
  syndefM (stype @ v) p_EIf (do
    ty1 ← at ch_EIf_if (stype @ v)
    ty2 ← at ch_EIf_else (stype @ v)
    cwt ← at ch_EIf_cond scwt
    return $ case (ty1, ty2, cwt) of
      (Right t1, Right t2, Right ()) → join t1 t2
      _ → combineErr ty1 ty2 cwt
  )
```

where the attribute *scwt* is defined as follows:

```
scwt :: Label ('Att "cwt" (Either String ()))
scwt = Label
```

```
scwt_top = syndefM scwt p_Top (return $ Right ())
scwt_neg = syndefM scwt p_Neg (at ch_Neg_e scwt)
scwt_and = syndefM scwt p_And (do
  l ← at ch_And_l scwt
  r ← at ch_And_r scwt
  return $ comb l r)
scwt_comp = λ(Proxy :: Proxy v) →
  syndefM scwt p_Comp (do
  l ← at ch_Comp_l (stype @ v)
  r ← at ch_Comp_r (stype @ v)
  return $ joinM l r)
```

Its semantics are simple, it collects type errors in conditions, whenever they occur.

6.8 L3 - Lists and enumerations.

We started with a language that manipulated integers and real numbers as its values. Now we aim to extend that set of terminals. We decided to add lists (sequences, in MateFun's jargon) and enumerations in this language level.

The implementation of this language level shows the power of the abstraction over terminals. We do not need to define either any new syntax in `AspectAG`, nor any new semantics. We just must define the new terminal types and implement the required interfaces.

6.8.1 Syntax definition.

The following data types are a possible implementation of the extended terminals:

```
data Op = Add | Times | Div | Minus | Cons
data Val = I Integer | R Double | List [Val] | Nil | Variant String
```

We implement MateFun's cons and nil as an operator and a value. Constructors of enumerations have a name. We internally use the term "variant" since that is what they are, in programming language theory terms.

We used the same names as before for the data types. We can since they are defined in a different module. Using qualified names we can eliminate any ambiguity. To build polymorphic ASTs with those new values we implement a class they satisfy, extending the *Values* class, as follows:

```
class (Values op v) ⇒ ListEnum op v | op → v, v → op where
  nil     :: v
  cons    :: v → v → v
  variant :: String → v
```

6.8.2 T1 - Pretty-printing.

The pretty-printer already implemented works, but it depends in the implementations of *Show* for *Op* and *Val*, and (*PPValues Op Val*). They can be defined easily.

6.8.3 T2 - Name binding, and T3 - Code generation.

Name binding and compilation to Core do not require any implementation.

6.8.4 T4 - Type checking.

We must provide a data type for types. The following implementation is given:

```
data Ty =
  TyR | TyZ | Sec Ty | Enum (Maybe String) [String] | Any
    deriving Eq
```

Enumerations have a name (the name of the set, if known) and a set of names, the names of all its members. We also introduced the constructor *Any*. It represents a polymorphic type. MateFun is monomorphic, but during type checking we face the issue that we do not know what type assigning to the empty list. We type the empty list with the type *Any* until more information is known.

Again, no new `AspectAG` definitions are required. The only thing to do is to implement instances of (*Types Op Val Ty*) and (*Show Type*) (for the pretty-printer).

6.9 L4 - Tuples.

Now, we want to add tuples to our language. Tuples are different with respect to the terminals introduced in L3 because we really need new syntax. Tuples of ground values can be implemented by the type of terminals, but tuples containing a pair of arbitrary expressions cannot. For instance, consider the following MateFun program:

```
f (x, y) = (g(x,y), 0)
```

In the language levels, we have implemented there is no way to represent neither the tuple returned nor the arguments of the function `g`. However, we already can represent the binder of `f`, which is also a tuple. Actually, in the languages defined until L3 the ability to write that pattern matching was completely useless since there was no way to call functions like `f`. We designed p_{Ecu} this way having possible extensions in mind. If we had not done it, we could just redefine the production p_{FDef} at this point.

We could represent tuples in the AST using just operators and values, such as we did with lists. We define an empty tuple as a value, and an operator that extends tuples. The type checking task could ensure that they behave as intended, having the size statically fixed. However, this solution is unnatural and hard to implement. Adding syntax makes much more sense to us.

To represent tuples of arbitrary length we can introduce a recursive production as we did with programs. Just for the sake of simplicity, we are going to introduce just pairs.

6.9.1 Syntax definition.

We extend the EBNF definition of the non-terminal *Expr* with the following production.

$$Expr \rightarrow (Expr, Expr)$$

We implement the new production of the expression language, representing pairs, as follows:

$\$\ (addProd\ \texttt{"Pair"}\ ''Nt\_Expr\ [(\texttt{"l"}, NonTer\ ''Nt\_Expr),$
$\qquad\qquad\qquad\qquad\qquad\qquad (\texttt{"r"}, NonTer\ ''Nt\_Expr)])$

The reified AST can be written as follows:

```
data Expr v = Val v
            | Var String
            | Op (Expr v) (Val2Op v) (Expr v)
            | Call String (Expr v)
            | Pair (Expr v) (Expr v)
```

We must redefine the semantic functions for *Expr*, and for all the ASTs that depend on it directly or indirectly. This means that we must redefine all ASTs, though they are syntactically identical (except because *Expr* has a new constructor) to their previous counterparts. While this is one annoying drawback of `AspectAG`, this is easily done with a couple of lines of Template Haskell as discussed in Chapter 3.

```
$ (closeNT Nt_Program)
```

Since we are using optimized records, we need to define one to store rules for expressions.

The following is given:

```
data SemExprPair var val op call pair
  = SemExprPair {gvar  :: var,
                 gval  :: val,
                 gop   :: op,
                 gcall :: call,
                 gpair :: pair}
```

Given an already defined aspect using *SemExpr* and a rule for pairs an extended record can be built in a simple way, for instance, as follows:

```
semExpr2semExprPair pair (SemExpr op var val call)
  = SemExprPair op var val call pair
```

We must also implement terminal values supporting tuples. The following is an implementation, including the features in L3:

```
data Op  = Add | Times | Div | Minus | Cons | Proj
           deriving (Eq, Read, Ord)
data Val = I Integer | R Double | List [Val] | Nil
           |  Variant String | Tuple [Val]
           deriving (Eq, Ord, Read)
```

To destruct tuples we use an operator *Proj* (for projection). Its second argument must be an integer literal.

6.9.2 T1 - Pretty-printing.

We must just give the semantics for the already defined attributes *spp* and *slastop* (just for the new syntax, of course!). We implement both at once:

```
pp_pair = const $
  (syndefM spp p_Pair (do
    lp ← at ch_Pair_l spp
    rp ← at ch_Pair_r spp
    return $ "(" ++ lp ++ ", " ++ rp ++ ")"
  ) ⋄ (syndefM slastop p_Pair (return Nothing))
```

6.9.3 T2 - Name binding.

Again, semantics are simple. Recall we had function names and variable names computed with the synthesized attributes *sfv* and *sffv*, and an environment *ifnames* with function names in scope. We combine all the rules on the fly:

```
nb_pair =
  const $   syndefM sfv  p_Pair ((++) <$> at ch_Pair_l sfv ⊛ at ch_Pair_r sfv)
          ⋄ syndefM sffv p_Pair ((++) <$> at ch_Pair_l sffv ⊛ at ch_Pair_r sffv)
          ⋄ copyAtChi ifnames ch_Pair_l
          ⋄ copyAtChi ifnames ch_Pair_r
```

6.9.4 T3 - Code generation.

To compile the new language to the Core we must encode pairs. Again we use Church encodings. In Figure 6.16 we showed the function *pair*. It takes two Core terms to build a Core term encoding the pair. Arguments can be recovered by applying the resulting term to *tt* or *ff* (the first and second component, respectively).

The rule to compute the attribute *scomp* (the resulting, compiled term) can be therefore given as follows:

```
scomp_pair = λ(Proxy :: Proxy v) →
  syndefM (scomp @ v) p_Pair (do
    l ← at ch_Pair_l (scomp @ v)
    r ← at ch_Pair_r (scomp @ v)
    return $ pair l r
  )
```

We must handle the operator case. In the case of destructing tuples we do not want to reduce to an operator application in the Core, but to use the encodings:

```
scomp_op = λ(Proxy :: Proxy Val) →
  synmodM (scomp @ Val) $p_{Op}$ (do
    op ← ter $ch_{Op_{op}}$
    l  ← at $ch_{Op_r}$ (scomp @ Val)
    r  ← at $ch_{Op_l}$ (scomp @ Val)
    case (op, r) of
      (Proj, CVal (I 1)) → return $ CApp l tt
      (Proj, CVal (I 2)) → return $ CApp l ff
      _                  → return $ COp l op r
  )
```

Note that this time we defined a monomorphic rule, fixing the variable *v* with *Val*. This allows us to pattern match on the constructors *Proj*[19] and *I*. However, in exchange we lost the ability to reuse the rule. If we want a polymorphic definition while performing the case analysis on projections, we must extend the interface for values, as we have done before. We leave it like that since no further terminal values will be defined in this document.

Note that *scomp_op* is defined using *synmodM* (i.e. modifying the previous definition of the attribute computation), this expression (of qualified name *MF.Tuple.Trans.scomp_op*) is intended to be combined with *MF.Expr.Trans.scomp_op*, but since we have defined semantics rule-wise, it would be valid to use *syndefM* and just replacing the old rule.

Finally, there is a last piece of semantics we must modify. Remember that those functions that take a pair as their argument are compiled curried. For that reason, when functions are applied to pairs we must subsequently apply the pair members, instead of applying just the encoded pair. How do we know if a function takes a pair? In the typed languages that we will implement later, it is easy: we just decide depending on its type. Here we do not have types available yet, but we can decide depending on the binders of the function equation. The issue is that we do not have them available in the production p_{Call}, so we must collect them. What we will do is to collect the information

[19] The constructor *Proj* has type *Op*. Note that due to the functional dependencies we used, once we fix *Val*, we also fix *Op*.

of the contents of the children *ch_Ecu_vars* in every equation of a function definition. Then, we distribute that information top-down.

To achieve this, we define two attributes:

```
sfpats :: Label ('Att "sfpats" (M.Map String [String]))
sfpats = Label

ifpats :: Label ('Att "ifpats" (M.Map String [String]))
ifpats = Label
```

To pass the information from the production p_{Ecu} to p_{FDef} we also define another attribute:

```
spat :: Label ('Att "pat" [String])
spat = Label
```

In Figure 6.17 we define the semantics. For the inherited attribute *ifpats* we use copy rules. We define two different aspects because we will need environments both when generating code from programs and from expressions.

Finally, we can define the rule *scomp_call*, as follows:

```
scomp_call = λ(Proxy :: Proxy v) →
  syndefM (scomp @ v) p_Call (do
    f      ← ter ch_Call_fun
    carg   ← at ch_Call_arg (scomp @ v)
    fpat   ← at lhs ifpats
    let pat = fromJust (M.lookup f fpat)
    return $ if length pat == 1
      then CApp (CVar f) carg
      else CApp (CApp (CVar f) (CApp carg tt)) (CApp carg ff)
  )
```

Whenever there is just one variable in the function equation we build an application. When there are two[20], we destruct the pair twice to generate a core term with two arguments.

Before going forward, let us show how we can use the semantic functions to build a proper traversal, since something somewhat technical emerges when using the semantics that we defined.

[20] Let us assume a well-formed tree. As we discussed before, this can be controlled by the parser. Otherwise, we could generate a core term that gets stuck.

$$spat\_ecu = const\ \$\ syndefM\ spat\ p_{Ecu}\ (ter\ ch\_Ecu\_vars)$$

$$\begin{aligned}
&sfpats\_fdef = const\ \$ \\
&\quad syndefM\ sfpats\ p_{FDef}\quad \begin{array}{ll} (& (\lambda a\ b \rightarrow M.singleton\ a\ b) \\ <\!\$\!> & ter\ ch\_FDef\_name \\ \circledast & at\ ch\_FDef\_ecu\ spat) \end{array}
\end{aligned}$$

$$\begin{aligned}
&sfpats\_programnil = const\ \$ \\
&\quad syndefM\ sfpats\ p_{ProgramNil}\ (return\ M.empty) \\
&sfpats\_programsnoc = const\ \$ \\
&\quad syndefM\ sfpats\ p_{ProgramSnoc}\quad \begin{array}{ll} (& M.union\ @\ String\ @\ [String] \\ <\!\$\!> & at\ ch_{ProgramSnoc_{init}}\ sfpats \\ \circledast & at\ ch_{ProgramSnoc_{last}}\ sfpats) \end{array}
\end{aligned}$$

$$\begin{aligned}
&att\_ifpats\_prog = program\_env\ ifpats \\
&att\_ifpats\_expr = expr\_env\ ifpats
\end{aligned}$$

Figure 6.17: Semantics to collect patterns in function equations.

Note that we built a synthesized attribute *sfpats* to latter pass down its contents as the inherited attribute *ifpats*. An AST for a program is built by many rewritings of the production $p_{ProgramSnoc}$. In an inner node *ifpats* is copied top-down, but in the root it should take the value of *sfpats*. The semantics to compute *ifpats* at an inner $p_{ProgramSnoc}$ production and at the root are different. There is no mechanism to decide whenever a node is in the root. We could solve this issue using more attributes to put global information about the AST in scope of the rule computation. However, we can do better, without defining more aspects. Remember that in the formal definition of a formal grammar given in Section 2.1, we defined grammars using a start symbol. While we lifted that requirement to define grammars in `AspectAG`, a start symbol could come handy to distinguish the root from inner nodes. This is a possible solution.

What we will do is conceptually the same, without defining an extra non-terminal and production. When defining a traversal over ASTs by applying a semantic function, we must define the initial inherited attributes, and we have the synthesized attributes of the root available. Nothing prevents us of defining them circularly.[21]

Given *asp_comp*, an aspect combining all the defined rules to compute *scomp*, *ifpats*, *sfpats*, and *fpats*, we can define the following function:

```
compileProgram p
  = let synatts = sem_Program asp_comp Proxy p inhatts
        inhatts = ifpats .=. (synatts #. sfpats) .*. emptyAtt
    in  synatts #. (spcomp @ Val)
```

Note that we use the synthesized attributes at the root, *synatts*, to extract *sfpats* and pass its value as the value of the inherited attribute *ifpats*.

6.9.5 T4 - Type checking.

We define a data type to represent types in this language:

```
data Ty =
  TyR | TyZ | Sec Ty | Enum (Maybe String) [String] | Cart [Ty] | Any
    deriving Eq
```

The new constructor is *Cart*, that represents cartesian products. We must provide an instance of (*Types Op Val Ty*).

Once done, semantics in the new production are straightforward, we define *stype_pair* building a cartesian product returning type:

```
stype_pair = λ(Proxy :: Proxy v) →
  syndefM (stype @ v) $p_{Pair}$ (do
    lt ← at $ch_{Pair_l}$ (stype @ v)
    rt ← at $ch_{Pair_r}$ (stype @ v)
    case (lt, rt) of
      (Right t, Right t′) → return $ Right $ Cart [t, t′]
      _                   → combineErr lt rt
  )
```

We also must pass down the environments *itypes* and *iftypes* at both children.

6.10 L5 - Simple types.

Finally, we add types to the syntax. In this language level we add the ability (and obligation) to write function types in function declarations.

[21] Thanks to lazy evaluation.

6.10.1 Syntax definition.

We extend function definitions by adding types in the corresponding production. In the definition of the context-free grammar using the EBNF notation, we change the production *FDef* as defined before, by the following:

$$\begin{aligned} FDef \rightarrow & f\ : \tau\ \rightarrow\ \tau' \\ & f\ Ecu \end{aligned}$$

with $\tau, \tau' \in \mathbb{T}$, where $\mathbb{T}$ denotes the terminal set of types (and $f \in \mathfrak{S}$ as before).

In the definition of the non-terminal *FDef* we use the same terminal f in the right hand side twice. Note that this definition is not context-free. The definition of context-free grammars (see Section 2.1.2) does not allow us to constraint the symbols in the right-hand side depening on each other. This is a typical case of context-dependence. Since MateFun syntax requires the same name in the function signature and then in the equation definition, it is a context-dependent language. There are two ways to handle this issue. One way is to define the language context-free and control that the two terminals are equal as part of the static semantics of the language (for instance in L2). The other way is to implement a context-dependent parser. This is very simple to do in this type of scenario by using monadic parsing. That is how the reference implementation solves the task.

So what we do here, is to simply assume the parser handles this and put the name once in the definition of our AST. We must just extend function definitions by adding two types in the production p_{FDef}, a domain and a codomain[22]:

$$\begin{aligned} ch_{FDef_{dom}} &= Label :: \forall\ ty.Label\ (\text{'}Chi\ \texttt{"FDef\_dom"}\ P_{FDef}\ (Terminal\ ty)) \\ ch_{FDef_{cod}} &= \ Label :: \forall\ ty.Label\ (\text{'}Chi\ \texttt{"FDef\_cod"}\ P_{FDef}\ (Terminal\ ty)) \end{aligned}$$

The reified data type can be writen as follows:

$$\textbf{data}\ FDef\ v = FDef\ String\ ((Val2Type\ v), (Val2Type\ v))\ (Ecu\ v)$$

Semantic functions for p_{FDef} and $p_{Program}$ must be redefined, but note that all the rules and aspects defined still work, even those defined for the old production p_{FDef}, because any referred child is still a child in the new definition.

[22] Of course we can use splices again in this context. We prefer in this case to add a new pair of labels and reusing the old ones in scope.

6.10.2 T1 - Pretty-printing.

The only thing we must do is to redefine semantics for the attribute *spp* for the production p_{FDef}:

```
spp_fdef = λ(Proxy :: Proxy v) →
  synmodM spp p_FDef (do
    fname ← ter ch_FDef_name
    dom   ← ter (ch_{FDef_dom} @ (Val2Type v))
    cod   ← ter (ch_{FDef_cod} @ (Val2Type v))
    ecu   ← at ch_FDef_ecu spp
    return $ fname ++ " :: " ++ show dom ++ " -> " ++ show cod ++ "\n"
          ++ fname ++ ecu
  )
```

6.10.3 T2 - Name binding, T3 - Code Generation

No big changes are needed. In the name binding step it can make sense to control if the arities of functions (when their domain is a cartesian product) match with the pattern performed in equations. It also makes sense to perform this check as a part of type checking, though. We consider the latter solution better to avoid making name binding dependent of the environment of function types, implemented as part of type checking.

Code generation is independent from type checking. Ideally, generated code from well-typed programs satisfies a set of properties. This is our intention, but defining and proving such properties is out of our scope.

6.10.4 T4 - Type checking.

We already implemented some type inference in previous iterations of T4. In this language level we must build the environments used to compute expression types, and control that the types inferred are consistent with the declared types.

Let us do a brief summary of what we have. Given an expression, or a guarded expression, the attribute (*stype* :: *Types op v t* ⇒ *Label* (*'Att* `"type"` (*Either String t*))) denotes its type. To compute *stype*, we used two environments: the type of all funcions in scope is passed down using an attribute (*iftypes* :: *Types op v t* ⇒

```
asp_sdecltypes = SemFuncST e e e e e e
  (λ(Proxy :: Proxy v) →
    syndefM (idecltypes @ v) $p_{FDef}$ (do
      fnam ← ter ch_FDef_name
      fdom ← ter ($ch_{FDef_{dom}}$ @ (Val2Type v))
      fcod  ← ter ($ch_{FDef_{cod}}$ @ (Val2Type v))
      return $ M.singleton fnam (fdom, fcod)
    )
    e
  (λ(Proxy :: Proxy v) →
    syndefM (idecltypes @ v) $p_{ProgramNil}$ (return M.empty))
  (λ(Proxy :: Proxy v) →
    syndefM (idecltypes @ v) $p_{ProgramSnoc}$ (do
      initTypes ← at $ch_{ProgramSnoc_{init}}$ (idecltypes @ v)
      lastType  ← at $ch_{ProgramSnoc_{last}}$ (idecltypes @ v)
      return $ M.union initTypes lastType))
  (SemExprPair e e e e e)
  where e = const emptyRule
```

Figure 6.18: Aspect to collect function environments.

Label ('*Att* "ftypes" (*M.Map String* (t, t)))); the types of all variables is passed down using the attribute (*itypes* :: *Types op v t* ⇒ *Label* ('*Att* "types" (*M.Map String t*))).

In this task we must build the two environments. The function environment *sftype* will be computed as we did with *sfpats* in Section 6.9.4, first collecting the information of all the defined functions to pass it down in the root of the AST. To collect the information in all functions we define the following aspect:

```
sdecltypes :: ∀ v op t.Types op v t ⇒
  Label ('Att "decltypes" (M.Map String (t, t)))
sdecltypes = Label
```

Figure 6.18 shows its declaration, we only need rules at p_{FDef}, $p_{ProgramNil}$ and $p_{ProgramSnoc}$. We defined all of them on the fly building a value of type *SemFuncST*, the optimized data structure for aspects in this language level.

The environment *stype* is easier to compute. At each equation in function declarations, knowing the declared type of the function and the pattern matching we perform, we build the environment with those variables.

From *stype* in guarded expressions we control if that computed type is a subset of the declared type of the function, wich is the neccesary and sufficient condition to consider it well-typed.

Then, we define an attribute to collect the information produced by the type checker in each function:

$swt = Label$ @ (*'Att* `"wt"` (*Either* [*String*] [*String*]))

In case of a type error the computed value of this attribute will be a *Left* constructor with a list of messages to print, we still keep a list of messages in well-typed cases, where values under the *Right* constructor are computed, to track information such as warnings.

The hard work is performed in the production p_{Ecu}. Figure 6.19 shows its implementation. It is the result of combining two rules, to compute *itypes* and *swt*.

To compute *itypes* we use *inhmodM*, so we can build an aspect to copy *itypes* at every production with a macro, an then combine the aspect with an aspect containing *swt_ecu*, to change the behaviour just at that production. What we do is pretty simple, we collect the current environments *itypes* (should be empty now, but perhaps we can add constants in the future, or local function definitions) and *iftypes*, the tuple of names in the pattern matching *vars*, and the function name in *ifnam*. The last attribute is defined, having type (*'Att* `"ifnam"` *String*) with simple semantics: passing it down at the production p_{FDef}. This is neccesary because the name of the function is not available at the production *swt_ecu*. With all that information we take the domain of the function (it should always be in scope, otherwise name binding should report an error) and extend the type of the variables in the pattern matching (many variables are allowed in the pattern matching only when the domain is a cartesian product). The calls to the functions *error* should never be performed: the function being checked should be in scope, and a mismatch in arities will imply a type error when defining *swt* without needing to use *itypes* (thanks to lazy evaluation).

To compute *swt* we also collect the values of all attributes we need. The new one we use is the type of the body, bound as *bodty*. If there is an error there, we just pass it up in *swt*. Otherwise, we control the computed type is a subset of the declared type. The function *handleType* does the job of computing *swt* from all the information available in scope.

```
swt_ecu = (λ(Proxy :: Proxy v) →
  inhmodM (itypes @ v) p_Ecu ch_Ecu_body (do
    types  ← at lhs (itypes @ v)
    gamma ← at lhs (iftypes @ v)
    vars   ← ter ch_Ecu_vars
    f      ← at lhs ifnam
      Just (dom :: Val2Type v, _) →
          if length vars == 1
          then return (M.insert (head vars) dom types)
          else case dom of
             Cart tys →
               if length vars == length tys
               then return (M.union types (M.fromList $ zip vars tys))
               else error "impossible: bug in swt"
      _   → error "impossible: bug in itypes/sdecltypes"
      )
  `extP`
          (λ(Proxy :: Proxy v) →
    syndefM swt p_Ecu (do
      gamma ← at lhs (iftypes @ v)
      bodty  ← at ch_Ecu_body (stype @ v)
      vars   ← ter ch_Ecu_vars
      f      ← at lhs ifnam
      let fty = M.lookup f gamma
      case bodty of
        Left err → return $ Left [err]
        Right _ → return $ handleType f vars gamma bodty
             )
```

Figure 6.19: Type checking semantics for function declarations.

It can be defined as follows:

```
handleType f vars gamma bodty =
  case M.lookup f gamma of
    Just (dom, cod) →
      case bodty of
        Left e → Left [e]
        Right ty → case vars of
          [x] → handleType' ty cod
          xs → case ty of
                Cart tys → if length xs == length tys
                  then handleType' ty cod
                  else Left ["arity mismatch"]
                _ → Left ["arity mismatch"]
      where handleType' ty cod = if isSubtype ty cod
        then Right ["well typed"]
        else Left ["function " ++ f ++ " is ill typed"]
      _ → error "impossible: bug in itypes/sdecltypes"
```

We also report an error in case of an arity mismatch. Note that the returning error messages can be much richer than just reporting that the function in discourse is ill-typed, using all the information in scope.

6.11 L6 - Type definitions and refinements.

Finally, in this language level we add the ability to declare new types using refinements over base types, and enumerations.

For instance, the following are valid MateFun declarations that should be represented within the new syntax:

```
set N    = { x in Z | x >= 0 }
set Rno0 = { x in R | x /= 0 }
set Day  = { Mon, Tue, Wed, Thu, Fri, Sat, Sun }
set Rno01 = { x in R | (x /= 0, x/= 1) }
```

6.11.1 Syntax definition.

Now, programs are defined as a list of function and set definitions. For that reason, we drop the non-terminal nt_{FDef} and introduce a more general one,

$$
\begin{aligned}
\mathit{Program} \to & [\mathit{Decl}] \\
\mathit{Decl} \to & \mathit{FDef} \\
\mathit{Decl} \to & \mathit{SDef} \\
\mathit{FDef} \to & f \ : \tau \ \to \ \tau' \\
& f \ \mathit{Ecu} \ (\mathit{where} \ f \in \mathfrak{S}, \tau, \tau' \in \mathbb{T}) \\
\mathit{SDef} \to & \ \textbf{set} \ s \ = \{x \in \tau \mid \mathit{Cond}\} \ (\mathit{where} \ x_i \in \mathfrak{S}, \tau \in \mathbb{T}) \\
\mathit{Ecu} \to & (x_1, \ldots, x_n) = \mathit{ExprG}, x_i \in \mathfrak{S} \\
\mathit{ExprG} \to & \mathit{Expr} \\
\mathit{ExprG} \to & \mathit{Expr} \ \textbf{if} \ \mathit{Cond} \ \textbf{or} \ \mathit{ExprG} \\
\mathit{Cond} \to & \mathit{Expr} \triangle \mathit{Expr} \ (\mathit{where} \ \triangle \in \{<, >, =\}) \\
\mathit{Cond} \to & \top \\
\mathit{Cond} \to & \mathit{Cond} \wedge \mathit{Cond} \\
\mathit{Cond} \to & \neg \mathit{Cond} \\
\mathit{Expr} \to & v \ (\text{where } v \in \mathbb{V}) \\
\mathit{Expr} \to & x \ (\text{where } x \in \mathfrak{S}) \\
\mathit{Expr} \to & \mathit{Expr} \ \Box \ \mathit{Expr} \\
\mathit{Expr} \to & f(\mathit{Expr}) \ (\text{where } f \in \mathfrak{S}) \\
\mathit{Expr} \to & (\mathit{Expr}, \mathit{Expr})
\end{aligned}
$$

Figure 6.20: Full syntax definition of L6.

nt_{Decl}, for declarations. Declarations can be function declarations (the production nt_{FDef}), or set declarations (the production nt_{SDef}). Note that nt_{FDef} must be actually redefined since now it has a different type, because we changed the non-terminal to which it belongs. Note that we could push modularity one step forward reusing the old definition by making it polymorphic if we want to. Figure 6.20 shows the full syntax of the language.

The new substantial production is $SDef$. A set has a name, a base type, and a condition that refines it. For the condition we use a binder (the x). In the concrete syntax we need a way to define sets as an enumeration, but as we will see soon this is enough for an AST.

We already used binders in equations. From our perspective as programmers, we felt comfortable by using a similar pattern. So we defined the syntax for *SDef* separating the refinement, as follows:

```
$ (addProd "SDef" "Nt_Decl  [("name", Ter "String),
                              ("base", Poly),
                              ("ecuS", NonTer "Nt_EcuS)])
$ (addNont "EcuS")
$ (addProd "EcuS" "Nt_EcuS [("vars", Ter "Vars),
                             ("cond", NonTer "Nt_Cond)])
```

We use two productions p_{SDef} end p_{EcuS}. In p_{SDef} there is a name, a base type and an equation. Equations have a list of binders and a condition. MateFun allows only one variable binder in sets refinemtnes, for instance the following definition is illegal:

```
set NotEq = { (x,y) in Z X Z | x /= y }
```

though the same set could be implemented as follows:

```
set NotEq = { p in Z X Z | p ! 1 /= p ! 2 }
```

We implement the reference behaviour, yet we consider a list of variables in the AST keeping future extensions in mind. In the evaluator we will assume that the length of this list is 1.

The new definitions can be reified to the following data types:

```
data Decl v = FDef String ((Val2Type v), (Val2Type v)) (Ecu v)
            | SDef String (Val2Type v) (EcuS v)


data EcuS v = EcuS Vars (Cond v)
```

For instance, the set Rno01 can be represented with the following AST (using the already defined data types *Val* and *Ty*):

```
rno01 =
  SDef "RNo01" TyR (EcuS ["x"] (Not (Comp (Var "x") EQ (Val (I 0)))
                                `And`
                                Not (Comp (Var "x") EQ (Val (I 0)))))
```

Sets defined by extension can be also defined using this syntax. Remember that the type *Ty* had a constructor for enumerations. The following is its representation in the AST for the set Day:

```
day = SDef "Day" (Enum (Just "Day")
                 ["Mon", "Tue", "Wed", "Thu", "Fri", "Sat", "Sun"])
                 (EcuS [""] Top)
```

Note that there is no refinement for this type. We used the trivial *Top* condition. We use the empty string as a fake binder, though we never use it at all (we could just put the empty list there). Where we can be worried for this representation is in the pretty-printer (we must discern enumerations from refinements to print them correctly), but this is easy: definitions with an enumeration as its base type and a *Top* condition are enumeration definitions.

When writing types in the signature of functions, to refer to refined types we use their names. To know wether a type of a function is a refinement or not, we implement an interface for refinement types as follows:

```
class Ref t where
  isRef :: t → Bool
  tName :: t → String
```

The predicate *isRef* returns if the type is a refinement, and *tName* returns its name.

We use the same definitions of *Val*, *Op* and *Type* that we had in the implementation of this language level. The type class *Ref* is implemented as follows:

```
instance Ref Ty where
  isRef (DeclTy _)  = True
  isRef _           = False
  tName (DeclTy n) = n
  tName _           = ""
```

6.11.2 T1 - Pretty-printing and T2 - Name binding.

For the pretty-printer we just implement semantics for the attribute *spp*. The only thing to decide is wether we print an enumeration set with the extension syntax, or as a comprehension. This is trivially performed looking up at the condition. As we discussed before.

Name binding requires also to extend the already defined attributes (because, for instance, set definitions have a condition that could have occurrences of free variables).

Note that we can also define refinement types referring already defined refinements. We must contol that the referred sets are in scope.

6.11.3 T3 - Code generation and T4 - Type Checking.

Since this language level mixes dynamic typing and static typing, part of the type checking is performed in runtime, so the complexity is handled in the code generation.

In the static type checking we do not need to add any behaviour, except for propagating the result of the type checker (the synthesized attribute *swt*) upwards in the set definitions, since static errors can occur in the refinement conditions. So what we implemented for T4 extends trivially to this new language level.

The code generation phase does not change in most scenarios, but the dynamic type checking must be implemented. When a function is called its domain is a refinement type, we check if the argument fulfills the refinement condition. If the codomain is a refinement type we check that the returned value fulfills its type condition. In Figure 6.21 we show the rule that implements this behaviour. Remember that the generation of code for functions was already performed at the production p_{Ecu} in previous language levels.

We do the same as in previous language levels at this production, but in the generated Core term (a lambda abstraction) we preprocess the argument (the computed result is *domchecked*) and then postprocess the result (the computed result is *codchecked*). When the types are not refinements those processings are just an identity. When the types are refinements, we check the corresponding refinement predicates. Remember that conditions reduced to a Church boolean, so the result of the evaluated condition is is applied to two arguments, the first is the “good” term, and the second an error.

For instance, the function *mkDomError* is defined as follows:

```
mkDomError fnam cod
  = CError $ "domain error: arg of function ‘"
    ++ fnam
    ++ "’ does not satisfy condition of its domain set "
```

Of course, the produced error message can be improved a lot, for instance by printing the condition (by using the *spp* attribute!).

```
scomp_ecu = λ(Proxy :: Proxy v) →
  syndefM (scomp @ v) p_Ecu (do
    vars  ← ter ch_Ecu_vars
    refty ← at lhs (ityenv @ v)
    fnam ← at lhs ifnam
    (dom :: Val2Type v,
      cod :: Val2Type v) ← at lhs (icurrtype @ v)
    body  ← at ch_Ecu_body (scomp @ v)
    let domchecked = domcheck fnam (head vars) dom refty body
    let codchecked = (codcheck fnam cod refty domchecked)
    return (foldr CLam codchecked vars)
  )
  where domcheck fnam var dom refty body
              = if isRef dom
                then ap3 ite
                  (CApp (snd $ fromJust (M.lookup (tName dom) refty))
                  (CVar var)) body (mkDomError fnam dom)
                else body
          codcheck fnam cod refty body
              = if    isRef cod
                then (ap3 ite
                  (CApp (snd $ fromJust (M.lookup (tName cod) refty)) body)
                    body (mkCodError fnam cod))
                else body
```

Figure 6.21: Semantics for dynamic type checking.

Example 7. *Consider the following MateFun program:*

```
set Rno0 = {x in R | x /= 0}
set Rno1 = {x in R | x /= 1}

f :: Rno0 -> Rno1
f (x) = 1 / x + 1
```

then `f` *compiles into the following Core term:*

$$\lambda x.ite\ (x \not\equiv 0)\ (ite\ ((1\ /\ x + 1) \not\equiv 1)\ (1\ /\ x + 1)\ coderr)\ (domerr)$$

where (ite) *is the conditional defined in Figure 6.16 and* $(domerr)$, and $(coderr)$ *are error terms.*

6.12 Conclusions

In this chapter, we have shown how to use the `AspectAG` library to implement a middle-sized compiler. We built the compiler front end and implemented an interpreter for the Core language, an intermediate representation we compile MateFun into.

We have shown how modular the system is. We started with a small subset of the final language and extended it both with new syntax and semantics without recompiling code. Parametrization over terminals proved to be particularly useful. Note that while we did not get to the reference implementation because, for instance, figures are missing, they can be trivially added by defining a new set of terminals. The issue we find with the parametrization approach is that each time we add a feature to a terminal set, we must define a new type class if we want to use keep the rules polymorphic. When defining a rule, if we pattern match over a terminal data type, the rule is no longer reusable. Moreover, each new terminal implementation must implement all previous interfaces, which can get tedious.

In an early iteration, we used the non-optimized approach, defining aspects with extensible records. We identified the performance issues there. In the presented implementation we switched to the optimization defined in Section 5.3.9 for all the languages except for the fixed Core that is still manageable.

While in this document we presented grammar labels and semantic functions defined by splices, in the codebase we initially defined labels by hand (and then switched to splices, testing the correctness of the code generation). The experience suggests that there are many opportunities to make mistakes when building the semantic functions by hand, especially thinking in the case of a beginner user. Building the boilerplate with metaprogramming ensures its consistency (at least if the code in the module *Language.Grammars.AspectAG.TH* is correct).

During the development we made some of the mistakes defined in Chapter 4, and we got good messages that helped us to debug the AG definitions we were working on. However, we also got long error messages with implementation leaks in some other scenarios. The most common was when using a free variable instead of an aspect or AG label because GHC prints the expected type.

Overall, we think the results are good. Both the library and its pragmatics are usable and allow us to extend our languages syntactically and semantically without recompiling the already defined code.

Chapter 7

Conclusions and future work.

In this thesis we introduced a reworked version of AspectAG, using modern techniques of type-level programming. We emphasized in having a strongly-typed system both at the level of terms, and at the level of types, and in DSL-oriented error reporting. Along the way we developed data structures and idioms that could be applied in more general contexts than just AspectAG, and we condensed them in the libraries requirements and poly-rec.

Then, we developed a case study: a modular implementation of the Mate-Fun programming language. We have shown how a language can be extended in a modular way by using AspectAG, tackling the expression problem.

In addition to these tangible results, we have shown a non-trivial example of type-level programming in Haskell, and how by using these techniques we can push the boundaries of what can be expressed within the host language itself.

We are satisfied with the obtained results. However, there are further interesting problems to solve, and rough edges in what we implemented, some of them related to our approach, and some of them related to the techniques themselves.

In chapters 3 and 6 we showed evidence of how AspectAG tackles the expression problem in practice by implementing modular languages. However, we always use semantic functions that traverse a reification of the defined grammar in form of a data type. While by using Template Haskell we relieve the users of rewriting that boilerplate code, behind the scenes both the semantic functions and the data types representing ASTs are rewriten and compiled. This is not the most ambitious solution to the problem, abeit we are still achieving that the code already written is untouched.

We would like to actually reuse the reified data types, avoiding rewriting them. There are approaches to encode extensible data types, such as *Data types à la carte* [Swierstra, 2008], or *Trees that grow* [Najd and Jones, 2017]. We were reluctant in integrating them in the solution since they require even more noisy types. Finding a way to do it harmoniously is an interesting future work.

Reusing data types is just half of the task, we would like to reuse the definition of the semantic functions. At a first glance it seems intuitive to abstract the semantic functions using an ad-hoc polymorphic function in a type class. The issue that arises is that in Haskell we need to write explicitly the class constraints in each instance definition. The type of the semantic functions have many constraints defining the shape that the rules must have to be valid. Figure 7.1 shows the type of the semantic function for the expression language defined in Section 3.1. By defining semantic functions by pattern matching as we have done in this document (whether by hand or using Template Haskell splices) GHC infers the constraints implicitly, freeing users from writing them.

As we discussed in Section 3.6, we can use a shallow embedding, representing ASTs directly with semantic function applications, but at the same time, this approach complicates the possibility of representing ASTs independently from a concrete definition of semantics. Again, the natural solution, encoding the syntax with type classes and concrete semantics by their instances (resembling the *final-tagless encoding* referred in the literature [Carette et al., 2007]) forces users to write constraints explicitly. However, those constraints should be easy to compute using type families, so we could find mechanisms to compute them.

Perhaps a more clever solution is to attack the problem from its root. Semantic functions have a constrained type because of our approach, where semantic functions are fixing the grammars. In `AspectAG` children depend on productions, and productions depend on non-terminals, but no dependencies are written the other way around. While this makes it easy to add new labels to the structure it delegates the work of defining the grammar structures to the semantic functions (by the *knit* function). A different approach would be to encode the full structure of a grammar by using data kinds. We could even compute the reified ASTs and the semantic functions from that structure. Making that explicit would require handling it, though, moving away

```
sem_Expr
  :: (Require (OpLookup PrdReco P_Add r)
        [Text "knit:(Add of Non-Terminal Expr)"],
    Require (OpLookup PrdReco P_Val r)
        [Text "knit:(Val of Non-Terminal Expr)"],
    Require (OpLookup PrdReco P_Var r)
        [Text "knit:(Var of Non-Terminal Expr)"],
    ReqR (OpLookup PrdReco P_Add r)
    ~ CRule
      []
      prd3
      [(Chi "Add_l" P_Add (Left (NT "Expr")), sp),
        (Chi "Add_r" P_Add (Left (NT "Expr")), sp)]
      ip
      [(Chi "Add_l" P_Add (Left (NT "Expr")), []),
        (Chi "Add_r" P_Add (Left (NT "Expr")), [])]
      []
      [(Chi "Add_l" P_Add (Left (NT "Expr")), ip),
        (Chi "Add_r" P_Add (Left (NT "Expr")), ip)]
      sp,
    ReqR (OpLookup PrdReco P_Var r)
    ~ CRule
      []
      prd2
      [(Chi "Var_var" P_Var (Right (T [Char])),
        [(Att "term" String, String)])]
      ip
      [(Chi "Var_var" P_Var (Right (T [Char])), [])]
      []
      [(Chi "Var_var" P_Var (Right (T [Char])), [])]
      sp,
    ReqR (OpLookup PrdReco P_Val r)
    ~ CRule
      []
      prd4
      [(Chi "Val_val" P_Val (Right (T Integer)),
        [(Att "term" Integer, Integer)])]
      ip
      [(Chi "Val_val" P_Val (Right (T Integer)), [])]
      []
      [(Chi "Val_val" P_Val (Right (T Integer)), [])]
      sp) ⇒
    CAspect [] r → Expr → Attribution ip → Attribution sp
```

Figure 7.1: Type of a semantic function.

from the design of `AspectAG`. Another reason to explore this approach is to have a tangible definition of a grammar to manipulate. While in `AspectAG` we achieved first-class attribute grammars in the sense that each ingredient of a grammar is a first-class citizen in the host language, we lack a way of manipulating grammars in a monolithic way. For example, an honest question we can ask ourselves is: what is the type of a grammar in `AspectAG`?

A future work already outlined in the original implementation of `AspectAG` [Viera et al., 2009] is to implement a similar system in a dependently-typed language. Again this would require explicit type annotations which can complicate the job.

Note that in all the previously discussed points appears in one way or another the fact that GHC infers types that are difficult to write for programmers. While by using type-level programming techniques we are pushing the boundaries of the Hindley-Milner type system to the land of dependent types, we keep the constraint solver working for us. This was already observed in previous works such as [Lindley and McBride, 2013]. For this reason we claim that we are not just mimmicking dependent types, but exploring a typing discipline which is worth on itself.

As we have shown in Chapter 4, we have achieved nice error messages to a set of domain-related type errors. This worked well in practice when those structural errors were made. However, as mentioned in Section 2.4 there are two main issues with DSL type errors: precise error reporting in domain-specific logic and implementation leaking. The latter one is still an issue. A common error we have seen during our developments is to use a free variable as an aspect (for instance, because we mispelled its name or we forgot to include a module). While this seems innocent, and the reported error will state clearly that there is a variable not in scope, the expected type of the variable is displayed. A look at Figure 7.1 should be enough to understand why this is an issue. With languages with dozens of productions it is much worse.

The proper type of the function *sem_Expr* of Figure 7.1 is problematic. Interactive development using the REPL and querying types is a common workflow in Haskell. Users could want to query the types in scope when developing programs. The type of *sem_Expr* leaks details and even the context information in the *Require* constraints.

A feature we desire to tackle this issue is to have a way of implementing opaque types. This can be given with compiler support, or perhaps encoding them.

As we discussed before, generating code with splices is a partial solution to avoid code duplication in the definition of data types and semantic functions. Apart from solving that issue in a better way interest us, there are more reasons to avoid metaprogramming. In general, template metaprogramming using Template Haskell has been considered "controversial" by some users[1]. For what concerns us, avoiding metaprogramming means we are actually testing the power of the type system. It is interesting theoretically to know to what extent it is possible to encode a full AG system as we did. Metaprogramming allows us to write a full AG system by using splices. The best way to set the limit between the EDSL encoded using an expressive type system and a theoretical uninteresting metaprogramming artifact is avoiding all splices at all. Unfortunately, a big issue arises: without metaprogramming, the consistency between the AG labels, the reified datatypes and the semantic functions depends on the correctness of users definitions.

Finally, in Section 5.3.9 we showed an optimization to solve the performance issues arised by the use of big aspects. We would want to write the data structures to avoid falling into the compile-time leak without defining dedicated data structures for aspects in each grammar version. The solution we implemented worked well for us as "advanced" users of `AspectAG`, but in general we should work in a solution with less responsibilities for final users. Overloading the interfaces to make aspects and using metaprogramming to generate the non-extensible records is a tempting solution, but it goes against the other proposed lines of future work we have outlined.

[1] See, for instance the following discussion:
`https://stackoverflow.com/questions/10857030/whats-so-bad-about-template-haskell`.

Bibliography

[Balestrieri, 2015] Balestrieri, F. (2015). *The productivity of polymorphic stream equations and the composition of circular traversals.* PhD thesis, University of Nottingham, UK.

[Barendregt, 1985] Barendregt, H. P. (1985). *The lambda calculus - its syntax and semantics*, volume 103 of *Studies in logic and the foundations of mathematics.* North-Holland.

[Bertot and Castéran, 2013] Bertot, Y. and Castéran, P. (2013). *Interactive theorem proving and program development: Coq'Art: the calculus of inductive constructions.* Springer Science & Business Media.

[Biere et al., 2009] Biere, A., Heule, M., van Maaren, H., and Walsh, T., editors (2009). *Handbook of Satisfiability*, volume 185 of *Frontiers in Artificial Intelligence and Applications.* IOS Press.

[Bird, 1984] Bird, R. (1984). Using circular programs to eliminate multiple traversals of data. Acta Informática. 21:239–250.

[Bove et al., 2009] Bove, A., Dybjer, P., and Norell, U. (2009). A brief overview of Agda – a functional language with dependent types. In Berghofer, S., Nipkow, T., Urban, C., and Wenzel, M., editors, *Theorem Proving in Higher Order Logics*, pages 73–78, Berlin, Heidelberg. Springer Berlin Heidelberg.

[Brady, 2013] Brady, E. C. (2013). Idris, a general-purpose dependently typed programming language: Design and implementation. *Journal of Functional Programming*, 23:552 – 593.

[Carboni et al., 2018] Carboni, A., Koleszar, V., Tejera, G., Viera, M., and Wagner, J. (2018). MateFun: Functional programming and math with adolescents. In *2018 XLIV Latin American Computer Conference (CLEI)*, pages 849–858.

[Carette et al., 2007] Carette, J., Kiselyov, O., and Shan, C. (2007). Finally tagless, partially evaluated. In *Programming Languages and Systems, 5th Asian Symposium, APLAS 2007, Singapore, November 29-December 1, 2007, Proceedings*, pages 222–238.

[Chakravarty et al., 2005a] Chakravarty, M. M. T., Keller, G., and Jones, S. P. (2005a). Associated type synonyms. *SIGPLAN Not.*, 40(9):241–253.

[Chakravarty et al., 2005b] Chakravarty, M. M. T., Keller, G., Jones, S. P., and Marlow, S. (2005b). Associated types with class. *SIGPLAN Not.*, 40(1):1–13.

[da Rosa et al., 2020a] da Rosa, S., Viera, M., and García-Garland, J. (2020a). A case of teaching practice founded on a theoretical model. In *Informatics in Schools. Engaging Learners in Computational Thinking - 13th International Conference, ISSEP 2020, Tallinn, Estonia, November 16-18, 2020, Proceedings*, pages 146–157.

[da Rosa et al., 2020b] da Rosa, S., Viera, M., and García-Garland, J. (2020b). Mathematics and MateFun, a natural way to introduce programming into school. Technical report, UdelaR, FI.

[da Rosa et al., 2021] da Rosa, S., Viera, M., and García-Garland, J. (2021). Training teachers in informatics: a central problem in science education. In *Proceedings of The 50th SADIO Conference, Simposio Argentino de Educación en Informática.*

[Danvy and Nielsen, 2001] Danvy, O. and Nielsen, L. R. (2001). Defunctionalization at work. In *Proceedings of the 3rd ACM SIGPLAN International Conference on Principles and Practice of Declarative Programming*, PPDP '01, page 162–174, New York, NY, USA. Association for Computing Machinery.

[de Bruijn, 1972] de Bruijn, N. (1972). Lambda calculus notation with nameless dummies, a tool for automatic formula manipulation, with application to the church-rosser theorem. *Indagationes Mathematicae (Proceedings)*, 75(5):381–392.

[de Moor et al., 2000] de Moor, O., Backhouse, K., and Swierstra, S. D. (2000). First-class attribute grammars. *Informatica (Slovenia)*, 24(3).

[de Moor et al., 1999] de Moor, O., Peyton Jones, S. L., and Wyk, E. V. (1999). Aspect-oriented compilers. In *Generative and Component-Based Software Engineering, First International Symposium, GCSE'99, Erfurt, Germany, September 28-30, 1999, Revised Papers*, pages 121–133.

[de Moura and Bjørner, 2008] de Moura, L. and Bjørner, N. (2008). Z3: An Efficient SMT Solver. In Ramakrishnan, C. R. and Rehof, J., editors, *Tools and Algorithms for the Construction and Analysis of Systems*, pages 337–340, Berlin, Heidelberg. Springer Berlin Heidelberg.

[Eisenberg and Stolarek, 2014] Eisenberg, R. A. and Stolarek, J. (2014). Promoting Functions to Type Families in Haskell. In *Proceedings of the 2014 ACM SIGPLAN Symposium on Haskell*, Haskell '14, page 95–106, New York, NY, USA. Association for Computing Machinery.

[Eisenberg et al., 2014] Eisenberg, R. A., Vytiniotis, D., Peyton Jones, S., and Weirich, S. (2014). Closed type families with overlapping equations. *SIGPLAN Not.*, 49(1):671–683.

[Ekman and Hedin, 2007] Ekman, T. and Hedin, G. (2007). The JastAdd system - modular extensible compiler construction. *Sci. Comput. Program.*, 69(1-3):14–26.

[Fowler, 2010] Fowler, M. (2010). *Domain Specific Languages*. Addison-Wesley Professional, 1st edition.

[Freeman and Pfenning, 1991] Freeman, T. and Pfenning, F. (1991). Refinement Types for ML. In *Proceedings of the ACM SIGPLAN 1991 Conference on Programming Language Design and Implementation*, PLDI '91, page 268–277, New York, NY, USA. Association for Computing Machinery.

[Gamma et al., 1994] Gamma, E., Helm, R., Johnson, R., and Vlissides, J. M. (1994). *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional, 1st edition.

[García-Garland et al., 2019] García-Garland, J., Pardo, A., and Viera, M. (2019). Attribute grammars fly first-class... safer!: dealing with DSL errors in type-level programming. In Stutterheim, J. and Chin, W., editors, *IFL '19: Implementation and Application of Functional Languages, Singapore, September 25-27, 2019*, pages 10:1–10:12. ACM.

[Gaster and Jones, 1996] Gaster, B. R. and Jones, M. P. (1996). A polymorphic type system for extensible records and variants. Technical Report NOTTCS-TR-96-3, University of Nottingham.

[Gibbons, 2003] Gibbons, J. (2003). Origami programming. In Gibbons, J. and de Moor, O., editors, *The Fun of Programming*, Cornerstones in Computing, pages 41–60. Palgrave.

[Gibbons, 2013] Gibbons, J. (2013). Functional Programming for Domain-Specific Languages. In *Central European Functional Programming School - 5th Summer School, CEFP 2013, Cluj-Napoca, Romania, July 8-20, 2013, Revised Selected Papers*, pages 1–28.

[Henry, 2020] Henry, S. (2020). `haskus-utils-variant`. https://hackage.haskell.org/package/haskus-utils-variant-3.1.

[Hudak, 1998] Hudak, P. (1998). Modular domain specific languages and tools. In *Proceedings of the Fifth International Conference on Software Reuse, ICSR 1998, Victoria, BC, Canada, June 2-5, 1998*, pages 134–142.

[Hughes, 1989] Hughes, J. (1989). Why Functional Programming Matters. *Computer Journal*, 32(2):98–107.

[Jansen, 2013] Jansen, J. M. (2013). Programming in the λ-calculus: From Church to Scott and Back. In Achten, P. and Koopman, P. W. M., editors, *The Beauty of Functional Code - Essays Dedicated to Rinus Plasmeijer on the Occasion of His 61st Birthday*, volume 8106 of *Lecture Notes in Computer Science*, pages 168–180. Springer.

[Jones, 1995] Jones, M. P. (1995). Functional programming with overloading and higher-order polymorphism. In *Advanced Functional Programming, First International Spring School on Advanced Functional Programming Techniques-Tutorial Text*, page 97–136, Berlin, Heidelberg. Springer-Verlag.

[Jones, 2000] Jones, M. P. (2000). Type classes with functional dependencies. In *ESOP*, volume 1782 of *Lecture Notes in Computer Science*, pages 230–244. Springer.

[Jones, 2002] Jones, S. P., editor (2002). *Haskell 98 Language and Libraries: The Revised Report*. http://haskell.org/.

[Kiselyov et al., 2004] Kiselyov, O., Lämmel, R., and Schupke, K. (2004). Strongly typed heterogeneous collections. In *Proceedings of the 2004 ACM SIGPLAN Workshop on Haskell*, Haskell '04, pages 96–107, New York, NY, USA. ACM.

[Kiss et al., 2019] Kiss, C., Eisenbach, S., Field, T., and Peyton Jones, S. (2019). Higher-order type-level programming in Haskell. In *International Conference on Functional Programming (ICFP'19)*. ACM.

[Kmett, 2009] Kmett, E. (2009). Haskell 98 phantom types to avoid unsafely passing dummy arguments. http://hackage.haskell.org/package/tagged.

[Knuth, 1968] Knuth, D. E. (1968). Semantics of context-free languages. In *In Mathematical Systems Theory*, pages 127–145.

[Knuth, 1984] Knuth, D. E. (1984). Literate Programming. *The Computer Journal*, 27(2):97–111.

[Knuth, 1990] Knuth, D. E. (1990). The genesis of attribute grammars. In *Attribute Grammars and their Applications*, pages 1–12. Springer.

[Leijen, 2004] Leijen, D. (2004). First-class labels for extensible rows. Technical Report UU-CS-2004-51. Utretch University.

[Leijen, 2005] Leijen, D. (2005). Extensible records with scoped labels. In *Proceedings of the 2005 Symposium on Trends in Functional Programming (TFP'05), Tallin, Estonia.*

[Lindley and McBride, 2013] Lindley, S. and McBride, C. (2013). Hasochism: The pleasure and pain of dependently typed haskell programming. *SIGPLAN Not.*, 48(12):81–92.

[Martins et al., 2013] Martins, P., Fernandes, J. P., and Saraiva, J. (2013). Zipper-based attribute grammars and their extensions. In Bois, A. R. D. and Trinder, P., editors, *Programming Languages - 17th Brazilian Symposium, SBLP 2013, Brasília, Brazil, October 3 - 4, 2013. Proceedings*, volume 8129 of *Lecture Notes in Computer Science*, pages 135–149. Springer.

[McBride, 2002] McBride, C. (2002). Faking it: Simulating dependent types in Haskell. *J. Funct. Program.*, 12:375–392.

[Mernik and Žumer, 2005] Mernik, M. and Žumer, V. (2005). Incremental programming language development. *Computer languages, Systems and Structures*, 31:1–16.

[Najd and Jones, 2017] Najd, S. and Jones, S. P. (2017). Trees that grow. *J. Univers. Comput. Sci.*, 23(1):42–62.

[Paakki, 1995] Paakki, J. (1995). Attribute grammar paradigms—a high-level methodology in language implementation. *ACM Comput. Surv.*, 27(2):196–255.

[Peyton Jones et al., 1997] Peyton Jones, S., Jones, M., and Meijer, E. (1997). Type classes: an exploration of the design space. In *Haskell workshop*.

[Pfenning and Elliott, 1988] Pfenning, F. and Elliott, C. (1988). Higher-Order Abstract Syntax. *SIGPLAN Not.*, 23(7):199–208.

[Pickering et al., 2016] Pickering, M., Érdi, G., Peyton Jones, S., and Eisenberg, R. A. (2016). Pattern synonyms. In *Proceedings of the 9th International Symposium on Haskell*, Haskell 2016, page 80–91, New York, NY, USA. Association for Computing Machinery.

[PureScript developers, 2017] PureScript developers (2017). Purescript language site. https://www.purescript.org.

[Ramsey, 1998] Ramsey, N. (1998). Unparsing expressions with prefix and postfix operators. *Softw. Pract. Exper.*, 28(12):1327–1356.

[Reynolds, 1972] Reynolds, J. C. (1972). Definitional interpreters for higher-order programming languages. In *Proceedings of the ACM Annual Conference - Volume 2*, ACM '72, page 717–740, New York, NY, USA. Association for Computing Machinery.

[Reynolds, 1975] Reynolds, J. C. (1975). User-defined types and procedural data structures as complementary approaches to data abstraction. In Schuman, S. A., editor, *New Directions in Algorithmic Languages 1975*, pages 157–168, Rocquencourt, France. IFIP Working Group 2.1 on Algol, INRIA.

[Reynolds, 1978] Reynolds, J. C. (1978). *User-Defined Types and Procedural Data Structures as Complementary Approaches to Data Abstraction*, pages 309–317. Springer New York, New York, NY.

[Saraiva, 2002] Saraiva, J. (2002). Component-based Programming for Higher-Order Attribute Grammars. In Batory, D., Consel, C., and Taha, W., editors, *Proceedings of the ACM SIGPLAN/SIGSOFT Conference on Generative Programming and Component Engineering, GCSE 2002, Held as Part of the Confederation of Conferences on Principles, Logics, and Implementations of High-Level Programming Languages, PLI 2002, Pittsburgh, PA, USA, October 3-8, 2002*, volume 2487 of *LNCS*, pages 268–282.

[Sheard and Jones, 2002] Sheard, T. and Jones, S. P. (2002). Template Meta-programming for Haskell. *SIGPLAN Not.*, 37(12):60–75.

[Silva and Visser, 2006] Silva, A. and Visser, J. (2006). Strong types for relational databases. In *Proceedings of the 2006 ACM SIGPLAN Workshop on Haskell*, Haskell '06, pages 25–36, New York, NY, USA. ACM.

[Sloane et al., 2009] Sloane, A. M., Kats, L. C. L., and Visser, E. (2009). A pure object-oriented embedding of attribute grammars. In *of the Ninth Workshop on Language Descriptions, Tools, and Applications*.

[Sterling, 2012] Sterling, J. (2012). vinyl: Extensible Records. http://hackage.haskell.org/package/vinyl.

[Swierstra et al., 1999] Swierstra, S. D., Azero Alcocer, P. R., and Saraiva, J. A. (1999). Designing and implementing combinator languages. In Swierstra, S. D., Henriques, P., and Oliveira, J., editors, *Advanced Functional Programming, Third International School, AFP'98*, volume 1608 of *LNCS*, pages 150–206. Springer-Verlag.

[Swierstra, 2008] Swierstra, W. (2008). Data types à la carte. *J. Funct. Program.*, 18(4):423–436.

[The GHC team, 2020] The GHC team (2020). GHC Manual. https://downloads.haskell.org/ghc/latest/docs/html/users_guide/. Acceded: 5-4-2022.

[van der Ploeg, 2013] van der Ploeg, A. (2013). CTRex: Open records using closed type families. http://hackage.haskell.org/package/CTRex.

[Van Wyk et al., 2010] Van Wyk, E., Bodin, D., Gao, J., and Krishnan, L. (2010). Silver: an extensible attribute grammar system. *Science of Computer Programming*, 75(1–2):39–54.

[Viera et al., 2018] Viera, M., Balestrieri, F., and Pardo, A. (2018). A Staged Embedding of Attribute Grammars in Haskell. In *Proceedings of the 30th Symposium on Implementation and Application of Functional Languages, IFL 2018, Lowell, MA, USA, September 5-7, 2018.*, pages 95–106.

[Viera et al., 2012] Viera, M., Swierstra, D., and Middelkoop, A. (2012). UUAG Meets AspectAG: How to make attribute grammars first-class. In *Proceedings of the Twelfth Workshop on Language Descriptions, Tools, and Applications*, LDTA '12, New York, NY, USA. Association for Computing Machinery.

[Viera and Swierstra, 2015] Viera, M. and Swierstra, S. (2015). Compositional compiler construction: Oberon0. *Science of Computer Programming*, 114.

[Viera et al., 2009] Viera, M., Swierstra, S. D., and Swierstra, W. (2009). Attribute Grammars Fly First-class: How to Do Aspect Oriented Programming in Haskell. In *Proceedings of the 14th ACM SIGPLAN International Conference on Functional Programming*, ICFP '09, pages 245–256, New York, NY, USA. ACM.

[Wadler, 1998] Wadler, P. (1998). The expression problem, mailing list discussion. http://homepages.inf.ed.ac.uk/wadler/papers/expression/expression.txt. Acceded: 5-4-2022.

[Wirth, 1988] Wirth, N. (1988). The Programming Language Oberon. *Softw. Pract. Exp.*, 18(7):671–690.

[Xia, 2018] Xia, L. (2018). First-class type families. https://hackage.haskell.org/package/first-class-families.