

Syntax Macros: Attribute Redefinitions

Master's Thesis

Joost Rommes

INF/SCR-03-31
July 2003

Abstract

In many applications of translations it is desired to return to the original input, for instance to report errors. More generally, there is need for a possibility to change the default behaviour of the translation. In this thesis, a technique is presented to redefine attributes that are specified in the attribute grammar of an abstract data structure at run-time. For the implementation an aspect-oriented approach to attribute grammars is required. The obtained attribute redefinition technique is combined with a syntax macro supporting compiler, which translates new concrete syntax to existing abstract syntax dynamically.

Contents

1	Introduction	4
2	Syntax Macros	5
2.1	Introduction	5
2.2	Syntax Macros: Terminology	6
2.2.1	Syntax Macros Source	8
2.2.2	Abstract Syntax	10
2.2.3	Macro Parser	10
2.2.4	Initial Grammar	10
2.2.5	Macro Interpreter	12
2.2.6	Extended Grammar	12
2.2.7	Program Source	12
2.2.8	Program Compiler	13
2.2.9	Compiled Program	13
2.3	Dynamic Typing	13
2.4	Restrictions	14
3	Attribute Redefinitions	16
3.1	Introduction	16
3.2	A First Approach	16
3.2.1	Pretty Print Attribute: Considerations	16
3.2.2	Attributes in General	17
3.3	Towards an Implementation	18
3.3.1	Elements of the Language	18
3.3.2	Extended Macro Language	21
3.3.3	Integration	22
3.3.4	Observations	25
3.3.5	λ -Abstractions	26
3.4	Inherited Attributes: Reconnaissance	26
3.4.1	Introduction	26
3.4.2	Silent Support	27
3.5	Attribute Grammar Combinators	29
3.5.1	Introduction	29
3.5.2	Attribute Grammars vs. Aspect-oriented Approach	29
3.5.3	Abstracting from the Structure and Attribute Computations	30
3.5.4	Separating the Aspects	33

3.5.5	A More General <i>knit</i>	34
3.5.6	Copy Rules	36
3.5.7	A Generic <i>knit</i>	36
3.6	Inherited Attributes: Solution	38
3.6.1	Rewriting the Semantic Functions	38
3.6.2	Attributes Revisited	38
3.6.3	Generated Code	43
3.6.4	Inherited Attributes Revisited	44
3.6.5	Extended Macro Language	44
3.6.6	Compiler	46
3.6.7	Non-constructor Translations	52
3.6.8	Nested Attribute Redefinitions	53
3.7	Considerations	57
4	Conclusions and Future Work	58
A	Extensions and Manual	63
A.1	Syntax Macro Library	63
A.2	Attribute Grammar System	63
A.3	Syntax Macro Grammar	64
A.4	Attribute Redefinition Manual	65

Chapter 1

Introduction

In the design of a language, one has to make a choice for either the compactness of the base language or the richness [4]. Both choices have their pros and cons, and some hybrid techniques are proposed in the literature [8, 12], combining the advantages of both sides. Syntax macros [1, 2] is one of the hybrid techniques. Syntax macros can be used to extend the concrete syntax of a programming language, and are especially useful for extending a small and simple core language with a domain specific language. Given an expressive core language, syntax macros provide the necessary syntactic sugar while leaving the core language unaltered.

The idea of syntax macros is more or less based on a one-way thought, namely the translation from the concrete syntax of a domain specific language to the abstract syntax of a core language. However, there are scenarios which require a sort of inverse transformation, from the core language to the domain specific language. Examples of such functionality are error reporting and pretty printing. More generally, one can think of the possibility to, besides giving the translation, redefine specific attributes that originate from the attribute grammar of the core language.

In this thesis, the difficulties of the attribute redefinition problem will be addressed and possible solutions will be presented. Chapter 2 gives an introduction to syntax macros and dynamic typing. Chapter 3 gives an analysis of the inverse transformation problem and describes a general solution. Chapter 4 concludes.

Chapter 2

Syntax Macros

2.1 Introduction

Syntax macros are introduced in [12] as a way to extend the concrete syntax of a programming language. This first idea of syntax macros has been used in various ways [1, 2, 4]. The syntax macros as discussed in [1, 2] will be used in this document, as they are built upon combinator parsers [15] and written in Haskell. The major advantage of this approach is that parsers can be generated on-the-fly. However, this flexibility requires that type checking must be performed at runtime as well. To get an idea of syntax macros, first an example of syntax macros will be given, followed by some syntax macro terminology.

The use and application of syntax macros will be illustrated on a simple expression language:

```
data Expr = Constant Int
         | Var String
         | Mul Expr Expr
         | Add Expr Expr
```

The corresponding context-free grammar, which will be used in the examples, is defined as follows:

```
Expression ::= Term "+" Expression
           | Term

Term        ::= Factor "*" Term
           | Factor

Factor      ::= IntLiteral
           | Varid

IntLiteral ::= ['0'... '9']+
Varid      ::= ['a'... 'z']+
```

The expression `1 + 2 * 3` is translated to

```
Add (Constant 1) (Mul (Constant 2) (Constant 3))
```

Usually, the concrete syntax as it is given by the grammar is fixed, that is, it cannot be extended with new syntax. Syntax macros offer the possibility to extend the concrete syntax with new syntax, provided a translation from the new concrete syntax to the existing abstract syntax is given. As an example, consider the following syntax macro:

```
Factor ::= "twice" x=Factor => Mul (Constant 2) x;
```

Without going into detail too much, the notion of this macro should be clear: concrete syntax of the form `twice x`, where `x` can be any `Factor`, should be translated to `Mul (Constant 2) x` for $x :: Expr$. In other words, writing `twice x` is "short-hand" for `2*x`.

Because new concrete syntax is added, the existing parser for the nonterminal concerned must be extended with a new alternative. Furthermore, a semantic function for the parser of the new alternative must be constructed. In order to do this at runtime, dynamic type-checking is needed, which will be described in Section 2.3.

With the idea of syntax macros more or less clear, also a glance at the main topic of this thesis can be given. A syntax macro defines a translation of concrete syntax to existing abstract syntax. In practice, it is sometimes desired to go back to the original representation of the macro. For instance when pretty printing an `Expr`, one would like to see the syntax macros to appear at the correct places: the pretty printed result of the translation of `twice x` should be `twice x` and not `2*x`. The current implementation of syntax macros does not support this functionality. What is wanted is a way to redefine the computation of the attributes for the abstract syntax tree of the translation.

In the following section, the terminology of syntax macros will be given. After that, type checking at runtime, or dynamic typing, will be described.

2.2 Syntax Macros: Terminology

Syntax macros extend the concrete syntax of a language by adding production rules for new concrete nonterminals and by adding alternatives for existing nonterminals. Such a production rule consists of the new concrete syntax and a translation to the abstract syntax of the core language. Globally, a compiler supporting syntax macros works as presented graphically in Figure 2.1.

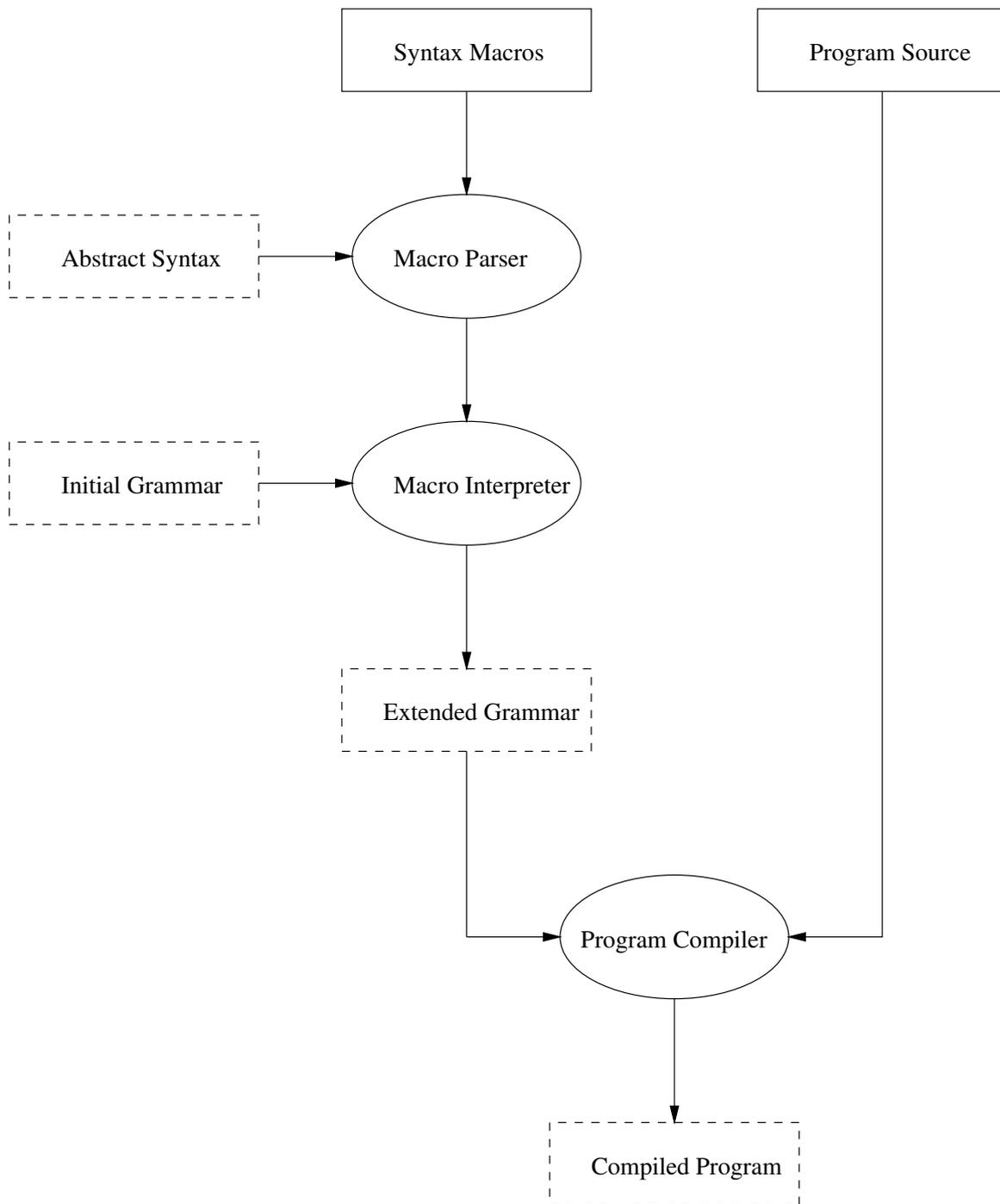


Figure 2.1: Schematic design of a compiler supporting syntax macros. Boxes with solid lines model source files, dashed boxes model internal entities and ellipses model processes (functions). Arrows model the flow of information.

Each of the elements in Figure 2.1 will be explained in the following sections. The major source for these subsections is the (unfinished) work of Arthur Baars [2].

2.2.1 Syntax Macros Source

A source file for syntax macros consists of a list of declarations of concrete nonterminals, and a list of production rules, defining the macros. In the following, EBNF notation will be used to define the context-free grammar [9].

```
MacroDefinitions ::= Nonterminals? Productions?
```

```
Nonterminals    ::= "nonterminals"  
                  Declaration+
```

```
Productions     ::= "rules"  
                  Production+
```

The concrete terminals `nonterminals` and `productions` are reserved keywords.

A `Declaration` can be used to define new concrete nonterminals (that will be used in the `Productions`), and consists of the new nonterminal and its `Type`, which can be either a type constant or a function type.

```
Declaration     ::= Nonterminal "::<" Type
```

```
Type           ::= SimpleType ("->" SimpleType)*
```

```
SimpleType     ::= TypeConstant  
                  | "(" Type ")"
```

The concrete nonterminal `TypeConstant` is a synonym for an uppercase identifier. Note that the set of `Types` that can be used in a `Declaration` is constant: one is restricted to the types defined in the core language. A `Nonterminal` is an uppercase identifier.

A `Production` defines the actual macro in a BNF-like style. For an existing or new concrete nonterminal, the concrete grammar is specified by a sequence of terminals and nonterminals (the `RHSElems`). Secondly, a translation from the new concrete syntax to the existing abstract syntax must be given. Associating variables with the concrete nonterminals in the right-hand side of the BNF rule makes it possible to refer to the nonterminals in the `Expression`-part that describes the translation of the recognized construct.

```
Production     ::= Nonterminal "::<=" RHSElem* "=>" Expression ";"
```

```
RHSElem       ::= StringLiteral  
                  | Varid "=" Nonterminal
```

A `Varid` is a lowercase identifier (keywords excluded), and a `StringLiteral` is a Haskell string literal (without white-spaces).

An `Expression` is an explicitly typed lambda expression (where `Type` must be defined in the core language). Besides translation to the abstract syntax, it is also possible to construct λ -abstractions over the existing abstract syntax.

```
Expression     ::= "\\\" Varid "::<" Type "." Expression  
                  | Factor+
```

```

Factor      ::= Constructor
              | Varid
              | Literal
              | StringLiteral
              | "(" Expression ")"

```

A **Constructor** is an uppercase identifier and a **Literal** is a language dependent literal. Note that a choice for the use of more intuitive concrete syntax for the translation would restrict the meaning of a macro translation to those semantics that are expressible by the original concrete syntax.

The abstract syntax of the context-free grammar describing the macro specification is as follows:

```

type Nonterminal = String
type Ident = String

data Program tp = Program [Declaration tp] [Production tp]

data Declaration tp = Decl Nonterminal (Exists tp)

data Production tp = Production Nonterminal [RHS_Elem] (Expression tp)

data RHS_Elem = Terminal String
                | Nont Identifier Nonterminal

```

The data type *Expression* is defined by

```

data Expression tp = Var Ident
                    | Apply (Expression tp)
                          (Expression tp)
                    | Const (Dynamic tp Id)
                    | Lambda Ident
                          (Exists tp)
                          (Expression tp)

```

The second argument of *Decl* and *Lambda* is of type *Exists tp*, a dynamic type representation (of the type of *Nonterminal* and *Ident* respectively). As will become clear in Section 2.3, the actual type is not visible right-away:

```

data Exists f = exists x o E (f x)

```

To conclude this section, an example macro source file for the simple expression language is presented.

```

nonterminals
Factor ::= Expr
rules
Factor ::= "square" f=Factor      => Mul f f ;
Factor ::= "pyth" a=Factor b=Factor => Add (Mul a a) (Mul b b)
Factor ::= "(" e=Expr ")"        => e ;

```

Because the translation expression must be defined in terms of the abstract syntax, it is not possible to reuse syntax macros (as they are defined in concrete syntax). Support for reusing macros would clearly improve the expressive power of the macro language.

2.2.2 Abstract Syntax

The abstract syntax of the original expression language (*Expr*) is as defined in the introduction of this section. The types and constructors must be known to the compiler of the syntax macros. Note that there are two kinds of abstract syntax (in fact even more), namely the abstract syntax of the language itself (the abstract syntax referred to in this section), and the abstract syntax of the small language used in the translation part of the macro syntax. In the following, it can be derived from the context which abstract syntax is concerned.

2.2.3 Macro Parser

The parser of the macro source file is built using the combinator parser library `UU_Parsing` as described in [14, 15]. Translation from the grammar in Section 2.2.1 to the corresponding parser is straightforward and will not be discussed in detail. However, two special cases are worth mentioning. The first concerns the parsing of `Types`: because type checking of the grammar for the new syntax must be done dynamically, parsed `Types` also must be represented dynamically. The correct type representation is looked up in an (automatically generated) associated list using the name of the type as key. How such a representation looks like internally will be described in Section 2.3.

Secondly, during parsing of the translation expression, a dynamic representation of the parsed `Constructor` must be available. This representation can be obtained by looking it up in an (automatically generated) associated list, using the `Constructor` as key. For the moment, such a representation can be taken to be a pair of a value (in this case the constructor) and its dynamic type: *Dynamic tpr value*, with *tpr* the type representation of *value*. For more details, see Section 2.3.

The final result of the macro parser is a *macroSpec@(Program decls prods)*.

2.2.4 Initial Grammar

A grammar associates concrete nonterminals with combinator parsers, representing the right-hand sides of the productions. A non-syntax macro supporting compiler can use a fixed associated list to represent the grammar, because the grammar will not be extended at run time. However, a syntax macro *supporting* compiler has to take care of two requirements:

1. The grammar may (and most likely will) be extended at run time;
2. The types of the extensions may (and most likely will) be different and must be checked at run time.

For the second item, the dynamic typing library [3] will be used (see also Section 2.3). To cope with the possible extensions (as defined by the macros) of the grammar, the definition

```
type Grammar tpr parser = [(Nonterminal, Dynamic tpr parser)]
```

is not sufficient. This insufficiency is caused by the fact that the parser for a nonterminal may change due to extensions, and is thus not available (completely) at compile time. In other words, the parsers for the nonterminals are not known until all macros have been parsed and interpreted. Once all macros are interpreted, the parser for a nonterminal can be looked up in the final grammar (including all extensions). This suggests defining the following type for the extended grammar:

```
type ExtGrammar tpr parser = Grammar tpr parser → Grammar tpr parser
```

With this definition, the initial grammar for the small language *Expr* can be defined:

```
initGrammar :: (IsParser p Char, TypeDescr tpr)
  ⇒ ExtGrammar tpr p
initGrammar final = [("Expr", expr :: tp_Expr)
  , ("Term", term :: tp_Expr)
  , ("Factor", factor :: tp_Expr)
  ] where
  expr = termRef
    ◇ Add <$> termRef <*> pToken "+" <*> exprRef
  term = factorRef
    ◇ Mul <$> factorRef <*> pToken "*" <*> termRef
  factor = Var <$> pVarid
    ◇ Constant <$> pInteger
  exprRef = getRule "Expr"
  termRef = getRule "Term"
  factorRef = getRule "Factor"
  getRule n = fromJust$
    do p ← lookup n final
        downcast tp_Expr p
```

The operators $\langle * \rangle$, $\langle \triangleright \rangle$ and $\langle \$ \rangle$ apply two parsers sequentially, choose a parser out of two alternatives and apply a function to the result of a parser respectively. The instance requirement *IsParser p Char* states that *p* is a parser that takes *Chars* as input (see [14, 15] for more information about the `UU_Parsing` library), and *TypeDescr tpr* requires that *tpr* is a dynamic type descriptor (representation). The term *tp_Expr* is a type descriptor of the type *Expr*, and $::$ is a constructor that creates a *Dynamic* out of a value and a type descriptor. If a dynamically typed value is to be used, the Haskell type checker must be convinced that the value is of the type represented by type descriptor. The *downcast* function takes care of this (see Section 2.3).

This initial grammar, without macro extensions, cannot be used right-away: it has to be fixed on *final*. The function *fix*, which computes the fixed point of a function, is defined by

```
fix :: (a → a) → a
fix f = let x = f x in x
```

The application of *fix* to *initGrammar* computes the fixed point of *initGrammar* in one iteration.

2.2.5 Macro Interpreter

The macro interpreter does the actual work by computing the new combinator parsers for the nonterminals and adding the new parsers to the initial grammar. To that end, the following steps must be taken for each of the rules of the given *macroSpec*@(*Program decls prods*):

1. Construct the semantic function for the parser of the new concrete syntax. This function uses the nonterminals of the concrete syntax (the *RHSElems*) as the arguments of a λ -abstraction. The body is the expression for the translation from the new syntax to the abstract syntax, and will be represented as an *Expression t* for some type descriptor *t*. Note that this construction is done dynamically. For example, the semantic function for the `twice` macro will look like (statically):

$$\lambda x \rightarrow \text{Mul } (\text{Constant } 2) x$$

2. Dynamically type check this semantic function.
3. Construct the extension of the parser by using parser combinators to apply the semantic function (*sem*) to the result of the parser. This parser is constructed out of the terminals and nonterminals in the right-hand side of the rule. Statically, the complete parser for the `twice` macro would be

$$\text{sem} \ll \$ pTerminal \text{"twice"} \langle * \rangle factor$$

where *pTerminal* parses a terminal string and *factor* is the parser for a `Factor` that is looked up in the initial grammar.

4. Extend *initGrammar*, which is passed to the macro interpreter, with the new parser. If a parser for the nonterminal already exists, then the new parser should be added as an alternative (using parser combinators); otherwise, a new nonterminal-parser pair must be added to the grammar.

2.2.6 Extended Grammar

The macro interpreter returns a pair of a list of terminal strings and a grammar of type *ExtGrammar tpr p*¹. Before the extended grammar can be used, it has to be fixed because still new alternatives may be added to the grammar. The fixed point must be computed to close the grammar.

2.2.7 Program Source

A program can now be coded using the original concrete syntax extended with the new syntactic sugar defined by the macros. An example program is

```
square 2 + square 3
```

¹The actual implementation in fact returns a grammar which given a grammar, returns a computation for a grammar, using monadic structures. Monadic structures ease certain actions such as error reporting.

2.2.8 Program Compiler

The program compiler fixes the extended grammar, takes a program as input, looks up the corresponding parser, convinces the type checker that it is of the correct type by *downcasting* it, and applies the parser to the input.

2.2.9 Compiled Program

The output of the program compiler is an abstract syntax tree in the original abstract syntax. For the example program of subsection 2.2.7 this becomes

```
Add (Mul (Constant 2) (Constant 2)) (Mul (Constant 3) (Constant 3))
```

2.3 Dynamic Typing

This section is based on the article *Typing Dynamic Typing* [3]. Because this section is rather technical and complete understanding is not necessary for the rest of this document, it can be skipped at a first reading. Knowing the notation and assuming that dynamic type checking is safely possible will be sufficient to understand the following chapters.

The idea is to define a universal data type *Dynamic* to represent a value of any type ($f\ x$) and a representation of that type ($tpr\ x$):

```
data Dynamic tpr f = exists x o f x :: tpr x
```

Note that the functor f is used for the sake of generality. The practical use of this will become clear in the rest of this section. Furthermore, the actual type is hidden by the existential quantor *exists*, and thus cannot be returned right-away.

A **class** of type descriptors is introduced to abstract from the type used for type representations, providing a compare operator *match*:

```
class TypeDescr td where
  match :: td a → td b → Maybe (Equal a b)
```

The result of an equivalence check on two type descriptions (labeled with the type concerned), is *Nothing* if the type descriptions are not equivalent, or *Just* a proof of the type equivalence:

```
newtype Equal a b = Equal (forall f o f a → f b)
```

The type *Equal* is based on Leibnitz' definition of equality and reflects type equality. Leibnitz' original definition is

$$a \equiv b \Leftrightarrow \forall f. f\ a \equiv f\ b \tag{2.1}$$

that states that if a and b are identical, then they must have identical properties.

One can see this intuitively as the identity function is the only non-diverging conversion function that can be used as an argument for the constructor: the conversion function cannot make any assumptions about the structure of f . Therefore, the existence of a value *Equal a b* implies $a \equiv b$, because the identity function is the only function that converts a into b in any context.

To represent constant types, such as *Int* and *Bool*, the data type *TypeCon* is defined as

```
data TypeCon a = Int (Equal a Int)
               | Bool (Equal a Bool)
```

To compare *TypeCons*, it is necessary to make *TypeCon* an instance of *TypeDescr* and implement the compare function *match*. For details on the implementation of *match* the reader is referred to [3].

In practice, one also wants to represent types that are constructed out of other types, such as function types:

```
data TpRep tpr a = TpCon (tpr a)
    | exists x y o Func (Equal a (x → y)) (FunTp tpr x y)
```

```
data FunTp tpr a b = TpRep tpr a :=> TpRep tpr b
```

To construct representations of function types, the function (*.→.*) can be used.

```
(.→.) :: TpRep tpr a → TpRep tpr b → TpRep tpr (a → b)
```

To make the data type *TpRep* an instance of the *TypeDescr* class, it is required that the type of type representations (*tpr*) is an instance of *TypeDescr* too (cf. [3]).

Having a dynamically typed value, and its expected type, the last function needed is a function which casts the value to its type:

```
downcast :: TypeDescr tpr => tpr a → Dynamic tpr f → Maybe (f a)
```

The implementation of various functions is not discussed here; however, an example may illustrate the use. Having representations for *Int* (*inttp* :: *TpRep TypeCon Int*) and *Bool* (*booltp* :: *TpRep TypeCon Bool*), integer and boolean values with their type representations can be constructed as

```
data Id x = Id x
    one = Id 1 :: inttp
    true = Id true :: booltp
```

Similarly, the dynamic representation of an operator of type *Int* → *Int* → *Bool* can be constructed as

```
dyn_op = Id op :: inttp .→. inttp .→. booltp
```

To actually use the dynamically typed values, the Haskell type checker has to be convinced that the type is the correct type. This can be done with the *downcast* function:

```
case downcast (inttp .→. inttp .→. booltp) dyn_op of
    Just op → op 1 1
    Nothing → error "Type_mismatch."
```

For a detailed definition of the dynamic typing library, see [3].

2.4 Restrictions

There are a number of restrictions in the current approach of syntax macros:

- It is not possible to reuse macro definitions in the macro source files. This becomes increasingly inefficient when implementing large domain specific extensions.

- Any connection between program source files using syntax macros and the transformed program (in the original abstract syntax) is lost: syntax macros are primarily designed to do one-way transformations. For instance, there is no way to pretty print the original input, because the input is transformed to original abstract syntax, and hence the original representation of the macro is lost.
- Error reporting also takes place in terms of the transformed program. As a result, errors may appear in pieces of code which are completely unknown to the user. Ideally, errors, especially type errors, should be reported in the *original* code.
- Currently, attributes are computed in the way they are defined for the abstract syntax of the translation. More generally, it may be desirable to adjust the existing semantics for existing attributes for the new syntax macros. A way to do this is to define the semantics for the new syntax macros in terms of existing semantics. This question will be studied in detail in the following chapter.

Chapter 3

Attribute Redefinitions

3.1 Introduction

The previous chapter explained the concept of syntax macros, and discussed some of the restrictions. This chapter will focus on these restrictions. More specifically, the attribute redefinition problem will be studied.

Section 3.2 gives an outline of a solution for the redefinition of synthesized attributes. Section 3.3 shows how to implement this idea. In Section 3.4, the problem of supporting inherited attributes will be studied. Section 3.5 discusses an aspect-oriented approach to attribute grammars, because this turns out to be needed for the support of inherited attribute redefinitions. In Section 3.6 the aspect-oriented approach will be used to implement both inherited and synthesized attribute redefinitions in the syntax macro system.

3.2 A First Approach

3.2.1 Pretty Print Attribute: Considerations

Pretty printing is a good example of the need for attribute redefinitions. Currently, the pretty print attribute will be computed according to its definition for the abstract syntax tree of the translation of the syntax macro. This definition is originating from the attribute grammar. In practice, one can think of situations where this behavior is not sufficient: pretty printing of the parsed source code of a language for example will lack any presence of syntax macros. Therefore, the pretty printing attribute will serve as the guiding attribute in the search for a solution of the attribute redefinition problem.

Pretty printing is the formatting of a data structure in such a way that it correctly reflects the abstract data structure using a concrete syntax, and is nicely readable. This definition clearly leaves some aesthetic issues open, but regarding pretty printing of programs using syntax macros, especially the following considerations must be made:

- As already remarked in Section 2.4, compiling a program throws away all the information about the original source program. This gives rise to the following problem: how to decide whether a certain sub-tree of the abstract syntax tree corresponds to the use of a syntax macro, or to original concrete syntax.
- Is the pretty printer expected to work on *all* programs defined in the abstract syntax, or only on the programs corresponding to the parsed source files? This becomes an

important issue if transformations are performed on the abstract syntax tree of the program afterwards.

- Is there any freedom for the user to define what the pretty printed program actually looks like? In other words, is there a small language to define the pretty printed output?

In this section, an approach similar to the inverting parsers [7] will be used. As a consequence, the constructed pretty print attributes will correctly reflect the original input: it is guaranteed that the inverse transformation preserves the original input, that is, syntax macros appear at the correct places.

3.2.2 Attributes in General

The pretty print attribute is just a specific *attribute*, in the context of attribute grammars [5]. With this in mind, the question comes up if there is a way to describe the computation of the pretty print attribute more generally, thus opening the way for manipulating and redefining other attributes, such as errors. In other words, the semantics used by the parser have to be adjusted. One way to do this is to define a small language for the attributes, just like the BNF-like language for the syntax macros. This language can then be used to manipulate the default semantics. Given some predefined operators, it thus becomes possible to print errors in the transformed program at the associated place in the original program (in which use is made of the extended notation introduced by the syntax macros). This functionality looks like a small but dynamic attribute grammar system without the possibility to define *new* attributes (but keep in mind that it is not the intention to create a real attribute grammar system).

More concretely, a small language can be added for the definition of the new attribute computations. The elements of this small language are the semantic functions of the original attribute grammar, the existing attributes themselves, and a limited set of functions and operators to manipulate the attributes.

To enlarge flexibility, it is desired to be able to identify the children of a node, no matter where the children reside in the tree. Such a functionality solves the problem of locating the child, and the problem of linearity (for instance in definitions of the form `square x => Mul x x`, where there is currently no distinction between the two `x` children).

Before defining the system in detail, some assumptions have to be made:

- Only constructions using the original abstract syntax are considered as valid expressions for the translation part of the syntax macros. Some elements of this expression language, such as λ -abstractions, are neglected (for the moment).
- The computation for existing attributes can be changed, but it is not possible to add new attributes dynamically.
- Only first-order attributes are considered.

Summarizing, the following sub-problems can be identified:

Elements of the Language The elements of the new language must be identified and created out of the attribute grammar definition automatically.

Extended Macro Language A new language, to express the new computations for existing attributes, in terms of existing attributes and computations must be designed, including interpretation and concrete syntax.

Integration Integration of the new functionality into the existing syntax macros module.

Note that the one and only moment to create the new semantics is during parsing and interpreting of the syntax macros, and the one and only moment to apply the new semantics is during parsing of the program. So the actual work is concentrated in the semantic function of the parser of the syntax macros. This approach focuses on programs that are actually parsed; applying transformations to the parsed programs is beyond the scope of this document.

In the following sections, steps towards an implementation are described. Of some steps the attended reader may think that things are missing or going wrong. However, reparations and refinements will follow in the subsequent sections.

3.3 Towards an Implementation

3.3.1 Elements of the Language

In general, there will be several attributes, of which the computation is defined (and generated) by an attribute grammar system. To reason about the semantic functions, some details of the attribute grammar system [5] must be mentioned. The AG system generates type synonyms for every data type. In a type synonym for a data type D , inherited attributes $inh_1 \dots inh_m$ appear as arguments, while synthesized attributes $syn_1 \dots syn_n$ appear in a tuple as result type¹:

type $T_D = Inh_1 \rightarrow \dots \rightarrow Inh_m \rightarrow (Syn_1, \dots, Syn_n)$

The type of the parser used for parsing such a data type then has the type

type $Parser_D = Parser\ Char\ T_D$

where a detailed description of $Parser$ can be found in [14].

Semantic functions for the constructors of a data type D take the semantics of their children $C_1 \dots C_k$ as arguments and return the semantics of D :

$$\begin{aligned} sem_{D_C} &:: T_{C_1} \rightarrow \dots \rightarrow T_{C_k} \rightarrow T_D \\ sem_{D_C} \ sem_{c_1} \ \dots \ sem_{c_k} &= \\ &\lambda inh_1 \ \dots \ inh_m \rightarrow \\ &\quad \mathbf{let} \ result_1 = sem_{c_1} \ inh_{c_1_1} \ \dots \ inh_{c_1_{s1}} \\ &\quad \dots \\ &\quad \quad \quad \ result_k = sem_{c_k} \ inh_{c_k_1} \ \dots \ inh_{c_k_{sk}} \\ &\quad \mathbf{in} \ (syn_1, \dots, syn_n) \end{aligned}$$

where $inh_{c_i_{sj}}$ is the j -th inherited attribute of the i -th child. These semantic functions are generated out of the attribute grammar, and this attribute grammar is defined for the base language. The macro interpreter extends the grammar with new parsers and semantic

¹In the most recent implementation of the AG system, the **types** are actually replaced by **newtypes** [13]. This is useful especially if the programmer makes a mistake: in the error message, only the name of the type is shown instead of the unfolded type.

functions. However, this interpretation is done in a fixed way: the computations as defined for the translation are used. Ideally there would be a default interpretation *and* a way to redefine the computations for the existing attributes. The most straightforward approach is to define the new attribute computations in terms of the existing attributes. Lifted to a higher level, what is needed is:

- Identification of attributes: every attribute is located in the tuple generated by the AG system, but the exact location is only known by the AG system. Functions to select individual attributes from the tuples must be generated by the AG additionally.
- Combinators to combine the attributes (of different types), such as pretty print combinators, in the new definition. Here a choice for a (small) selection of combinators must be made, because there are infinitely many combinators to think of. This choice has to be made by the designer of the language. An additional language to define combinators is not the intention.
- Inserter functions that take a computation for the selected attribute and insert the result into the tuple containing all attributes.
- The original semantic functions as defined by the attribute grammar. These are already available, but have to be represented as dynamic values. Because the new semantics for the attributes have to be type checked dynamically, the type representations have to be available as well. Abstraction from the number of elements (and types) in a tuple is required to avoid creating type representations of all kinds of cartesian products. This can be done by using the type synonyms that are generated by the AG system, instead of the unfolded types.

Assuming that there are no inherited attributes, a type signature of the semantics for a data type D looks like

type $T_D = (Syn_1, \dots, Syn_n)$

Selection functions for each of the attributes $syn_i :: Syn_i$ have to be generated:

$select_syn_i :: T_D \rightarrow Syn_i$
 $select_syn_i (syn_1, \dots, syn_i, \dots, syn_n) = syn_i$

Similarly, insertion functions for each of the attributes into the result product have to be generated:

$insert_syn_i :: Syn_i \rightarrow T_D \rightarrow T_D$
 $insert_syn_i syn (syn_1, \dots, syn_i, \dots, syn_n) = (syn_1, \dots, syn, \dots, syn_n)$

The order of the attributes in the tuple is hidden by the insertion and selection functions; as a result, one is not restricted to a fixed order of inserting and defining the new semantics. Both the selector and inserter functions have to be available as *Dynamic* values. The functions are identified by the name of the attributes that they respectively select and insert:

$selectors :: [(String, Dynamic (Type TypeCon) Id)]$
 $selectors = [("syn_i", Id select_syn_i :: tp_T_D \rightarrow. tp_Syn_i)]$

$inserters :: [(String, Dynamic (Type TypeCon) Id)]$
 $inserters = [("syn_i", Id insert_syn_i :: tp_Syn_i \rightarrow. tp_T_D \rightarrow. tp_T_D)]$

The prefix *tp_* is used to denote the dynamic type representation of the name of the type in the suffix. The original semantic functions are available in the same way, with the name of the constructor to be used as key²:

```

semantics :: [(String, Dynamic (Type TypeCon) Id)]
semantics =
  [("C_i", Id sem_D_C_i ::: tp_T_C_i_1 .->. tp_T_C_i_... .->. tp_T_C_i_k -> tp_T_D)]

```

Regarding the attribute combinator functions, such as pretty print combinators, it should be mentioned that their representation as dynamic values can be generated, but that the combinators themselves have to be provided by the designer of the language.

As an example, consider the simple expression language (*Expr*) with attributes *ast* (abstract syntax tree) and *pp* (pretty printed). The following code (among other code) or code with the same semantics will have to be generated by the AG system in order to be used in the syntax macros library, starting with the type synonym and the corresponding type description for the synthesized attributes:

```

type Expr_attrs = (Expr, PP_Doc)

tp_Expr_attrs = TpCon (Expr_attrs_tp_equal)
data TypeCon a = ...
  | Expr_attrs_tp (Equal a Expr_attrs)

```

Note that the type synonym *Expr_attrs* is used to construct the dynamic type representation, instead of the unfolded type. Note also that the *match* function of **class** *TypeDescr* must be extended to cope with the new constructor of *TypeCon*.

The selection and insertion functions are generated straightforwardly:

```

select_ast :: Expr_attrs -> Expr
select_ast = fst

select_pp :: Expr_attrs -> PP_Doc
select_pp = snd

insert_ast :: Expr -> Expr_attrs -> Expr_attrs
insert_ast e (ast, pp) = (e, pp)

insert_pp :: PP_Doc -> Expr_attrs -> Expr_attrs
insert_pp p (ast, pp) = (ast, p)

```

The semantic functions as defined by the attribute grammar:

```

sem_add (t, pp1) (e, pp2) =
  (Add t e, pp1 >#< text "+" >#< pp2)
sem_mul (t, pp1) (e, pp2) =
  (Mul t e, pp1 >#< text "*" >#< pp2)
sem_var v = (Var v, text v)
sem_constant i = (Constant i, (text ∘ show) i)

```

²For notational simplicity the name of the constructor is assumed to be unique. The AG system has an option **rename** to ensure that constructor names are unique by prefixing each constructor with the corresponding data type.

The selection and insertion functions, and the semantic functions are collected in associated lists, to be able to use them dynamically:

```

selectors :: [(String, Dynamic (Type TypeCon) Id)]
selectors = [("ast", Id select_ast ::: tp_Expr_attrs .->. tp_Expr)
              , ("pp", Id select_pp ::: tp_Expr_attrs .->. tp_PP_Doc)
              ]

inserters :: [(String, Dynamic (Type TypeCon) Id)]
inserters = [("ast", Id insert_ast ::: tp_Expr .->. tp_Expr_attrs .->. tp_Expr_attrs)
              , ("pp", Id insert_pp ::: tp_PP_Doc .->. tp_Expr_attrs .->. tp_Expr_attrs)
              ]

semantics :: [(String, Dynamic (Type TypeCon) Id)]
semantics = [("Mul", Id sem_mul ::: tp_Expr_attrs .->. tp_Expr_attrs .->. tp_Expr_attrs)
              , ("Add", Id sem_add ::: tp_Expr_attrs .->. tp_Expr_attrs .->. tp_Expr_attrs)
              , ("Var", Id sem_var ::: tp_String .->. tp_Expr_attrs)
              , ("Constant", Id sem_constant ::: tp_Int .->. tp_Expr_attrs)
              ]

```

3.3.2 Extended Macro Language

The macro language must be extended with concrete syntax for (1) the identification of attributes and (2) the operators and functions for manipulating the attributes. The grammar as defined in 2.2.1 needs to be adjusted at some points. A `Production` is extended with a list of `AttrExpressions`:

```

Production ::= Nonterminal " ::= " RHSElem*
              "=>" Expression ";" (AttrExpression ";")*

```

```

Expression ::= Factor+

```

```

Factor ::= Constructor
         | Varid
         | Literal
         | StringLiteral
         | "(" Expression ")"

```

The production for `RHSElem` remains unaltered.

An `AttrExpression` consists of a left-hand side that identifies the synthesized attribute of the parent that is to be redefined, and a right-hand side that is the attribute definition.

```

AttrExpression ::= Attr "=" AttrCompExpression

```

```

AttrCompExpression ::= AttrFactor+

```

```

AttrFactor ::= Constructor
            | Literal
            | StringLiteral

```

```

| AttrSel
| Operator
| "(" AttrCompExpression ")"

```

Attributes can be identified by their name as defined in the attribute grammar; an attribute a of a nonterminal variable c (that is associated internally with a product of synthesized attributes) can be identified by $c.a$. The combinators are available as **operators**.

```

Attr      ::= Varid
AttrSel   ::= Varid "." Attr

Operator  ::= OpSymbols+
OpSymbols ::= "#" | "$" | "%" | "^" | "&" | "*" | "-" | "<" | ">"

```

An example macro definition, that redefines the pretty print attribute `pp`, now takes the form

```

Factor ::= "double" x=Factor => Mul (Constant 2) x ;
      pp = Text "double" >#< x.pp ;

```

The function `Text` and the operator `>#<` are provided by the pretty print library as `text :: String → PP_Doc` and `(>#<) :: PP_Doc → PP_Doc → PP_Doc`, to print a string and to concatenate two `PP_Docs` using a space respectively. The machinery behind the interpretation of the macros is described in the following section.

3.3.3 Integration

Macro Scanner

Because additional operators may be used in the definition of the semantics, the scanner of the `UU_Parsing` library [14] must be instructed with the corresponding operator characters.

Macro Parser

To integrate the new ideas in the existing implementation, new parsers for the extension of the macro language must be added. These parsers can be constructed out of the grammar using combinator parsers. There are a number of changes to both the parsers and their semantic functions that are worth mentioning. First, the data type *Production* is adjusted to represent the translation expression, and a list of tuples of expressions. The tuple is a pair of a function that inserts the attribute into the result attribute-tuple, and the new computation for the corresponding attribute.

```

data Production tp = Production Nonterminal
                        [RHS_Elem]
                        (Expression tp)
                        [(Expression tp, Expression tp)]

```

The translation is parsed as follows (neglecting the λ -abstractions):

```

pTranslation :: Parser (Token (Dynamic (Type t) Id)) (Expression (Type t))
pTranslation = ( $\lambda c$  args → foldl Apply (getSemantics c) args) <$>
              pConid <*> pList1 pTranslation

```

```

    ◇ pTransFactor
where pTransFactor = Const <$> pLiteral
          ◇ Var <$> pVarid
          ◇ pParens pTranslation
    getSemantics c = maybe (Var c) Const (lookup c semantics)

```

```
lookup :: Eq a => a -> [(a, b)] -> Maybe b
```

However, the semantic function of *pTranslation* has changed, as all constructor functions are now replaced by their corresponding semantic functions generated by the AG system. Recall that such a semantic function expects the semantic functions of its children (in this case the synthesized attributes). The result for the syntax macro is then the representation as a dynamically typed expression of the semantics for the abstract syntax of the translation. In other words, the result is the same as would the concrete syntax of the translation of the syntax macro have been parsed, and hence can be seen as the default semantics for a syntax macro. Considering the `double x` macro, the result for the `pp` attribute would be `2 * x`, because `2 * x` is exactly the pretty print attribute as defined for *Mul (Constant 2) (Var "x")*. A value `double x` for the pretty print attribute would be more desirable.

The second part, the list of attribute-expressions, defines the new attribute computations. The first expression in the resulting tuple of *pAttrExpression* represents the insertion of the attribute redefinition in the result tuple; the second expression is the redefinition of that attribute.

```

pAttrExpression :: Parser (Token (Dynamic (Type t) Id))
  (Expression (Type t), Expression (Type t))
pAttrExpression = (,) <$> pInserter <*> pKeyword "=" <*> pSemExpression

```

The parser *pInserter* parses an attribute name and looks up the corresponding attribute inserter.

```

pInserter :: Parser (Token (Dynamic (Type t) Id)) (Expression (Type t))
pInserter = getInserter <$> pVarid
where getInserter v = maybe (Var v) Const (lookup v inserters)

```

The parser *pSemExpression* parses an attribute computation with operators. The operators are available in a list *operators* :: [(String, Dynamic (Type TypeCon) Id)]. The application of constructors and functions has higher priority than the use of operators.

```

pSemExpression :: Parser (Token (Dynamic (Type t) Id)) (Expression (Type t))
pSemExpression
  = pChainl ((λop f1 f2 -> (Apply (Apply (getOperator op) f1) f2)) <$> pOperator)
    pConstrExpr

```

```
getOperator op = maybe (Var op) Const (lookup op operators)
```

The operands are expressions with constructors and functions, attribute selections, literals, string literals, and parenthesized attribute computations. The operands are parsed by *pConstrExpr*. For pragmatical reasons constructors and functions are assumed to be upper-case identifiers. Internally, constructors and functions are treated in the same way. The constructors and functions are available in *constructors* :: [(String, Dynamic (Type TypeCon) Id)].

```
pConstrExpr :: Parser (Token (Dynamic (Type t) Id)) (Expression (Type t))
```

$$pConstrExpr = (\lambda c \text{ args} \rightarrow \text{foldl } Apply \text{ (getConstructor } c) \text{ args}) \langle \$ \rangle$$

$$pConid \langle * \rangle pList1 pConstrExpr$$

$$\langle \diamond \rangle pFactor$$

$$pFactor = pParens pSemExpression$$

$$\langle \diamond \rangle Const \langle \$ \rangle pLiteral$$

$$\langle \diamond \rangle pAttrSel$$

$$\text{getConstructor } n = \text{maybe (Var } n) Const (\text{lookup } n \text{ constructors})$$

An attribute selection consists of a nonterminal variable and an attribute. The parser $pAttrSel$ parses the nonterminal variable and uses $pAttr$ to parse the attribute; $pAttr$ returns the corresponding selector of the attribute, and $pAttrSel$ constructs an $Apply$ of this selector to the nonterminal variable. Note again that the nonterminal variable has the type of the semantic function corresponding to type of the nonterminal, i.e. a tuple of attributes. This variable is supplied by the macro interpreter when it constructs the λ -abstraction for the semantic function of the parser for the syntax macro. Hence, this variable represents the parse result of the nonterminal of the right-hand side of the production.

$$pAttrSel :: Parser (Token (Dynamic (Type t) Id)) (Expression (Type t))$$

$$pAttrSel$$

$$= (\lambda v \text{ sel}_a \rightarrow Apply \text{ sel}_a (Var v)) \langle \$ \rangle pVarid \langle * \rangle pKeyword "." \langle * \rangle pAttr$$

$$pAttr :: Parser (Token (Dynamic (Type t) Id)) (Expression (Type t))$$

$$pAttr = \text{getSelector} \langle \$ \rangle pVarid$$

$$\text{where } \text{getSelector } v = \text{maybe (Var } v) Const (\text{lookup } v \text{ selectors})$$

Initial Grammar

The initial grammar must use the semantic functions constructed by the AG system, or at least functions with the same types and semantics.

Macro Interpreter

The macro interpreter builds the complete parser for the new syntax macro. This is done in four steps:

1. Construct the new semantic function that takes the parsed tokens and returns a tuple with the redefined attributes.
2. Dynamically type check the new semantic function.
3. Combine this semantic function with the parser for the syntax macro.
4. Extend the grammar with the new parser.

In comparison with the steps for the interpreter in section 2.2.5, only the first step, i.e. the construction of the semantic function has changed, as described below.

The interpreter must combine the default semantic function and the list of tuples of attribute inserters and attribute redefinitions into the complete semantic function for the parser of the syntax macro. This is done by calling the function $extend$ that takes the default

computation (*default_sem*) and the list of tuples (*ins_comp_exprs*) and returns the body of the semantic function.

```

extend :: (Expression (Type a))
          → [(Expression (Type a), Expression (Type a))]
          → Expression (Type a)
extend default_sem ins_comp_exprs =
  if null exprs then orig_sem else foldl1 Apply exprs_fixed
    where zipped_exprs = map (uncurry Apply) ins_comp_exprs
          exprs_fixed = (init zipped_exprs)
                        ++ [(Apply (last zipped_exprs) (default_sem))]

```

What happens is the following. The default attribute definitions need to be adjusted by the redefined attributes. This is done by first providing all inserter functions with the new attribute definitions (*map (uncurry Apply) ins_comp_exprs*), then providing the *last* inserter function with its final argument, i.e. the default attribute definitions, (*(init zipped_exprs) ++ [(Apply (last zipped_exprs) (orig_sem))]*), and finally folding this list of insertions into one expression. Essentially, the original product with the attributes is passed to the last inserter function; the result of that application is a new product with one attribute replaced; this result is passed to the next inserter function. If there are no attribute redefinitions, just the original attributes are used (*orig_sem*). This expression will be the new body of the semantic function for the new parser. The remainder of the macro interpreter is not changed.

3.3.4 Observations

There are some improvements and work to think of:

- Support for λ -abstractions over the abstract syntax in the translation part of the syntax macro.
- Inherited attributes must be supported.
- In addition to redefining attributes in terms of the attributes of the children in the translation of the syntax macro, it would also be useful to locally redefine the attributes of the children themselves.
- The attribute grammar system must be adjusted to generate the code needed for the syntax macros.
- Support for higher order attributes.
- Besides redefining existing attributes, the possibility the define new attributes would be useful.

The first four points will be addressed in the remainder of this document. The AG system does not yet support higher order attributes, so these will not be considered in this thesis (although supporting higher order attributes does not introduce big issues). Defining new attributes is beyond the scope of this document (see also Section 3.7).

3.3.5 λ -Abstractions

Silent support for λ -abstractions can be added straightforwardly by adding the following alternative to *pTranslation*:

```

pTranslation = ..
  <> Lambda <$ pKeyword "\\\"
    <*> pVarid
    <*> pKeyword "::\"
    <*> pType
    <*> pKeyword "."
    <*> pTranslation

```

Unfortunately, this does not enable the possibility to redefine attributes for macros with an **Expression**-part that uses λ -abstractions, nor to refer to attributes of these macros in other attribute redefinitions. The reason for this is that the translation is an *abstraction* over the original abstract syntax, and thus the (dynamic) type of the translation is not equal to the type of the catamorphism of the abstract data type (e.g. *Expr_attrs*), but a function type with as result type that type (e.g. *Expr_attrs* \rightarrow *Expr_attrs*). As a result, the application of inserter functions to that translation will result in a type error during dynamical type checking (because an insertion function for a datatype expect values of types corresponding to that datatype). So why not reason about the attributes of the body of the λ -abstraction, which is of the correct type? By bringing the *Lambda* part of the default semantics to front and replace its body with the new body for attribute redefinitions, this can be achieved:

```

insertBody :: Expression t  $\rightarrow$  Expression t  $\rightarrow$  Expression t
insertBody (Lambda v tp body) newBody = Lambda v tp (insertBody body newBody)
insertBody _ newBody = newBody

```

What happens is that the attributes of the result of the application are redefined. Note that the attribute selection and insertion functions in this case must correspond to the type of the result of the λ -abstraction. In Section 3.6.8 a more subtle approach will be given.

However, it is still not possible to refer to synthesized attributes of variables corresponding to function type nonterminals in other macros, for the same reason as above.

3.4 Inherited Attributes: Reconnaissance

3.4.1 Introduction

Handling inherited attributes is different from handling synthesized attributes because inherited attributes appear as arguments in the type synonym of the data type. Support for inherited attributes appears to be more difficult, and the problem will therefore be divided in three sub-problems.

1. Silent support: inherited attributes may be used in the default semantics, but it is not possible to use inherited attributes in synthesized attribute redefinitions, nor to redefine inherited attributes.
2. Redefine inherited attributes: it is possible to redefine inherited attributes.

3. Inherited attribute use: inherited attributes can now be used in redefinitions of synthesized attributes.

In the following subsection, a start for the support of inherited attributes will be made by first reconsidering the insertion and selection functions as they are used for synthesized attributes. However, as will become clear, this approach will fail for more fundamental reasons.

3.4.2 Silent Support

To illustrate the changes in the insertion and selection functions, the inherited attribute $ppStyle :: Int$ (that denotes the style of the pretty print attribute) is introduced for the $Expr$ data type.

Because the AG type synonyms change if inherited attributes are introduced, the attribute selection and insertion functions have to be changed too. Attributes can only be selected *after* the semantic function has been applied to the inherited attributes. This results in the following selector function prototype:

$$\begin{aligned} select_D_syn_i &:: T_D \rightarrow (Inh_1 \rightarrow \dots \rightarrow Inh_m \rightarrow Syn_i) \\ select_D_syn_i\ sem &= project_syn_i \circ sem \end{aligned}$$

where $project_syn_i$ selects the i -th out of a cartesian product. Notice that T_D , the type synonym for the semantic function, is now a function type from inherited attributes to the product of synthesized attributes. The inserter function prototype is

$$\begin{aligned} insert_D_syn_i &:: (Inh_1 \rightarrow \dots \rightarrow Inh_m \rightarrow Syn_i) \rightarrow T_D \rightarrow T_D \\ insert_D_syn_i\ def_syn\ sem &= \lambda inh_1 \dots inh_m \\ &\rightarrow \mathbf{let} (syn_1, \dots, syn, \dots, syn_n) = sem\ inh_1 \dots inh_m \\ &\quad syn_i = def_syn\ inh_1 \dots inh_m \\ &\mathbf{in} (syn_1, \dots, syn_i, \dots, syn_n) \end{aligned}$$

Clearly some remarks must be made. Firstly, this implementation is not efficient: the expression $sem\ inh_1 \dots inh_m$ is executed more than once (in fact once for every attribute redefinition). Secondly, the expression representing the new definition of a (synthesized) attribute cannot be computed by simply folding $Apply$ around the list of selections and combinators. The whole expression must be preceded by an abstraction from the inherited attributes, and these attributes must be passed to each application of a selection function, because attributes can only be selected *after* application of the semantic function to the inherited attributes³.

These issues can be explained with an example, again for the simple expression language $Expr$. The selection functions for the two synthesized attributes now become

$$\begin{aligned} \mathbf{type}\ Expr_attrs &= Int \rightarrow (Expr, PP_Doc) \\ select_expr_ast &:: Expr_attrs \rightarrow (Int \rightarrow Expr) \\ select_expr_ast\ sem &= fst \circ sem \\ \\ select_expr_pp &:: Expr_attrs \rightarrow (Int \rightarrow PP_Doc) \\ select_expr_pp\ sem &= snd \circ sem \end{aligned}$$

³An alternative approach is to select directly out of the product of synthesized attributes. Such an approach however has some subtle issues concerning the use of redefined attributes.

The inserter functions take a function that computes the new attribute, apply that function to the inherited attribute(s) and insert the result in the tuple of synthesized attributes.

$$\begin{aligned} \text{insert_expr_ast} &:: (Int \rightarrow Expr) \rightarrow Expr_attrs \rightarrow Expr_attrs \\ \text{insert_expr_ast } e \text{ sem} &= \\ &\lambda ppStyle \rightarrow \mathbf{let} (ast, pp) = \text{sem } ppStyle \mathbf{in} (e \text{ } ppStyle, pp) \end{aligned}$$

$$\begin{aligned} \text{insert_expr_pp} &:: (Int \rightarrow PP_Doc) \rightarrow Expr_attrs \rightarrow Expr_attrs \\ \text{insert_expr_pp } p \text{ sem} &= \\ &\lambda ppStyle \rightarrow \mathbf{let} (ast, pp) = \text{sem } ppStyle \mathbf{in} (ast, p \text{ } ppStyle) \end{aligned}$$

The mapping from nonterminals to corresponding inherited attributes is generated as

$$\begin{aligned} \text{inherited_attrs} &:: [(String, [(String, Exists (Type TypeCon))])] \\ \text{inherited_attrs} &= [(\text{"Expr"}, [(\text{"ppStyle"}, E \text{ } tp_Int)]) \\ &\quad , (\text{"Factor"}, [(\text{"ppStyle"}, E \text{ } tp_Int)]) \\ &\quad , (\text{"Term"}, [(\text{"ppStyle"}, E \text{ } tp_Int)]) \\ &\quad] \end{aligned}$$

Consider the syntax macro (that does not use any inherited attributes)

```
Factor ::= "simple" x=Factor => x;
          pp = Text "simple" >#< x.pp;
```

For the part `Text "simple" >#< x.pp`, the following code is generated (after down casting and type checking):

$$pp_redef = \lambda ppStyle \rightarrow \text{text "simple"} \gg\ll \text{select_expr_pp } x \text{ } ppStyle$$

The variable `pp_redef` is introduced for notational reasons and should be replaced with its right hand-side. Combined with the inserter `insert_expr_pp`, this becomes

$$\begin{aligned} \lambda ppStyle \rightarrow \mathbf{let} (ast, pp) = x \text{ } ppStyle \\ \mathbf{in} (ast, pp_redef \text{ } ppStyle) \end{aligned}$$

Finally, the variable `x :: Expr_attrs`, that corresponds to the nonterminal of the concrete syntax and hence is of function type, is introduced by the function `buildLambda` in the macro interpreter, which completes the semantic function for the new parser alternative by adding the abstraction part:

$$\begin{aligned} \lambda x \rightarrow \lambda ppStyle \rightarrow \mathbf{let} (ast, pp) = x \text{ } ppStyle \\ \mathbf{in} (ast, pp_redef \text{ } ppStyle) \end{aligned}$$

Note that there are two abstractions of `ppStyle`, but that these are not conflicting.

Sadly, serious problems can be announced. Although there is silent support for inherited attributes, there is no way to use the actual values as defined by the original attribute grammar. Furthermore, inherited attributes only used for the children (for instance attributes not inherited by the parent), cannot be accessed at all. As a consequence, the so called silent support cannot be called support at all: only inherited attributes that are unchanged are supported, and this is rarely the case. The trivial, but true conclusion is that attributes that are to be used, need to be accessed. Only the argument (i.e. the inherited attribute) and

the result of the function can be accessed and changed, but not the inherited attributes and results for the children. At the moment, the solution is either to rebuild the complete **let**-construction, as produced by the AG system, dynamically, or to attack the problem at its root. The first alternative roughly is a dynamic attribute grammar system, which is clearly not the goal, while the second alternative may result in a more elegant solution. An aspect-oriented approach to attribute grammars may help, as will be described in the following section.

3.5 Attribute Grammar Combinators

3.5.1 Introduction

The problem as it is encountered in the previous section can be summarized as follows: one can change the input of a function from the outside, one can change the output of a function from the outside, but one cannot change the semantics of the function itself. What is needed is a way to manipulate a specific part of the function. This section will discuss an aspect-oriented approach to the attribute grammar system, and how this approach can be used to solve the problem just described.

3.5.2 Attribute Grammars vs. Aspect-oriented Approach

The attribute grammar system [5] as it is known takes a data type definition (that models the underlying context-free syntax) and the semantics for the attributes, and constructs the semantic functions as folds over the data type to compute the attributes. These semantic functions are defined by the production rules. As a consequence, modifications can only be made by scanning the complete grammar, which is error-prone. Another drawback is the problem of the inherited attributes as explained above. Instead of focusing on the computation per production rule, the aspect-oriented approach of De Moor, Peyton-Jones, Van Wyk and Swierstra [6] focuses on attribute computations. The result is compositionality of attribute computations: because the attribute computations are now isolated, they can be combined into the semantic function. Moreover, modifications can be made simply by changing the isolated computation of the attribute concerned.

Attribute Grammars

The traditional example of the *repmim* problem [5] will be used throughout this section to explain the new approach. The *repmim* problem is defined for binary trees:

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

and consists of producing a tree of the same shape with all leaf-values replaced by the minimum element of the tree. Using attribute grammars⁴, the solution can be formulated by first defining the abstract data type:

```
DATA Root | Root tree:Tree
DATA Tree | Node le:Tree ri:Tree
           | Leaf i:Int
```

⁴For a complete manual of the AG system being developed at the Utrecht University, see [13]

The three attributes can be defined by specifying the computation per constructor:

```

ATTR Root [|sres:{Tree}]
ATTR Tree [imin:{Int}||smin:{Int},sres:{Tree}]

SEM Root | Root lhs.sres = @tree.sres
          tree.imin = @tree.smin

SEM Tree | Node lhs.sres = Node @le.sres @ri.sres
          lhs.smin = min @le.smin @ri.smin
          le.imin = @lhs.imin
          ri.imin = @lhs.imin
          Leaf lhs.sres = Leaf @lhs.imin
          lhs.smin = @i

```

with two synthesized attributes (the minimum element and the resulting tree) and one inherited attribute (the minimum element). The attribute grammar system [16] creates the following Haskell code for this attribute grammar:

```

sem_Root (Root tree) =
  let (smin, sres) = (sem_Tree tree) smin
  in (sres)

sem_Tree (Node le ri) =
  \imin → let (lmin, lres) = (sem_Tree le) imin
             (rmin, rres) = (sem_Tree ri) imin
  in (min lmin rmin, Node lres rres)

sem_Tree (Leaf i) =
  \imin → (i, Leaf imin)

```

Attribute grammars offer syntactic compositionality, because the specification of the computation of the attributes can be separated. However, the generated definition of the attribute grammar in the target language (Haskell), still lacks this compositionality into aspects: all computations are located in one function. To make full use of all features of the target language, such as type-checking and the richness of the expression language, semantic compositionality is desired, that is, isolation of the computation per attribute. Semantic compositionality can be achieved by introducing a number of abstractions.

3.5.3 Abstracting from the Structure and Attribute Computations

To abstract from the structure, a slight change of view must be made. Considering a production rule $P \rightarrow C_1 \dots C_k$, the computed attributes are the synthesized attributes of P and the inherited attributes of $C_1 \dots C_k$. For these computations, the inherited attributes of P and the synthesized attributes of $C_1 \dots C_k$ are used. In other words, the complete computation can be seen as a function from input attributes to output attributes. The input attributes are the inherited attributes of P and the synthesized attributes of its children, $C_1 \dots C_k$. The output attributes are the synthesized attributes of P and the inherited attributes of $C_1 \dots C_k$.

Interpreting the attribute computations as such, the semantic functions that define the catamorphism do not longer have to depend on the specific shape of the production rule and

the attribute computations. Having the attribute computations for a constructor C available as cf , with c to be interpreted as the name of the constructor in lower-case letters ($rootf$, $nodef$ and $leaff$ respectively), and a function $knitj$ ($j > 0$) that takes care of supplying the computations with the correct arguments, the catamorphism can be redefined as

$$\begin{aligned} sem_Root (Root\ tree) &= knit1 (rootf\ ()) (sem_Tree\ tree) \\ sem_Tree (Node\ le\ ri) &= knit2 (nodef\ ()) (sem_Tree\ le) \\ &\quad (sem_Tree\ ri) \\ sem_Tree (Leaf\ i) &= knit1 (leaff\ ()) (\lambda_ \rightarrow i) \end{aligned}$$

Note that the semantic functions now only depend on the number of children of the constructor. The attribute computations $rootf$, $nodef$ and $leaff$, which are functions from the computation input (the inherited attributes of the parent and the synthesized attributes of the children) to the computation output (the inherited attributes of the children and the synthesized attributes of the parent), will be studied in detail in the rest of this section. The argument $()$ will be discussed as well.

The $knit$ functions generalize the attribute flow, and thus depend on the shape of the production rules; it takes the attribute computation of the parent and the semantic functions of the children, and it results in the classic semantic function of the inherited attributes of the parent to the synthesized attributes of the parent. The shape of a production rule is defined by the number of children of the parent. Consequently, the number of production rules with different numbers of children defines the number of $knit$ functions (in Section 3.5.7 a more sophisticated approach will be described). For binary trees, there is a $Root$ with one child, a $Node$ with two children and a $Leaf$ with one child; two $knit$ functions are needed, $knit1$ for a parent with one child, $knit2$ for a parent with two children. To be complete, also $knit0$ for a parent with no children is given:

$$\begin{aligned} knit0\ f &= \lambda pi \rightarrow \mathbf{let}\ def = pi \\ &\quad po = f\ def \\ &\quad \mathbf{in}\ po \\ knit1\ f\ c &= \lambda pi \rightarrow \mathbf{let}\ co = c\ ci \\ &\quad (ci, po) = f\ (co, pi) \\ &\quad \mathbf{in}\ po \\ knit2\ f\ l\ r &= \lambda pi \rightarrow \mathbf{let}\ lo = l\ li \\ &\quad ro = r\ ri \\ &\quad (li, ri, po) = f\ (lo, ro, pi) \\ &\quad \mathbf{in}\ po \end{aligned}$$

The function f is the computation that transforms the input attributes to output attributes. Thanks to lazy evaluation, the inherited attributes of the children (li and ri) can be computed and passed to the semantic functions for the children (l and r) to compute the synthesized attributes of the children (lo and ro). Finally, these synthesized attributes can be used together with the inherited attributes of the parent (pi) to compute the synthesized attributes of the parent (po). The final result is thus the classic semantic function from inherited attributes to synthesized attributes. The idea is depicted graphically in Figure 3.1.

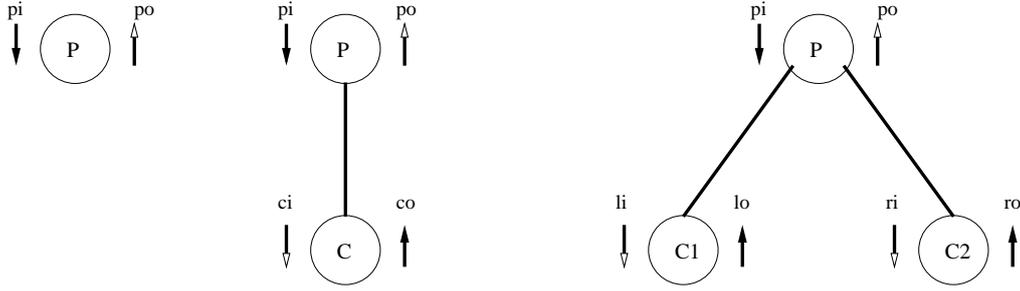


Figure 3.1: The idea behind the *knit* functions is that the computation for a parent and children takes as input the inherited attributes of the parent and the synthesized attributes of the children (arrows with filled heads). The output of the computation is formed by the inherited attributes of the children and the synthesized attributes of the parent (arrows with open heads). It is the task of *knit* to supply the semantic functions of the children with their inherited attributes, and to supply the attribute computation with the synthesized attributes of the children and the inherited attributes of the parent.

The actual attribute computations have to be defined in terms of the computation input (the inherited attributes of the parent and the synthesized attributes of the children). The types corresponding to the inherited and synthesized attributes for the *repmim* problem are

```
type Tree_Inh = (Int, ())
type Tree_Syn = (Int, (Tree, ()))
```

Notice that *nested* cartesian products are used, and that there is also a unit element (). The reason for using nested products will become clear in Section 3.5.7. With these types, the types for the computation input and output for *Node* can be defined as

```
type Node_In = (Tree_Syn, Tree_Syn, Tree_Inh)
type Node_Out = (Tree_Inh, Tree_Inh, Tree_Syn)
```

For *Node_In*, there are two values of type *Tree_Syn*, the synthesized attributes of its both children, and one value of *Tree_Inh*, its inherited attributes. *Node_Out* is defined similarly: two values of type *Tree_Inh*, the inherited attributes of the children, and one value of type *Tree_Syn*, its synthesized attributes. The function *nodef* contains the computations for the attributes *imin* of its both children, and of its synthesized attributes *smin* and *sres*:

```
nodef :: Node_In → Node_Out
nodef (lo@(lsmim, (lsres, ())), ro@(rsmin, (rsres, ())), pi@(imin)) =
    ((imin, ())
     , (imin, ()))
     , (lsmim lsmim rsmin
        , (Node lsres rsres
           , ()))
     )
    )
```

The first two elements of the result are the inherited attributes of the children, implementations of the copy rule. The third element is a product with the computations of the synthesized

attributes. In a similar way, the attribute computations for the other production rules can be defined:

$$\begin{aligned}
 \mathit{rootf} \ (co@(smin, (sres, ())), pi@(imin)) &= \\
 &\quad ((imin, ()), \\
 &\quad \quad (sres, ())) \\
 &\quad) \\
 \mathit{leaff} \ (i, pi@(imin)) &= \\
 &\quad ((\\
 &\quad \quad (i \\
 &\quad \quad \quad (Leaf \ iimin, ()))) \\
 &\quad) \\
 &\quad)
 \end{aligned}$$

The single unit value $()$ is used because the child of *Leaf* does not have any inherited attributes, and hence the computation is empty.

3.5.4 Separating the Aspects

The attribute computations can be refined further if its aspects can be separated. The four aspects of *nodef* are the distribution of *imin* to the left and to the right child, the computation of *smin* and the computation of *sres*. What happens essentially, is the extension of a nested cartesian product with a new element (the distribution of the inherited attribute for the children), or two new elements (the computation of the synthesized results). Such an extension can be expressed as

$$\lambda product \rightarrow (new_comp, product)$$

and using this λ -abstraction, the aspect of distributing *imin* to the left child of *node* can be separated as

$$\begin{aligned}
 \mathit{nodef_imin_li} \ (lo, ro, pi@(imin, ())) &= \\
 \lambda(li, ri, po) \rightarrow ((\lambda v \rightarrow (imin, v)) \ li, ri, po)
 \end{aligned}$$

The other aspects can be handled in a similar way:

$$\begin{aligned}
 \mathit{nodef_imin_ri} \ (lo, ro, pi@(imin, ())) &= \\
 \lambda(li, ri, po) \rightarrow (li, (\lambda v \rightarrow (imin, v)) \ ri, po) \\
 \mathit{nodef_smin_po} \ (lo@(lsmmin, (lsres, ())), ro@(rsmmin, (rsres, ())), pi) &= \\
 \lambda(li, ri, po) \rightarrow (li, ri, \\
 &\quad (\lambda v \rightarrow (min \ lsmmin \ rsmmin, v) \ po)) \\
 \mathit{nodef_sres_po} \ (lo@(lsmmin, (lsres, ())), ro@(rsmmin, (rsres, ())), pi) &= \\
 \lambda(li, ri, po) \rightarrow (li, ri, \\
 &\quad (\lambda v \rightarrow (Node \ lsres \ rsres, v) \ po))
 \end{aligned}$$

Note that initially, a unit value $()$ is passed to the product extension functions.

The computation output (li, ri, po) is computed partially by each of the separated computations for the aspects. Because the computations are functions that take a product (or better, a product containing nested products), and result in a product, they can be composed with the following operator:

$$\begin{aligned}
 \mathit{ext} \ :: \ (a \rightarrow b \rightarrow c) \rightarrow (a \rightarrow d \rightarrow b) \rightarrow a \rightarrow d \rightarrow c \\
 f \ 'ext' \ g = \lambda input \rightarrow f \ input \circ g \ input
 \end{aligned}$$

The original *nodef* function, containing all computations can be restored by combining the aspect computations:

$$\text{nodef } () = \text{nodef_imin_li 'ext' nodef_imin_ri 'ext' nodef_smin_po 'ext' nodef_sres_po 'ext' emptyProd}$$

$$\text{emptyProd } _ = (((), ()), (), ())$$

where the *emptyProd* function takes care of the initial products, which are unit values. These unit values are extended to nested cartesian products by the functions that define the aspects (*nodef_...*). The shape of *emptyProd* is the same as the shape of the computation output.

3.5.5 A More General *knit*

By grouping the arguments of the children in a production rule (*li* and *ri*, *lo* and *ro*) in a cartesian product, a more general *knit* can be defined:

$$\begin{aligned} \text{knit } kn \ f \ ch_fn \ pi = \\ \mathbf{let} \ (ch_arg, loc, po) = f \ (kn \ ch_fn \ ch_arg, loc, pi) \\ \mathbf{in} \ po \end{aligned}$$

$$\begin{aligned} kn_0 \ f \ () = f \ () \\ kn_1 \ f \ a = f \ a \\ kn_2 \ \sim (f, g) \ \sim (a, b) = (f \ a, g \ b) \end{aligned}$$

The *knit* function now abstracts from the number of children; the *kn_i* functions take care of that. Note that lazy pattern matching (\sim) is used. Local arguments are introduced as a nested cartesian product (*loc*). The function *kn₀* for a parent with zero children takes $()$ as argument because the result of its children is empty. In Section 3.5.7, this will be refined a little bit more.

Another generalization that can be made is for the definition of inherited and synthesized attributes in computations such as *nodef_imin_li*. The structure of those functions is always the same, and can thus be generalized in a new function. With *f* the function that inserts the new attribute into the cartesian product, the inherited attribute of the first child of a parent with one child can be defined with

$$\text{def_1_1} = \lambda f \ (ch1, loc, po) \rightarrow (f \ ch1, loc, po)$$

and for the first and second child of a parent with two children by

$$\begin{aligned} \text{def_2_1} = \lambda f \ ((ch1, ch2), loc, po) \rightarrow ((f \ ch1, ch2), loc, po) \\ \text{def_2_2} = \lambda f \ ((ch1, ch2), loc, po) \rightarrow ((ch1, f \ ch2), loc, po) \end{aligned}$$

Generally, the function *def_n_k* will define the inherited attributes of the *k*-th child of a parent with *n* children. In Section 3.5.7, the special case of a constructor with zero children will be described.

The definition of synthesized attributes of the parent is even shorter, because only one element is changed:

$$\text{def_syn} = \lambda f \ (chs, loc, po) \rightarrow (chs, loc, f \ po)$$

Local arguments can be defined in a similar way:

$$\text{def_loc} = \lambda f (chs, loc, po) \rightarrow (chs, f\ loc, po)$$

The attribute computations must be redefined in their input arguments and can use the new definition functions:

$$\begin{aligned} \text{nodef_imin_li} ((lo, ro), loc, pi@(imin, ())) &= \\ &\text{def_2_1} (\lambda v \rightarrow (imin, v)) \\ \text{nodef_imin_ri} ((lo, ro), loc, pi@(imin, ())) &= \\ &\text{def_2_2} (\lambda v \rightarrow (imin, v)) \\ \text{nodef_smin_po} ((lo@(lsmín, (lsres, ())), ro@(rsmin, (rsres, ())), loc, pi) &= \\ &\text{def_syn} (\lambda v \rightarrow (\text{min } lsmín\ rsmin, v)) \\ \text{nodef_sres_po} ((lo@(lsmín, (lsres, ())), ro@(rsmin, (rsres, ())), loc, pi) &= \\ &\text{def_syn} (\lambda v \rightarrow (\text{Node } lsres\ rsres, v)) \\ \text{leaff_smin_po} ((i, ()), loc, pi) &= \\ &\text{def_syn} (\lambda v \rightarrow (i, v)) \\ \text{leaff_sres_po} ((i, ()), loc, pi@(imin, ())) &= \\ &\text{def_syn} (\lambda v \rightarrow (\text{Leaf } imin, v)) \\ \text{rootf_imin_ci} (co@(smin, (sres, ())), loc, pi) &= \\ &\text{def_1_1} (\lambda v \rightarrow (smin, v)) \\ \text{rootf_sres_po} (co@(smin, (sres, ())), loc, pi) &= \\ &\text{def_syn} (\lambda v \rightarrow (sres, v)) \end{aligned}$$

The final compositions are constructed as follows:

$$\begin{aligned} \text{nodef} () &= \text{nodef_imin_li} \text{'ext'} \text{nodef_imin_ri} \text{'ext'} \text{nodef_smin_po} \text{'ext'} \\ &\quad \text{nodef_sres_po} \text{'ext'} \text{emptyProdNode} \\ \text{leaff} () &= \text{leaff_smin_po} \text{'ext'} \text{leaff_sres_po} \text{'ext'} \text{emptyProdLeaf} \\ \text{rootf} () &= \text{rootf_imin_ci} \text{'ext'} \text{rootf_sres_po} \text{'ext'} \text{emptyProdRoot} \\ \\ \text{emptyProdRoot} _ &= ((), (), ()) \\ \text{emptyProdLeaf} _ &= ((), (), ()) \\ \text{emptyProdNode} _ &= (((), ()), (), ()) \end{aligned}$$

The catamorphisms use the composed computations and the general *knit* functions:

$$\begin{aligned} \text{sem_Root} (\text{Root } tree) &= \text{knit } kn1 (\text{rootf} ()) (\text{sem_Tree } tree) \\ \text{sem_Tree} (\text{Node } le\ ri) &= \text{knit } kn2 (\text{nodef} ()) (\text{sem_Tree } le, \\ &\quad \text{sem_Tree } ri) \\ \text{sem_Tree} (\text{Leaf } i) &= \text{knit } kn1 (\text{leaff} ()) (\lambda_ \rightarrow (i, ())) \end{aligned}$$

Note that for the semantic function of *Leaf*, the third argument to *knit* is $(\lambda_ \rightarrow (i, ()))$. The only synthesized attribute of a *Leaf*'s child is the value *i*; this value must be tupled with $()$, and not just *i*, to make a difference between no attributes $()$ and an empty attribute $()$ (that would result in $((), (()))$).

The argument $()$ to the attribute computations *rootf*, *nodef* and *leaff* is a (nasty) workaround to avoid unresolved top-level overloading. Discarding this argument requires an additional *Int* ascription in the definition of *nodef_smin_po*, because of the type of *min*: $\text{Ord } a \Rightarrow a \rightarrow a \rightarrow a$. Annotating *nodef* with the correct type signature also solves the problem⁵.

⁵In the revised Haskell 98 Report this phenomenon is called monomorphic restriction [11].

3.5.6 Copy Rules

Top-down copy rules are defined as computations that pass inherited attributes from parent to child. Because this is a common pattern in attribute grammars, inherited attributes of a parent are by default copied to the children. The *knit* function defines the flow of computations; consequently, the *knit* function must be changed to implement default top-down copy rules:

```

knit kn f ch_fn pi =
  let (ch_arg, loc, po) = f (ch_res, loc, pi) initProd
      (ch_res, initProd) = kn ch_fn ch_arg pi
  in po

kn0 f () pi = (f (), ((), (), ()))
kn1 f a pi = (f a, (pi, (), ()))
kn2 ~ (f, g) ~ (a, b) pi = ((f a, g b), ((pi, pi), (), ()))

```

The *kn* functions implement the copy rules by constructing an initial cartesian product that will be passed to the attribute computations as the initial computation output. The *kn* function knows the number of children (*kn1* is the function for a parent with one child, *kn2* for a parent with two children) and hence knows the shape of the cartesian product for the initial product. The elements of this product are the inherited attributes *pi* of the parent, that replace the empty products $()$ and $((), ())$ for one and two children respectively. The initial product also stores the initial local definitions $(())$ and the initial synthesized attributes of the parent $(())$. It must be noted that *all* inherited attributes of the parent are copied to its children, even if they are not used (or specified by the attribute grammar).

Another advantage of this definition of *knit* is that the compositions do not need to be initialized with *emptyProds* anymore:

```

nodef () = nodef_smin_po 'ext' nodef_sres_po
leaff () = leaff_smin_po 'ext' leaff_sres_po
rootf () = rootf_imin_ci 'ext' rootf_sres_po

```

3.5.7 A Generic *knit*

The first argument of *knit* is a *kn* function, that only depends on the number of children of a constructor. The *kn* function takes a cartesian product containing the catamorphisms for the children and a cartesian product of the same shape containing the inherited attributes for the children. It results in a function that, given the inherited attributes of the parent, computes the synthesized attributes of the children (and constructs the initial product). This functionality, neglecting the top-down copy rules for the moment, can be captured in a class definition:

```

class Knit a b c d | a → b c d where
  kn :: a → b → (c, d)

```

The type signature of the function *kn* exactly shows the desired form: given a structure with the catamorphisms for the children (type variable *a*), and the input for the children (*b*), compute the results for the children (*c*) and the initial product of type *d*. The functional

dependency $a \rightarrow b\ c\ d$ is needed to help the Haskell type-checker to resolve ambiguities: a determines b , c and d . One can say that the shape of b , c and d depends on a , rather than the type. However, in Haskell this can only be expressed by a functional dependency.

The kn function for one child can now be defined as

```
instance Knit (a → b) a b () where
  kn f a = (f a, ())
```

The kn function for more than one child is little bit more complicated. The catamorphisms and arguments are stored in a nested cartesian product, but the shape is unknown. However, the first element of the product is a single element, and can thus be used. For the remainder of the nested product, kn is called recursively. The class system guarantees that the correct instance is chosen. Note that the initial product $empties$ is built together with the result, and is a nested cartesian product of the same shape.

```
instance Knit d e f g ⇒ Knit (a → b, d) (a, e) (b, f) ((), g) where
  kn ~ (a2b, d) ~ (a, e) = let (f, empties) = kn d e
                        in ((a2b a, f), ((), empties))
```

Finally, the $knit$ function becomes

```
knit f ch_fn = λpi → let (ch_arg, loc_args, po) = f (ch_res, loc_args, pi) (empties, (), ())
                        (ch_res, empties) = kn ch_fn ch_arg
                        in po
```

The nested product $empties$ keeps track of the number of children of the parent: for each call to kn , $empties$ is extended with a new $()$.

The catamorphisms become a bit more general because the kn functions are now determined by the class system:

```
sem_Root (Root tree) = knit (rootf ()) (sem_Tree tree)
sem_Tree (Node le ri) = knit (nodef ()) (sem_Tree le,
                               sem_Tree ri)
sem_Tree (Leaf i) = knit (leaff ()) (λ() → (i, ()))
```

The case of a parent with zero children can also be handled by the **class** $Knit$. Just like zero inherited attributes is represented by the empty product $()$, zero children can also be represented by $()$. If for example an $EmptyLeaf$ is added to $Tree$, computations for the synthesized attributes become

```
emptyleaff_smin_po ((), loc, pi) =
  def_syn (λv → (0, v))
emptyleaff_sres_po ((), loc, pi@(imin, ())) =
  def_syn (λv → (EmptyLeaf, v))
```

The catamorphism becomes

```
sem_Tree (EmptyLeaf) = knit (emptyleaff ()) (λ() → ())
```

Notice the subtle difference with the definitions for *Leaf*, which has one child: the second argument of the call to *knit* is $\lambda() \rightarrow ()$, while for *Leaf* the argument is $\lambda() \rightarrow (i, ())$. In other words, $()$ is used to represent the result of a constructor with zero children. The difference between constructors with zero and one child respectively cannot be based on the structure of *empties*.

As a final remark, it should be mentioned that the nested products containing the attributes always have $()$ as deepest element, but that the nested products containing *those* attribute products have not: contrary to the attributes, the children of a constructor are never redefined or extended, and thus the product is fixed. As a consequence, the *def_n_k* functions that define inherited attributes for the *k*-th child of a constructor with *n* children, should use this nested structure for $n > 2$. For example:

$$\text{def_3_1} = \lambda f ((\text{chin1}, (\text{chin2}, \text{chin3})), l, p) \rightarrow ((f \text{ chin1}, (\text{chin2}, \text{chin3})), l, p)$$

3.6 Inherited Attributes: Solution

3.6.1 Rewriting the Semantic Functions

The semantic functions as defined by the attribute grammar are computed by knitting the attribute computations to the semantic functions of the children. These attribute computations are generated out of the attribute grammar by the AG system. A summary of the changes and extensions to the AG system can be found in Appendix A.

The general structure of the catamorphism for a constructor *C* with *k* children *c_i* becomes

$$\text{sem}_C (C \ c_1 \dots \ c_k) = \text{knit} (cf \ ()) (\text{sem}_{c_1} \ c_1, (\text{sem}_{c_2} \ c_2, (\dots, \text{sem}_{c_k} \ c_k)))$$

3.6.2 Attributes Revisited

Setting the Goal

Because of the compositionality of attribute computations, the structure of the semantic functions has changed. However, the philosophy used in Section 3.3 for the manipulation of synthesized attributes remains unaltered; still functionality is needed to select, redefine and insert synthesized and inherited attributes. The actual changes are in the implementation. This boils down to some subtle composition of selection functions.

The attribute computation for the attribute *attr* of constructor *C* (of data type *D*) is parameterized by a 3-tuple (*children_out*, *local*, *parent_in*), where *children_out* is a nested cartesian *k*-product containing the synthesized attributes of each of its children *c_1* ... *c_k* (in nested cartesian products as well). The nested cartesian product *local* contains the local definitions. The nested cartesian *m*-product *parent_in* contains the inherited attributes of *C*. This 3-tuple will be referred to as the *computation input*, because the synthesized attributes of the children and the inherited attributes of the constructor are the input for the attribute computations. For example, the computation input of *Mul* (from the *Expr* language), with one inherited attribute *ppStyle* :: *Int*, no local attributes and two synthesized attributes *ast* :: *Expr* and *pp* : *PP_Doc*, has the following type signature:

$$\begin{aligned} \text{type } \text{Mul_input} = & (((\text{Expr}, (\text{PP_Doc}, ())), (\text{Expr}, (\text{PP_Doc}, ()))) \\ & , () \\ & , (\text{Int}, ())) \end{aligned}$$

for a constructor C of data type D , with n synthesized attributes and m inherited attributes. Constructor C has k children of type D_i for $i = 1 \dots k$.

```

type  $D\_attrs\_syn = (D\_Syn\_1, (\dots, (D\_Syn\_n, ())))$     -- synthesized attributes
type  $D\_loc = (D\_Loc\_1, (\dots, (D\_Loc\_l, ())))$     -- local definitions
type  $D\_attrs\_inh = (D\_Inh\_1, (\dots, (D\_Inh\_m, ())))$     -- inherited attributes
type  $D\_attrs = D\_attrs\_inh \rightarrow D\_attrs\_syn$     -- semantics
type  $D\_C\_input = (D\_C\_children\_out$     -- attr computation input
                  ,  $D\_loc$ 
                  ,  $D\_attrs\_inh$ )
type  $D\_C\_output = (D\_C\_children\_in$     -- attr computation output
                  ,  $D\_loc$ ,
                  ,  $D\_attrs\_syn$ )
type  $D\_C\_initial = (((() \_1, ((() \_2, (\dots, () \_k))))$ 
                  ,  $()$ 
                  ,  $()$ 
                  )
type  $D\_C\_children\_in$ 
    =  $(D\_1\_attrs\_inh, (\dots, (D\_k\_attrs\_inh)))$     -- inh attrs of children
type  $D\_C\_children\_out$ 
    =  $(D\_1\_attrs\_syn, (\dots, (D\_k\_attrs\_syn)))$     -- syn attrs of children

```

It must be noted that local attributes are not supported. The problem is that the types of the local attributes that are needed for the type synonyms are not available (contrary to the types of the inherited and synthesized attributes). Nevertheless, if the AG system is changed to require type annotations for local attributes as well, this problem can be solved easily by treating local attributes the same as inherited and synthesized attributes.

The type synonym for the attribute computation function needs some additional explanation. It takes the unit element, the computation input and a nested cartesian product of $()$'s, that has the same structure as the result (the computation output). The attribute computation uses the computation input to compute the attribute, and extends the result sofar with the new attribute. Therefore the products should be $()$ initially: the attribute computations construct the nested cartesian products by extending the $()$'s.

```

type  $D\_Cf =$     -- attribute computation function
     $() \rightarrow$ 
     $D\_C\_input \rightarrow$ 
     $D\_C\_initial \rightarrow$ 
     $D\_C\_output$ 

```

Selection Functions

To access elements in the computation input, especially synthesized and inherited attributes, selection functions for the following elements are needed:

- The output of the children of C as a whole:

```

select_D_C_children ::  $D\_C\_input \rightarrow D\_C\_children\_out$ 
select_D_C_children (ch, loc, inh) = ch

```

- The output of child i of C :

$$\begin{aligned} \text{select_D_C_child_i} &:: D_C_input \rightarrow D_C_children_out_i \\ \text{select_D_C_child_i} &((c_out_1, (\dots, (c_out_i, (\dots, (c_out_k))))), loc, inh) = c_out_i \end{aligned}$$

where $D_C_children_out_i$ is the projection on the i -th element of $D_C_children_out$. Note that the result is a nested cartesian product that contains the synthesized attributes of the child.

- The synthesized attributes of each data type D :

$$\begin{aligned} \text{select_D_syn_attr_i} &:: D_attrs_syn \rightarrow D_Syn_i \\ \text{select_D_syn_attr_i} &(d_syn_1, (\dots, (d_syn_i, (\dots, (d_syn_n, ()))))) = d_syn_i \end{aligned}$$

- The tuple of inherited attributes out of the computation input for C :

$$\begin{aligned} \text{select_D_C_inherited} &:: D_C_input \rightarrow D_attrs_inh \\ \text{select_D_C_inherited} &(ch, loc, inh) = inh \end{aligned}$$

- The inherited attributes of each data type D :

$$\begin{aligned} \text{select_D_inh_attr_i} &:: D_attrs_inh \rightarrow D_Inh_i \\ \text{select_D_inh_attr_i} &(d_inh_1, (\dots, (d_inh_i, (\dots, (d_inh_m, ()))))) = d_inh_i \end{aligned}$$

If for example the second synthesized attribute of the first child (of data type $D1$) of C (of data type D) is needed, the following composition of selection functions can be applied to the computation input:

$$\text{select_D1_syn_attr_2} \circ \text{select_D_C_child_1}$$

Computation Input

The selection functions have to be applied to the computation input. This means that the computation input for each of the constructors must be available while parsing the syntax macros. More specifically, a unique variable name for the computation input, simply the presentation of its type (" D_C_input "), and its type (D_C_input) are needed to use it dynamically.

Insertion Functions

New computations for attributes have to be combined with the existing computations for the other attributes. This is done by extending the existing computations with the new computation from the left, i.e. the existing computation is overwritten. It is stressed that computations must be extended from the left, because the nested cartesian product must have been built to its final shape first by the initial computation. Insertion functions are needed for synthesized attribute definitions and for inherited attribute definitions of the children:

- The synthesized attributes of a datatype constructor:

$$\text{insert_D_C_syn_attr_i} :: (D_C_input \rightarrow D_Syn_i) \rightarrow D_Cf \rightarrow D_Cf$$

$$\begin{aligned} \text{insert_D_C_syn_attr_i } new_comp \text{ orig_comp} &= \lambda() \rightarrow \text{extend 'ext' (orig_comp ())} \\ \text{where extend comp_input} &= \\ &\text{def_syn } (\lambda v \rightarrow \text{insert_D_syn_i (new_comp comp_input) v}) \end{aligned}$$

Note that computations and computation inputs vary per constructor, and thus the insertion functions have to be specified per constructor. Essentially, the function replaces the computation of attribute i by extending (overwriting) the original computation $orig_comp$ of all attributes with the new computation for that specific attribute: the definition of $extend$ by using def_syn is similar to initial computations such as $nodef_smin_po$, except that an element in the resulting product is replaced instead of that a product is extended. Note also that the new attribute computation new_comp expects the computation input as an input argument, and hence must be applied to the computation input ($comp_input$).

- The inherited attributes (in a nested cartesian product of type D_j_Inh) of the children of a constructor can be redefined in a similar way. The function for replacing the computation of the i -th inherited attribute of the j -th child becomes:

$$\begin{aligned} \text{insert_D_C_inh_attr_i_c_j} &:: (D_C_input \rightarrow D_j_Inh_i) \rightarrow D_Cf \rightarrow D_Cf \\ \text{insert_D_C_inh_attr_i_c_j } new_comp \text{ orig_comp} &= \\ &\lambda() \rightarrow \text{extend 'ext' (orig_comp ())} \\ \text{where extend comp_input} &= \\ &\text{def_k_j } (\lambda v \rightarrow \text{insert_D_j_Inh_i (new_comp comp_input) v}) \end{aligned}$$

where k is the number of children of C .

The insertion functions make use of helper functions that insert synthesized and inherited attributes into the tuple containing the other attributes:

$$\begin{aligned} \text{insert_D_syn_i} &:: D_Syn_i \rightarrow D_attrs_syn \rightarrow D_attrs_syn \\ \text{insert_D_syn_i } syn \text{ (syn_1, (... (syn_i, (... (syn_n, ())))))} &= \\ \text{(syn_1, (... (syn, (... (syn_n, ())))))} & \end{aligned}$$

$$\begin{aligned} \text{insert_D_inh_i} &:: D_Inh_i \rightarrow D_attrs_inh \rightarrow D_attrs_inh \\ \text{insert_D_inh_i } inh \text{ (inh_1, (... (inh_i, (... (inh_m, ())))))} &= \\ \text{(inh_1, (... (inh, (... (inh_m, ())))))} & \end{aligned}$$

Original Computation

The new attribute computations have to overwrite the original computations of the attributes. This means that the original computation of all attributes, $d_cf :: D_Cf$, has to be available as a *Dynamic* value. It is this computation that will be used as the initial argument ($orig_comp$) to the $insert_D_C_inh_attr$ and $insert_D_C_syn_attr$ functions, that extend (replace) a given computation with a new computation.

Semantic Function Constructors

Given the new attribute computations, the new semantic function that takes the semantics for the children and results in a function of type D_attrs , can be constructed by knitting the new attribute computations to those of the children:

$$\begin{aligned}
& \text{make_sem_D_C} :: D_Cf \rightarrow \\
& \quad D_C_1_attrs \rightarrow \dots \rightarrow D_C_k_attrs \rightarrow D_attrs \\
& \text{make_sem_D_C} \text{ new_comp} = \lambda c_1 \dots c_k \rightarrow \text{knit} (\text{new_comp } ()) (c_1, (\dots, c_k))
\end{aligned}$$

where $D_C_i_attrs$ is the type of the semantics of the i -th child of C .

Note that the constructed catamorphism differs slightly from the catamorphisms that are used in the explanation of **class** *Knit* (Section 3.5): the final result is the semantic function of a parser and it is thus not necessary to build (and destroy) the abstract syntax tree explicitly.

Recall that *knit* can handle nested cartesian products with $k \geq 0$ children: the **class** *Knit* chooses the correct instance.

3.6.3 Generated Code

Using the prototypes of the previous section, it is straightforward to generate code out of the AG definition.

Additionally, the functions, types and variables names as described in the previous section have to be available dynamically. This is done by creating associated lists. For selection functions for the computation input, additional information about the type of the selection is returned, for instance the type of the selected child. This information will be used for looking up selection functions to be applied to that selection, such as attribute selectors. In the remainder of this subsection, "to extend" in the context of concatenating strings should be read as to concatenate two strings with an underscore as separator.

Input selectors select either a product with the synthesized attributes of a child or the product with the inherited attributes out of the computation input. The key is the name of the constructor extended with either the name of the child or "lhs". The associated element is a tuple with the name of the type of the child (or the constructor corresponding to the parent in case of "lhs"), and the selector as a *Dynamic* value.

$$\text{input_selectors} :: [(String, (String, Dynamic (Type TypeCon) Id))]$$

Attribute selectors select inherited and synthesized attributes out of a nested cartesian product. The key is the name of the datatype extended with the name of the attribute. The associated element is the selector as a *Dynamic* value.

$$\text{attr_selectors} :: [(String, Dynamic (Type TypeCon) Id)]$$

The computation input is a 3-tuple with the synthesized attributes of the children, the local attributes and the inherited attributes of the parent. The key is the name of the constructor. The associated element is a tuple with the variable name for the computation input (to be used during interpreting of the syntax macros) and the type representation of the computation input.

$$\text{comp_inputs} :: [(String, (String, Exists (Type TypeCon)))]$$

Attribute inserters redefine the original computation with computations for inherited and synthesized attributes. The key is the name of the constructor extended with the name of the attribute if a synthesized attribute is concerned. For an inherited attribute, the key is the name of the constructor extended with the name of the child and the name of the attribute. The associated element is the inserter.

$$\text{attr_inserters} :: [(String, Dynamic (Type TypeCon) Id)]$$

The original attribute computation functions are to be extended with new computations for the attributes, and thus have to be available dynamically. The key is the name of the constructor, the associated element is the attribute computation as a *Dynamic* value.

```
attr_computations :: [(String, Dynamic (Type TypeCon) Id)]
```

The semantic function constructors construct the new catamorphism using the new attribute computations. The key is the name of the constructor, the associated element is the semantic function constructor as a *Dynamic* value.

```
sem_constructors :: [(String, Dynamic (Type TypeCon) Id)]
```

Finally, the constructors and functions have to be available as well. The key is the name of the constructor or function, the associated value is the constructor or function respectively as a *Dynamic* value. Functions must be made available by the designer of the language, together with the types and the uppercase identifiers representing these functions in attribute redefinitions.

```
constructors :: [(String, Dynamic (Type TypeCon) Id)]
```

The operators are available in a similar way:

```
operators :: [(String, Dynamic (Type TypeCon) Id)]
```

During the discussion of the machinery in the following sections, these associated lists will be used.

3.6.4 Inherited Attributes Revisited

With the compositionality of attribute computations, the solution for supporting inherited attributes has become within reach. In fact, the only difference between handling synthesized and inherited attributes is the kind of selection and insertion functions used. However, to make inherited attributes available for syntax macro definitions, the macro language must be extended. This will be the main topic of the following subsections.

3.6.5 Extended Macro Language

The macro language as defined in 2.2.1 must be extended with concrete syntax for (1) the selection of synthesized and inherited attributes, (2) the redefinition of attributes and (3) the new combinators. Attributes can be identified by their name as defined in the attribute grammar; an attribute *a* of a child *c* can be identified by *c.a*. Note that the child and attribute names have to be exactly the same names as the names used in the attribute grammar. The combinators are available as operators, while functions on attributes should start with an uppercase letter. The rule for `Production` becomes:

```
Production ::= Nonterminal "==" RHSElem* "=>" Translation ";"
              (AttrDefExpression ";")*
```

The production for `RHSElem` remains unaltered. The `Translation` after the `"=>"` symbol is the translation to the abstract syntax of the underlying language.

```
Translation ::= "\\\" Varid "::" Type "." Translation
```

```
    | Factor+
```

```
Factor ::= Varid
        | Constructor
        | Literal
        | StringLiteral
        | "(" Translation ")"
```

The `AttrDefExpressions` are the redefinitions of the synthesized and inherited attributes. An `AttrDefExpression` consists of a `Lefthand`, that identifies the attribute to be redefined, and an `AttrExpression` for the redefinition.

```
AttrDefExpression ::= Lefthand "=" AttrExpression
```

A `Lefthand` identifies a synthesized attribute by `lhs`, and an inherited attribute by the name of the child.

```
Lefthand ::= LElement "." Attr
```

```
LElement ::= "lhs"
            | Varid
```

An `AttrExpression` consists of `AttrFactors`, of which `AttrSel` is the selection of an inherited attribute (`lhs`) or a synthesized attribute of the children.

```
AttrExpression ::= AttrFactor+
```

```
AttrFactor ::= Constructor
            | Function
            | AttrSel
            | Operator
            | "(" AttrExpression ")"
            | StringLit
            | Literal
```

```
AttrSel ::= RElement "." Attr
```

```
RElement ::= "lhs"
            | Varid
```

```
Attr ::= Varid
```

Functions, that have to be supplied by the designer of the language, are supposed to start with an uppercase identifier.

```
Function ::= Conid
Operator ::= OpSymbols+
```

An example macro definition, which redefines the pretty print attribute `pp`, now takes the form

```
Factor ::= "double" x=Factor => Mul (Constant 2) x ;
        pp = Text "double" >#< ri.pp ;
```

where `ri` is the identification of the right-hand child as defined by the attribute grammar. The function `Text` and the operator `>#<` are provided by the pretty print library as `text` and `>##<`, to print a string and to concatenate two `PP_Docs` using a space respectively. They have been made available for the syntax macro library by adding them to the function and operator lists as `("Text", Id text :: tp_String .->. tp_PP_Doc)` and `(">#<", Id (>##<) :: tp_PP_Doc .->. tp_PP_Doc .->. tp_PP_Doc)` respectively. The machinery behind the interpretation of the macros is described in the following section.

3.6.6 Compiler

Macro Scanner

Because additional operators may be used in the definition of the semantics, the scanner of the `UU_Parsing` library [14] must be instructed with the corresponding operator characters (via `specchars` and `opchars`).

Macro Parser

To integrate the new ideas in the existing implementation, new parsers for the extension of the macro language must be added. These parsers can be constructed out of the grammar using combinator parsers. The new functionality, i.e. the redefinition of attributes, can only be implemented as semantic functions of the parsers.

Starting with `Translation`, there are two synthesized attributes⁶:

```
ATTR Translation [| sem_comp : Expression t -> Expression t
                  , orig_comp: (TranslationType, Expression t)]
```

In the type `Expression t`, `t` should be an instance of `TypeDescriptor`. The attribute `sem_comp`, the semantic function constructor, is a function that, given as argument the redefined attribute computations (or the original computation), uses this computation to construct the new semantic function for the parser of the syntax macro. The attribute `orig_comp` is a tuple with a value of the following data type:

```
data TranslationType = TransLiteral
                    | TransConstructor String
                    | TransVar String
```

```
instance Show TranslationType where
  show TransLiteral = "_Literal_"
  show (TransConstructor c) = c
  show (TransVar tp) = tp
```

and the original attribute computation of the parsed constructor. Data type `TranslationType` is used to make a difference between kinds of translation(-factor)s, e.g. `TransLiteral` for a literal, `TransConstructor` for a constructor (with as child the name of the constructor) and

⁶Some AG-notation will be used here for notational convenience, although the actual code is not generated by the AG-system. Code generation is not possible because the current AG-implementation does not support parameterized data types and higher order attributes.

TransVar for a variable (with as child the corresponding type; this will become clear in Section 3.6.7). For example, for the constructor *Mul* this attribute will be

```
(TransConstructor "Mul", Id mul f :: tp_Expr .->. tp_Expr .->. tp_Expr)
```

The value *mul f* will be used if there are no attribute redefinitions for the current macro and thus the original attribute computation has to be passed to *sem_comp*. This decision will be made by *pProduction*, which is defined later on in this section.

For the moment, it will be assumed that only inherited attributes of the direct children and synthesized attributes of the top constructor can be redefined. This means that for a syntax macro like

```
Factor ::= "product" f=Factor g=Factor h=Factor => Mul (Mul f g) h
```

the inherited attributes of *(Mul f g)* and *h* can be redefined, and the synthesized attributes of the translation itself, but that for example the inherited attributes of *f* and *g* cannot be redefined. In Section 3.6.8, the compiler will be extended with this feature. The parser *pTranslation* for the translation then becomes as follows, where it is noted that λ -abstractions are deferred until the end of Section 3.6.8.

```
pTranslation :: Parser (Token (Dynamic (Type t) Id))
  ((Expression (Type t)) → Expression (Type t)
   , (TranslationType, Expression (Type t))
  )
pTranslation = (λc args →
  (λnewAttrcomp →
    ((foldl Apply (buildNewSemantics c newAttrcomp)
      (map (λ(f, tup) → f newAttrcomp) args)))
    , (TransConstructor c, getAttrComputation c)
  )) <$> pConid <*> pList1 pTransFactor
  <◇> (λv args →
    (λnewAttrcomp →
      ((foldl Apply (Var v) (map (λ(f, tup) → f newAttrcomp)
        args)))
      , (TransVar v, Var v)
    )) <$> pVarid <*> pList pTransFactor
  where pTransFactor = (λl → (λ_ → Const l, (TransLiteral, Const l))) <$> pLiteral
    <◇> (λv → (λ_ → Var v, (TransVar v, Var v))) <$> pVarid
    <◇> (λ(f, tup) → (λcomp → (f (snd tup)), tup))
      <$> pParens pTranslation
    <◇> (λc → (λ_ → buildSemantics c, (TransConstructor c,
      buildSemantics c)))
      <$> pConid
  getSemConstructor c = maybe (Var c) Const (lookup c sem_constructors)
  getAttrComputation c = maybe (Var c) Const (lookup c attr_computations)
  buildNewSemantics c newAttrComp = Apply (getSemConstructor c)
    newAttrComp
  buildSemantics c = buildNewSemantics c (getAttrComputation c)
```

The semantic function of $pProduction$ uses sem_comp and $orig_comp$ to construct the complete attribute computation for the production. Notice that in the case of a nonterminal variable, $TransVar$ is supplied with the name of the variable instead of its type. This will be refined in Section 3.6.7.

The second part, the list of attribute-expressions, defines the new computations for the attributes. An `AttrDefExpression` consists of a left-hand side (the attribute to be redefined) and a right-hand side (the attribute redefinition):

```
DATA AttrDefExpression | AttrDefExpr lhs:InsertExpr
                        rhs:AttrExpr

ATTR AttrDefExpression [ constructor:String
                        |
                        | insert_attr_expr:Expression (Type t)
                        ]

ATTR InsertExpr [ constructor:String || insert_expr:Expression (Type t) ]
ATTR AttrExpr   [ constructor:String || attr_expr :Expression (Type t) ]
```

The inherited attribute `constructor` is the name of the constructor to which the attribute redefinitions should apply. This name, for example "Mul", is needed to lookup the correct attribute selection and insertion functions. The synthesized attribute `insert_attr_expr` is the dynamic representation of a function that inserts the redefined attribute (`attr_expr`). Suppose the attribute pp is redefined for the constructor Mul . The value of the insertion function itself (`insert_expr`) can then be determined by looking up "Mul_pp" in the associated list of attribute inserters.

Concerning the `AttrExpr`, all operators have equal priority, lower than function (and constructor) application. The inherited attribute `constructor` is typically used in combination with an identifier corresponding to a child: the function `makeName` concatenates a list of strings together using an underscore as separator. The result, for example "Mul_li" representing the selection of child li of constructor Mul , can be used to look up the corresponding selection function in the associated list of computation input selection functions.

```
makeName :: [String] → String
makeName = foldr1 (λl r → l ++ "_" ++ r)
```

An `AttrDefExpression` can be parsed by first parsing the inserter, either of an inherited or synthesized attribute, followed by parsing the attribute redefinition. The redefinition is the body of a *Lambda* abstraction that abstracts from the computation input (to be looked up in $comp_inputs$). This function is the argument of the insertion function. Recall that the final composed function will be used to extend the existing attribute computations (see $pProduction$).

```
pAttrDefExpression :: Parser (Token (Dynamic (Type t) Id))
                    (String →
                     Expression (Type t)
                    )
pAttrDefExpression = makeComp <$> pInserter
                    <*> pKeyword "="
```

```

                                <*> pAttrExpression
where makeComp ins expr constructor
      = let (attr_comp_input_nm, attr_comp_input_tp)
        = maybe (error ("input_arg_name_not_found_for_"
                        ++ constructor))
              id (lookup constructor comp_inputs)
        in Apply (ins constructor)
              (Lambda attr_comp_input_nm
                attr_comp_input_tp
                (expr constructor))

```

The semantic function of *pInserter* uses *constructor*, the name of the child for an inherited attribute or "lhs" for a synthesized attribute (*ch_lhs*), and the name of the attribute to look up the corresponding attribute inserter.

```

pInserter :: Parser (Token (Dynamic (Type t) Id))
           (String → Expression (Type t))
pInserter = getInserter <$> pVarid <*> pKeyword "." <*> pVarid
where getInserter ch_lhs attr constructor =
      let nm = makeName [constructor, ch_lhs, attr]
        e = error ("inserter_not_found_for_" ++ nm)
      in maybe e Const (lookup nm attr_inserters)

```

The redefinition of an attribute is an expression that consists of operators and expressions with functions and attribute selections.

```

pAttrExpression :: Parser (Token (Dynamic (Type t) Id))
                (String → (Expression (Type t)))
pAttrExpression = pChainl (fun <$> pOp) pConstrExpr
where pOp = pAny pOperator (map fst operators)
      fun = λop f1 f2 constr → let e1 = f1 constr
                                e2 = f2 constr
      in Apply (Apply (getOperator op) e1) e2

```

```

getOperator op = maybe (error ("operator_not_found_for_" ++ op))
                    Const (lookup op operators)

```

```

pConstrExpr :: Parser (Token (Dynamic (Type t) Id))
             (String → (Expression (Type t)))
pConstrExpr = (λargs constructor → (foldl1 Apply ∘ map (λf → f constructor)) args)
              <$> pList1 pFactor

```

```

pFactor = pParens pAttrExpression
        ◊ (λl c_name → Const l) <$> pLiteral
        ◊ (λc c_name → getConstructor c) <$> pConid
        ◊ pAttrSel

```

```

getConstructor c = maybe (error ("constructor_not_found_for_" ++ c))
                    Const (lookup c constructors)

```

The selection function for a synthesized attribute depends on the type of the child and the name of the attribute, and thus *pAttr* expects the type (as a *String*) of the child as inherited attribute.

```

pAttr :: Parser (Token (Dynamic (Type t) Id))
          (String → (Expression (Type t)))
pAttr = getAttrSelector <$> pVarid
  where getAttrSelector v tp =
        let nm = makeName [tp, v]
            in maybe (error ("attr_selector_not_found_for_" ++ nm))
                  Const (lookup nm attr_selectors)

```

The semantic function of *pAttrSel* has the responsibility for inserting the computation input argument (*attr_comp_input*) at the correct places, i.e. applying each selection function to it. The computation input is the three-tuple containing the output of the children, the local definitions and the inherited attributes of the parent. In Section 3.6.2, the name and type (that is, the type representation) of this computation input are discussed; using *constructor* (*constructor*), the name and type representation of the computation input can be looked up.

```

pAttrSel :: Parser (Token (Dynamic (Type t) Id))
            (String → (Expression (Type t)))
pAttrSel = semAttrSel <$> pVarid <*> pKeyword "." <*> pAttr
  where semAttrSel v sel_a constructor =
        let attr_comp_input
            = maybe (error ("input_arg_name_not_found_for_"
                          ++ constructor))
                  (\(s, tp) → Var s) (lookup constructor input_args)
            nm = makeName [constructor, v]
                (ch_tp, childSelector)
            = maybe (error ("input_sel_not_found_for_" ++ nm))
                  (\(tp, sel) → (tp, Const sel)) (lookup nm input_selectors)
        in Apply (sel_a ch_tp) (Apply childSelector attr_comp_input)

```

Finally, the semantic function of *pProduction*, that parses a complete production rule, constructs the new attribute computation by passing the resulting attribute redefinition to the semantic function constructor *sem_comp*.

```

pProduction :: Parser (Token (Dynamic (Type t) Id)) (Production (Type t))
pProduction =
  production_sem
    <$> pConid <*> pKeyword "::=" <*> pList pRHS_Elem
    <*> pKeyword "=>"
    <*> pTranslation <*> pKeyword ";"
    <*> pList (pAttrExpression <*> pKeyword ";")
  where production_sem nont rhss
        trans@(sem_comp, (constr, orig_comp))
        attr_exprs
        = let foldr_f c = foldr (\fcomp a → Apply (fcomp c) a)
            new_attr_comp = case constr of

```

$$\begin{aligned}
 & (\text{TransConstructor } c) \rightarrow \\
 & \quad \text{sem_comp } (\text{foldr_f } c \text{ orig_comp attr_exprs}) \\
 & \quad \rightarrow \\
 & \quad \text{foldr_f } "" (\text{Apply } (\text{sem_comp orig_comp})) \text{ attr_exprs} \\
 & \text{in Production nont rhss new_attr_comp}
 \end{aligned}$$

Notice that in the computation of *new_attr_comp*, the original computation is taken as a starting point and that this computation is extended one by one with redefinitions of the attribute computations. These computations have to be supplied with the name of the constructor (or the empty string for non-constructor translations). Finally, the result is passed to *sem_comp*, to compute the new catamorphism.

Initial Grammar

The initial grammar must use the semantic functions constructed by the AG system, or at least functions with the same types and semantics.

Macro Interpreter

The interpreter is back in its original form of the syntax macro library, because the macro parser computes the body of semantic function for the parser of the new production rule.

Observations

There is now support for synthesized and inherited attribute redefinitions. However, some improvements can be made:

- Non-constructor translations: currently, it is not possible to redefine attributes for translations that do not start with a constructor, such as

```
Factor ::= "(" e=Expr ")" => e;
```

Although it is not possible to change computation within *e*, an improvement would be to be able to redefine the computation of the inherited and synthesized attributes of *e*.

- Nested attribute redefinitions: sometimes, translations can be rather complex and consist of multiple constructors. It may then be needed to redefine attributes for constructors other than the top constructor.
- The attribute grammar system must be adjusted to generate the code needed for the syntax macros.
- Besides redefining semantics for existing attributes, the possibility to define new attributes would be useful.

The first three improvements will be discussed in the following sections. The fourth improvement is beyond the scope of this document.

3.6.7 Non-constructor Translations

Non-constructor translations are translations that do not start with a constructor, but with a nonterminal variable or λ -abstraction. Because the looking up of selection and insertion functions depends on the name of the constructor, there is currently no way to redefine attributes of such translations. However, if it is possible to find the type of the nonterminal, it is also possible to look up the corresponding selection and insertion functions. This idea suggests the use of a small environment.

Variable Environment

During parsing of a syntax macro like

```
Factor ::= "(" e=Expr ")" => e;
```

the nonterminal variable e will be identified as an `Expr` nonterminal. This information can be used to lookup the type of e : in the initial grammar, there already is a mapping from nonterminals to types. Furthermore, this mapping will be extended with the optional new definitions in the `nonterminals` section of the macro definition. Note that this mapping is the same as the one constructed by the macro interpreter, except that the initial grammar instead of the final grammar will be used to construct the mapping from nonterminals to types. The mapping from a nonterminal variable to its type can then be constructed by looking up the type of the associated nonterminal in that map.

Inherited and Synthesized Attributes

Having the type of a nonterminal variable, the corresponding insertion and selection functions for the attributes can be looked up. Just like the computation input for the constructor types, a computation input for the data type must be generated:

```
type D_input = (D_attrs_inh, D_attrs_syn)
```

The approach is much like the initial approach described in Section 3.3. This approach will suffice because translations consisting of only nonterminal variables can only be redefined for their input and output. The selection and insertion functions are also similar:

```
select_D_inh :: D_input → D_attrs_inh
select_D_inh = fst
```

```
select_D_syn :: D_input → D_attrs_syn
select_D_syn = snd
```

```
insert_D_syn_attr_i :: (D_input → Syn_i) → D_attrs → D_attrs
insert_D_syn_attr_i syn_i inh_syn =  $\lambda inh \rightarrow$  let  $syn = inh\_syn\ inh$ 
                                     in  $insert\_D\_syn\_i\ (syn\_i\ (inh, syn))\ syn$ 
```

```
insert_D_inh_attr_i :: Inh_i → D_attrs → D_attrs
insert_D_inh_attr_i inh_i inh_syn
  =  $\lambda inh \rightarrow$  let  $inh' = insert\_D\_inh\_i\ (inh\_i\ (inh, syn))\ inh$ 
                   $syn = inh\_syn\ inh$ 
                  in  $inh\_syn\ inh'$ 
```

The original computation and the semantic function constructor as described in Section 3.6.2 are in this case not needed because only the input and output can be changed (and hence the computation is hidden anyway).

Practically, the attribute `constructor` will be used to be either the top-level constructor or the type of the nonterminal variable. Because of the choice of the new insertion and selection functions, there are no additional changes for the compiler of the attribute redefinitions: it can use `constructor` to look up the insertion and selection functions for both types of translations. However, a new inherited attribute `nont_type` must be added, that maps nonterminals to their types, and an inherited attribute `var_type_env` that maps nonterminal variables to their types (type presentations):

```
ATTR Productions Production [nont_type:[(String, Exists (Type t))]]|]
ATTR Translation [var_type_env:[(String, String)]|]
```

The parser *pProduction* constructs the environment `var_type_env` by looking up the nonterminal types in `nont_type`. The map `nont_type` is available after parsing the declarations of the nonterminals.

The parser *pTranslation* then uses `var_type_env` for the parsing of nonterminal variables: the attribute `orig_comp` will be a tuple containing a *TransVar* with as child the type corresponding to the nonterminal variable (that will be used as the `constructor`), and its original computation, i.e. just the variable itself (recall that this variable corresponds to the semantic function of a data type and that the value will be supplied by the parser of the corresponding nonterminal).

In a macro definition, one has to make clear whether a synthesized or an inherited attribute is selected by prefixing the selector with `syn.` or `inh.`, just like the selection of a specific child of a constructor. In the left-hand side of an attribute redefinition, `lhs` and `inh` are used to redefine synthesized and inherited attributes respectively. Note that the synthesized or inherited attributes of the *complete* translation expression are selected and redefined. In the current implementation, it is not possible to select attributes of individual nonterminal variables.

```
Factor ::= "(" e=Expr ")" => e;
          { lhs.pp = Text "(" >#< syn.pp >#< Text ")" ; }
```

For the complete implementation of *pProduction* and *pTranslation*, see 3.6.8.

3.6.8 Nested Attribute Redefinitions

In order to redefine attributes of children of constructors, the idea used to redefine attributes for the top-level has to be generalized to any level. This asks for a new data structure and some new syntax.

Data Structure for Translation

In the current implementation, the name of the constructor (`constructor`) is used to look up the original computation of the attributes. Only for the top-level constructor it is possible to define a new computation. The attribute `sem_comp` will use the new computation, which is passed as an argument, to extend the original computation and define the new catamorphism. For constructors used for the children of the top-level constructor, the original computation

will be used. The principle used for the top-level constructor can be generalized by using a *Tree* data structure:

```
data Tree a = Node a [Tree a]
type TranslationTree t = Tree ((Expression t) → Expression t
                               , (TranslationType, Expression t)
                               )
```

In a *TranslationTree*, each *Node* is labeled with the attributes `sem_comp` and `orig_comp` corresponding to the constructor, and has a list of *TranslationTrees* as a child. This list represents the children of the constructor, and by handling constructors in the children the same as the top-level constructor, i.e. by defining `sem_comp` to use the new computation, it is possible to redefine attributes for sub-level constructors as well. Of course, the attribute redefinitions have to be administrated in a correct way. To that end, a similar data structure for the nested attribute redefinitions is introduced:

```
type AttrTree t = Tree [String → (Expression t)]
```

In an *AttrTree*, each *Node* is labeled with a list of functions that given the `constructor` attribute, compute the attribute `insert_attr_expr`. These attribute redefinitions correspond to the constructor represented by the label of the *Node* at the same level of the corresponding *TranslationTree*. Similarly, the attribute redefinitions of the list of child-trees correspond to the children of the *TranslationTree*. Both trees can be combined in the same way as it is done for the top-level constructor. The resulting new semantic function can then be passed as an argument to the parent constructor.

Extended Grammar

The grammar has to be extended to cope with nested attribute definitions:

```
Production ::= Nonterminal " ::= " RHSElem* " => " Translation ";"
              ("{" AttrDefBlock "}")?

AttrDefBlock ::= (AttrDefExpression ";")* ("{" AttrDefBlock "}")*
```

The `AttrDefExpressions` apply to the constructor or nonterminal variable at the corresponding level in the `TranslationTree`; these expressions may be redefinitions of synthesized attributes, and of inherited attributes of the children. The list of `AttrDefBlocks` corresponds to the children of the constructor; attribute redefinitions in these blocks can thus be redefinitions of synthesized attributes of the children, and of inherited attributes of the children's children. One and other is organized syntactically using curly braces.

Compiler

The attributes `sem_comp` and `orig_comp` are now stored in a new synthesized attribute `trans_tree`, and the attributes `constructor` and `insert_attr_expr` are now stored in a new attribute `attr_tree`. This has some consequences for *pTranslation* and *pAttrDefBlock*:

```
pTranslation :: Parser (Token (Dynamic (Type t) Id))
                [(String, String)] → TranslationTree (Type t)
```

)

```

pAttrDefBlock :: Parser (Token (Dynamic (Type t) Id))
                (AttrTree (Type t))
pAttrDefBlock = Node <$> pList (pAttrExpression <*> pKeyword ";" )
                <*> pList (pAccs pAttrDefBlock)

```

The parser *pProduction* has the responsibility to combine the `trans_tree` and `attr_tree` into the new semantic function for the complete translation. This will be done by a recursive function *zipTree* that has the following semantics: for the children of a constructor, construct the attribute computation and use this computation to construct the new semantic function for each of the children. Then construct the attribute computation for the parent, and use this computation together with the new semantic functions of its children to construct the semantic function for the complete translation. For a *Node*, this function is first called recursively on the children to compute the new semantics of the children; then it computes the attribute redefinitions of the top-level and finally constructs the computation for the top-level by using *foldl Apply* to the results of the children, with the attribute redefinitions of the top-level as zero element. Note that the number of children in an *AttrTree* must be equal to the number of children of the corresponding constructor; if this is not the case, this can be fixed by simply consing the necessary *Node [] []*s (stating that there are no attribute redefinitions for those children).

```

pProduction :: Parser (Token (Dynamic (Type t) Id))
                ((String, String)] → Production (Type t))
pProduction = production_sem
                <$> pConid <*> pKeyword " ::= " <*> pList pRHS_Elem
                <*> pKeyword " => "
                <*> pTranslation <*> pKeyword ";"
                <*> ((pAccs pAttrDefBlock) 'opt' (Node [] []))
where production_sem nont rhss transTree attrsTree typeEnv
      = Production nont rhss new_attr_comp
      where new_attr_comp = zipTree (transTree varTypeEnv) attrsTree
      varTypeEnv =
      [(v, (λ(E t) → showF t) (resultType tp)) |
      (Nont v nont) ← rhss, (nont', tp) ← typeEnv, nont ≡ nont']
      ]
      zipTree (Node (sem_comp', (constr, orig_comp)) args)
      (Node attr_exprs nested_exprs) =
      let new_attr_comps = map (λf → f (show constr)) attr_exprs
      nested_ress = zipWith zipTree args nested_exprs'
      nested_exprs' = nested_exprs ++ (take n (repeat (Node [] [])))
      where n = length args - length nested_exprs
      sem_comp new_comp =
      foldl Apply (sem_comp' new_comp) nested_ress
      res_attr_comp = case constr of
      (TransConstructor c) →
      foldr Apply (sem_comp orig_comp) new_attr_comps
      - → sem_comp (foldr Apply orig_comp new_attr_comps)

```

in *res_attr_comp*

The complete semantic function is thus constructed by passing the semantic functions of the children and the new attribute computation *new_comp* of the top-level constructor to the semantic function constructor *sem_comp*. Non-constructor translations can be handled by this algorithm as well.

λ -Abstractions Revisited

With the new translation structure, it is also possible to redefine attributes of the body of a λ -abstraction. This requires two modifications, in *pTranslation* and *pProduction*. In *pTranslation*, λ -abstractions have to be handled slightly different: the function **sem_comp** will now be expecting the body of the abstraction, instead of the new attribute computation as usual. The translation of the body becomes the only child of the **trans_tree** for a λ -abstraction. The data type *TranslationType* must be extended:

```
data TranslationType = TransLiteral
                    | TransConstructor String
                    | TransVar String
                    | TransLambda
show TransLambda = "_Lambda_"
```

The parser *pTranslation* is changed as follows:

```
pTranslation :: Parser (Token (Dynamic (Type t) Id))
              <|> [(String, String)] → Translation Tree (Type t)
pTranslation = ...
  <|> (λv tp@(E tp') treeComp env →
    let bodyTree = treeComp ((v, showF tp') : env)
    in Node (λbody → Lambda v tp body
      , (TransLambda, Var v)
      ) [bodyTree] <$ pKeyword "\\\" <*> pVarid
      <*> pKeyword ":@"
      <*> pType
      <*> pKeyword "."
      <*> pTranslation
```

This change has consequences for *pProduction*, because there now is a third alternative (next to literals and constructors and variables): the λ -expression. This alternative is identified by *TransLambda*. It is then known that the only child will be the body of a λ -abstraction, that thus must be passed to **sem_comp**.

With the body playing the role of the *n*-th nested child, where *n* is the number of enclosing λ -abstractions, it is possible to redefine attributes for the body just as if it was a constructor translation.

```
MulFun ::= "mul" e=Expr => \x::Expr . Mul e x ;
      {
      {
      lhs.pp = Text "const" >#< ri.pp >#< le.pp;
      }
      }
      }
```

A construction like `MulFun` will be used in combination with another construction like

```
MulApp1 ::= f=MulFun e=Expr => f e ;
```

As has been explained in Section 3.6.7, the attributes of the complete translation `f e` can be selected or redefined, but not of `f` or `e`.

With this functionality, there is a way to preserve the usage of λ -abstractions, for instance in pretty printing.

3.7 Considerations

The language used to define the translations and attribute redefinitions is a very simple language. One could argue that it would be better to use the complete λ -calculus. However, that would be a bridge too far. The goal is to be able to redefine attributes, and nothing more than that. Adding more and more features would ultimately result in the conclusion that it would have been better to implement Haskell dynamically, which is clearly not the goal.

Related to this issue is the way to access children and attributes: the original names as specified in the attribute grammar must be used, and hence these names must be known by the implementor of the syntax macros. Although it would be nice to also have the possibility to identify children with their variable names as defined in the macro, this would again be another extension of the compiler. Furthermore, identification via AG names is needed anyway because macro translations may also use expressions for the children that are more complex than a single variable.

The designer of the language has to provide the combinators and functions available to the implementor of the syntax macros. Again this is a conscious choice: it is up to the designer of the language to decide to what extent attributes can be redefined (and which).

Then there is the question of defining new attributes. This feature is currently not supported, but it may be fruitful to have the possibility to define new attributes. However, this requires changes in both the dynamic typing library and way the attribute grammars are implemented (using typable records [10] instead of tuples for the attributes for example).

Chapter 4

Conclusions and Future Work

The power of syntax macros has been enlarged by using dynamic typing [3] to give the programmer the possibility to redefine attributes for the translation of the new concrete syntax at run-time. The existing syntax macro library [2] has been extended with a small language for the redefinition of the attributes. To support the redefinition of both inherited and synthesized attributes, it has appeared that an aspect-oriented approach to attribute grammar systems is needed [6].

Virtually all of the ingredients needed for this extension can be generated out of the attribute grammar definition automatically. The only work for the designer of the language is to specify the functions and combinators that will be available for the programmer of the syntax macros to manipulate the attributes.

At the same time, one of the weaknesses of syntax macros is exposed. Just giving a translation to the abstract syntax and redefining attributes in a limited way is not always enough. It requires a full language to control the translation completely. The danger is that this will end up in a dynamic attribute grammar system.

Nevertheless, the following future work for both the dynamic typing library and the syntax macro library can be identified:

- In addition to redefining existing attributes, it would be nice to also be able to add new attributes at run-time (although such an extension would approach a dynamic attribute grammar system).
- The error-handling of the dynamic typing library and the extended syntax macro library must be improved.
- Support for polymorphic types in the dynamic typing library.
- The redefinition of attributes proceeds by extending the original computations and replacing the original result with the new result. For efficiency reasons it is desirable to replace the original computation instead of extending it.
- The language for redefining attributes can be made more expressive. For example, adding syntactic sugar for identifying nonterminals by their associated variables (in addition to identifying the children by their original names) would enlarge the expressiveness. Another improvement is to be able to write inline attribute redefinitions, so that it is no longer necessary to duplicate the translation tree (which is cumbersome for more complicated translations).

- The current implementation uses a lot of nested cartesian products, which may become increasingly inefficient if the number of attributes grows. Finding alternative implementations is a challenging task.
- Concerning the code generation, support must be added for AG specialties such as `TYPE` declarations, non-simple attribute types and local attributes.

Acknowledgements

The first person I thank is Arthur Baars. The discussions with Arthur often led to new ideas and approaches. Especially his patient explanations and quick software fixes were a great help.

The second person I am thankful to is Doaitse Swierstra. In the several meetings we have had he occasionally carried out monologues about politics, the administration and holidays, always ending with "but that was not what you have been coming for". Usually, however, he gave fair comments and fruitful instructions, with as climax the key to his treasure house full of attribute grammar combinators.

Bibliography

- [1] BAARS, A. I. Typed Syntax Macros. Master's thesis, Utrecht University, 2000.
- [2] BAARS, A. I., AND SWIERSTRA, S. D. Syntax Macros. <http://www.cs.uu.nl/people/arthurb>, 2002.
- [3] BAARS, A. I., AND SWIERSTRA, S. D. Typing Dynamic Typing. In *ICFP 2002 Proceedings* (2002).
- [4] CARDELLI, L., MATTHES, F., AND ABADI, M. Extensible syntax with lexical scoping. Tech. Rep. 121, 1994.
- [5] DE MOOR, O., BACKHOUSE, K., AND SWIERSTRA, S. D. First-class Attribute Grammars. In *Third Workshop on Attribute Grammars and their Application* (2000).
- [6] DE MOOR, O., PEYTON-JONES, S., AND VAN WYK, E. Aspect-oriented compilers. In *Generative and Component-Based Software Engineering* (1999), Springer Lecture Notes in Computer Science, Springer.
- [7] GUERRA, R. Inverting non-order preserving parsers. <http://www.cs.uu.nl/staff/rui.html>, 2002.
- [8] GUY L. STEELE, J. Growing a language. *Journal of Higher-Order and Symbolic Computation*, 12 (1999), 221–236.
- [9] JEURING, J. T., AND SWIERSTRA, S. D. *Grammars and Parsing*. Utrecht University, 2001.
- [10] JONES, M. P., AND REID, A. *The Hugs 98 User Manual*. Yale Haskell Group, 2002. <http://www.haskell.org/hugs/>.
- [11] JONES, S. P. *Haskell 98 Language and Libraries: The Revised Report*. Cambridge University Press, 2003. <http://www.haskell.org/>.
- [12] LEAVENWORTH, B. Syntax macros and extended translation. *CACM* 11, 9 (1966), 790–793.
- [13] LÖH, A., BAARS, A. I., AND SWIERSTRA, S. D. Using the AG system. <http://www.cs.uu.nl/groups/ST/twiki/bin/view/Center/Software>, 2002.
- [14] SWIERSTRA, S. D. Fast, Error Repairing Parser Combinators. http://www.cs.uu.nl/groups/ST/Software/UU_Parsing/.

- [15] SWIERSTRA, S. D. Combinator Parsers. *Electronic Notes in theoretical Computer Science* (2001).
- [16] SWIERSTRA, S. D., AND BAARS, A. I. Attribute Grammar System. <http://www.cs.uu.nl/groups/ST/twiki/bin/view/Center/Software>.

Appendix A

Extensions and Manual

A.1 Syntax Macro Library

The module *MacroParser* of the syntax macro library has been extended to cope with attribute redefinitions. Furthermore, the module *GenerateDynOps* has been added to generate some additional code that may ease the use of the syntax macro library.

The source code for the syntax macro library is available online:

<http://www.cs.uu.nl/people/arthurb>

A.2 Attribute Grammar System

The attribute grammar system of the Utrecht University [16] has been extended with an option `syntaxmacro`. Enabling this option will force the system to create all additional source code that is necessary for attribute redefinitions in syntax macros. The module *GenerateCode* has been adjusted to generate the additional code for:

- Catamorphisms that use `class Knit`;
- Type synonyms;
- Dynamic type representations;
- Attribute selection functions;
- Attribute insertion functions;
- Semantic function constructors;
- Aspect computations.

To use these functions for attribute redefinitions, they must be available in associated lists; these lists are generated by the AG system as well.

The latest version of the UU_AG system is available online:

```
cvs -d:pserver:anonymous@cvs.cs.uu.nl:/data/cvs-rep login
cvs -d:pserver:anonymous@cvs.cs.uu.nl:/data/cvs-rep checkout uust
```

The AG code is in `uust/tools/ag`.

The official release is available on

<http://www.cs.uu.nl/groups/ST/twiki/bin/view/Center/Software>

A.3 Syntax Macro Grammar

For the lexical elements it must be noted that `Varid` and `Conid` must not be equal the reserved keywords (such as functions).

```

Uppercase      ::= ['A'..'Z']
Lowercase      ::= ['a'..'z']
IdentLetter    ::= Uppercase | Lowercase | ['0'..'9'] | '_'
Varid          ::= Lowercase IdentLetter*
Conid          ::= UpperCase IdentLetter*

StringLiteral  ::= ''' StringChar* '''

OpSymbols      ::= "#" | "$" | "%" | "^" | "&" | "*" | "-" | "<" | ">"

Constructor    ::= Conid
Function       ::= Conid
TypeConstant   ::= Conid
Nonterminal    ::= Conid
Operator       ::= OpSymbols+

MacroDefinitions ::= Nonterminals? Productions?

Nonterminals   ::= "nonterminals"
                Declaration+

Declaration    ::= Nonterminal "::" Type

Type           ::= SimpleType ("->" SimpleType)*

SimpleType     ::= TypeConstant
                | "(" Type ")"

Productions    ::= "rules"
                Production+

Production     ::= Nonterminal "::=" RHSElem* "=>" Translation ";"
                ("{" AttrDefBlock "}")?

RHSElem        ::= StringLiteral
                | Varid "=" Nonterminal

Translation    ::= "\\\" Varid "::" Type "." Translation

```

```

        | Factor+

Factor      ::= Varid
             | Constructor
             | Literal
             | StringLiteral
             | "(" Translation ")"

AttrDefBlock ::= (AttrDefExpression ";")* ("{" AttrDefBlock "}")*
AttrDefExpression ::= Lefthand "=" AttrExpression

Lefthand    ::= LElement "." Attr

LElement  ::= "lhs"
             | "inh"
             | Varid

AttrExpression ::= AttrFactor+

AttrFactor  ::= Constructor
             | Function
             | AttrSel
             | Operator
             | "(" AttrExpression ")"
             | StringLit
             | Literal

AttrSel     ::= RElement "." Attr

RElement  ::= "lhs"
             | "syn"
             | "inh"
             | Varid

Attr       ::= Varid

```

A.4 Attribute Redefinition Manual

In order to successfully use attribute redefinitions, one must be known with the attribute grammar system and the syntax macro library:

1. Define your language and attributes in an AG file (`lang.ag`).
2. Compile `lang.ag` using the following command:

```
uuag -mcdif --syntaxmacro lang.ag
```

3. Enable the code for the use of syntax macros by adding an initial grammar (see the examples in the syntax macro library). Part of this code can be generated by using *GenerateDynOps* of the syntax macro library. This function takes a file `file.sm` with the following (optional) contents:

```
imports    ::= "imports" "{" ("import" String)* "}"
functions ::= "functions" "{" (Conid "=" String ":" Type)* "}"
operators ::= "operators" "{" (Oper "=" Oper ":" Type)* "}"
```

A `Conid` is an uppercase identifier, and an `Oper` is an operator in Haskell syntax. A `Type` is a type in Haskell syntax. The result of the call

```
runhugs GenerateDynOps file.sm
```

is a file `fileDynFuncs.hs` containing definitions of additional dynamic type representations and lists of operators and functions that can be used in the attribute redefinitions. Follow the instructions in the generated file to incorporate this code with the code of the new language.

4. Define the syntax macros, including any attribute redefinitions. Use the names for attributes and children as they are defined in the original attribute grammar.
5. Call the syntax macro compiler on the macro file and a program file that uses the new language.