Parsing Mixfix Expressions

Dealing with User-Defined Mixfix Operators Efficiently

vorgelegt von
Diplom Informatiker
Jacob Wieland
aus Berlin

Von der Fakultät IV - Elektrotechnik und Informatik
Technische Universität Berlin
zur Erlangung des akademischen Grades
Doktor der Ingenieurwissenschaften
Dr. ing.
genehmigte Dissertation

Tag der wissenschaftlichen Aussprache: 12.10.2009

Berlin 2009

D 83

Acknowledgements

I want to thank my mother Karla Wieland for always supporting and motivating me through childhood and education, and my father Robert Wieland who sparked my interest into languages and mathematics.

I also want to thank my professor Prof. Peter Pepper for inspiring and allowing me to write this thesis on the basis of his vision of future programming languages, as well as giving me lots of feedback while working on it. Prof. Sabine Glesner also gave me some useful pointers into the right direction.

I found the lively discussions with my colleagues Baltasar Trancon-y-Widemann and Markus Lepper very enlightening, while implementing the algorithms together with Diez B. Roggisch in a few Extreme Programming sessions helped me a lot to finish the experimental aspect of the work.

Finally, I want to thank my wife Jessica for discussing my work with me and making me finish it and also my friend Alexander Lorenz for proof-reading and English language consultation.

Contents

Chapter 1

Introduction

1.1 Problem

1.1.1 Computer Languages - What Are They Good For?

Computer languages are used foremost as a means of communicating the demands of the computer programmer to the machine that is supposed to process those demands. For this purpose, they need to be formal and unambiguous, as well as mechanically or automatically processable. High-level computer languages are also used as a more or less abstract way of formulating algorithms, i.e. descriptions of processes that solve certain problems to be used both by humans and machines. Towards this purpose, computer languages have been developed which allow abstraction from the underlying architecture of the machine and to focus more on the process description and program architecture, leaving the machine-specific details to the automatic interpreter of the program. Modern programming languages even include constructs that allow re-usability of existing programs to avoid constant repetition of the same work by the programmers, and formal automatic or semi-automatic reasoning about programs to ascertain specific properties they need to fulfils to meet their purpose, e.g. that the number of computation steps needed by the program is no larger than a certain amount to ensure that the program terminates before the computation result is needed.

1.1.2 Documentation: Formal vs. Natural Language

Normally, computer programs are often not easily readable[1], and thus not easily understandable. In the modern world of computers, computer programs have become so large and complex that it is imperative they are well documented, using natural language, so that they can be understood by the *human* reader. But they should also be well documented/specified for the machine, e.g. for automatically finding errors in the program.

However, the problem with *natural language* documentation is that in general it lacks the aforementioned properties of formality and unambiguity, therefore often being a source of confusion to human readers, even if the documentation is in their mother-tongue. The problem of course becomes even worse when natural language barriers have to be crossed between writer and reader[2].

To avoid such confusion, these properties of computer languages could make them ideal as a means of *communication between programmers* as well, easing re-

[1]because of lots of parentheses, strange and arbitrary precedences between different operators, etc.

[2]e.g. when everybody uses English, although it is not their mother-tongue

use or refactoring of programs for new goals or new programming environments and finding of errors.

The reasons for the unreadability of common computer languages, even though they resemble natural languages in some respects superficially, are mostly of a technical nature, stemming from restrictions imposed upon computers in the early days of computer language development like very limited memory and slow processing speed.

1.2 Solution

1.2.1 Natural-Language-Like Computer Language

The ideal solution for the understandability problem of computer programs would be a computer language that is as close as possible to natural language, and thus better understandable, but is still formal, unambiguous and efficiently processable by machines. This would make the language both useful as a tool for communication between humans as well as between human and machine. Such a language would have to resemble natural language more closely than common computer languages, or at least allow using those formal constructs normally used by the programmers in their specific domains, and thus would be easier to read, write and understand[3].

1.2.2 Different Approaches To Parsing

Parsing Natural Language

A lot of the understandability problems of computer languages come from the different ways that languages are processed by humans and machines. In natural languages, parsing, i.e. syntactic analysis and disambiguation, is often guided or helped by semantic analysis and heuristic inferences from the context.

Parsing Common Computer Languages

In contrast, syntactic analysis of a computer program is normally done without reference to context or semantics whatsoever, but instead simply uses rules described by a context-free grammar. If that were not enough restriction already, this grammar has to be *syntactically* unambiguous (i.e. allowing only one *syntactic* interpretation of each sentence), or given additional rules to allow at most one syntactic interpretation (like precedence rules between infix operators). Certain classes of these grammars allow the automatic construction of efficient parsers using a formal method, implemented by a computer program (e.g. yacc), that way largely avoiding human programming errors during this task.

Parsing Natural-Language-Like Computer Languages

Taking the idea of the processes to parse natural languages, this thesis proposes a different way of automatically parsing a computer language which allows user-defined natural-language-like constructs, called *mixfix operators*. By tightly coupling the syntactic and semantic analysis, using semantic and context information for syntactic disambiguation[4] purposes, we will show that natural-language-like computer languages are possible which have all the necessary properties and for which there exist, nevertheless, efficient parsing methods.

[3]e.g. by allowing mathematicians or physicists to write down their formulae the same way they would write them on paper

[4]i.e. parsing

1.3 Motivation

Many computer languages have built-in natural-language-like constructs. This indicates that their designers agree that such constructs make a language more readable. Following this train of thought, a language where the users can decide which natural-language-like constructs to include into their programs could have even greater readability, *at least to the users themselves.* Supposing that people with similar backgrounds speak the same language (i.e. use the same domain-specific nomenclatures etc.), it is conceivable that such programs are likely to be more understandable to other people with a similar background, as well, even without a lot of additional documentation.

We introduce a programming language which gives the users the freedom to design their own dialects of the language, hoping that this will make reading, understanding, but also writing of computer programs easier, thus heightening productivity of software development and support, as well as re-usability of computer programs.

Many software engineering tasks have been proposed to minimize the cost of development and support of software. Readable programming languages could be an important aspect to further this effort. Instead of taking the state-of-the-art in computer languages as a given that cannot be made better, then introducing lots of methods to deal with the problems caused by the hard-to-understand programming languages used, we propose to use languages that cause fewer of these problems by being hopefully less hard to understand to begin with.

1.4 Algorithmic Approach

1.4.1 Integration of Parsing and Checking Phase

Instead of the classical pipe-line approach[5], where there can be no feedback from the semantical analysis toward the syntactical analysis, we propose to split the parsing phase into a coarse parsing phase, where built-in meta constructs are parsed[6], followed by a fine parsing phase which is integrated with the checking phase, thereby being able to use inferred semantic properties for syntactical disambiguation.

The fine parsing phase is split again into two parts: a *context-free* parsing phase where all purely syntactical disambiguation is taken care of, and a *type and precedence dependent*[7] parsing phase, where types and user-defined precedence relations are taken into account for resolving any further ambiguity.

1.4.2 Efficiency

We will show that this approach is efficient enough for normal programming purposes, especially in the positive case that there is only one type-correct syntactical interpretation for each parsed expression. We will also point out factors which make the algorithm — understandably — less efficient, developing some guide-lines for the usage of the features of our language.

[5]lexical analysis – scanning, syntactical analysis – parsing, semantical analysis – type and context checking

[6]like declaration keywords FUN , VAR and PREC

[7]i.e. context-sensitive

1.5 Outline

In chapter 2, we will introduce our concept of mixfix operators *informally* and give many examples of mixfix operators common in existing programming languages to show their usefulness, acceptance and expressiveness in general. Afterwards, we will also informally explore the different causes for ambiguity in mixfix expressions. Finally, we will summarize the goals we want to pursue with our language and the means necessary to achieve them.

In chapter 3 we will *formally* introduce the tools to describe the solutions to the ambiguity problems, namely the concepts of mixfix operator signatures, backbone grammars, operator fixities, precedence relations, unification and two-level grammars.

Chapter 4 will define the mixfix expression language together with its type system and its integration into a functional language that allows the introduction of mixfix operators.

Chapter 5 formally explores the problem of ambiguity and discusses common approaches to its solution. Furthermore, the different causes of ambiguities for mixfix expressions and how they can be dealt with are discussed.

Chapter 6 defines the mapping of the given mixfix operator signatures and adhoc precedence relations into two-level grammars with grammar constraints incorporating our language restrictions.

Chapter 7 generalizes common grammar transformations to our concept of two-level grammars, allowing the automatic derivation of top-down parsers for the mixfix expression language for such grammars.

Chapter 8 discusses an implementation of our approach using a one-level Earley parser and type-driven top-down disambiguation of the resulting parse tree representation. We show the results of experiments supporting our thesis and discuss several optimizations that we find useful. We also add some guidelines for the usage of the features of the mixfix expression language to avoid less efficiently parsable programs.

Chapter 2

Overview

To make a language more flexible and more readable, the idea of *mixfix operators* that can be introduced into the language by the user comes to mind.

However, this can cause a lot of problems, mainly by the introduction of possible ambiguities into the language which cannot be allowed in programming languages.

This thesis explores the possibilities of how these problems can be dealt with efficiently, especially during syntactic analysis, i.e. parsing of the language.

2.1 Notes on Notation

In the following section, we give a little overview of some notations to make the following chapters more understandable.

2.1.1 Mixfix Operator Declaration

We follow the classical algebraic approach to declare mixfix operators in *signatures* where every operator is given a *type*.

Because they are *mixfix operators*, we also have to give a *pattern* that defines the *form* of the operator declaring the places of the operands in the operator by use of *placeholders*.

Example 1 *In the signature in figure 2.1 which describes operators for expressions for the natural numbers, we declare*

- *the* sort `nat` *as a nullary mixfix operator,*
- *the* constant 0*, also as a nullary mixfix operator of type* `nat`*,*
- *the* successor function `succ _` *as a unary prefix operator,*
- *the* addition function `_ + _` *as a binary infix operator on natural numbers,*
- *the* identity function `( _ )` *as a unary closed operator on natural numbers.*

Declaring the variables `x` *and* `y`*, we can then give a* pattern based definition *of the addition operator.*

```
FUN nat          : SORT
FUN 0            : nat
FUN succ _       : [nat] → nat
FUN _ + _        : [nat , nat] → nat
FUN ( _ )        : [nat] → nat
VAR x            : nat
VAR y            : nat
DEF x + 0        == x
DEF x + succ y   == succ(x + y)
```

Figure 2.1: Signature of example 1

We use mixfix operators to both describe the *type language* and the *value language*. So, mixfix expressions can be used both on the right hand side of *declarations* of mixfix operators and on both sides of their *definitions*.

2.1.2 Built-In Mixfix Operators

To be able to declare mixfix operators for *types*, we need the predefined *sort symbol* `SORT` (a nullary operator), as well as the infix arrow operator _ → _ to describe *function types*.

Since the *operand types* in the type declarations can *also* be complex mixfix expressions (which are normally sequences of more than one symbol), we need to distinguish them syntactically from each other, denoting them as *operand lists* like `[nat , nat]` (which are basically products of types). Thus, there must be other built-in mixfix operators that allow us to denote such *lists of terms*, i.e. the operators `[ _ ]` and _ , _ and the empty operator for denoting the empty list.

2.1.3 Mixfix Operator Variables

Finally, we need to be able to distinguish the *variables* in any declaration from the other operators. But we do not want an *implicit variable declaration* because this is a too limited approach which would allow only single-symbol variables. Therefore, we enforce the *separate* declaration of variables which allows them to be complex mixfix operators as well.

Again, there are *type* variables for generic type declarations, but also variables for *value* parameters to be considered. We declare variables *syntactically* like other mixfix operators with a pattern and a type, but using the distinguishing keyword VAR instead of FUN at the beginning of the declaration. All operator declarations and definitions in a scope are considered quantified over the variables that occur in them. Thus, every variable can have a different binding in each declaration or definition. This saves us from having to repeat the same variable declaration for every generic declaration.[1]

Example 2 *In the signature in figure 2.2, we declare a* sort variable `A`*.*

We also declare a generic sequence type *operator* `seq _` *to be able to describe the types of the sequence operators.*

All the following declarations are generic over the variable `A`*, but it can have a different binding for every one of these declarations.*

[1] This approach mimics the use of variables in PROLOG with the difference that in that language variables are distinguished from other operators syntactically via their capital first letter.

The indexing operator _ [_] *that is supposed to select the indexed element from a sequence is a full-fledged* mixfix operator *as it is neither purely prefix, infix nor postfix in any classical sense.*

```
VAR  A        : SORT
FUN  seq _    : [SORT] → SORT
FUN  <>       : seq A
FUN  _ :: _   : [A , seq A] → seq A
FUN  _ ++ _   : [seq A , seq A] → seq A
FUN  _ [ _ ]  : [seq A , nat] → A
```

Figure 2.2: Signature of example 2

But why do we want to be able to declare variables also as mixfix operators? In our view, it can be useful in different ways.

Example 3 *To define the* reduce *operator* _ / _ | _ *(see figure 2.3) that reduces a list* $x_1 :: \ldots :: x_n ::$ `<>` *over an infix operator* _ ⊕ _ *with a start element* `e` *to* $x_1 \oplus \ldots \oplus x_n \oplus e$, *we need to declare several variables.*

First of all, all operators and variables used in the definitions are generic over the type variables `A` *and* `B`.

Since we want the definition of the reduce operator parameterized with the operator _ ⊕ _, *it also has to be declared as a variable. As we can see, this allows us to use it* mixfix fashion *on the right-hand-side of the definition of the reduce operator.*

Finally, declaring the rest list *as the very complex nullary mixfix variable* $x_2 :: \ldots :: x_n ::$ `<>` *lets us denote the definition of the reduce operator in a way that is very close to our intuition.*

```
VAR  A                                   : SORT
VAR  B                                   : SORT
VAR  e                                   : B
VAR  $x_1$                               : A
VAR  $x_2$ :: ... :: $x_n$ :: <>         : seq A
VAR  _ ⊕ _                               : [A , B] → B
FUN  _ / _ | _                           : [seq A , [A , B] → B , B] → B
DEF  <> / (_ ⊕ _) | e                    == e
DEF  $x_1$ :: $x_2$ :: ... :: $x_n$ :: <> / (_ ⊕ _) | e == $x_1$ ⊕ $x_2$ :: ... :: $x_n$ :: <> / (_ ⊕ _) | e
```

Figure 2.3: Signature of example 3

Summarizing, we enforce the declaration of variables *outside of the declarations and definitions they are used in* for the following reasons:

1. to allow them to be mixfix operators which can also be used syntactically like other mixfix operators,

2. to distinguish them from other mixfix operators

3. to be able to easily *re-use* them in different declarations (as a shorthand notation for ∀-quantification of these declarations).

Also, this way of declaration is more robust against later change of the signature.[2]

2.2 Mixfix Operators — General Motivation

Mixfix operators ranging from simple ones like those in figure 2.4 to more complex ones like in figure 2.5 are built into almost every programming language because they heighten the readability of programs.

```
FUN int   : SORT
FUN 0     : int
FUN _ + _ : [int , int] → int
VAR A     : SORT
FUN ( _ ) : [A] → A
```

Figure 2.4: Simple Mixfix Operators

```
FUN _ [ _ ]                : [array A , nat] → A
FUN if _ then _ else _     : [bool , A , A] → A
FUN for _ := _ to _ do _   : [var int , int , int , stmt] → stmt
```

Figure 2.5: Complex Mixfix Operators

However, while every programming language allows the definition and use of prefix operators and of variable or constant identifiers by the user, very few programming languages allow the definition of operators, much less variables, of other fixities. Even if they do, the freedom to do so is always either very restricted or the language analyzation tools deal very poorly with them, both of which is unsatisfactory.

This thesis shall explore how it can be possible to give users the ability to define and use their own powerful mixfix operators in a full-fledged functional programming language. The concepts we develop are, however, not restricted to functional languages. Our explorations using a functional language can be seen as only an example to introduce mixfix operators into *any* kind of textual programming language.

We deem a language incorporating user-defined mixfix operators to have greater expressive power and possibly a more natural feel to it than standard programming languages, functional or otherwise. User-defined mixfix operators can help to customize the language, and hopefully this shall allow easier writing and reading of programs for the non-computer-scientist, if used carefully.

Unfortunately, introducing mixfix operators without *any* restrictions can cause similar problems in parsing the language as those encountered when trying to parse *natural* languages. But while natural languages inherently have ambiguities that humans resolve heuristically, having to deal with possible misunderstandings, programs *must not* be ambiguous. A compiler, unlike a human, cannot safely make

[2]In functional languages like Opal and Haskell where all undeclared names in definitions are seen as variables, later introduction of a constant operator with the same name into the program changes the semantics of the definition.

probabilistic decisions to assign meaning to a program. *There should be no misunderstanding possible between the compiler and the programmer.* Consequently, the compiler needs to be able to find *all* ambiguities in a program and either resolve them deterministically or revoke the program as (possibly) ambiguous, helping the user to resolve the (possible) ambiguities.

Because of this problem, we will focus mainly on the issue of ambiguities that user-defined mixfix operators might introduce into the language, especially the different *causes* of ambiguities that exist and how they can be found, avoided and controlled efficiently.

Efficiency will be treated both in respect to parsing and from the viewpoint of the user. The amount of information that needs to be provided in addition to the actual program by the user shall be kept as small as possible to make mixfix operators a *useful* tool.

We will critically discuss common approaches to parsing programming languages which use *unambiguous context-free grammars* to describe the language and derive parsers for it. We will give reasons why these approaches are ill-fitted for dealing with user-defined mixfix operators and thus are the main reason why such operators have not been incorporated with great success into programming languages.

Motivated by this and inspired by the success of using *two-level grammars* in dealing with the parsing of natural languages, we will give an approach using such grammars derived from the mixfix operator declarations and the ad-hoc precedence relations given by the user. These grammars shall be *unambiguous by construction* as long as certain properties in regard to the information given by the user are satisfied.

The restrictions on the mixfix operators also shall be as natural as possible, i.e. easy to understand by users that are not experts in programming language theory. Thus, they shall be largely independent of the parsing scheme or type system to be used, instead following only from the natural causes of ambiguity that we will discuss. We will try to avoid all idiosyncratic design decisions that favour one approach over another, when different approaches to a problem are possible, leaving the path to all possibilities open.

Instead, we will give the users of the language tools to make these decisions – if necessary – themselves, either locally or globally. One such tool is the introduction of *user-defined precedence relations* on the user-defined operators. Such so called *ad-hoc precedence relations* will only describe a *preference* of the user how the operators shall be used when several semantically valid syntactic interpretations might be possible. This is different from the normal approach where precedences *always* enforce a certain *syntactic* interpretation, regardless of the actual existence of ambiguity or semantic validity of that interpretation, often making the programmer use a lot of syntactic structuring where none should be necessary.

2.3 Motivations for Mixfix Operators

Mixfix operators are present in almost every programming language and therefore, obviously useful in general.

This section shall give some examples on how user-defined mixfix operators could help to introduce such normally built-in operators into a language that does not have them built in.

2.3.1 Common Types of Mixfix Operators

Most programming languages allow the programmer to introduce parametric operations only as some kind of functions or procedures. But the possibilities as to how to use these functions *syntactically*, i.e. denote expressions involving their application or instantiation, are often very limited. In most cases, either only prefix or, sometimes, also postfix applications (OPAL/Haskell) are allowed.

Binary Infix Operators

In more advanced scenarios, it is also possible to define binary infix operators and restrict their usage syntactically (Prolog [34] / OPAL [31] / Haskell [18]) by associating them with precedence levels or precedences. More restrictively, some languages allow the overloading of the built-in infix operators (C++).

Example 4 *Something like the specification for sequence operators in figure 2.6 could be written in Opal.*

Here, the definitions for `L`$_1$ *and* `L`$_2$ *are equivalent because of the* `RIGHT BRACKET` *precedence declarations (given as pragmas) which signify that any occurrence of the different binary operators should be seen as bracketed to the right.*

```
SORT        A
SORT        seq[A]
FUN  <> :   seq[A]
FUN  ::  :  A**seq[A] → seq[A]
FUN  ++:    seq[A]**seq[A] → seq[A]
         /$ RIGHT BRACKET[ :: ][ + +] / $
         /$ RIGHT BRACKET[ + +][ :: ] / $
FUN   x  :  A
FUN   L1 :  seq[A]
FUN   L2 :  seq[A]
DEF   L1 == x :: x :: <> + +x :: <> + +x :: x :: <>
DEF   L2 == x :: (x :: (<> + +(x :: (<> + +(x :: (x :: <>))))))
```

Figure 2.6: Signature of example 4

Arithmetic Operators

However, most programming languages also already have some built-in *infix* operations for the built-in types.

Example 5 *Using a language that can define operator patterns with placeholders and add precedences for them like in 2.7 (via* PREC *and* EQPREC *declarations, as defined in definition 30 on page 53), we can declare the standard precedence for arithmetic operators.*

The one we are giving here allows disambiguation of the usual arithmetic expressions without parentheses, but it also allows additional unambiguous ones like $4 * -5$.

```
FUN     _ + _              : [int , int] → int
FUN     _ − _              : [int , int] → int
FUN     _ ∗ _              : [int , int] → int
FUN     _ / _              : [int , int] → int
FUN     _ ∧ _              : [int , int] → int
FUN     − _                : [int] → int

EQPREC (_ + _)(_ − _)        -- same precedence for + and -
EQPREC (_ ∗ _)(_ / _)        -- same precedence for * and /
PREC   (_ + _) + _           -- + is left associative
PREC   (_ ∗ _) ∗ _           -- * is left associative
PREC   (_ ∧ _) ∧ _           -- ∧ is left associative
PREC   (_ ∗ _) + (_ ∗ _)     -- * is precedent to +
PREC   (_ ∧ _) + (_ ∧ _)     -- ∧ is precedent to +
PREC   (_ ∧ _) ∗ (_ ∧ _)     -- ∧ is precedent to *
PREC   (− _) + _             -- prefix - is left precedent to +
PREC   (− _) ∗ _             -- prefix - is left precedent to *
PREC   (− _) ∧ _             -- prefix - is left precedent to ∧
```

Figure 2.7: Signature of example 5

Common Built-in Real Mixfix Operators

We call operators that do not have to be either prefix or postfix, *mixfix operators* of which the prefix and postfix operators (as well as the infix operators) are just special cases.

But there are operators even more complex than infix operators, some of which are very common in most programming languages.

Example 6 *Operators that build number or string expressions could be expressed by mixfix operators like the ones in figure 2.8 on the lexical level (with the restriction that there is no white-space allowed in between the different tokens). The terminal symbols* $\underline{\texttt{digit}}$ *and* $\underline{\texttt{stringchar}}$ *represent character classes,* $\underline{\texttt{digit}}$ *containing all digit characters and* $\underline{\texttt{stringchar}}$ *containing all characters that can occur in a string literal. They are thus a shorthand notation saving us from having to write down a declaration for every possible character in these classes.*

As we can see, we define empty operators (those with pattern ϵ) for both constructions which contain neither terminal symbols nor operands. Such operators in general are very useful for optional parts of expressions and allow us to have single-digit numbers and empty string literals. We also define the string-denotation operator "_" which is both prefix and postfix.

```
FUN digit _        : [digits] → digits
FUN ϵ              : digits
FUN digit _        : [digits] → number
FUN stringchar _   : [chars] → chars
FUN ϵ              : chars
FUN "_"            : [chars] → string
```

Figure 2.8: Signature of example 6

Invisible Mixfix Operators

Apparently, some mixfix operators do not even contain *any* terminal symbols, but only operands (if any), making them neither prefix, postfix nor infix. We call these operators *invisible*. Several of these are common in most programming languages.

Of special interest to us are those invisible operators that are nullary (i.e. empty operators for optional expression parts), unary (i.e. converter operators for implicit coercion) and binary (i.e. concatenation operators for juxtaposition), because these occur most often. That is why we do not want to forbid them in general as user-defined operators.

Example 7 *In programming languages, where there are primitive types with inclusion relations, there exist unary invisible converter operators for implicit type coercion. These can be mimicked with operators like the ones in figure 2.9.*

```
FUN int                                   : SORT
FUN long                                  : SORT
FUN float                                 : SORT
FUN double                                : SORT
         -- int ⊆ long ⊆ float ⊆ double
FUN _                                     : [int] → long
FUN _                                     : [int] → float
FUN _                                     : [int] → double
FUN _                                     : [long] → float
FUN _                                     : [long] → double
FUN _                                     : [float] → double
```

Figure 2.9: Signature of example 7

To avoid problems of circular application of invisible operators, we will introduce restrictions on their applicability in the course of this thesis which prevent the occurrence of these problems.

The invisible operators can also be used for other purposes like in the following example.

Mixfix Operators for Container Types

Example 8 *The mixfix operator signature in figure 2.10 would allow the denotation of comma-separated display or comprehension expressions like* $\{1\ ,\ 5+3\ ,\ 7\}$ *or* $\{\mathsf{a}|\mathsf{a} > 5\}$[3]*. This is achieved by combining several different kinds of mixfix operators, including converters and empty operators.*

The indexing operation _ [_] *to be used on* `array` *or* `map` *expressions is another example of a full-fledged mixfix operator.*

We introduce the type variable operators `A` *and* `B` *which allows us to introduce type constructor operators generic over these variables.*

```
VAR   A                  : SORT
VAR   B                  : SORT
FUN   _ *                : [SORT] → SORT
FUN   _ +                : [SORT] → SORT
FUN   ϵ                  : A*
FUN   _                  : [A + ] → A*
FUN   _ , _              : [A , A + ] → A+
FUN   _ , _              : [A , A] → A+
PREC  _ , (_ , _)

FUN   _ | _              : [A , bool] → A*

FUN   seq _              : [SORT] → SORT
FUN   array _            : [SORT] → SORT
FUN   set _              : [SORT] → SORT
FUN   bag _              : [SORT] → SORT
FUN   map _ to _         : [SORT , SORT] → SORT
FUN   maplet from _ to _ : [SORT , SORT] → SORT

FUN   < _ >              : [A * ] → seq A
FUN   { _ }              : [A * ] → array A
FUN   { _ }              : [A * ] → set A
FUN   { _ }              : [A * ] → bag A
FUN   _ ↦ _              : [A , B] → maplet from A to B
FUN   { _ }              : [(maplet from A to B) * ] → map A to B

FUN   _ [ _ ]            : [array A , nat] → A
FUN   _ [ _ ]            : [map A to B , A] → B
```

Figure 2.10: Signature of example 8

[3]Such expressions might look strange in a functional language at first, but here we are not concerned with the semantics or implementation details of such constructions, but only with their parsing. We are aware that the denotation of set comprehensions necessitates mechanisms to identify the variables the set is quantified over.

Application and Tupling Operators

Most languages allow the definition of *functions* that can be applied to *tuples*. Both these constructions, application and tupling can be treated as mixfix operations.

Example 9 *Tuples are similar to arrays in form, but more complex in typing, because they require* structures over types *to describe them. Since we allow the introduction of mixfix operators also as type constructors, this leads us to the* multi-level mixfix signature *in figure 2.11.*

We declare an auxiliary type constructor `commalist _` *over sequences of types which we can use to describe all comma lists where the types of the individual elements correspond to the types in the type-sequence. For example, the expression* 3 , "`foo`" *would be of type* `commalist(nat , string)`.

The type constructor `product _` *in turn describes the tuples, i.e. comma-lists inside parentheses, corresponding to a sequence of types.*

Finally, the application operator _ _ *allows the denotation of prefix application of a function to an argument of its domain type. If the domain type is a product type, the function can be applied to tuple expressions, as expected.*

```
FUN seq _        : [SORT] → SORT
FUN <>           : seq SORT
FUN _ :: _       : [SORT , seq SORT] → seq SORT

FUN commalist _  : [seq SORT] → SORT
VAR A            : SORT
VAR B            : SORT
VAR L            : seq SORT
FUN ϵ            : commalist <>
FUN _            : [A] → commalist(A :: <>)
FUN _ , _        : [A , commalist(B :: L)] → commalist(A :: B :: L)

FUN product _    : [seq SORT] → SORT
FUN ( _ )        : [commalist L] → product L

FUN _ _          : [A → B , A] → B

FUN int          : SORT
FUN long         : SORT
FUN float        : SORT
FUN double       : SORT
VAR i            : int
VAR l            : long
VAR f            : float
FUN d            : double
FUN g            : product(int :: long :: float :: <>) → double
DEF g(i , l , f) == d
```

Figure 2.11: Signature of example 9

Programming Statements

Finally, most built-in *programming statements* can not be characterized as only prefix or postfix or even infix operations, but could be easily defined as mixfix operations.

Example 10 *The imperative programming statements of Pascal could be declared using the mixfix operator signature in figure 2.12.*

The boolean operand of the type `stmt _` *has been introduced because of the so-called* dangling else ambiguity *caused by the operators* `if _ then _` *and* `if _ then _ else _`*. If the operand is true, the statement does not have the operator* `if _ then _` *at the end. Only those statements which have that property are allowed between* `then` *and* `else`*. This is why there is only one correct interpretation for the right-hand-side of statement* `P`*. Thus, the ambiguity is solved by the* typing *of the statement operators.*

```
FUN int                  : SORT
FUN bool                 : SORT
FUN stmts                : SORT
FUN stmt _               : [bool] → SORT
FUN var _                : [SORT] → SORT
VAR A                    : SORT
VAR m                    : bool
FUN if _ then _          : [bool , stmt m] → stmt false
FUN if _ then _ else _   : [bool , stmt true , stmt m] → stmt m
FUN while _ do _         : [bool , stmt m] → stmt m
FUN for _ := _ to _ do _ : [var int , int , int , stmt m] → stmt m
FUN _ := _               : [var A , A] → stmt true
FUN repeat _ until _     : [stmt m , bool] → stmt true
FUN begin _ end          : [stmts] → stmt true
FUN ϵ                    : stmts
FUN _                    : [stmt m] → stmts
FUN _ ; _                : [stmt m , stmts] → stmts

FUN P                    : stmt m
VAR S                    : stmt true
DEF P ==                   if m then while m do if m then S else S
```

Figure 2.12: Signature of example 10

Conclusion

Apparently, the declaration of mixfix operators is closely related to the definition of grammars and type systems on the resulting parse trees which is one of the reasons for the approach we will take with dealing with user-defined mixfix operators.

2.3.2 Why User-Defined Mixfix Operators

Naturally, there could be multitudes of applications where a user-defined mixfix operator would be much more expressive than the given ones of a language, since they can resemble natural language constructs and common complex mathematical constructs more closely than a simple prefix function application expression. The readability of programs could thus be greatly enhanced by their usage with all advantages that this entails from a software engineering standpoint.

Especially in non-computer-science communities which nevertheless use a lot of programming (like physics, mathematics, engineering, economics or psycholinguistics), but do not necessarily share the same nomenclatures or denotation styles in their theoretic work, it could greatly benefit both the programming process as well as the understanding of the program for others in the same field, if those nomenclatures and styles could be introduced into the used programming language *by the users themselves* and consequently freely used.

We can go even further and say that programming languages incorporating user-defined mixfix operators could be customized to fit the native language used by the programmers, as that is probably the language they tend to think in and understand best. Such customizations could lead to very literate programming (in one language) instead of the usual possibilities the users are stuck with, i.e. using only the programming language's host language, e.g. English, or mixing the constructs of the host language with identifiers from the native language or using some sort of preprocessing outside of the language to mimic the host language constructs with native ones, basically adding the need for another compiler.

However, programmers don't want to have to write compilers to make their programs more readable — most of them wouldn't know how or want to —, they want to get on with solving the problems at hand and to do that use a language that gives them the possibility to write readable programs from the start.

It should be noted that such language-customizations can also lead to situations where programs from one community (or individual) are totally unreadable to other people (outside this community). But the same is true for domain-specific language-solutions anyway (and possible in any all-purpose programming language), so we do not see a drawback in this. As long as the syntactic libraries used by a community are accessible by those that want to read their programs, processes of making such programs understandable to them should be possible (for instance by mapping the operator set of a program into another operator set and then using an according automatic program transformation).

2.3.3 Common Practices

Domain Specific Languages

Almost no general programming language allows the definition of mixfix operators by the user. Thus, if a programming community opts for more readable programs, they often have to design a programming language dedicated specifically for their domain of interest. The result, not very surprisingly, is often less than acceptable from a compiler constructor's point of view, because it cannot be efficiently parsed or compiled[4].

On the other hand, if compiler construction experts are involved in the language design process, common practices that are mostly aimed at deriving an *efficient* compiler often restrict the language so much that readability is greatly hampered for the actual users.

[4]The author of this thesis has had to deal with these kinds of problems designing the first TTCN-3[12] compiler, TTCN-3 being a language designed and standardized by the telecommunication industry and ETSI[13] for the specification of test cases

So, there seems to be a dichotomy between *readability* and *parsing efficiency*, even though that sounds like a contradiction. *The easier readable a text is, the easier it should be parsable.*

All-Purpose Programming Languages

The other choice open for the users is to use a general all-purpose programming language that is *not* specifically tailored to any domain and thus suited equally (badly) to most of them regarding readability. The established general programming languages force their syntactic idiosyncrasies and restrictions on the user, only so that the compiler is able to efficiently parse and compile the programs. A lot of design decisions taken by the language-designers serve this purpose, so that in effect the compiler or at least the parsing algorithm to be used for a language influences the language itself, instead of the other way around.

This has very undesirable side-effects. If the language is not so restricted that *every* program is easily parsable[5], the programmers need to understand the *parsing process* to write efficiently parsable programs. They have to add a lot of formal noise (parentheses, annotations, etc.) to the essence of their program which leads to distortion and unreadability, clouding the actual intention of the programmer. Also, when the compilation process changes for the language, for whatever reason, formerly easily parsable programs might suddenly be less easily parsable because the parsing algorithm was not part of the language specification[6].

Context-Free Grammars and LR Parser Generators

Normal compiler architectures use parsers for *unambiguous context-free grammars* as the tool for the syntactic analysis of the program. This means that every program should only have one *syntactic* interpretation describable by such a grammar. If this syntactic interpretation obeys certain semantic constraints (like type- correctness) imposed by the semantics of the language, the program is accepted by the compiler and can be processed further. Otherwise, it is rejected.

To make a context-free language unambiguous, it is often restricted in such a way that it belongs to the class of languages which are LR(k)-parsable (which is known to be unambiguous). For these languages, comfortable bottom-up parser generators exist which make the actual implementation of the parser easy. Mostly only the grammar has to be provided and actions added to it that the parser should take on recognising the different language constructs (normally constructing an abstract syntax-tree that is the basis for further analysis).

One restriction aimed at unambiguity of context-free grammars is the use of *precedence levels* for the different operators that are built into the language. If such an operator has a higher precedence than another, then it should be applied first in an expression which uses a combination of both operators.

Example 11 *If* $\_ * \_$ *has a higher precedence than* $\_ + \_$*, then* $\mathtt{a} * \mathtt{b} + \mathtt{c} * \mathtt{d}$ *should be interpreted as* $(\mathtt{a} * \mathtt{b}) + (\mathtt{c} * \mathtt{d})$*.*

Such merely syntactic restrictions often result in behavior which seems very strange to users which do not know all of the reasons behind them, especially when combinations of operators not known from basic algebra are involved. Expressions which are completely type-correct and semantically unambiguous in the unrestricted language are nonetheless *rejected* by the parser. Adversely, semantically unambiguous, but syntactically ambiguous expressions in the unrestricted

[5]most are that restricted

[6]which it almost never is

grammar are *accepted* by the restricted grammar, but with the wrong syntactic interpretation, which leads to their rejection by the semantics check, afterwards.

Example 12 *Take any of the possible precedence level relations for the operators in the signature in figure 2.13. Even though the definitions of* $\texttt{E}_1$, $\texttt{E}_2$, $\texttt{E}_3$, $\texttt{E}_4$ *are all unambiguous, no precedence level relation satisfies all of them and* $\texttt{E}_4$ *cannot be satisfied by any precedence level relation.*

```
FUN 1      : nat
FUN _ + _  : [nat , nat] → nat
FUN <>     : seq nat
FUN _ :: _ : [nat , seq nat] → seq nat
FUN # _    : [seq nat] → nat

DEF E1     == # 1 :: <> + 1
                -- (# (1 :: <>)) + 1
                -- (_ :: _) > (# _) > (_ + _)
DEF E2     == # 1 + 1 :: <>
                -- # ((1 + 1) :: <>)
                -- (_ + _) > (_ :: _) > (# _)
DEF E3     == # <> + 1 :: <>
                -- ((# <>) + 1) :: <>
                -- (# _) > (_ + _) > (_ :: _)
DEF E4     == # <> + 1 :: # 1 + 1 :: <> :: <>
                -- ((# <>) + 1) :: (# ((1 + 1) :: <>)) :: <>
```

Figure 2.13: Signature of example 12

Even if typing is *not* considered, there are nevertheless many context-free languages which are not in LR(k) and still unambiguous. *Also, even in most ambiguous languages, as we know from natural languages, lots of expressions are still unambiguous.* Allowing the users to freely define their own mixfix operators can easily and will in most cases make the language generally ambiguous. Those are some reasons why normal programming languages don't allow it and restrict the mixfix operators built into the language so much. However, this approach to disallow all constructs that introduce the *possibility* of ambiguity restricts the languages more than necessary, excluding not only all the pathological ambiguous cases, but also many perfectly valid unambiguous ones. Thus, the main goal of this work is to find ways of dealing efficiently with the latter kind of expressions without having to restrict the language in general on a purely syntactic level.

Generalized Parsers and Semantic Filters

Because of their inherent ambiguities, user-defined mixfix operators demand different parsing algorithms as the grammar or the parser of the language must change according to the operators defined to recognize them. This is not easily incorporated into the classical parser generator framework, another reason why these operators are normally avoided.

Even if *generalized parsers* are used that can deal with arbitrary, possibly context-free grammars by computing all syntactically correct interpretations, the ambiguity problem still arises and cannot be dealt with efficiently in general as mixfix expressions can have an exponential or even infinite[7] number of syntactic interpretations.

Example 13 *If we take the ambiguous variant of the dangling else problem, as introduced by the operators in the signature in figure 2.14, and an expression with* $2n$ *if-statements but only* n *else-clauses (which has length* $\omega(n)$*) there are* $\omega(3^n)$ *syntactic interpretations possible.*

```
FUN  if _ then _        : [bool , stmt] → stmt
FUN  if _ then _ else_  : [bool , stmt , stmt] → stmt
FUN  S                  : stmt
FUN  B                  : bool
DEF  E ==               (if B then)^2n S (else S)^n
```

Figure 2.14: Signature of example 13

Example 14 *If we have a binary operator* $\_ \oplus \_$ *and nullary operators* $\mathtt{x_i}$ *then an expression* $\mathtt{x_0} \oplus \ldots \oplus \mathtt{x_n}$ *has close to* 4^n *syntactic interpretations.*

To find a single unambiguous syntactic interpretation amongst these myriads of possible ones, so-called *semantic filters* can be used. This technique is explored by Visser [19], [45] and Thorup [40], [39].

If such filters are employed *after* parsing, in most cases they *cannot* be efficient and their integration with the parsing algorithm to make them more efficient is complicated (and very close to our two-level grammar approach), especially in a parser with a changing grammar during parse-time[8].

We can conclude that the state-of-the-art approaches to parsing do not seem to be adequate for dealing with user-defined mixfix operators.

2.3.4 The Problems to Solve

Obviously, the main problem caused by freely user-definable mixfix operators is the possible ambiguity of the expression language. Thus, if the programmer can introduce mixfix operators, several questions arise.

- How can we ensure that expressions using these operators remain syntactically unambiguous, i.e. have exactly one type-correct syntactic interpretation?
- How can we ensure that, if several syntactic interpretations are possible, the one intended or preferred by the programmer is chosen by the compiler?
- How can we provide efficient parsing means for possibly ambiguous expressions?
- How can we resolve the ambiguities that may arise between the mixfix operations introduced by the programmer and those of the built-in core language, i.e. the meta operators for annotations and introduction of operators?

All these problems shall be addressed in this thesis and we will try to solve them in a satisfactory manner, yielding a very expressive language that is still efficiently parsable in the positive case that there actually is an acceptable type-correct syntactic interpretation for every expression to be parsed. Searching for such

[7]if invisible operators without restrictions are allowed
[8]due to local introduction of operators

an interpretation of an expression which has none or several of them can still be very inefficient in some cases and can be solved only heuristically and pragmatically (e.g. by using a parsing timeout and rejecting some expressions as *too complex to parse*).

The key to our solution is the use of *two-level grammars* that are derived from certain kinds of given mixfix operator sets, called *mixfix operator signatures* and so called *ad-hoc precedences* for these operators in such a way that these grammars, together with a few restrictions, are *unambiguous by construction.* This is a very small, but nevertheless very powerful, subclass of the class of two-level grammars. It is known that for *arbitrary* two-level grammars there cannot be efficient parsing algorithms dealing with their languages *in general.*

Therefore, we will characterize those signatures and precedence relations for which it is *possible* to find such efficient algorithms. We will give these characterizations as efficiently computable properties and discuss them formally. We also will discuss the possibilities of *parsing* the languages of this class of two-level grammars efficiently.

2.4 A Functional Mixfix Expression Language

Before exploring the possibilities for ambiguity in mixfix expressions, let us envision the kind of functional mixfix expression language we want to develop.

We want to be able to integrate the following constructs into a functional mixfix expression language:

- user-definable mixfix operators with ad-hoc precedences
- higher-order functions and built-in application and composition
- type and scope annotations
- overloading and genericity
- partially instantiated operators (called *sections*)

The functional host-language itself must provide additional means to declare and define the actual mixfix operators to be used and their ad-hoc precedences, as well as mechanisms for modularization, export and import.

2.4.1 Goals

Let us summarize the goals that we want to achieve with our functional mixfix expression language.

- *unambiguous expressions*

  All accepted expressions of the language shall be type-correct, as well as syntactically unambiguous. When there is an ambiguity, the parser shall detect and report it. The set of unambiguous accepted expressions shall be as large as possible.

- *efficient analyzation and parsing*

  The techniques employed to cope with the language, i.e. acceptance and rejection of expressions, shall be efficient, preferably as efficient as normal parsing approaches. It is our view that by early, local disambiguation the efficiency of the parsing of otherwise ambiguous grammars is greatly heightened as opposed to global filtering of the set of parse trees, though the latter might be able to recognize more unambiguous expressions.

- *independence of parser paradigm*

  Since ambiguity is a concept that is independent of the paradigm which is used to parse the language, the users should only be required to understand the ways their introduced operators *could* cause ambiguity, but not necessarily the parsing algorithm, to be able to denote efficiently parsable, unambiguous expressions.

  Thus, all ambiguities are treated from the viewpoint of the user-defined operators instead of the viewpoint of the parsed grammar.

- *freedom and power of expressiveness*

  By allowing the definition of almost arbitrary mixfix operators, the freedom for the denotation of expressions is greatly enhanced. By adding the ability to mix these operations with high-level functional language constructs, greater readability can be achieved in such a language. Also, domain-specific notations can be introduced more easily.

- *composability for modularization*

  Since the grammars should be derivable from different modules of the program, they shall be fully compositional, allowing for easy integration.

2.4.2 Means

What means will we use to arrive at the above goals?

- *mixfix operator signatures and ad-hoc precedences*

  The means for the programmer to describe the mixfix operators are *mixfix operator signatures* where each mixfix operator is assigned a functionality, i.e. a type, describing the types of the operands and the result of the operator. Additionally, the user can give *ad-hoc precedences* between operators, describing *preferred* syntactic interpretations, if necessary or desirable.

- *composable two-level grammars*

  From these signatures and precedence relations, specific kinds of unambiguous, composable two-level grammars and grammar constraints are generated that allow the efficient parsing of expressions containing these operators.

- *two-staged parsing*

  The parsing process becomes two-staged, where the first stage parses the coarse structure of the host language and collects the user-defined operators and precedences for a scope, and the second stage parses the unparsed fragments inside that scope as mixfix expressions with the help of the grammars derived from the signatures.

2.5 Causes of Ambiguity

The restrictions on the mixfix operator signatures and precedences imposed by our language shall be as few and as natural as possible to make them as understandable as possible to the users. We will examine all causes for ambiguity following naturally from the form and functionality of the given operators and the type system. Every restriction we impose shall be motivated directly by these causes of ambiguity.

Unlike in classical language design approaches, the restrictions shall *not* be dictated by restrictions of the possible parsing algorithms to be used for the grammars. Though the grammars can be optimized for different parsing algorithms, this should not matter to the user. The user should not have to know about the parsing algorithm to be able to write efficiently parsable programs.

Since our grammars only recognize a subset of the unambiguous expressions, we would have to use the more general and possibly inefficient approaches to include the rest of the unambiguous expressions into our recognized language. However, if we want to remain efficient, this should be avoided.

Some possible causes for ambiguity are only syntactic in nature, while others also depend on typing/semantics which is why ignoring typing completely in describing an unambiguous grammar does not seem sensible.

2.5.1 Ambiguity of Mixfix Expressions in General

In this section, we will describe our concept of user-defined mixfix operators and the mixfix expressions that can be formed using such operators.

Mixfix Operator Patterns: Operand Separators and Placeholders

Definition 1 *Every* mixfix operator pattern *is a sequence of n* separators*, interleaved by $n-1$* operand placeholders $\mathtt{s}_0$ _ $\mathtt{s}_1$ . . . _ $\mathtt{s}_\mathtt{n}$.

Definition 2 *A* separator *is a (possibly empty) sequence of* token symbols *which is produced from the actual input by lexical analysis*[9] *before parsing. Each separator separates either:*

- *two operand placeholders,*
- *the first placeholder from the beginning of the operator,*
- *the last placeholder from the end of the operator.*

Definition 3 *If a separator between two operand placeholders is the empty sequence, these operands are called* adjacent*.*

We refer to the placeholders inside an operator by the index of the separator *after* that placeholder. So, they are numbered 1 .. n.

Definition 4 *If separators s_{i-1} and s_i are both non-empty the i-th placeholder is called* enclosed*.*

Syntactic Operator Categorizations: Fixity, Openness, Visibility, Arity

Definition 5 *If a separator at the beginning of a pattern is empty, the operator is called* left-open*, otherwise it is called* left-closed*. An operator with an empty last separator is called* right-open*, if the last separator is non-empty, the operator is called* right-closed*.*

Definition 6 *Left-closed operators are also called* prefix *operators, while right-closed operators are also called* postfix *operators. Operators which are both prefix and postfix are called* closed *operators.*

Example 15 *The built-in parentheses operator* `(_)` : `[A]` $\rightarrow$ `A` *is an example of a closed unary operator.*

[9]The lexical token analysis is outside the scope of this thesis. We assume that an input sequence of characters is translated by the lexical analysis into a sequence of token symbols by grouping the character symbols together into token symbols

Definition 7 *Operators where all separators are the empty token sequence are called* invisible*. All other operators are called* visible*.*

Definition 8 *The* arity *of a mixfix operator pattern is equal to the number of placeholders in the pattern.*

Definition 9 *The* outer operands *of a mixfix operator pattern are those operand placeholders which appear before the first non-empty separator and after the last non-empty separator in that pattern.*

The inner operands *of a mixfix operator pattern are those operand placeholders which appear after the first non-empty separator, but before the last non-empty separator in that pattern.*

Example 16 *The operator pattern* `if _ then _ else _` *has the four separators* `if`*,* `then`*,* `else` *and* ϵ*. It is a visible prefix operator which is right-open and has no adjacent operands. Its arity is 3. It has two inner operands (after* `if` *and after* `then`*) and one outer operand (after* `else`*).*

Mixfix Expressions

A *mixfix expression* in general is either a pattern of a nullary mixfix operator (i.e. one with no operand placeholders) or the substitution of the operand placeholders in a non-nullary mixfix pattern with expressions which again are mixfix expressions.[10] An expression is thus a sequence of the separators of the operator pattern interspersed with operand expressions between the separators.

This constructive top-down point of view already implies a syntactic *interpretation* of such an expression, i.e. the application of the instantiated operator to its operands. It thus also is closely linked to the notion of *parse tree*, as a parser tries to find such an interpretation for an expression. Only those expressions which are *type-correct*[11] are of real interest to us. This means that typing can not be ignored in parsing mixfix expressions.

Unfortunately, an expression, before parsing takes place, is just a sequence of token symbols which can potentially have several of the above-mentioned type-correct interpretations. A parser should assign an interpretation to such a sequence of symbols. In our approach, this assignment should depend also on the *context* of the expression, specifically, its *demanded type*, i.e. the expected result type for the expression.

Ambiguous Mixfix Expressions

If there exist several interpretations of an expression which have the same type we call the expression *ambiguous*. Such ambiguity comes in two flavours.

If we can find several structurally different parse trees for an expression, we have a *syntactic ambiguity*.

If a single parse tree can have different meanings (i.e. type annotations *inside* the tree that do not influence the type of the whole tree), we have a *semantic ambiguity*.

In parsing, we are mostly interested in finding a unique type-correct parse tree for every expression in any given demanded type where we ignore the possible semantic ambiguities occurring in that tree. The semantic ambiguities should be dealt with afterwards by a semantic analysis of the syntactically unambiguous tree found by the parser.

[10] The case of the nullary operator is obviously just a special case of the instantiated operator, since all operands in a nullary operator are already instantiated — there are none.

[11] i.e. where the types demanded for the operands of the instantiated operator match the inferred types of the operand expressions it is applied to

Operator Backbones

Another helpful syntactical characterization of expressions is the *segmentation* into expression parts which we call *operator backbones* (which are not necessarily proper expressions themselves). An operator backbone is the *inner part* of a *visible* operator pattern where the operand placeholders are replaced by expressions (which again can be segmented into backbones). The inner part of an operator pattern is the sub-pattern that starts with the first token of the first non-empty separator and ends with the last token of the last non-empty separator (i.e. everything except the outer operands). It is obvious that only visible operators can have backbones, because all invisible operators have only empty separators.

Example 17 *The backbone of the operator pattern* `if _ then _ else _` *is* `if _ then _ else`. *The backbone of pattern* $\_ + \_$ *is* $+$.

2.5.2 Ensuring Syntactic Unambiguity

Steps toward Syntactic Unambiguity

Our road-map of ensuring syntactic unambiguity of the expressions to be parsed are the following three steps:

1. Find a unique segmentation of the token sequence into *operator backbones* of visible operators. If more than one such segmentation is found, reject the expression as *backbone ambiguous*. Also, find a unique segmentation of the token sequence of each operator backbone into its *separator* and *operand* parts. This means that it should be determined for every token of the backbone sub-sequence which specific separator token of the instantiated operator it corresponds to. Thereby, also the operand parts *between* the found separators are uniquely identified. If no such segmentation can be found for one backbone sub-sequence, then also reject the expression as *backbone ambiguous*.

2. Find a unique segmentation of the operand parts of *adjacent inner operands* of the operator of each backbone to find out which sub-expression token sequence corresponds to which inner operand of the operator, respecting several restrictions pertaining to adjacent operands[12]. If more than one possible segmentation can be found, reject the expression as *adjacent-operand ambiguous*, i.e. ambiguous because of violated adjacent operand restrictions.

3. Find a unique *precedence* ordering of the operator backbones towards each other into a type-correct parse tree respecting *natural* and *ad-hoc* precedence relations between operator instantiations, and also taking restrictions on invisible operators and adjacent *outer* operands into account. If more than one such type-correct precedence ordering can be found, reject the expression as ambiguous because of precedence conflicts.

Since inner and outer operands are again expressions, these steps must also be applied recursively to the operand expressions to ensure that the whole expression is syntactically unambiguous.

Parsing Methods

- We achieve step 1 by backbone parsing of the expression. This determines, as long as there is no backbone ambiguity, both the unique segmentation of an expression into backbones, as well as the unique mapping of each token symbol to its place inside the backbone it occurs in.

[12]introduced in section 5.5

- Step 2 is achieved by finding a unique left-weighted interpretation via left-weighted parsing[13] of the expression, we also find the unique segmentation of the operand parts of the expression, if it exists. Actually, this achieves already more than is necessary for step 2, i.e. it also assigns expression parts to *outer* adjacent operands.
- Finally, we can achieve step 3 by finding a unique re-ordering of the operator instantiations in the left-weighted parse tree (found in step 3) into a different, type-correct parse tree consistent with the precedence relations, which yields a syntactically unambiguous parse tree for our expression.

Example 18 *Take the right-precedent operators*

- `_ _ a _ _ b _ _` : `[T , T , T , T , T , T]` $\rightarrow$ `T`
- `c` : `[]` $\rightarrow$ `T`
- ϵ : `[]` $\rightarrow$ `T`

Determining the correct syntactic interpretation for the token sequence `c c a b c c c c a c c b a a c c b c c c c a b b c c c a b` *would be done in the following steps:*

1. *backbone parsing would yield the following backbone structure:*
   `c c (a b) c c c c (a c c b) (a (a c c b) c c c c (a b) b) c c c (a b)`
2. *left-weighted parsing would yield the following parse tree*[14]*:*
   `((c c a b c c) (c c a c c b) a (a c c b c c) (c c a b) b c c) c a b`
3. *finally, reordering the nodes of the tree according to right-precedence of the operators, we get:*
   `(c c a b c c) (c c a c c b) a (a c c b c c) (c c a b) b c (c c a b)`

Actually, since all unambigous mixfix expressions must have exactly one left-weighted interpretation[15], we could also first try left-weighted parsing of the expression. If this parsing process yields no ambiguity, then no backbone ambiguity can exist, either. But, if there *is* an ambiguity, we have to use backbone parsing to determine whether it is backbone ambiguity or ambiguity caused by adjacent operands.

Proof of Unambiguity

- Once we have found a unique backbone segmentation of an expression, we only have to resolve precedence ambiguity issues *between* these backbones in step 3. The ambiguity issues *inside* the backbones can be addressed separately for each backbone, using step 2.
- Once we have found a unique segmentation of one backbone into its separator and its operand parts, we only have to resolve ambiguity between *adjacent* inner operands of that backbone, which is addressed in step 3.
- After having found for each inner backbone operand a unique segmentation into the adjacent inner operands of the original operator pattern of that backbone, we can apply the disambiguation process recursively on each inner operand, because each operand must be a proper mixfix expression.

[13]see section 5.5.1
[14]the ϵ operator is not shown
[15]as we will show in section 5.5.1

- After we have found a unique precedence order of the backbones, finding out for each sub-expression, which backbone is the backbone of the root operator and which backbone concatenation corresponds with which outer operand of that operator (the same way as for inner operands), we have found a unique parse tree. This means that there cannot be any syntactic ambiguity present.

If all of these tasks can be completed successfully, there is no more room for ambiguity in the interpretation of the expression and parsing is thus successfully completed. Therefore, we must identify those situations where one of the above tasks cannot be (efficiently) completed and find the causes of the ensuing ambiguity.

Special care must be taken of the segmentation of adjacent operands and the topological ordering of operator backbones into parse trees when invisible operators are involved. Unfortunately, these operators introduce a lot of pathological ambiguity situations. But since there are very common and useful applications of such operators, forbidding them cannot be the answer. That is why we have to devote some effort to deal with invisible operators, even though it might seem pointless from a cursory observation of these matters.

Since all of these kinds of ambiguity are independent from each other, we will treat them separately in the rest of this section, assuming for each of them that the other kinds of ambiguity do not arise simultaneously.

2.5.3 Fixity and Precedence

Precedence-related ambiguity is of a syntactic nature, causing an expression to have different syntactic interpretations if conflicting precedences between operators are defined. This can be the case if for any subexpression, it can not be unambiguously determined, which operator is the root of the parse tree of that subexpression.

Right-Open vs. Left-Open Operators

Lemma 1 *Precedence-related ambiguity can only occur if more than one* open *operator instantiation is present in an expression. One of these instantiations must be of a* left-open *operator and one must be of a* right-open *operator. The right-open operator instantiation must occur to the left of the left-open operator instantiation in the expression.*

Proof 1 *Assume that there occurs no backbone ambiguity and no ambiguity in regard to adjacent operands.*

- *If there is no left-open operator instantiation present in the expression, then all instantiated operators are prefix operators and thus, the root must be the leftmost prefix operator and all its rightmost descendants (if it is a right-open operator) also must be prefix operator instantiations. None of these descendants can be the root of the expression, because then the leftmost operator instantiation could not have a root operator above it, since it would have to be a leftmost descendant of that root. Thus, the root of every subtree can unambiguously be determined.*

- *Analogously, if there is no right-open operator instantiation present in the expression, then the rightmost postfix operator must be the root of the expression.*

- *If at least one left-open operator instantiation and at least one right-open operator instantiation is present and all right-open operators appear to the right of all left-open operators in an expression, then the root must be the leftmost right-open operator and also the rightmost left-open operator, i.e. an infix operator. If there were no such infix operator present, then the rightmost* *left-open operator could not have a root above it (as all operators to its right are not left-open), neither could the leftmost right-open operator have a root above it (as all operators to its left are not right-open). No ambiguity of its rightmost operand can occur because there are no left-open operators present and no ambiguity of the leftmost operand can occur as there are no right-open operators present in it. Hence, the root of the tree and the roots of all subtrees can unambiguously be determined.*

- *Thus, we need at least one right-open operator instantiation to the left of at least one left-open operator instantiation to get a possible precedence-related ambiguity, i.e. where both the left-open and the right-open operator could be the root of an interpretation of a subexpression.*

A common special case here is that an *infix* operator (i.e. one that is both left-open and right-open) is instantiated twice in an expression.

Type-Related Precedence Ambiguity

Another condition for this ambiguity to occur is that the two operator instantiations must both have the same result type, meaning that in the same context demanding that type, both of them could be the topmost operator in an interpretation of the expression. **Because of this type-related issue, resolving precedence ambiguities cannot be done reasonably without using type information.**

Example 19 *The situations that can occur can become quite complex, even if no polymorphism occurs. The same expression can be ambiguous in one type and unambiguous in another, as shown in figure 2.15. Here, the expression* `abs n − n!` *is ambiguous for type* `nat`*, but it is unambiguous for type* `int`.

```
FUN abs _ : [int] → nat
FUN _ − _ : [nat , nat] → int
FUN _!    : [nat] → nat
FUN n     : nat
FUN E₁    : nat
DEF E₁ == abs n − n!
            -- abs(n − (n!))
            -- (abs(n − n))!
FUN E₂    : int
DEF E₂ == abs n − n!
            -- (abs n) − (n!)
```

Figure 2.15: Signature of example 19

Such typical precedence-problems can be solved by letting the user add so called *ad-hoc precedence relations*, stating the user's preferred precedence of possibly ambiguous operator instantiations. If one of the possibly ambiguous variants is preferred by the user, this variant will be accepted if it is type-correct. Otherwise, both variants will be rejected for being *possibly ambiguous.*

Example 20 *As can be seen in figure 2.16, no infix operators have to be involved so that precedence-ambiguity can occur between non-prefix and non-postfix operators.*

As these examples show, the programmer must be given a means to describe ad-hoc precedence relations, as shown in example 5 on page 17. These user-given

```
FUN  a _  : [B] →  A
FUN  b _  : [C] →  B
FUN  c _  : [A] →  C
FUN  _ e  : [A] →  A
FUN  d    : A
FUN  E    : A
DEF  E == a b c d e
              -- a (b (c (d e)))
              -- (a (b (c d))) e
```

Figure 2.16: Signature of example 20

relations must then also be checked for consistency, since they must fulfil several criteria to be acceptable as a disambiguating tool. We will give a full account of these conditions in section 5.6.2.

2.5.4 Converter Operators

We call *unary invisible* operators *converters* because that is what they are most useful for — implicit type conversion. Unfortunately, these operators can cause both syntactic and semantic ambiguities. For completeness sake, we still do not want to forbid them entirely, since their usefulness is widely known from normal programming languages. We thus have to restrict them in a manner that prevents the above-mentioned ambiguity.

How does this ambiguity manifest itself? Since the result of an application of a converter operator looks exactly like its operand, it is not clear how many such instantiations take place. The number of such instantiations could even be *infinite*. While typing might help sometimes in disambiguating such situations, it is still prudent to generally forbid multiple applications of converters to the same expression.

Example 21 *If multiple applications of converters were allowed, there are infinitely many parse trees for the expression* a *existent as shown in figure 2.17.*

```
FUN  _    : [A] →  seq A
FUN  _    : [seq A] →  A
FUN  a    : A
FUN  E    : seq A
DEF  E == a
              -- (_)a
              -- (_)((_)((_)a))
              -- ...
```

Figure 2.17: Signature of example 21

We must also take care with all converters where the operand type is unifiable with its result type (which could be useful for normalization purposes). For such operators, whenever they are applicable, it is not clear, whether or not an instantiation should take place.

Hence, pragmatically, we declare that *exactly one* converter is applied to every non-converted expression. For every type where no normalization operator is defined by the user, we must assume that the *identity* normalization operator exists.

2.5.5 Backbone Ambiguity

As explained before, there is another cause of possible ambiguity unrelated to precedence. It is purely syntactic in nature and is more related to the concepts of *conflicts* occurring in the construction of unambiguous LR-parsers.

Operator Backbones

Abstracting from precedence between operators and of typing, every expression can be seen as a sequence of so-called *operator backbones* (or *operator backbone instantiations*). An operator backbone is the sub-sequence of tokens of an instantiated visible operator, starting from the beginning of its leftmost visible separator up to the end of its rightmost visible separator.

If an expression is split into such a sequence, the hierarchy between the operator instantiations inside the expression is forgotten.

We call concatenations of backbones *backbone expressions.* The set of backbone expressions is a superset of the set of type-correct mixfix expressions, ignoring precedence, typing and problems caused by adjacent operands.

A more formal definition of backbones can be found in section 3.2 on page 47.

Example 22 *If* $\mathtt{v}_1$*,* $\mathtt{v}_2$ *and* $\mathtt{v}_3$ *are backbone expressions, the list of terminal symbols of the form* $\mathtt{[for]} +\!+\ \mathtt{v}_1\ +\!+ \mathtt{[:=]} +\!+\ \mathtt{v}_2\ +\!+ \mathtt{[to]} +\!+\ \mathtt{v}_3\ +\!+ \mathtt{[do]}$ *is a possible backbone of the operator with pattern* `for _ := _ to _ do _` *.*

We can derive a (generally ambiguous) context-free *backbone grammar* which describes all such concatenations for a given set of mixfix operators (cf. 3.2.1). The language recognized by this grammar is the set of backbone expressions.

In essence, this grammar abstracts from typing, precedences between operators and adjacent operands inside operators in the following way:

- All left-open or right-open operands (and those directly and indirectly adjacent to them) are ignored (by stripping them from the operator patterns) and the parse trees thus are *flattened*, causing the inner parts of the operators to stand beside each other instead of being put into a hierarchy.

- The problems caused by adjacent operands between non-empty separators are ignored by subsuming adjacent inner operands into a single operand. This does not cause our backbone language to change, because adjacent operands are already a concatenation of expressions. Therefore, since *every expression* is a sequence of backbones, sequences of adjacent instantiated operands can already be seen as a backbone expression.

It should be easy to see how a mapping from parse trees to backbone interpretations (which is a sequence of sequence of tokens) can be defined (cf. definition 17 on page 47).

Thus, if an expression does not have an interpretation in the backbone language, it cannot have an interpretation in the expression language.

Moreover, if an expression is *unambiguous* in the backbone language, all possible syntactic ambiguity in the expression can only be related to precedence (which is influenced by typing) or adjacent operands.

If, on the other hand, there is more than one backbone interpretation for an expression we have an ambiguity in the expression language that can *not* in general be resolved by use of precedences or by the restrictions on adjacent operands.

Therefore, we must reject expressions which have this so-called *backbone ambiguity* as ambiguous during parsing, as we have no means to disambiguate them (other than letting the user add parentheses).

Backbone Ambiguity Caused By Overlap

The cause of backbone ambiguity are so-called *overlapping expressions*. Keep in mind that every expression can be seen as a concatenation of separators.

There are two cases of overlap:

- A backbone expression can be the prefix of a suffix of another expression. This situation we call *full overlap*.
- A backbone expression has a suffix (which must be a concatenation of separators) which can also be the prefix of a backbone expression. This situation we call *partial overlap*.

More generally, partial overlap occurs if there exist operators that can build both the backbone expression `u v` and the backbone expression `v w` where `u`, `v`, `w` are all separator concatenations. Then, the backbone expression `u v u v w v` exists and has the two backbone grammar interpretations `(u v) (u v w v)` and `u (v (u v) w) v` . Likewise, `v u v w v w` has the interpretations `(v (u v) w) (v w)` and `v (u (v w) v) w` .

Example 23 *Ignoring the ambiguities caused by precedences, the signature in figure 2.18 allows two type-correct interpretations of the expression* `| a | a | a |`*, either* `abs(a) * a * abs(a)` *or* `abs(a * abs(a) * a)` .

```
FUN | _ |  : [real] → real                 -- abs
FUN _ _    : [real , real] → real          -- _ * _
FUN a      : real
DEF a == −1
DEF E == |a|a|a|
               -- | (a (| a |) a) | == 1
               -- (| a |) a (| a |) == − 1
```

Figure 2.18: Signature of example 23

Full overlap occurs if there exist operators that can build backbone expression `u v w` where `u` and `w` are separator concatenations and `v` is a backbone expression. In that case, the backbone expression `u v v w` exists and has the interpretations `u (v) v w` and `u v (v) w`.

Example 24 *Consider the backbones of the operators of the signature in figure 2.19. The backbone expression* `if B then` *is a prefix to the separator suffix* `if B then S else` *of backbone expression* `if B then if B then S else`*. Therefore, there are two backbone concatenations for expression* `if B then if B then S else S` *possible,* `(if B then) (if B then S else) S` *and* `(if B then ((if B then) S) else) S`.

```
FUN  if _ then _ else _ : [bool , stmt , stmt] →  stmt
FUN  if _ then _        : [bool , stmt] →  stmt
FUN  B                  : bool
FUN  S                  : stmt
FUN  E                  : stmt
DEF  E ==                if B then if B then S else S
                            -- (if B then) (if B then S else) S
                            -- (if B then ((if B then) S) else) S
```

Figure 2.19: Signature of example 24

Dealing with Backbone Ambiguity

Determining ambiguity of context-free grammars is undecidable in general (see [4]), but we can try to find satisfying conditions for unambiguous backbone grammars. If these conditions hold, all syntactic ambiguities that can occur must thus have to do with precedences, as long as the restrictions for adjacent operands and converters are adhered to.

Deterministic Parser Construction

The existence of a deterministic parser is one satisfying condition for unambiguity. Therefore, we could try to construct such a parser, e.g. an $LL(1)$, $LR(1)$ or $LALR(1)$ parser. Constructing such a parser can be very costly, though.

Generalized Parsing

The most straightforward, still somewhat costly approach is the construction of a *generalized parser*[16] for the backbone grammar of the given operator set. Such a parser normally has a worst-case time efficiency of $O(n^3)$ and yields a representation of all syntactically valid derivations in a space of $O(n^2)$, where n is the length of the expression.

After constructing such a parser from the given operator set, the mixfix expression in question can be parsed with this parser. If only one parse tree is yielded, the grammar is unambiguous *for that expression.* The resulting parse tree then could be used to find the right precedence by other parsing means[17]. If possible, such a parser does not have to find *all* solutions, but can stop as soon as more than one is found.

As a rule of thumb, we can assume that most mixfix expressions will either be very short or highly regular and repetitive, as otherwise, a human reader would seldom be able to understand them. Also, almost no programmer will explicitly try to lead the parser up the garden path[18]. If the programmer has an unambiguous understanding of an expression in mind, then the ambiguities actually occurring are probably few. That is why such a general approach will make sense for most programs.

Deterministic Operators

Yet another solution could be to analyze the operator set itself, finding satisfying conditions that the construction of a deterministic parser *would* succeed without

[16]see [29] for a treatise on generalized parsers
[17]This approach was taken in our implementation, discussed in chapter 8
[18]except to test its capabilities which is pathological

actually constructing it, or finding the reasons *why it would not succeed*, giving the user hints why the operator set might be syntactically ambiguous.

A big help in making an operator set deterministic is the use of *identifying terminal symbols*, i.e. terminals that only occur in one (possibly overloaded) operator. This is not surprising — the same trick is used to make normal programming languages deterministic by using *keywords* for different constructions.

Operator Filtering

If the actual operator set is *filtered* for every expression as to the operators that can actually occur in it by looking at the terminal symbols present in that expression, this can greatly improve performance.

This approach might make sense especially for very long expressions where the set of actually occurring operators is usually very small in comparison to the length of the expression, so that the cost of analyzing the operators might be much lower than the cost of actually backbone-parsing the expression.

Assertion

If the operator set and then the expression is found to be possibly syntactically ambiguous, but the user knows that the expression is unambiguous anyway, they have to be provided with means to assure the parser of unambiguity (i.e. by annotation). If thus unambiguity can be taken for granted by the mixfix parser, it has to find at most one correct solution and can then terminate instead of continuing a costly search for other solutions which do not exist.

One should be aware, though, that this can be very risky, as the user can then also write down ambiguous expressions and declare them as unambiguous which will never be checked by the parser. If the user was wrong in assessing an ambiguous expression as unambiguous, this could lead to serious problems if the parser chooses a different interpretation as the one intended by the user.

Thus, a prudent programming environment should present, for every expression annotated as unambiguous, the interpretation found by the parser to the user so that the user can *check* their interpretation against the one found. Unfortunately, such an approach is probably not useful in general practice. But it could safely be applied to programs which already have been parsed and all expressions found unambiguous to speed up re-parsing time if that should become necessary.

2.5.6 Adjacent Operands

Our general mixfix operator pattern definition allows for two phenomena uncommon in most programming languages, *non-unary invisible operators* [19] and *invisible separators between operand placeholders.*

We recall that two operands are called *adjacent* if the separator between them is the empty sequence.

Again, for completeness sake, we do not want to generally forbid the occurrence of these phenomena, but, since they can cause ambiguities in their interaction, we must find restrictions that prevent these ambiguities.

Of course, first the question arises what invisible operators could be useful *for* in general.

[19] *Unary* invisible operators do not clash with adjacent operands.

Application Operator

Most importantly in our view, the absence of the application operator
`_ _ : [A → B , A] → B` which is an example of a binary invisible operator would be very inelegant in a functional language with higher-order functions (i.e. where operations can result in functions). But since this operator also has adjacent operands, forbidding the concept of adjacent operands altogether but allowing this special case would seem idiosyncratic at best.

Optional Expression Parts

Likewise, we can think of useful applications of *nullary invisible operators*, called *empty* operators, especially as a possibility to leave out parts of expressions which are optional.

Example 25 *In the mixfix operator signature in figure 2.20, we introduce the empty operator* ϵ : `A*` *to gain the possibility of denoting the empty sequence* $<>$ *without having to introduce the operator* `<>` : `seq A` *which should be avoided because it would cause an backbone ambiguity with the operator* $<$ `_` $>$: `[A * ] → seq A`.

```
VAR  A       : SORT               -- type variable

FUN  _ +     : [SORT] → SORT      -- non-empty commalist type
FUN  _       : [A] → A+           -- type inclusion
FUN  _ , _   : [A , A] → A+       -- commalist constructor
FUN  _ , _   : [A , A + ] → A+    -- commalist constructor

FUN  _ *     : [SORT] → SORT      -- possibly empty commalist type
FUN  ϵ       : A*                 -- empty comma sequence
FUN  _       : [A] → A*           -- type inclusion
FUN  _       : [A + ] → A*        -- type inclusion

FUN  seq _   : [SORT] → SORT      -- polymorphic sequence type
FUN  < _ >   : [A * ] → seq A     -- sequence bracket operator
```

Figure 2.20: Signature of example 25

Operator Name Expressions

The use of operator patterns stripped of all their operand placeholders, like
`if then else` instead of `if _ then _ else _`, (which we call *operator name expressions*), i.e. where all placeholders are instantiated with *empty placeholder expressions*, is another sensible and useful one in a language where functions and operators can be passed as parameters. We can allow such expressions because the lack of backbone ambiguity between operator patterns ensures also the unambiguity of such operator name expressions.

Prefix and Postfix Expressions

Unfortunately, beside these useful applications, there are myriads of unuseful or downright stupid ones that the user could introduce with the help of our general operator pattern concept. Lots of pathological ambiguity situations could arise which must be prevented by either disallowing certain kinds of operator patterns containing adjacent operands or by rejecting expressions where these ambiguities actually occur. Therefore, we must identify the situations where adjacent operands can have ambiguities, i.e. where it is not clear where the first operand ends and the next one begins.

Fortunately, our concepts of fixity help us in that respect. They allow us to introduce the notions of *prefix expression* and *postfix expression.* A prefix expression in our context is either an instantiation of a prefix operator (i.e. a visible, non left-open operator) or of a left-open operator with a prefix expression as its leftmost operand. A postfix expression can be defined analogously with the help of postfix operators.

We will see that we arrive at a useful and unambiguous subset of our mixfix expression language if we restrict the instantiation of adjacent operands with non-empty expressions such that

- the left operand must be a postfix expression (it shall not have ϵ as its root or one of its rightmost operands), while
- the right operand must be a prefix expression (it shall not have ϵ as its root or one of its leftmost operands), and
- both operands shall not be concatenation expressions.

Example 26 *If the above restrictions are adhered to, the instantiation of the operator* `f _ _` *with the postfix expression* `x+ x` *as its left adjacent operand and the prefix expression* `x+ x` *as its right adjacent operand is unambiguous in the mixfix operator signature in figure 2.21.*

```
FUN _ + _ : [nat , nat] → nat
FUN f _ _ : [nat , nat] → nat
FUN x     : nat
FUN E     : nat
DEF E ==  f x+ x x+ x
              -- unambiguous: f(x + x)(x + x)
```

Figure 2.21: Signature of example 26

While this might seem confusing at first glance because normally no such constructions exist in established programming languages, it significantly enlarges the expression language where otherwise a lot of parantheses would have to be used instead.

Of course, the notion of adjacent operands (with the exception of the concatenation operator) could be abolished altogether to avoid such confusion, but since this feature is manageable with a few sensible restrictions, we don't deem this necessary.

However, it would not be a problem to leave this feature out of the expression language altogether. This decision will be up to the language designers who try to incorporate the ideas of this thesis into a programming language as fits their needs.

Empty Operands

We shall see that in case of *visible* operators with adjacent operands, we can also allow these operands to be instantiated with *empty operators.* But to this end, we must add the restriction that either both or none of the adjacent operands must be instantiated with an empty expression so that it remains unambiguous.

Instantiation of *invisible* operators with empty expressions must thus be disallowed in general, as such instantiations would then yield the empty expression which would lead to confusion with the empty operator.

Concatenation Operands

Finally, we have to relax our restrictions somewhat in regard to concatenation operators, so that we can allow them to be left-precedent towards each other. Also, it is only possible to have concatenation operators of *one* arity in one context, as mixing concatenation operators of different arities can always introduce ambiguities.

2.5.7 Polymorphism and Semantic Ambiguity

There are two prominent kinds of polymorphism present in different functional languages and as such, they might be interesting to incorporate into our framework, as well.

One is so called *parametric* polymorphism, sometimes called *genericity* or simply *polymorphism*, the other is *ad-hoc* polymorphism, also called *overloading*.

Genericity allows operators with a single definition to have types with type variables, possibly allowing infinitely many instantiations of different type for these operators, which nevertheless all have the same implementation.

Overloading, in turn, allows the same operator to have several definitions with different implementations. That is why overloading can cause semantic ambiguities in such a way that an expression can have two syntactically equal interpretations of the same type which still have different semantics.

Example 27 *There are no purely syntactic means (except type annotation) to disambiguate the expression* | v | *in the signature in figure 2.22.*

```
FUN | _ |  : [real] → real
FUN | _ |  : [vector] → real
FUN  v     : real
FUN  v     : vector

FUN  E     : real
DEF  E == | v |
              -- | (v : real) | : real
              -- | (v : vector) | : real
```

Figure 2.22: Signature of example 27

The same is true for converter operators as they can have effects very similar to those of overloading and can be seen as a shorthand way of defining a lot of overloaded operators.

Example 28 *Again, there are no purely syntactic means to disambiguate the expression* | v | *in the signature in figure 2.23. The converter operator has the same effect as if there were two operators* v : seq nat *and* v : seq seq nat*.*

```
FUN |_| : [seq A] → nat
FUN _   : [seq nat] → seq seq nat
FUN v   : seq nat

FUN E   : nat
DEF E ==| v |
            -- | (v : seq nat) | : nat
            -- | ((v : seq nat) : seq seq nat) | : nat
```

Figure 2.23: Signature of example 28

As we have already stated, we deal with the problem of converters by restricting their use in such a way that *exactly one* converter is applied to every non-converted expression, yielding a *converted expression.* To this end, we must introduce the default *identity converter* which is applied when no other converter operator is applicable. But this would still not help us dissolve the ambiguity in the above example.

Fortunately, overload ambiguity cannot happen when only generic and no converter or overloaded operators are used. Though differently typed interpretations for the same parse tree are possible in the presence of genericity, these different interpretions would still be mapped to the same implementation, i.e. they have in essence only one semantics because every generic operator has only one implementation, regardless of its different instantiations[20].

Unfortunately, while genericity does not introduce *overload ambiguity*, it often introduces *precedence ambiguity* and thus should be handled very carefully, as well.

In general, it can be said that all three concepts, overloading, genericity and converters can lead to a very fast explosion of ambiguity and thus should be used only with great care.

Pragmatically, the following guidelines should be followed by the users to avoid such explosion.

- import only those operators that are actually used
- use mostly only fully instantiated import of generic operators
- use converter operators and overloading sparsely

While this forces some additional workload onto the user, the amount of work is comparable with other programming languages, and, as such acceptable.

[20]This is, of course, only true as long as the implementation cannot take the type information into account.

Chapter 3

Description Tools

In this chapter, we will formally introduce the concepts of mixfix operator signatures, fixities, precedence relations and two-level grammars.

We will use OPAL-like functional programs to help define some of the concepts.

3.1 Signatures

Definition 10 *A* signature Σ *is a pair* $(\mathtt{S}\ ,\ \mathtt{OP})$ *where*

- $\mathtt{S}$ *is a set of* sort symbols *and*
- $\mathtt{OP}$ *is a set of* operators $\mathtt{op}:\ \mathtt{t}_1 \ldots \mathtt{t}_\mathtt{n} \to \mathtt{t}_0$ *where*
  - $\mathtt{op}$ *is the* operator name*,*
  - $\mathtt{t}_1 \ldots \mathtt{t}_\mathtt{n} \in \mathtt{S}^\star$ *are the* operand types*, and*
  - $\mathtt{t}_0 \in \mathtt{S}$ *is the* result type *of the operator.*

We will adhere to this classical approach, but we will also enlarge it somewhat towards usability for mixfix expressions.

Definition 11 *A* mixfix operator signature *is a signature with the following additional characteristics:*

- *There is a given set of* separator token symbols *and a special* placeholder token symbol _. *These symbols together make up the set* $\mathtt{token}$*. Each* operator name *is a sequence of* $\mathtt{token}$ *symbols called* pattern *of the form* $\mathtt{s}_0\ \_\ \mathtt{s}_1\ \ldots\ \_\ \mathtt{s}_\mathtt{n}$ *where each* separator $\mathtt{s}_\mathtt{i} \in \mathtt{token}^\star$ *is a sub-sequence with* $\_ \notin \mathtt{s}_\mathtt{i}$.
- *the* sort symbols *are represented by* expressions of a type expression language $\mathcal{L}$ *(which in turn may be an expression language induced by another mixfix operator signature). Therefore, we use the list denotation* $[\mathtt{t}_1\ ,\ \ldots\ ,\ \mathtt{t}_\mathtt{n}]$ *to denote the operand types instead of* $\mathtt{t}_1 \ldots \mathtt{t}_\mathtt{n}$ *because we need a way to find the right segmentation of these expressions into the respective operand types.*
- *the set of operators* $\mathtt{OP} = (\,\mathtt{FUN} \cup \mathtt{VAR}\,)$ *is divided into two disjoint subsets. The set* $\mathtt{VAR}$ *contains the* variable operators*, while the set* $\mathtt{FUN}$ *contains the* function operators*.*

  For $\mathtt{op}:\ \mathtt{t}_1 \ldots \mathtt{t}_\mathtt{n} \to \mathtt{t}_0 \in \mathtt{FUN}$ *we write:*

$$\textsc{fun}\ \mathtt{op} : [\mathtt{t}_1\ ,\ \ldots\ ,\ \mathtt{t}_\mathtt{n}] \to \mathtt{t}_0$$

For $\texttt{op}: \ \texttt{t}_1 \ldots \texttt{t}_\texttt{n} \rightarrow \texttt{t}_0 \in \texttt{VAR}$ *we write:*

$$\text{VAR}\ \texttt{op} : [\texttt{t}_1\ ,\ \ldots\ ,\ \texttt{t}_\texttt{n}] \rightarrow \texttt{t}_0$$

For convenience sake we allow to write `t` *instead of* `[]` $\rightarrow$ `t`*.*

Example 29 *A classical signature which only allows the declaration of prefix functions and constants would look like the one in figure 3.1, declaring operators for natural numbers, list expressions and boolean expressions.*

```
FUN nat    : SORT
FUN list   : SORT
FUN data   : SORT
FUN bool   : SORT

FUN zero   : []                  → nat
FUN succ   : [nat]               → nat
FUN plus   : [nat , nat]         → nat

FUN empty  : []                  → list
FUN cons   : [data , list]       → list
FUN first  : [list]              → data
FUN rest   : [list]              → list
FUN concat: [list , list]        → list

FUN true   : []                  → bool
FUN false  : []                  → bool
FUN not    : [bool]              → bool
FUN and    : [bool , bool]       → bool
FUN or     : [bool , bool]       → bool
FUN equals: [data , data]        → bool
FUN cond   : [bool , data , data] → data
```

Figure 3.1: Signature of example 29

Now, using mixfix signatures, we can give more readable patterns for these operators, as shown in the following examples.

Example 30 *Suppose, that we have a given type expression language* $\mathcal{L}_0$ *with the expression* $\texttt{SORT} \in \mathcal{L}_0$*, then we can define the mixfix operator signature* $\Sigma_1 = (\mathcal{L}_0\ ,\ \texttt{OP}_1\ )$ *where* $\texttt{OP}_1$ *is characterized by the declarations in the signature in figure 3.2. The operators* `nat`*,* `bool` *and* `list_` *are type constructor functions, while the operator* `data` *is a type variable. This allows the denotation of generic types like* `list data`*.*

```
FUN nat    : SORT
FUN list _ : [SORT] → SORT
FUN bool   : SORT
VAR data   : SORT
```

Figure 3.2: Signature of example 30

Example 31 *Suppose, the signature* Σ_1 *induces the language* $\mathcal{L}_1$ *with* {`nat` , `bool` , `data` , `list data`} $\subseteq \mathcal{L}_1$*, then we can define the mixfix operator signature* $\Sigma_2 =$ ($\mathcal{L}_1$, $\mathtt{OP}_2$) *where* $\mathtt{OP}_2$ *is characterized by the declarations of the signature in figure 3.3. We declare the same operators as in the classical examples, but this time as mixfix operators.*

```
FUN 0           : nat                                      -- zero
FUN succ _      : [nat]                  → nat
FUN _ + _       : [nat , nat]            → nat             -- plus

FUN <>          : list data                                -- empty
FUN _ :: _      : [data , list data]     → list data       -- cons
FUN first _     : [list data]            → data
FUN rest _      : [list data]            → list data
FUN _ ++ _      : [list data , list data]→ list data       -- concat

FUN true        : bool
FUN false       : bool
FUN ¬ _         : [bool]                 → bool            -- not
FUN _ ∧ _       : [bool , bool]          → bool            -- and
FUN _ ∨ _       : [bool , bool]          → bool            -- or
FUN _ = _       : [data , data]          → bool            -- equals
FUN _ ⇒ _ | _   : [bool , data , data]   → data            -- cond
```

Figure 3.3: Signature of example 31

We will see in chapter 4 how an expression language is induced by a given mixfix signature.

By combining different signatures where some operators describe the type language of others, we get a so called *multi-level* signature. Unfortunately, the foundation of multi-level signatures is outside the scope of this thesis. They have been successfully used with mixfix operators by Visser [42] in the context of SDF+ASF ([41], [17], [45]).

Definition 12 *The* arity `arity(op)` *of an operator* `op` = (`p` : [$\mathtt{t}_1$, . . . , $\mathtt{t_n}$] → $\mathtt{t}_0$) *is equal to the number of operand placeholders in the operator pattern* `p`.

A mixfix signature (`S` , `OP`) *is* well-formed *if the arity of every operator* `op` ∈ `OP` *is equal to the number of operands in its type, i.e the following property holds:*

Property 1

$$\forall i \in 0..n : \exists s_i \in (\mathtt{token} \setminus \{\_\})^\star : \mathtt{op} = (s_0 \ \_ \ s_1 \ldots \_ \ s_n : [t_1, \ldots, t_n] \rightarrow t_0)$$

3.2 Separators, Identifiers and Backbones

Definition 13 *The* separator tokens `separatorTokens(s)` *of a token sequence* `s` *are all tokens that are not the placeholder symbol .*

```
FUN  separatorTokens : seq token →  seq token
DEF  separatorTokens ==filter(λ x .  x ≠ "_")
```

Definition 14 *A* unique identifier *(or short* identifier*) of an operator signature* Σ *is a token that occurs only in one operator pattern in* Σ*. It thus* identifies *the operator pattern it belongs to in regard to that signature.*

Definition 15 *The* separators *of an operator pattern are the sequences of separator tokens before and behind placeholder tokens.*

Thus, every `n`*-ary operator pattern can be characterized as a sequence* $\mathtt{s}_0$ _ $\mathtt{s}_1$ . . . _ $\mathtt{s_n}$ *with* _ $\notin$ $\mathtt{s_i}$ *where all* $\mathtt{s_i}$ *are the separators.*

```
FUN  separators          : seq token →  seq seq token
DEF  separators(<>)      ==<> :: <>
DEF  separators("_" :: s) ==<> :: separators(s)
DEF  separators(t :: s)   ==(t :: S) :: R
     WHERE (S :: R)       ==separators(s)
```

Definition 16 *The* operator backbone pattern *of a visible operator pattern is the the sub-sequence of tokens starting with the beginning of the first non-empty separator continuing to the end of the last non-empty separator of the operator pattern. The separators of this sub-sequence are called the* inner separators*.*

Example 32 *The operator backbone pattern of the an operator pattern* _ ⇒ _ | _ *would be* ⇒ _ |*.*

The operator backbone pattern of `if _ then _ else _` *is* `if _ then _ else`*.*

The operator backbone pattern of `_ _ a _ _ b _ c _` *is* `a _ _ b _ c`*.*

Definition 17 *An* operator backbone instantiation *(or short* operator backbone *) is an operator backbone pattern where all placeholder symbols are replaced by expressions.*

We can define a function `backbones` *which maps the parse tree of every expression to its corresponding sequence of operator backbones.*

1. *Suppose we have the operator instantiation* $\mathtt{op(T_1\ ,\ \dots\ ,\ T_n)}$ *where* $\mathtt{op} = \mathtt{s}_0$ _ . . . _ $\mathtt{s_n}$
2. *let* $\mathtt{S_i}$ *be the sequence of operator backbones for each operand* $\mathtt{T_i}$ *and let* $\mathtt{F_i}$ *be the flattened sequence of* $\mathtt{S_i}$*.*
3. *if* $\mathtt{s_{lm}}$ *is the leftmost non-empty separator, i.e.* $\mathtt{s}_0$, . . . , $\mathtt{s_{lm-1}}$ *are all empty separators, then let* `L` *be the concatenation of the sequences* $\mathtt{S}_1$, . . . , $\mathtt{S_{lm}}$
4. *if* $\mathtt{s_{rm}}$ *is the rightmost non-empty separator, i.e.* $\mathtt{s_{rm+1}}$, . . . , $\mathtt{s_n}$ *are all empty separators, then let* `R` *be the concatenation of the sequences* $\mathtt{S_{rm+1}}$, . . . , $\mathtt{S_n}$
5. *let* `M` *be the sequence* $\mathtt{s_{lm}}\ \mathtt{F_{lm+1}}\ \mathtt{s_{lm+1}}\ \dots\ \mathtt{F_{rm}}\ \mathtt{s_{rm}}$*, i.e. where every inner placeholder before separator* $\mathtt{s_i}$ *is replaced by* $\mathtt{F_i}$*,*
6. *then, the resulting sequence of operator backbones for the overall exression is* `L ++[M] ++ R`*.*

3.2.1 Backbone Grammar

Since our mixfix expressions will be operator patterns where all placeholders are replaced by expressions, we can interpret every mixfix expression also as a *sequence of backbone instantiations.* We can describe these sequences with the following construction.

Definition 18 *A* backbone grammar *of an operator set is a context-free grammar* $(\mathtt{V_T}\ ,\ \{\mathtt{E}\}\ ,\ \mathtt{P}\ ,\ \mathtt{E})$ *which describes all concatenations of backbone instantiations that can be constructed with the help of the operator set. The function* `backboneRules` *computes the set of production rules* `P` *of this grammar. The set of production rules contains one rule for every operator pattern in the operator set. We arrive at such a rule in the following way:*

1. *take the sequence of visible separators of the pattern*
2. *insert the nonterminal symbol* `E` *before every visible separator*
3. *take the resulting sequence of symbols as the right-hand-side of the production rule, where the left-hand-side is the nonterminal* `E`*.*

The set of production rules also contains at least *the rule* $\mathtt{E} ::= \epsilon$*.*

Example 33 *The operator pattern* `if _ then _ else _` *is mapped to the production rule* $\mathtt{E} ::= \mathtt{E}\ \underline{\mathtt{if}}\ \mathtt{E}\ \underline{\mathtt{then}}\ \mathtt{E}\ \underline{\mathtt{else}}$*.*

The operator pattern `_ _ a _ _ b _ c _` *is mapped to the production rule* $\mathtt{E} ::= \mathtt{E}\ \underline{\mathtt{a}}\ \mathtt{E}\ \underline{\mathtt{b}}\ \mathtt{E}\ \underline{\mathtt{c}}$*.*

We have chosen to make each backbone rule left-recursive, i.e. starting again with the nonterminal `E`, because that way, the resulting grammar can be more easily processed by generalized bottom-up parsers, which are less efficient for right-recursive grammar rules. However, if one were to choose to derive a top-down parser for a backbone grammar, it would be wiser to put the `E` after each visible separator instead of before it.

The concept of the backbone grammar is useful to determine the occurrence of backbone ambiguity. All expressions which only have one derivation in the backbone grammar do not have a backbone ambiguity.

3.3 Fixities

Fixities are a tool to characterize mixfix operators (and in extension mixfix expressions) syntactically.

Definition 19 *The three characterizing features that we will need to restrict our mixfix expression language are, whether an operator is* empty*,* prefix *or* postfix.[1]

- *An operator with pattern* ϵ *(where* ϵ *is the empty sequence) is called* empty*.*
- *An operator with pattern* $\mathtt{s_0}\ \_\ \mathtt{s_1} \ldots \_\ \mathtt{s_n}$ *is called* prefix *if* $\mathtt{s_0} \neq \epsilon$*, i.e the operator pattern is not empty and does not start with a placeholder symbol.*

[1]Our characterizations of prefix and postfix operators is taken from [1]. It might seem unusual at first glance. Classically, an operator is called prefix if written in front of its operand and postfix if written behind its operand. Instead, we are only interested in whether or not operators start or end in non-empty separators to see them as prefix or postfix. Of course, these characterizations will coincide in most cases with the classical ones, but also include others, since we are much more free in the declaration of operator patterns.

- *An operator with pattern* $\mathtt{s}_0$ _ $\mathtt{s}_1$ … _ $\mathtt{s}_n$ *is called* postfix *if* $\mathtt{s}_n \neq \epsilon$*, i.e the operator pattern is not empty and does not end with a placeholder symbol.*

Definition 20 *Additional operator pattern characterizations are:*

- *We call an operator* closed *if it is prefix and postfix.*
- *We call a non-empty operator* infix *if it is neither prefix nor postfix.*
- *We call an operator* invisible *if it has a pattern* $\mathtt{s}_0$ _ $\mathtt{s}_1$ … _ $\mathtt{s}_n$ *where all* $\mathtt{s}_i = \epsilon$.
- *We call an operator that is not invisible* visible .
- *We call an invisible operator of arity 1* converter .
- *We call an invisible operator of arity greater than 1* concatenation.
- *We call a non-empty non-postfix operator* right-open .
- *We call a non-empty non-prefix operator* left-open .

3.4 Syntactic Interpretations and Parse Trees

Definition 21 *A* syntactic interpretation *of an expression* $\mathtt{s}_0\ \mathtt{E}_1\ \mathtt{s}_1\ \ldots\ \mathtt{E}_n\ \mathtt{s}_n$ *is a parse tree* $(\mathtt{s}_0$ _ $\mathtt{s}_1$ … _ $\mathtt{s}_n)(\mathtt{T}_1, \ldots, \mathtt{T}_n)$ *where* $\mathtt{T}_i$ *is a syntactic interpretation of* $\mathtt{E}_i$ *and* $\mathtt{s}_0$ _ $\mathtt{s}_1$ … _ $\mathtt{s}_n$ *is an operator pattern.*

Sometimes, we write $\mathtt{s}_0\ (\mathtt{T}_1)\ \mathtt{s}_1\ \ldots (\mathtt{T}_n)\ \mathtt{s}_n$ *as the syntactic interpretation of* $\mathtt{s}_0\ \mathtt{E}_1\ \mathtt{s}_1\ \ldots\ \mathtt{E}_n\ \mathtt{s}_n$.

Definition 22 *A* type-correct *syntactic interpretation of an expression is a syntactic interpretation* $\mathtt{op}(\mathtt{T}_1, \ldots, \mathtt{T}_n)$*, with* $\mathtt{op} : [\mathtt{t}_1, \ldots, \mathtt{t}_n] \to \mathtt{t}_0$*, where each* $\mathtt{T}_i$ *is type-correct and the inferred type of operand parse tree* $\mathtt{T}_i$ *is compatible with* $\mathtt{t}_i$.[2]

Definition 23 *A* syntactic interpretation of an expression $\mathtt{E}$ in type τ *is a type-correct syntactic interpretation of expression* $\mathtt{E}$ *for which type* τ *can be inferred.*

3.5 Unification

Our mixfix expression language can include generic operators, i.e. operators whose types are quantified over type variables.

The types in our language are also expressions of the language itself, i.e. mixfix expressions, where mixfix variable expressions can occur. We assume for every definition which refers to a set of declared variables, that it is ∀-quantified over these variables. Normalization which removes the quantor then yields types which contain only free variables.

As is well known (i.e. from polymorphic functional languages like ml [26]), a type-universe with polymorphic or generic types necessitates *unification* of type expressions during the type inference process to determine whether or not two generic types can stand in for the same type.

To that end, a unification algorithm must compute a *substitution function* which can be applied to the types. Such a substitution function maps the free type variables to type expressions. To be a unification substitution for two types, it must, if applied consistently to all occurrences to the substituted variables in both types, yield the same result.

[2]The inferred type of $\mathtt{T}_i$ and compatibility will be discussed in section 4.4.

Definition 24 *A* unification substitution *for the types* $\mathtt{T}_1$ *and* $\mathtt{T}_2$ *is written* $[\mathtt{T}_1 := \mathtt{T}_2]$[3]

If the substitution $[\mathtt{T}_1 := \mathtt{T}_2]$ *is applied to the type* $\mathtt{T}$*, we write* $\mathtt{T}[\mathtt{T}_1 := \mathtt{T}_2]$*.*

If there exists a substitution which applied to the types $\mathtt{T}_1$ *and* $\mathtt{T}_2$ *yields the same type* $\mathtt{T}'$*,* $\mathtt{T}_1$ *and* $\mathtt{T}_2$ *are called* unifiable.

If two types $\mathtt{T}_1$ *and* $\mathtt{T}_2$ *are not unifiable, i.e. there exists no substitution function to make them equal, the substitution* $[\mathtt{T}_1 := \mathtt{T}_2]$ *is defined as* $\lambda\mathtt{T}\ .\ \bot$*, i.e. the function that returns* undefined *for every type it is applied to.*

A simultaneous substitution *computed by simultaneously unifying all the pairs* $(\mathtt{T_i}\ ,\ \mathtt{T'_i})$ *for all* $i \in 1 \ldots n$ *is written* $[\mathtt{T_i} := \mathtt{T'_i}]_{\mathtt{i=1}}^{\mathtt{n}}$ *or* $[\mathtt{T}_1 := \mathtt{T}'_1\ ,\ \ldots\ ,\ \mathtt{T_n} := \mathtt{T'_n}]$*. It is the same as the normal substitution* $[(\mathtt{T}_1\ ,\ \ldots\ ,\ \mathtt{T_n}) := (\mathtt{T}'_1\ ,\ \ldots\ ,\ \mathtt{T'_n})]$*, where we must assume that the comma operator exists.*

The first requirement for unification to work is that our types must be structures, which means terms where an operator is applied to an operand list. Thus, we view the types as their unambiguous interpretations, i.e. parse trees, which always consist of a *constructor*, which is the root operator, applied to a *sequence of operands* which again are parse trees. The leafs of the trees thus have to be nullary operators.

However, we shall write down the respective types by using mixfix expressions where possible to facilitate reading.

Instead of `[seq _[seq _[A[]]] := seq _[B[]]]`, we would write `[seq seq A := seq B]`.

Definition 25 *A substitution is* type-consistent *if all variables are substituted by expressions whose type is unifiable with the type of the variable*[4]*.*

In our multi-level type universe, we are only interested in type-consistent substitutions when inferring types.

Algorithms for computing term unification substitutions can be found in [3] or [27].

3.5.1 Unification and Higher-Order-Functions

Even though we are dealing in our functional language with higher-order functions which may appear both as value and as type expressions, both on the value as well as the type level, we are not faced with the undecidability problems of higher-order unification, as treated in [8].

The reason for this is, that we only unify first-order type terms (with the restriction that the type annotations of the terms must also be unifiable without need for type-consistency), disregarding semantic equalities of these terms, an undertaking for which E-Unification would be necessary. Enlarging our unification algorithm to include also type equations, as in [43] would be a topic for further research.

3.6 Precedence Relations

Definition 26 Precedence relations *describe allowed syntactic combinations of operators, i.e whether an instantiation of an operator* $\mathtt{op}_1$ *can be used* syntactically *as an operand to another operator* $\mathtt{op}_2$*. If so,* $\mathtt{op}_1$ *is said to be* precedent *towards* $\mathtt{op}_2$ *and* $\mathtt{op}_2$ *is called the* dominant *operator over* $\mathtt{op}_1$*.*

[3]We have chosen the assignment symbol := rather than an equality symbol to show that in case of a variable-variable unification, the variable from the left term is substituted with the variable of the right term. In our experience, this yields much fewer fresh variables in the resulting substituted terms during our type-inference algorithm.

[4]For this unifiability of types, we do not demand type-consistency, as this could lead to an infinite process.

More specifically, we are interested in the following: which operators can be allowed syntactically as *rightmost* operands to *right-open* operators and which ones can be allowed as *leftmost* operands to *left-open* operators.

We call relations which describe such operator combinations *right-precedence* relations or *left-precedence* relations, respectively.

Definition 27 *A* right-precedence **Right** *is a subset of* $\mathbf{RightOpen} \times \mathbf{Op}$ *where* **Op** *is a set of operator patterns and* $\mathbf{RightOpen} = \mathbf{Op} \cap \{\mathrm{s}_0\ \_\ \mathrm{s}_1 \ldots \_\ \mathrm{s_n} | \mathbf{n} > 0\ ,\ \mathrm{s_n} = \epsilon\}$ *is the subset of right-open operator patterns.*

If $\mathbf{op}_1$ **Right** $\mathbf{op}_2$*,* $\mathbf{op}_2$ *is called* right-precedent *to* $\mathbf{op}_1$ *in respect to* **Right***.*

If $\mathbf{op}_2 = \mathrm{s}_0\ \_\ \mathrm{s}_1 \ldots \_\ \mathrm{s_n}$ *and* $\mathbf{op}_1 = \mathrm{s}'_0\ \_\ \mathrm{s}'_1 \ldots \_\ \mathrm{s}'_{\mathrm{n}'}$*, then* $\mathbf{op}_1$ **Right** $\mathbf{op}_2$ *allows expressions of the form* $\mathrm{s}'_0\ \mathbf{E}'_1\ \mathrm{s}'_1\ \ldots \mathbf{E}'_{\mathrm{n}'-1}\ \mathrm{s}'_{\mathrm{n}'-1}\ \mathrm{s}_0\ \mathbf{E}_1\ \mathrm{s}_1 \ldots \mathbf{E_n}\ \mathrm{s_n}$ *with interpretation* $\mathbf{op}_1(\mathbf{T}'_1\ , \ldots , \mathbf{T}'_{\mathrm{n}'-1}\ , \mathbf{op}_2(\mathbf{T}_1\ , \ldots , \mathbf{T_n}))$ *where all* $\mathbf{T_i}$ *are allowed interpretations of* $\mathbf{E_i}$ *and* $\mathbf{T}'_\mathbf{i}$ *are allowed interpretations of* $\mathbf{E}'_\mathbf{i}$*.*

A left-precedence **Left** *is a subset of* $\mathbf{Op} \times \mathbf{LeftOpen}$ *where* **Op** *is a set of operator patterns and* $\mathbf{LeftOpen} = \mathbf{Op} \cap \{\mathrm{s}_0\ \_\ \mathrm{s}_1 \ldots \_\ \mathrm{s_n} | \mathbf{n} > 0\ ,\ \mathrm{s}_0 = \epsilon\}$ *is the subset of left-open operator patterns.*

If $\mathbf{op}_1$ **Left** $\mathbf{op}_2$*,* $\mathbf{op}_1$ *is called* left-precedent *to* $\mathbf{op}_2$ *in respect to* **Left***.*

If $\mathbf{op}_2 = \mathrm{s}_0\ \_\ \mathrm{s}_1 \ldots \_\ \mathrm{s_n}$ *and* $\mathbf{op}_1 = \mathrm{s}'_0\ \_\ \mathrm{s}'_1 \ldots \_\ \mathrm{s}'_{\mathrm{n}'}$*, then* $\mathbf{op}_1$ **Left** $\mathbf{op}_2$ *allows expressions of the form* $\mathrm{s}'_0\ \mathbf{E}'_1\ \mathrm{s}'_1\ \ldots \mathbf{E}'_{\mathrm{n}'}\ \mathrm{s}'_{\mathrm{n}'}\ \mathrm{s}_1 \ldots \mathbf{E_n}\ \mathrm{s_n}$ *with interpretation* $\mathbf{op}_2(\mathbf{op}_1(\mathbf{T}'_1\ , \ldots , \mathbf{T}'_\mathbf{n})\ , \mathbf{T}_2\ , \ldots , \mathbf{T_n})$ *where all* $\mathbf{T_i}$ *are allowed interpretations of* $\mathbf{E_i}$ *and* $\mathbf{T}'_\mathbf{i}$ *are allowed interpretations of* $\mathbf{E}'_\mathbf{i}$*.*

We distinguish between *natural* and *ad-hoc* precedence relations.

Natural precedence relations are restricted by the types of the operators involved. They are used to describe semantically sensible syntactic interpretations of expressions.

Example 34 *A typical example for a natural precedence relation is the precedence of the length operator* $\#\ \_ : [\mathbf{seq\ nat}] \rightarrow \mathbf{nat}$ *towards the faculty operator* $\_\ ! : [\mathbf{nat}] \rightarrow \mathbf{nat}$*. Since* $\#\ \_$ *can appear as left operand to* $\_\ !$*, but* $\_\ !$ *cannot be the right operand of* $\#\ \_$*, we have the natural precedence relations* $(\mathbf{Left}, \mathbf{Right})$*, constrained by the following properties:*

$$\{(\#\ \_, \_\ !), (\_\ !, \_\ !)\} \subseteq \mathbf{Left}$$

$$\{(\#\ \_, \#\ \_), (\#\ \_, \_\ !)\} \cap \mathbf{Right} = \emptyset$$

Ad-hoc precedence relations are only restricted by the definition of precedence relations. They are used to describe the user-preferred syntactic interpretations of expressions, if several are naturally possible.

Example 35 *A typical example for an ad-hoc precedence relation is the precedence of the arithmetic operators* $\_ + \_ : [\mathbf{nat}\ , \mathbf{nat}] \rightarrow \mathbf{nat}$ *and* $\_ * \_ : [\mathbf{nat}\ , \mathbf{nat}] \rightarrow \mathbf{nat}$*. To achieve that* $\_ * \_$ *has a general precedence towards* $\_ + \_$*, we could constrain the ad-hoc precedence relations* $(\mathbf{Left}_A, \mathbf{Right}_A)$ *by the following properties:*

$$\{(\_ + \_, \_ + \_), (\_ * \_, \_ + \_), (\_ * \_, \_ * \_)\} \subseteq \mathbf{Left}_A$$

$$\{(\_ + \_, \_ * \_)\} \subseteq \mathbf{Right}_A$$

A combination of both kinds of precedence-relations can be used for disambiguation of expressions which might have both left- and right-precedent syntactic interpretations.

Definition 28 *A pair of precedence relations* $(\mathbf{Left}, \mathbf{Right})$ *is said to have a* conflict *for an expression e in type τ, if it allows two type-correct syntactic interpretations, in type τ* $\mathbf{i}_1 = \mathbf{op}_1(\mathbf{T}_1 , \dots , \mathbf{T_n})$ *and* $\mathbf{i}_2 = \mathbf{op}_2(\mathbf{T}'_1 , \dots , \mathbf{T}'_\mathbf{m})$ *where*

- $\mathbf{op}_1$ *is right-open and*
- $\mathbf{op}_2$ *is left-open and*
- $\mathbf{op}_2$ *is the root operator of a rightmost descendant of* $\mathbf{i}_1$ *(while all its ancestors are right-open) and*
- $\mathbf{op}_1$ *is the root operator of a leftmost descendant of* $\mathbf{i}_2$ *(while all its ancestors are left-open)*

, i.e. such that $(\mathbf{op}_1 , \mathbf{op}_2) \in \mathbf{Left}^\star \cap \mathbf{Right}^\star$*.*

Definition 29 *A pair of precedence relations* $(\mathbf{Left}, \mathbf{Right})$ *is* conflict-free*, if for all expressions e there is at most one type-correct syntactic interpretation of e in every type τ.*

Example 36 *The pair of the largest possible natural precedence relations* $(\mathbf{Left}, \mathbf{Right})$ *for the arithmetic operators* $\mathbf{op}_1 = \_ + \_ : [\mathbf{nat} , \mathbf{nat}] \to \mathbf{nat}$ *and* $\mathbf{op}_2 = \_ * \_ : [\mathbf{nat} , \mathbf{nat}] \to \mathbf{nat}$ *would be constrained by the following properties:*

$$\{\_ + \_, \_ * \_\} \times \{\_ + \_, \_ * \_\} \subseteq \mathbf{Left}$$

$$\{\_ * \_, \_ + \_\} \times \{\_ * \_, \_ + \_\} \subseteq \mathbf{Right}$$

All left-precedent and all right-precedent syntactic interpretations of expressions using combinations of op_1 and op_2 are type-correct.

Thus, these natural precedence relations have a conflict for every such expression.

Conflict-freeness of precedence relations ensures that only one syntactic interpretation can be derived for every expression[5]. If, on the other hand, the used precedence relations are not conflict-free, we cannot ensure that there is only one type-correct syntactic interpretation for every expression which is our general aim. Thus, we are interested in conditions that make precedence relations conflict-free, but are as least restrictive as possible (i.e. cause the fewest rejections of expressions as precedence-ambiguous).

Proposition 1 *A pair of precedence relations* $(\mathbf{Left}, \mathbf{Right})$ *is not conflict-free if* $\mathbf{Left} \cap \mathbf{Right} \neq \emptyset$*.*

Proof 2 *If* $\mathbf{Left} \cap \mathbf{Right} \neq \emptyset$*, then there exists a pair of operators* $(\mathbf{op}_1 = \mathbf{p}_1 \; \_ \; , \mathbf{op}_2 = \_ \; \mathbf{p}_2)$ *such that* $\mathbf{op}_1$ $\mathbf{Left}$ $\mathbf{op}_2$ *and* $\mathbf{op}_1$ $\mathbf{Right}$ $\mathbf{op}_2$*.*

This would mean that, according to $(\mathbf{Left}, \mathbf{Right})$*, both* $(\mathbf{p}_1\ (\mathbf{E}))\ \mathbf{p}_2$ *and* $\mathbf{p}_1\ ((\mathbf{E})\ \mathbf{p}_2)$ *would be allowed syntactic interpretations for expression* $\mathbf{p}_1\ \mathbf{E}\ \mathbf{p}_2$ *.*

We call a situation as in proposition 1 a *direct conflict* of the precedence relations. However, while this condition is easily checked and avoided, not all conflicts that can occur need to be direct. We will go into further detail on that topic in section 5.6.2.

Proposition 2 *A pair of precedence relations* $(\mathbf{Left}, \mathbf{Right})$ *is conflict-free , if both* $\mathbf{Left}$ *and* $\mathbf{Right}$ *are transitive, i.e.* $\mathbf{Left} = \mathbf{Left}^\star$ *and* $\mathbf{Right} = \mathbf{Right}^\star$*, and do not have a direct conflict, i.e.* $\mathbf{Left} \cap \mathbf{Right} = \emptyset$*.*

[5]as long as no other cause for ambiguity is present

Proof 3 *This is a special case of fully hierarchical precedence relations as described in section 5.6.2. Since these do not have conflicting operator pairs, they cannot have a conflict.*

This condition which guarantees conflict-freeness is used in the classical *precedence-level* approach and can, of course, still be used in our approach. But it is unsatisfactory in general because it is too restrictive, not allowing enough unambiguous expressions.

Therefore, we will introduce a less restrictive condition for conflict-freeness that takes demanded types and the actual occurrence of operators in the expression to be disambiguated into account. The precedence relations we use will be conflict-free by construction, but will — unfortunately — still not allow *all* expressions which are unambiguous in regard to precedence.

Definition 30 *If we have two operator patterns* $\mathbf{op}_1 = \mathbf{p}_1$ *and* $\mathbf{op}_2 = \_\ \mathbf{p}_2$ *we can write* PREC $(\mathbf{p}_1)\ \mathbf{p}_2$ *to declare* $\mathbf{op}_1\ \mathbf{Left_A}\ \mathbf{op}_2$*, where* $\mathbf{Left_A}$ *is the ad-hoc left-precedence relation.*

If we have two operator patterns $\mathbf{op}_1 = \mathbf{p}_1\ \_$ *and* $\mathbf{op}_2 = \mathbf{p}_2$ *we can write* PREC $\mathbf{p}_1\ (\mathbf{p}_2)$ *to declare* $\mathbf{op}_1\ \mathbf{Right_A}\ \mathbf{op}_2$*, where* $\mathbf{Right_A}$ *is the ad-hoc right-precedence relation.*

If two operator patterns $\mathbf{op}_1$ *and* $\mathbf{op}_2$ *shall have the same ad-hoc precedence towards all operator patterns, we can write* EQPREC $(\mathbf{op}_1)(\mathbf{op}_2)$.

Example 37 *For the use of precedence declarations, refer to example 5 on page 17.*

3.7 Two-Level Grammars

Definition 31 *A* two-level grammar *is a tuple* $(V_N, V_T, V_A, V_V, P, \mathcal{L}(\Gamma))$.

- V_N *is the set of* nonterminal symbols.
- V_T *is the set of* terminal symbols.
- V_A *is the set of* action symbols *(i.e. null-nonterminals that describe semantic actions on the parser state).*
- V_V *is the set of* variable symbols.
- P *is a set of* production rules.
- $\mathcal{L}(\Gamma)$ *is the* annotation language *which is generic over the set of variables* V_V *(and* Γ *is a subset of* V_V*).*

V_N*,* V_T*,* V_A *and* V_V *must be pairwise disjoint and the set of all* grammar symbols V *is the union of* V_N*,* V_T *and* V_A.

Definition 32 *An annotated symbol is a pair* $(s, a) \in (V \times \mathcal{L}(\Gamma)^\star)$*. We write* $s(a_1, \ldots, a_n)$ *for the annotated symbol* $(s, (a_1, \ldots, a_n))$.

Definition 33 *A* production rule *of a two-level grammar* $(V_N, V_T, V_A, V_V, P, \mathcal{L}(\Gamma))$ *is a tuple* (Γ, C, N, w)*, written* $\forall \Gamma \mid C.N ::= w$*, where*

- $N \in V_N \times \mathcal{L}(\Gamma)^\star$ *is an annotated nonterminal symbol,*
- $w \in (V \times \mathcal{L}(\Gamma)^\star)^\star$ *is a sequence of annotated grammar symbols,*
- $\Gamma \subseteq V_V$ *and*

- *C is a boolean predicate over the variables in Γ, also called a* constraint.

If the constraint C is true, then we can also write $\forall\Gamma.N ::= w$.

If Γ is non-empty and N or w contains variables from Γ, the production rule is called generic.

If Γ is empty, i.e. the rule is not generic, it is a one-level grammar production and can be written $N ::= w$.

Therefore, one-level context-free grammars are a special case of two-level grammars which only contain non-generic productions.

Definition 34 *A* generic grammar *is a two-level grammar that contains generic production rules.*

Example 38 *In a* two-level grammar *the nonterminal symbols can be attributed.*

To allow disambiguation by type, our attributes must consist of at least *the inferred* type *of the parse tree generated for each nonterminal symbol.*

Hence, if a type of one nonterminal in a production is generic, then the production rule is also generic.

One such generic two-level grammar representing constructor and selector operands for generic sequence types could look like this:

$$\begin{array}{lll}
\forall\{A\}. & E(seq\ A) & ::= \underline{<>} \\
\forall\{A\}. & E(seq\ A) & ::= E(A)\ \underline{::}\ E(seq\ A) \\
\forall\{A\}. & E(A) & ::= \underline{\texttt{ft}}\ E(seq\ A) \\
\forall\{A\}. & E(seq\ A) & ::= \underline{\texttt{rt}}\ E(seq\ A)
\end{array}$$

Definition 35 *A* sentential form *is a pair (Γ, v), written $\forall\Gamma.v$ where $v \in (V \times \mathcal{L}(\Gamma)^\star)^\star$ is a sequence of annotated grammar symbols.*

A constrained sentential form *is a sentential form together with a constraint C over its variables Γ, i.e. a tuple (Γ, C, v) written $\forall\Gamma \mid C.v$.*

Definition 36 *A* derivation step *is a consistent application of a production rule to a nonterminal symbol in a sentential form, by substituting the nonterminal on the left-hand side of the production with its right-hand side and unifying the annotation parts of the nonterminal to be substituted and the nonterminal on the left-hand side of the production.*

If we have the sentence s_1 and the production rule p, then $s_1 \Rightarrow s_2$ is a derivation step if and only if there exists an injective variable renaming $\sigma \in V_V \to V_V$ (lifted to a substitution function $\overline{\sigma}$) such that:

$$\begin{array}{rcl}
s_1 & = & \forall\Gamma_1 \mid C_1.u\ N(T_1)\ w \\
p & = & \forall\Gamma_2 \mid C_2.N(T_2) ::= v \\
\sigma(\Gamma_2) \cap \Gamma_1 & = & \emptyset \\
s_2 & = & \forall\Gamma_1 \cup \sigma(\Gamma_2) \mid \overline{\sigma}(T_2) = T_1 \wedge \overline{\sigma}(C_2) \wedge C_1.u\ \overline{\sigma}(v)\ w
\end{array}$$

Definition 37 *A* sentence *of the language of the grammar is a sentential form $\forall\Gamma \mid C.w$ with $w \in (V_T \times \mathcal{L}(\Gamma)^\star)^\star$.*

Example 39 *Suppose, we have the grammar from example 38 with the additional rule:*

$$\forall\{A\}. \quad E(seq\ seq\ A) \quad ::= \underline{\texttt{s}}$$

Then, the following is a derivation step:

$$\forall\{A\}\ .\ \underline{\texttt{ft}}\ E(seqA)\ \underline{::}\ \underline{<>} \Rightarrow \forall\{A, A'\} \mid seq\ A = seq\ seq\ A'\ .\ \underline{\texttt{ft}}\ \underline{\texttt{s}}\ \underline{::}\ \underline{<>}$$

Chapter 4

A Functional Mixfix Expression Language

This chapter will define the mixfix expression language together with its type system and its integration into a functional language that allows the introduction of mixfix operators.

4.1 Mixfix Expressions

4.1.1 Phrases

All expressions are interpretations of sequences of *phrases*. A sequence of phrases is called a *sentence*.

One kind of phrase is the *token phrase* which represents a word made up of alphanumeric symbols (i.e. **foo**4), sequences of graphemic symbols (i.e. $++$) or denotations of arbitrary characters (i.e. "**foo** + +4"). A *screener* takes care of grouping symbols together as tokens.[1]

Another kind of phrase that we use is the *group phrase* which is a sequence of *declarations* enclosed by parentheses and looks like **(**FUN **op** : **T** DEF **op==E)**. Each group introduces a *scope*.

The declarations that interest us here are either *variable declarations* or *operator declarations*, both describing the pattern of the declared operator and its type. Also, *precedence declarations* between operator patterns are used for disambiguation.

Definition 38 *A* scope *contains a set of operator declarations, a set of definitions, as well as ad-hoc precedence relations over the declared operators. An operator declaration consist of an operator pattern and a declared type.*

4.1.2 Meta Operators

We need some built-in *meta-operators* to provide means to disambiguate expressions by annotating them with a type **(E : T)** or a different scope **(S . E)** than the one they appear in syntactically.

Also, some additional operators must be predefined to allow for the actual denotation of types for the newly introduced operators **(**$[\mathbf{E}_1 , \mathbf{E}_2] \rightarrow \mathbf{E}_0$**)**.

These operators at least must be recognizable by the mixfix parser in every scope so that declarations can be parsed.

[1]Again, this is outside the scope of this thesis.

However, we allow the use of the meta-operators only in specific places inside our mixfix expressions, so they do not cause more syntactic ambiguities than necessary.

4.1.3 Mixfix Parser

The *mixfix parser* in general takes a sentence and a scope and tries to find a syntactically unambiguous interpretation of the sentence as a mixfix expression in that scope. It also can identify erroneous and possibly ambiguous sentences (i.e. those that have no type-correct syntactic interpretation or more than one).

4.2 Meta Operators

Some meta operators are needed for disambiguation purposes. They allow the user to constrain the interpretation of a sentence by annotation with a type or scope.

Other predefined mixfix (meta-)operators allow the convenient denotation of type expressions, so it is possible for the users to declare their own mixfix operators.

The built-in operations are:

- group structuring — (FUN **op** : **T** DEF **op == E** ...)
  The parentheses are used to make a *record* out of a list of declarations.
- type annotation — **E** : **T**
  Some expressions are only unambiguous in a certain type. Therefore, type annotation is sometimes necessary for disambiguation.
- interpretation — **S** . **E**
  We generalize the *selection operator* so that the expression **S** . **E** is the value of **E** interpreted in the scope induced by structure **S**, overriding the declarations in the current scope.

  This makes several things possible:

  - normal selection — **Seq** . **nil**
  - let-in-like expressions[2]— (FUN **x** : **A** FUN **y** : **B** ...) . **x** + **y**
  - interpretation — (USE **Nat**) . $3 + 4 * \mathbf{n}$
- brackets — [**E**] Brackets are used in the description of *sections*[3].

  Every section has a list of *operands* which is denoted as a bracketed comma-separated list of expressions.
- comma — $\mathbf{E}_1$, $\mathbf{E}_2$ The comma separator can be used to build arbitrary comma-separated lists. It is always interpreted as right-precedent and in essence only used for building *pairs* of expressions. They are *not* associative, i.e. $(\mathbf{a} , \mathbf{b}) , \mathbf{c} \neq \mathbf{a} , (\mathbf{b} , \mathbf{c}) = \mathbf{a} , \mathbf{b} , \mathbf{c}$.
- arrow — $\mathbf{E}_1 \to \mathbf{E}_2$
  Mapping one expression $\mathbf{E}_1$ to another expression $\mathbf{E}_2$ is done with the arrow operator. It can used for denotation of:

  - function types — $\mathbf{nat} \to \mathbf{nat}$,
  - section types — $[\mathbf{nat} , \mathbf{nat}] \to \mathbf{nat}$, and

[2]Since the selection operator has weaker precedence than all user-defined mixfix operators, the whole expression to the right of the . is interpreted in the scope to its left.

[3]described in sections 4.2.3 and 4.2.4

  – lambda function expressions with pattern matching — **succ n** → **m** → **succ add n m** .

- syntactic structuring — **(E)**
  Since it is not always possible to disambiguate everything by typing or given precedences, it is possible to give syntactic structure to an expression by adding a parenthesis around it.

- tupling — **(a , b)** If there is a comma-expression inside a parenthesis, the resulting expression is interpreted as a tuple.

  We also use tuples to represent products, i.e. **(nat , bool)** is both the tuple of the two types **nat** and **bool**, as well as the product of all tuples where the components have the respective types.

- section lifting — (_ + _)
  Sometimes a section expression should be interpreted as a *higher-order function*, e.g. allowing (_ + _) to be used as a fully instantiated function value of type **(nat , nat)** → **nat** instead as a binary operator section of type **[nat , nat]** → **nat** .

  This is also made possible by the parentheses operator, additional to its purpose for syntactic structuring.

  We shall see that the different tupling operators, though closely related have slightly different semantics which makes them useful for different purposes.

These expressions are identified by a so-called *meta parser* that knows how to recognize the built-in meta-operators. We will describe them in the rest of this section, both in regard to their precedences to other expressions, as well as their typing relations.

4.2.1 Meta Type Expressions

There are some meta type constructor operators, for instance **SORT** and **LIFT** _ that are useful for denoting type expressions. They are needed because the type language itself is to be defined by the user with mixfix operators.

Definition 39 *The type* **SORT** *is the* topmost type of the type universe *and is its own type, i.e.* **SORT** : **SORT**.

It can be used to declare anything that doesn't have a more specific type.

Definition 40 *The type* **LIFT E** *is a type expression that describes only the value* **E***, i.e.* **E** : **LIFT E**.

It is useful when we want to refer to the *value* of an operand inside the type universe. Basically, the value **E** is *lifted* to the type universe. For example, this can help us in describing the type annotation operator _ : _ : **[A , LIFT A]** → **A**.

Example 40 *The generic structure* **Seq data** *is described by using a type variable for the actually given type* **data**. *Inside the structure, the type of that type-variable is not really interesting, but its value is.*

VAR **data** : **SORT** -- *type variable*
FUN **Seq** _ : **[LIFT data]** → (FUN_ :: _ : **[data , seq]** → **seq**. . .)

4.2.2 Enclosed Expressions

For the sake of syntactic structuring, we have a built-in parenthesis operator which serves (among other things) as an identity function.

FUN (_) : $[\mathbf{A}] \to \mathbf{A}$

This way, we can enclose expressions, thereby enforcing *natural precedence*.

Example 41 *In the signature in figure 4.1, the operator _ − _ is declared as left precedent. To enforce a right-precedent interpretation of the expression* $10 - 5 - 3$*, we have to use the parenthesis operator.*

```
FUN  _ − _        : [nat , nat] →  nat
PREC (_ − _) − _
FUN  3            :  nat
FUN  5            :  nat
FUN  10           :  nat
FUN   E           :  nat
DEF   E           ==10 − (5 − 3)
```

Figure 4.1: Signature of example 41

4.2.3 Built-In Type Constructors

Section Expression Constructors

Every declaration of an n-ary operator shall have as its type a *section expression* $[\mathbf{T}_1 , \ldots , \mathbf{T_n}] \to \mathbf{T}_0$ where it is allowed to write simply $\mathbf{T}_0$ instead of $[] \to \mathbf{T}_0$ for nullary operators.

To this end, we introduce the following type constructor operators which exist on every level of the type-universe.

```
VAR A
VAR B
FUN _ , _ : [A , B] → SORT
FUN [ _ ] : [A] → SORT
FUN _ → _ : [A , B] → SORT
FUN ϵ     :  SORT
FUN _ , _ : [A , B] → (A , B)
FUN [ _ ] : [A] → [A]
FUN _ → _ : [A , B] → (A → B)
FUN ϵ     : ϵ
```

Tupling Constructor

Combining the comma-operator with the parenthesis operator we automatically get *tuple* expressions. We treat a tuple made up of types as the *product* of these types.

Union Constructor

We also introduce union expressions $\bigcup[\mathbf{V}_1 , \ldots , \mathbf{V_n}]$ to describe overloading.

FUN $\bigcup$ _ : $[[\mathbf{A}]] \to$ **SORT**
FUN $\bigcup$ _ : $[[\mathbf{A}]] \to \bigcup[\mathbf{A}]$

If an operator is overloaded, it can be seen as a union of its definitions. The type of such a union can again be seen as the union of the types of the definitions.

Example 42 *In the signature in figure 4.2, the operator _ + _ is overloaded. Therefore, it has the type* $\bigcup[[\mathbf{nat}, \mathbf{nat}] \to \mathbf{nat}, [\mathbf{real}, \mathbf{real}] \to \mathbf{real}]$. *It also has two definitions which can also be seen as a union.*

FUN _ + _ : $[\mathbf{nat}, \mathbf{nat}] \to$ **nat**
DEF _ + _ ==**addNat** _ _
FUN _ + _ : $[\mathbf{real}, \mathbf{real}] \to$ **real**
DEF _ + _ ==**addReal** _ _
LAW _ + _ = $\bigcup$[**addNat** _ _ , **addReal** _ _]
    -- _ + _ : $\bigcup[[\mathbf{nat}, \mathbf{nat}] \to \mathbf{nat}, [\mathbf{real}, \mathbf{real}] \to \mathbf{real}]$

Figure 4.2: Signature of example 42

4.2.4 Placeholders and Sections

Like in every higher-order functional language, the operators shall be first-class citizens and therefore shall also be usable *uninstantiated* as operands to other operators.

To this end, we define two *placeholder operators* that help us denote *uninstantiated* operator instantiations: the *underscore operator* and the *empty operator*.

FUN "_" : $[] \to [\mathbf{A}] \to \mathbf{A}$
FUN ϵ : $[] \to [\mathbf{A}] \to \mathbf{A}$

Even though both operators are nullary, they yield a section type as their result. Thus, we will allow them also to be used as if being of type $[\mathbf{A}] \to \mathbf{A}$.

By instantiating a non-nullary operator only with underscore operators, we get an expression that looks exactly like the operator pattern itself. Also, we can infer exactly the type declared for that operator pattern for this expression. *Thus, operator patterns are already mixfix expressions.*

Example 43 *If we want to pass operators (like _ = _) as arguments to higher-order functions like _* **compared by** *_ in the signature in figure 4.3, we can do this by either instantiating the operator pattern with underscore placeholder expressions or, for shortness sake, with empty placeholder expressions.*

In the signature, the operator _ = _ is also lifted from a section to a full-fledged function by enclosing it in parentheses, as explained in 4.2.5.

```
FUN _ = _              : [A , A] → bool
FUN _ compared by _ : [seq A , (A , A) → bool] → seq seq A
FUN E1 : seq seq nat
DEF E1 == [1 , 3 , 5 , 1] compared by(_ = _)
FUN E2    : seq seq nat
DEF E2 == [1 , 3 , 5 , 1] compared by( = )
```

Figure 4.3: Signature of example 43

Even better, this approach is compositional in such a way that operators can also be *partially instantiated* , i.e. some of their operands are instantiated with placeholders while other operands are instantiated with other expressions.

Definition 41 *We call an operator instantiation* partially instantiated *or* operator section expression *when at least one of its operands is either a placeholder expression or a partially instantiated expression.*

We call an operator instantiation fully instantiated *if it is not partially instantiated, i.e. all placeholders have been instantiated with fully instantiated expressions.*

The *empty placeholder operator* in turn is useful to arrive at expressions which we call *operator name expressions* . These expressions are useful for passing operators as operands to higher-order functions without having to write down the full operator pattern.

Definition 42 *An* operator name expression *is an instantiation of an operator with only empty placeholder operands, yielding the sequence of the separator tokens of that operator, which we call the* operator name.

Example 44 *In the signature in figure 4.4, both expressions* $\mathbf{S}_1$ *and* $\mathbf{S}_2$ *have the same semantics, but the first is defined with the help of the underscore placeholder operator while the second is defined using the empty placeholder operator, using the operator name* **if then else**.

```
FUN while _ do _         : [bool , stmt] → stmt
FUN if _ then _ else _ : [bool , stmt , stmt] → stmt
FUN S1                         : [bool , bool , stmt , stmt] → stmt
DEF S1 ==                      while _ do if_ then _ else _
FUN S2                         : [bool , bool , stmt , stmt] → stmt
DEF S2 ==                      while do if then else
```

Figure 4.4: Signature of example 44

4.2.5 Lifted Sections

Section expressions can be lifted to function values by enclosing them in a parenthesis.

```
FUN ( _ ) : [[A] → B] → (A) → B
```

Such a lifting is necessary because of the distinction between the inhomogeneous operand type lists $[\mathbf{T}_1 , \ldots , \mathbf{T_n}]$ of section types and the product types $(\mathbf{T}_1 , \ldots , \mathbf{T_n})$.

With its help, every mixfix expression, if enclosed in a parenthesis, can be used as a function, e.g. applied to other expressions via the apply operator or given to higher-order functions that demand such a function as their operand.

Example 45 *In the signature in figure 4.5, we import the type constructors* **seq** _ *and* **nat** *from the structures* **Seq** *and* **Nat** *to be able to declare the map operator* _ $*$ _. *It can then be used to map expression* $\mathbf{E}_1$ *which is a sequence of triples of type* **nat** *to a sequence of elements of type* **nat** *by lifting the expression* _ + _ + _ *to the type* **(nat , nat , nat)** $\rightarrow$ **nat** *via the lifting operator.*

```
USE  Seq
USE  Nat
VAR  A     : SORT
VAR  B     : SORT
FUN  _ * _ : [A → B , seq A] → seq B
FUN  E₁    : seq(nat , nat , nat)
FUN  E₂    : seq nat
DEF  E₂ ==(_ + _ + _) * E₁
```

Figure 4.5: Signature of example 45

Although section lifting could possibly be allowed at any point in an expression where an operand of an operator is instantiated, this could lead to very confusing expressions, which is why we only allow it in this limited fashion.

Example 46 *Given the* application operator *as declared in the signature in figure 4.6, every mixfix expression (if enclosed in parentheses) can be used in prefix application to a tuple of its uninstantiated operands. The same arity is maintained so it is clear which argument instantiates which operand.*

```
FUN  _ _ : [A → B , A] → B
FUN  fst: (A , B) → A
                -- (_ + fst _ + _) : (nat , (nat , B) , nat) → nat
LAW     (_ + fst _ + _)(3 , (4 , 5) , 6) = 3 + fst(4 , 5) + 6
```

Figure 4.6: Signature of example 46

Example 47 *With the help of an additional* currying converter operator *as in the signature in figure 4.7, the same can even be achieved without the use of tuples. If there is such a converter present that takes a function as its operand and yields its curried version, application of former uncurried functions can take place without the use of parentheses.*

4.2.6 Annotation Expressions

Scope Annotation

If the user wants to interpret an expression inside a different scope than the one the expression is syntactically present in, they can do so by using the *selection operator* _ . _ which can also be seen as a prefix scope annotation .

FUN _ : $[(\mathbf{A}\ ,\ \mathbf{B}) \to\ \mathbf{C}] \to\ \mathbf{A} \to\ \mathbf{B} \to\ \mathbf{C}$
  -- (_ + **fst** _ + _) 3 : $((\mathbf{nat}\ ,\mathbf{B})\ ,\ \mathbf{nat}) \to\ \mathbf{nat}$
  -- (_ + **fst** _ + _) 3 (4 , 5) : $\mathbf{nat} \to\ \mathbf{nat}$
LAW (_ + **fst** _ + _) 3 (4 , 5) 6 $= 3 + \mathbf{fst}(4\ ,5) + 6$

Figure 4.7: Signature of example 47

FUN _ . _ : $[\mathbf{LIFT\ S}\ ,\ \mathbf{A}] \to\ \mathbf{S}\ .\ \mathbf{A}$

Type Annotation

Likewise, the user can annotate an expression postfix via the _ : _ operator to constrain the type the expression should be interpreted under.

FUN _ : _ : $[\mathbf{A}\ ,\ \mathbf{LIFT\ A}] \to\ \mathbf{A}$

These annotations can be especially useful in environments where overloading is present to determine which of the different versions of an operator is meant by the user.

4.2.7 Precedences of the Meta Operators

The infix meta-operators are ordered precedence-wise in the following transitive ordering relation (where $\mathbf{o}_1 < \mathbf{o}_2$ means that $\mathbf{o}_2$ is both left- and right-precedent towards $\mathbf{o}_1$): (_ , _) < (_ : _) < (_ $\to$ _) < (_ . _)

The operators (_ , _) , (_ : _) and (_ $\to$ _) are right-precedent, while the operator (_ . _) is left-precedent.

This is achieved by the following precedence relation:

PREC _ , (_ , _)
PREC (_ : _) , (_ : _)
PREC (_ $\to$ _) , (_ $\to$ _)
PREC (_ . _) , (_ . _)
PREC _ : (_ : _)
PREC (_ $\to$ _) : (_ $\to$ _)
PREC (_ . _) : (_ . _)
PREC _ $\to$ (_ $\to$ _)
PREC (_ . _) $\to$ (_ . _)
PREC (_ . _) . _

The meta-operators and other mixfix operators are treated separately by the parser because the meta-operators (i.e. the selection operator) might change the scope for part of the expression while the scope must be considered fixed for any real mixfix sub-expression.

Also, the type annotation operator must first evaluate its *right* operand before the mixfix parser can be applied to its left operand, if top-down type information is to be used in the mixfix parsing process.

Mixfix expressions shall only contain meta-expressions in place of *enclosed operands*, i.e. placeholders that have a token directly to their left and right. Thus, such a meta-expression can and must be syntactically separated via such enclosing operators as [_] or (_).

4.2.8 Context-Free Meta Grammar

We can describe the meta expressions also with the help of an (unfortunately ambiguous) context-free grammar .

Definition 43 *The context-free* meta grammar *is defined as follows:*

$$
\begin{array}{lll}
\textbf{COMMA} & ::= & \textbf{COLON}\ \underline{,}\ \textbf{COMMA} \\
\textbf{COMMA} & ::= & \textbf{COLON} \\
\textbf{COLON} & ::= & \textbf{ARROW}\ \underline{:}\ \textbf{COLON} \\
\textbf{COLON} & ::= & \textbf{ARROW} \\
\textbf{ARROW} & ::= & \textbf{DOT}\ \underline{\to}\ \textbf{ARROW} \\
\textbf{ARROW} & ::= & \textbf{DOT} \\
\textbf{DOT} & ::= & \textbf{DOT}\ \underline{.}\ \textbf{MIXFIX} \\
\textbf{DOT} & ::= & \textbf{E} \\
\textbf{MIXFIX} & ::= & \underline{\texttt{phrase}}^{\star} \\
\textbf{E} & ::= & \textbf{GROUP}
\end{array}
$$

The nonterminal **COMMA** will be used for enclosed operands inside mixfix expressions while the **COLON** nonterminal is the start symbol for every expression appearing on the right-hand-side of a declaration (and both on the left-hand-side and the right-hand-side of every definition).

The nonterminal symbol **MIXFIX** represents the actual mixfix expressions which can contain any phrase, even the group phrases as well as the token symbols of the meta operators. This is because the operator set may change according to the rule **DOT** $\underline{.}$ **MIXFIX** where the **MIXFIX** part will be parsed by the mixfix parser in the scope represented by the **DOT** part which might introduce new operators or override old ones.

Remark 1 *It could also be possible to let the left-hand-side of the arrow-expressions introduce variable operators that can be used on the right-hand-side, thereby getting a lambda-like expression* $(?\mathbf{x}\ ,?\mathbf{y}) \to \mathbf{x} +\ \ \mathbf{y}$ *so we do not have to use the clumsy version* $(\text{VAR } x \text{ VAR } y).((x, y) \to x + y)$ *or introduce an additional lambda-operator.*

Thus, we cannot parse the right operands of the arrow or scope annotation expressions without first evaluating the parse result of the left operands.

The rule **E** ::= **GROUP** is added so that group phrases are accepted as expressions[4].

Example 48 *If we would not include the token* $\underline{,}$ *under the terminal symbol* $\overline{\texttt{phrase}}$*, the expression in the signature in figure 4.8,* $(\text{USE } \mathit{Nat}).(\text{USE } \mathit{Set\ nat}).\{1, 2, \overline{3}\}$ *could only be interpreted as* $(((\text{USE } \mathit{Nat}).(\text{USE } \mathit{Set\ nat}).\{1), (2, (3\})))$ *which has no viable interpretation and is clearly not the intended use.*

The benefit of this approach, thus, is that we can use — without using any built-in parentheses — comma-separated lists of type- and scope-annotated mixfix expressions that introduce any number of new operators in place of every enclosed operand, for instance inside the built-in operator [_] which is used to describe our operator types.

As can be seen in the above example, it should also be possible to overload the built-in operators like _ , _ to give them additional semantics.

[4]**E** being the start symbol for all mixfix operator instantiations

```
VAR  A : SORT
FUN  Set _ : [LIFT A] → (
   FUN  set : SORT
   FUN  commalist : SORT
   FUN  _ , _ : [A , commalist] →  commalist
   FUN  _ , _ : [A , A] →  commalist
   FUN  { _ }  : [commalist] →  set
   FUN  { _ } : [A] →  set
   FUN  { _ } : [ϵ] →  set
        )

   FUN  S : ( USE Nat) . ( USE Set nat) . set
   DEF  S == (USE Nat) . ( USE Set nat) . {1 , 2 , 3}
```

Figure 4.8: Signature of example 48

4.3 Mixfix Expression Language

Given the meta-operators, we can now define the language of mixfix expressions as all those mixfix operator instantiations that can be formed using an operator set in a given scope.

4.3.1 Mixfix Expressions

Definition 44 *The* mixfix expression language $\mathcal{L}(\Sigma)$ *for a given mixfix signature* Σ *is the set of all* meta operator instantiations *inside* Σ*, all* mixfix operator instantiations *inside* Σ *and all* group expressions *that are valid in* Σ*.*

A mixfix operator instantiation *inside a mixfix signature* Σ *is a sequence* $\mathbf{s_0}\ \mathbf{E_1}\ \mathbf{s_1}\ \ldots \mathbf{E_n}\ \mathbf{s_n}$ *where* $\mathbf{s_0}\ \_\ \mathbf{s_1} \ldots \_\ \mathbf{s_n}$ *is a mixfix operator pattern from* Σ *with* $\mathbf{s_i} \in \mathbf{token}^\star$*, and* $\mathbf{E_i} \in \underline{\texttt{phrase}}^\star$*, with*

- $\mathbf{E_i}$ *is a mixfix operator instantiation inside* Σ *or a group expression that is valid in* Σ*, if* $s_{i-1} = \epsilon$ *or* $s_i = \epsilon$*, or*
- $\mathbf{E_i} \in \mathcal{L}(\Sigma)$*, otherwise.*

A meta operator instantiation *inside a mixfix signature* Σ *is an operator instantiation of a meta operator inside* Σ *where the non-enclosed operands can also be instantiated with meta operator instantiations inside* Σ *according to the precedences between the meta operators.*

A group expression *that is valid inside a mixfix signature* Σ *is a* group *symbol, representing a list of parsable declarations in parentheses that can use operators from* Σ*. Every group expression represents its own operator signature .*

Example 49 *In the signature in figure 4.9, the expression* **if x** $=$ 0 **then** 1 **else** (**x** $-$ 1) ! $*$ **x** *is a mixfix operator instantiation of the mixfix operator pattern* **if** _ **then** _ **else** _ *with the mixfix expressions* **x** $=$ 0*,* 1 *and* (**x** $-$ 1) ! $*$ **x***.*

The language $\mathcal{L}(\Sigma)$ only constrains the token sequences that are recognizable as mixfix expressions *syntactically.*

But of course, many of these expressions might *still* be syntactically ambiguous. For instance, the number of viable syntactic interpretations can be infinite if invisible operators are present in the given operator set.

```
FUN  if _ then _ else _ : [bool , nat , nat] → nat
FUN  _ = _              : [nat , nat] → bool
FUN  0                  : nat
FUN  1                  : nat
FUN  _ − _              : [nat , nat] → nat
FUN  _ ∗ _              : [nat , nat] → nat
FUN  _ !                : [nat] → nat

VAR  x                  : nat
FUN  fac _              : [nat] → nat
DEF  fac x              == if x = 0 then 1 else (x − 1) ! ∗ x
```

Figure 4.9: Signature of example 49

Restricting operand instantiation with the help of operator fixities, precedence relations and type compatibility will help us restrict the language to an unambiguous subset.

But as we shall see, the possible operator sets must also be restricted syntactically because not all syntactic ambiguities can be removed with the above-mentioned means.

4.3.2 Context-Free Mixfix Grammar

The context-free grammar for a given mixfix operator signature describing the mixfix operator instantiations inside that signature is very straightforward.

Definition 45 *The context-free mixfix grammar for mixfix operator signature Σ is defined as follows:*

- *There is one nonterminal symbol* **E** *representing the mixfix operator instantiations.*
- *There is one rule for* **E** *for every operator pattern inside Σ. The right-hand side for that operator pattern is derived as follows:*
  - *map every separator token to its respective terminal symbol*
  - *map every enclosed placeholder token*[5] *to nonterminal* **COMMA**
  - *map every left-open, right-open or adjacent placeholder token to nonterminal* **E**

Adhering to definition 44, we treat enclosed operands differently from open or adjacent ones, because **COMMA** would make the grammar ambiguous if used in place of an adjacent, left- or right-open operand, but does not for enclosed operands. We want to be able to use comma-separated lists and concatenation expressions in as many places as possible, so we do not restrict enclosed operands more than necessary while applying the necessary restrictions to other operands.

Example 50 *The context-free mixfix grammar for the operator signature from example 49 would look like the one in figure 4.10.*

[5]a placeholder symbol which appears between two non-empty separators

E ::= <u>`if`</u> **COMMA** <u>`then`</u> **COMMA** <u>`else`</u> **E**
E ::= **E** <u>=</u> **E**
E ::= <u>`0`</u>
E ::= <u>`1`</u>
E ::= **E** <u>−</u> **E**
E ::= **E** <u>∗</u> **E**
E ::= **E** <u>!</u>
E ::= <u>`x`</u>
E ::= <u>`fac`</u> **E**

Figure 4.10: Grammar of example 50

4.4 Mixfix Language Type System

In a language like our mixfix expression language, a lot of ambiguities can occur that are only resolvable by use of typing information.

4.4.1 Types and Values

In our multi-level scenario, the type language of a mixfix expression language describing the *values* is also a mixfix expression language. However, there are usually some restrictions what kind of constructions are allowed for the values of the type language[6].

If we want to talk about type inference for a mixfix expression language, we view the types not as mixfix expressions, but as already interpreted expressions, i.e. their unambiguously parsed parse trees. This entails, of course, that type annotations have to be parsed and unambiguously interpreted before it is possible to use that annotation for the disambiguation of the annotated expression.

4.4.2 Type-Correctness and Ambiguity

All expressions can be characterized by their types and we can distinguish between type-correct and type-incorrect expressions.

Type-incorrect expressions are syntactically valid mixfix operator instantiations where the types of the operand expressions do not agree with the corresponding operand-types of the instantiated operator.

If an expression has no type-correct interpretation for a given result type, it is considered erroneous and should be rejected by the parser.

But an expression might have type-correct and type-incorrect interpretations at the same time, even for the same result type. It also might be erroneous for one result type and have only type-correct interpretations for another type.

If an expression has more than one type-correct interpretation, for a given result type, it is called ambiguous *for that type.*

The parser should be able to reject syntactically ambiguous expressions. But by use of precedence relations a *preferred* type-correct syntactic interpretation might be unambiguously *chosen* for an otherwise syntactically ambiguous expression and therefore accepted by the parser.

Example 51 *Taking only type information to disambiguate, the operator* $\_+\_$ *will yield ambiguous interpretations for all mixfix expressions that contain several occurrences of it.*

[6]even though this is outside the scope of this thesis

However, by use of ad-hoc left precedence for this operator, the left-precedent of the two possible variants is chosen among the type-correct ones, making the expressions unambiguous.

```
FUN  nat       : SORT
FUN  _ + _     : [nat , nat] → nat
FUN  a         : nat
FUN  E         : nat
PREC (_ + _) + _
DEF  E ==        a+ a+ a+ a
                    -- chosen: ((a+a)+ a)+ a
                    -- not chosen: (a+(a+a))+ a
                    -- not chosen: (a+a)+(a+ a)
                    -- not chosen: a+((a+ a)+ a)
                    -- not chosen: a+(a+(a+ a))
```

4.4.3 Multi-Level Signatures

Following this chain of reasoning, our type system must be powerful enough to describe all those expressions which should be denotable and help us to find as many type-errors as possible as well as distinguish the different interpretations for an ambiguous expression.

We want to be able to introduce our own types, define generic and overloaded operators and use types, functions, sections, products and groups as first-class citizens. This shall allow us to write multi-level specifications without the need to invent new terminology for every new meta-level, as it is done in most programming languages. The values of all levels above level n can be used as the types of level n.

Since every actual mixfix expression in a scope lives only on one level, we can clearly distinguish between values and types in that level and do not have to cope with different type levels for any one expression. Thus, the type system we introduce can be considered to exist separately for every level. Though this is only true if it can be determined for every expression on which level it lies, this is possible via an inference algorithm[7].

Besides the built-in type constructors, the user can introduce other mixfix operators as type constructors, using types of higher levels for their declaration.

The distinction into levels is actually an artificial one. In essence, we simply need to establish a topological order between the different declarations. Each declaration is dependent on the operators used on its right-hand side. If the declarations of all operators on the right-hand side have parsable declarations, then it is also possible to parse the declaration.

It is even possible to overload type and value operators, as long as one does not use the other in its declaration (as that would violate the topological order condition). In such a case, the operators have to be introduced in different scopes, so they can be distinguished by scoping annotation.

[7]see [37])

Example 52 *The multi-level signature in figure 4.11 has 4 levels 0 to 3.*

1. *Level 3 only declares the type* **SORT** *to be used for the declarations of the type constructors on level 2.*
2. *On level 2, we declare*
   - *the normal simple types* **nat**, **bool** *as well as*
   - *the type constructor* **seq** _, *and*
   - *a* type class **ord** *for orderable types and a type class constructor* **container** _ *for container types*
   - *two type variables* **A** *and* **B** *allowing us to declare the type constructors on level 1 and the operators on level 0 in a generic way*
3. *On level 1, we declare*
   - *again, the types* **nat** *and* **seq** _, *this time as instances of orderable types and container types, respectively*[8]
   - *a container type constructor* **map** _ **to** _ *for mappings where the contained elements are indexable by key-elements of an orderable type*
   - *the container type variables* $\mathbf{C}_1$, $\mathbf{C}_2$ *and the orderable type variable* **O** *which enable us to declare the operators on level 0 in a generic way*
4. *On level 0, we declare*
   - *a comparison operator* _ < _ *for any pair of values of the same orderable type*
   - *the value* **s** *as a sequence of natural numbers which in turn is a container type for natural numbers*
   - *the value* **m** *as a mapping from natural numbers to sequences of natural numbers which is a container type for sequences of natural numbers*
   - *the function* **sort** *which takes a sequence of any orderable type and yields a (sorted) sequence of elements of the same type*
   - *the map operator* _ ∗ _ *which takes a function of type* $\mathbf{A} \to \mathbf{B}$ *and applies it to a container of elements of type* **A**, *yielding a container of elements of type* **B** *(with the same structure).*

[8]This is necessary to allow type-consistent unification of these types with type variables of type **ord** or **container** respectively.

FUN	**SORT**	: **SORT**
		-- *level 2*
FUN	**bool**	: **SORT**
FUN	**nat**	: **SORT**
FUN	**seq** _	: [**SORT**] → **SORT**
FUN	**ord**	: **SORT**
FUN	**container** _	: [**SORT**] → **SORT**
VAR	**A**	: **SORT**
VAR	**B**	: **SORT**
		-- *level 1*
FUN	**nat**	: **ord**
FUN	**seq** _	: [**LIFT A**] → **container A**
FUN	**map** _ **to** _	: [**ord** , **LIFT A**] → **container A**
VAR	**O**	: **ord**
VAR	$\mathbf{C}_1$	: **container A**
VAR	$\mathbf{C}_2$	: **container A**
		-- *level 0*
FUN	_ < _	: [**O** , **O**] → **bool**
FUN	**s**	: **seq nat** : **container nat**
FUN	**m**	: **map nat to seq nat** : **container seq nat**
FUN	**sort**	: **seq O** → **seq O**
FUN	_ ∗ _	: [**A** → **B** , $\mathbf{C}_1$: **container A**] → ($\mathbf{C}_2$: **container B**)

Figure 4.11: Signature of example 52

Since it is not really relevant to our problems with disambiguation, we consider further explorations of this topic outside the scope of this thesis.

A scoping and type-level inference algorithm for multi-level signatures can be found in [37].

In [42], Visser explores the topic of multi-level specifications, given quite a few examples of their usefulness and expressive power, while he tackles the problems of type unification in [43].

Comparison with Haskell's Type Classes

In Haskell [18], there exists a construction for categorizing (i.e. typing types of values) by declaring *type classes* [16] which are predicates over a type, asserting the existence of certain operators for that type and then explicitely declaring given types as instances of such type classes, supplying the actual implementation of the demanded operators. Thus, a kind of overloading can be achieved in Haskell.

We can achieve the same in our multi-level specification approach by introducing a sort for every such type class and then declaring different types to be of that sort so they become instances of that class.

By declaring a type variable to be of different type classes, we can achieve the same as is done by *type contexts* of *generic type schemes* $\forall\overline{u}.\pi \Rightarrow \tau$ which is a type generic over the type variables $\overline{u}$ where for all instantiations of the variables $\overline{u}$ in τ the property π must hold. A similar construction exists in our language. Using **C** . **t** where **C** is a group declaring type variables with their type constraints, we get a type **t** which is generic over these variables.

But, obviously, we have a greater freedom of expression since we can also categorize type classes, etc. .

A different approach to implementing type classes in multi-level specifications can also be found in [42].

4.4.4 Bottom-Up Type Inference

We could describe our type system by bottom-up inference rules, using predicates like $S \vdash E : [L] \rightarrow \tau$ which signifies that expression E can have section type $[L] \rightarrow \tau$ in scope S.

Even though this algorithm does not provide us with an efficient disambiguation tool, it is more easy to understand than the actually needed top-down bottom-up inference algorithm, presented thereafter. The latter algorithm is basically a refinement of the former.

Mixfix Instantiation

The type of an operator instantiation can be inferred from the type the operator is declared with and the inferred types of the operand expressions.

$$\frac{\forall i \in 1, \ldots, n : S \vdash E_i : [L_i] \rightarrow \tau_i}{S \vdash s_0(E_i\ s_i)_{i=1}^{n} : ([L_1] \ldots [L_n] \rightarrow T_0)[T_i := \tau_i]_{i=1}^{n}}\, s_0(\_\ s_i)_{i=1}^{n} : [T_1, \ldots, T_n] \rightarrow T_0 \in S$$

We build a section type $[\mathbf{L}_1] \ldots [\mathbf{L_n}] \rightarrow \mathbf{T}_0$ which has the concatenation of the operand-type lists $[\mathbf{L_i}]$ of the inferred section types of all operand expressions as its operand-types and the result type $\mathbf{T}_0$ of the declared section type of the operator as its result type. Then, we apply a substitution $[\mathbf{T_i} := \tau_\mathbf{i}]_\mathbf{i=1}^\mathbf{n}$ to that section type which is computed by unifying each operand type $\mathbf{T_i}$ in the operator declaration with the result type $\tau_\mathbf{i}$ of the inferred section type for the corresponding operand expression.

Nullary Operator Unlifting

For nullary operators $\mathbf{s}_0 : [] \rightarrow [\mathbf{L}] \rightarrow \mathbf{T}$ which yield a section type, like our placeholder operators $"\_" : [] \rightarrow [\mathbf{A}] \rightarrow \mathbf{A}$ and $\epsilon : [] \rightarrow [\mathbf{A}] \rightarrow \mathbf{A}$, we need a rule that lets us use these nullary operator instantiations , as if they were partially instantiated section expressions.

$$\frac{}{S \vdash s_0 : [L] \rightarrow T}\, s_0 : [] \rightarrow [L] \rightarrow T \in S$$

This way, the user can introduce his own placeholder symbols or achieve similar effects with other nullary operators resulting in section types.

This behavior could probably be generalized to automatically unlift other fully instantiated expressions that yield a section type as their result,i.e. all expressions of an inferred type like $[] \rightarrow [\mathbf{L}] \rightarrow \mathbf{T}$. But we are not sure, whether this feature would not be too confusing to use.

Section Lifting

Another useful feature is the lifting of partially instantiated section expressions to fully instantiated function expressions by putting a parenthesis around the expression to be lifted.

$$\frac{S \vdash E : [L] \rightarrow T}{S \vdash (E) : [] \rightarrow (L) \rightarrow T}$$

Again, this rule could be generalized to apply to non-parenthesized expressions, but we deem this a too confusing concept.

LIFT construct

The bottom-up inference rule for the **LIFT** construct is very simple, following directly from the law that every expression **E** should be interpretable as a fully instantiated expression of type **LIFT E** which is just the shorthand notation for [] → **LIFT E**.

$$\frac{}{S \vdash E : [] \rightarrow \texttt{LIFT}\, E}$$

Type Annotation

For type-annotated expressions **E** : **T**, the type-annotation **T** has the inferred type [] → **LIFT** τ_2. The type τ_2 is more special than **T** and must agree with the inferred type τ_1 of the annotated expression **E**.

$$\frac{S \vdash T : [] \rightarrow \texttt{LIFT}\ \tau_2 \quad S \vdash E : \tau_1}{S \vdash E \mathrel{\underset{-}{:}} T : \tau_1[\tau_1 := \tau_2]}$$

Therefore, we apply the substitution $[\tau_1 := \tau_2]$ to the inferred type τ_1 and take this as the inferred type of the whole expression.

Non-Empty-Section Type Annotation

We assume that every type-annotation is meant as a section type by the user, so it is possible to describe section expressions via type-annotation.

However, since it should also be allowed to use non-section expressions as annotations to describe fully instantiated section expressions, we treat these as if the user had given a fully instantiated section expression as annotation.

Also, it should be possible to annotate an expression which results in a section type [**L**] → **T** without having to annotate it with [] → [**L**] → **T**.

Both these shortcut-notations are made possible by the following rule. The inferred type τ_2 of the annotation expression **T** is treated as the result type of an empty section expression of type [] → τ_2. If τ_1 agrees with this type, we get the overall inferred type by unifying them.

$$\frac{S \vdash T : [] \rightarrow \texttt{LIFT}\ \tau_2 \quad S \vdash E : \tau_1}{S \vdash E \mathrel{\underset{-}{:}} T : \tau_1[\tau_1 := [] \rightarrow \tau_2]}\ \tau_2 \neq [] \rightarrow T'$$

Scope Annotation

Finally, for scope annotation, we must evaluate the scope **S′** the expression **E** is annotated with, add it to the scope **S**, using some scoping rules in the process, and infer the actual type of the expression **E** in the new scope **S** ⊕ **S′**.

$$\frac{S \oplus S' \vdash E : \tau}{S \vdash S' \mathrel{\underset{-}{.}} E : \tau}$$

4.4.5 Top-Down Bottom-Up Type Inference

Inherent Deficiency of the Bottom-Up Inference

Unfortunately, pure bottom-up type inference as described above is not sufficient as an *efficient* disambiguation tool. An expression might have an exponential number of syntactic interpretations, all of which have different types, where the selection of the right interpretation can only be a filtering process over this exponential set.

Example 53 *The generic type-constructor* **pair _ _** *can be used to describe any kind of binary structure. For each of the three different structures, given by the types of* $\mathbf{E}_1$*,* $\mathbf{E}_2$ *and* $\mathbf{E}_3$*, the same expression* **a & a & a & a** *has a different interpretation.*

```
VAR A        : SORT
VAR B        : SORT
FUN pair _ _ : [SORT , SORT] → SORT
FUN _ & _    : [A , B] → pair A B
FUN a        : nat
FUN E1       : pair (pair nat nat) (pair nat nat)
DEF E1 ==     a & a & a & a
                  -- (a & a) & (a & a)
FUN E2       : pair nat (pair nat (pair nat nat))
DEF E2 ==     a & a & a & a
                  -- a & (a & (a & a))
FUN E3       : pair(pair (pair nat nat) nat) nat
DEF E3 ==     a & a & a & a
                  -- ((a & a) & a) & a
```

Top-Down Approach Using Demanded Result Types

If the demanded result type for an operator instantiation is taken into account, much better disambiguation can be achieved using a top-down bottom-up inference algorithm.

We give this algorithm using statements of the form $S, \tau \vdash E : [L] \rightarrow T$, meaning that E can be inferred to be of section type $[L] \rightarrow T$ in scope S, if τ is the demanded result type of E.

For instance:

$$\{\_ + \_ : [nat, nat] \rightarrow nat, \text{"\_"} : [] \rightarrow [A] \rightarrow A\}, nat \vdash \_ + \_ + \_ : [nat, nat, nat] \rightarrow nat$$

We additionally demand that, if $S, \tau \vdash E : [L] \rightarrow T$, then T is more special than τ, i.e. there is a substitution σ for the variables in τ so that $\sigma(\tau) = T$.

Operator Instantiation

The operator instantiation inference rule works much like in the bottom-up approach. However, the demanded types for the operands are computed by unifying the demanded type of the overall expression τ_0 with the topmost operator's result type T_0 and applying the resulting substitution to each respective operand type T_i.

Finally, also agreement between T_0 and τ_0 is enforced so that the result type of the inferred section is definitely more special than the demanded result type.

$$\frac{\forall i \in \{1, \ldots, n\} : S, T_i[T_0 := \tau_0] \vdash E_i : [L_i] \rightarrow \tau_i}{S, \tau_0 \vdash s_0\ (E_i\ s_i)_{i=1}^n : ([L_1] \ldots [L_n] \rightarrow T_0)[T_i := \tau_i]_{i=0}^n} s_0\ (\_\ s_i)_{i=1}^n : [T_i]_{i=1}^n \rightarrow T_0 \in S$$

It should be noted that here we do not take inferred type information from one operand expression as a demanded type for another operand into account. Such an influence from one operand to another would have to be arbitrarily chosen (e.g. from left to right) which could be confusing to users as to why some expressions can be correctly inferred and some very similar ones can not.

However, the built-in prefix application operator should behave differently in that respect, which is why we treat it specially.

Special Operator Instantiation: Application

For the prefix application operator $\_\ \_$: $[\mathbf{T}_2 \to \mathbf{T}_0\ ,\ \mathbf{T}_2] \to \mathbf{T}_0$, the type $\mathbf{T}_2$ of the right operand of the application does not directly depend on the result type $\mathbf{T}_0$. This dependence is only established via the type of the left operand $\mathbf{T}_2 \to \mathbf{T}_0$.

Thus, if we used the type inference as for every normal operator, we would use the demanded type $\mathbf{T}_2[\mathbf{T}_0 := \tau_0] = \mathbf{T}_2$ where $\mathbf{T}_2$ is a type-variable which does not restrict the demanded type for the right operand at all.

Using $\mathbf{T}_2[\mathbf{T}_0 := \tau_0\ ,\ \mathbf{T}_2 \to \mathbf{T}_0 := \tau_1]$, instead, where τ_1 is the result type of the inferred section type of the left operand of the application, we have a very good restriction for the right operand.

This is one of the few cases where the order of type inference for the operands of an expression actually matters while it is normally insignificant in our approach.

$$\frac{S, T_1[T_0 := \tau_0] \vdash E_1 : [L_1] \to \tau_1 \quad S, T_2[T_i := \tau_i]_{i=0}^{1} \vdash E_2 : [L_2] \to \tau_2}{S, \tau_0 \vdash E_1\ E_2 : ([L_1][L_2] \to T_0)[T_i := \tau_i]_{i=0}^{2}} T_1 = T_2 \to T_0$$

It would be interesting to find denotational means in the language itself to describe such type-dependencies between the operands, and how they are allowed to influence each other during type-inference, as this could again lead to more unambiguous expressions. Unfortunately this is outside the scope of this thesis.

LIFT Construct

The type-inference for the **LIFT** construct for lifting a value to the type language is as straightforward in the top-down bottom-up approach as in the pure bottom-up approach.

Of course, because the inferred result type must be more special than the demanded type, this rule can only be applied in instances where the demanded type is of the form **LIFT T**.

$$\frac{S, \tau \vdash T : [] \to \tau_1 \quad S, \tau_1 \vdash E : [] \to \tau_2}{S, \texttt{LIFT}\ T \vdash E : [] \to \texttt{LIFT}\ E[E := T]}$$

Type Annotation

Using the inference for **LIFT** constructs, we can change our type annotation expression inference rule for expressions of the form **E** : **T** to top-down bottom-up behavior the following way.

First, we infer the type of **T** with demanded type **LIFT** τ_0 where τ_0 is the demanded type of the whole expression. The inferred type $[] \to$ **LIFT** $([\mathbf{L}] \to \tau_2)$ then contains $[\mathbf{L}] \to \tau_2$ which is more special than τ_0 and it is also more special than **T**.

Taking the result type τ_2 as the demanded type for expression **E** and afterwards enforcing agreement between the inferred type τ_1 with the inferred type $[\mathbf{L}] \to \tau_2$, we get the inferred type for the whole expression: $\tau_1[\tau_1 := [\mathbf{L}] \to \tau_2]$.

$$\frac{S, \texttt{LIFT}\ \tau_0 \vdash T : [] \to \texttt{LIFT}([L] \to \tau_2) \quad S, \tau_2 \vdash E : \tau_1}{S, \tau_0 \vdash E \mathrel{\underline{:}} T : \tau_1[\tau_1 := [L] \to \tau_2]}$$

Thus, the type annotation operator is another operator where the inferred type of one operand is used to determine the demanded type of the other. But in this case the dependency is from right to left.

While we could take the demanded type τ_0 also as the demanded type for the left operand, and check agreement with the annotation afterwards, the above approach is more sensible.

The annotation expression is usually more special than the actually demanded type, therefore having better disambiguation quality. We can argue, that if no disambiguation were necessary, the user would not have given an annotation. Therefore, naturally, the user's annotation should be taken into account for disambiguation.

Non-Empty-Section Type Annotation

The rule for type annotation with non-empty-section types works exactly the same as for the bottom-up inference.

$$\frac{S, \texttt{LIFT}\ \tau_0 \vdash T : [] \to \texttt{LIFT}\ \tau_2 \quad S, \tau_2 \vdash E : \tau_1}{S, \tau_0 \vdash E \mathrel{\underline{:}} T : \tau_1[\tau_1 := [] \to \tau_2]} \tau_2 \neq [] \to T'$$

Nullary Operator Unlifting

Unlifting of nullary operators almost works the same as for the bottom-up inference, but it also ensures the agreement between the inferred and the demanded result types.

$$\frac{}{S, \tau \vdash s_0 : ([L] \to T)[T := \tau]} s_0 : [] \to [L] \to T \in S$$

Section Lifting

Because we have to ensure the agreement between demanded type τ_0 and the result types of the inferred types, we unify these to get the actually inferred type of a lifted section expression.

$$\frac{S, T' \vdash E : [L] \to \tau_1}{S, (L') \to T' \vdash (E) : ([] \to (L) \to \tau_1)[(L) \to \tau_1 := (L') \to T']}$$

Basically, we only take the result type of the demanded function type and try to infer a section type with that result type for **E**. If the tuple made from the contents of the section-operands **(L)** is then unifiable with the domain type **(L′)** of the demanded function type, **E** can be lifted to a section of the appropriate type.

4.4.6 Relationship between Bottom-Up and Top-Down Type Inference

As already mentioned, the top-down bottom-up algorithm is a refinement of the simple bottom-up algorithm.

Every rule of the top-down algorithm is a refinement of a rule of the bottom-up algorithm which is done by converting every statement $S \vdash E : [L] \to T$ into $S, \tau \vdash E : [L] \to T$ and adding additional constraints on the applicability of each rule using the newly introduced variables.

Thus, the applicability of each rule is reduced, depending on the demanded result type of an expression. Whenever a type can be inferred with the top-down algorithm, a more general type can also be inferred by the bottom-up algorithm. The reverse is not true. There can be cases where the bottom-up algorithm can infer a type for the expression where the top-down algorithm can not, dependent on the given overall demanded result type of the expression.

If the bottom-up algorithm can infer a type for an expression and this inferred type can be unified with the overall demanded result type, then the top-down algorithm can also infer a more special type for the same expression.

The top-down algorithm has two advantages over the bottom-up algorithm. It yields more exact inferred types, and it allows for earlier disambiguation of the expression as operator applications whose result type does not agree with the demanded type need not be considered for rule application during the algorithm, therefore, pruning the paths the algorithm can actually take, thus making it more efficient.

If we had only operators where the demanded type of one operand is never influenced by the inferred type of another operand (like function application, type and scope annotation), only a top-down pass would be necessary.

4.4.7 Sufficiency of Top-Down Bottum-Up Inference for Parsing

One might wonder why two inference passes (top-down and bottom-up) are sufficient for such a complex type system such as ours while other languages which incorporate genericity and overloading need at least three passes.

The reason for that is that we only do as much type-inference as we need for syntactic disambiguation of expressions in the parsing phase. All semantic ambiguity remaining after the parsing phase needs still to be treated in the normal semantic analysis phase, where it is very probable that another top-down pass, using the demanded and inferred types, is necessary.

However, since the scope of our thesis is the syntactic analysis phase, we consider the semantic analysis of our mixfix expressions out of its scope.

Chapter 5

Ambiguity

After informally investigating the normal approaches to ambiguity in existing programming languages, this chapter formally explores the different causes of ambiguities and the restrictions and tools necessary to deal with them.

5.1 Kinds of Ambiguity

First, we have to determine what ambiguity of an expression actually means: An expression is ambiguous in a language, if there exist multiple interpretations of that expression in the language.

We differentiate between two kinds of ambiguity: syntactic and semantic.

Definition 46 *An expression is* syntactically ambiguous *for a given type, if two or more different syntactic structures (parse trees stripped of their type annotations) can be derived from it for that type.*

While this seems to be confusing syntactic and semantic information, this definition makes sense in our context because we use the type information to *syntactically* disambiguate an expression, i.e. to find the only type-correct parse tree which has the demanded result type. Thus, even if an expression might be syntactically ambiguous in general, it is still possible to be syntactically unambiguous for a specific given type.

Example 54 *Under the generic type α (a type variable), both given interpretations of the expression #<> :: <> would be type correct in the signature in figure 5.1.*

```
FUN <>    : seq A
FUN _::_ : [A , seq A] → seq A
FUN # _  : [seq A] → nat
FUN E1   : nat
FUN E2   : seq nat
DEF E1   == #<> :: <>
               -- # ((<> : seq A) :: (<> : seq seq A) : seq seq A) : nat
DEF E2   == #<> :: <>
               -- (#(<> : seq A) : nat) :: (<> : seq nat) : seq nat
```

Figure 5.1: Signature of example 54

If there are different unambiguous interpretations for an expression for different types, this can still lead to syntactic ambiguities if generic types are involved.

Definition 47 *An expression is* semantically ambiguous *for a given type, if two or more different semantic structures (type-annotated parse trees) with the same syntactic structure can be derived from it for that type.*

Example 55 *There are no purely syntactic means to semantically disambiguate the expression* |**v**| *in the signature in figure 5.2. It has two parse trees which have the same structure, but different type annotations at the inner node* **v**.

FUN | _ |: [**real**] → **real**
FUN | _ |: [**vector**] → **real**
FUN **v** : **real**
FUN **v** : **vector**
-- | **v** |== | (**v** : **real**) | : **real**
-- | **v** |== | (**v** : **vector**) | : **real**

Figure 5.2: Signature of example 55

Obviously, we need type analysis to determine if there occurs either of these kinds of ambiguity.

While syntactic ambiguity can always be circumvented by adding parentheses or type annotations to the expression, semantic ambiguity can sometimes only be circumvented by using type annotations for inner parts of the expression or by other means like scope restriction, i.e. overriding an overloaded operator in the scope of the ambiguous expression to exclude the other declarations of that operator from the scope.

5.2 Normal Approaches to Ambiguity

Before we present our own approach to disambiguation, let us examine the state of the art. How is the ambiguity problem tackled in practice in existing programming languages?

There are three standard approaches to cope with these ambiguities. We will give examples of these approaches in the rest of this section.

5.2.1 Syntactic Language Restrictions

The predominant solution is to restrict the *syntax*.

- Most programming languages do not allow the definition of true mixfix operators additional to the ones built into the grammar of the language to be defined at all.
- The precedences between the *predefined* mixfix operators are hard-coded into the grammar of the language by *precedence levels* (e.g. in [3]), also called *priorities* (e.g. in [45]).
- For most repetitive constructions, *longest or shortest match* rules are added to the grammar to avoid so-called *shift/reduce conflicts* for the generated LR parser ([3], [5]).
- Even if the language allows the use of prefix, infix or postfix operators that can be defined by the user in the language itself, often rules are added as to how they can be combined.

Example 56 *In OPAL [31], where every function can be used as prefix, postfix or, if it has multiple arguments, as infix operator, it is not possible to mix different usages in one expression without using parentheses.*

- Another approach is to use different application syntaxes for different fixities.

  Example 57 *In Haskell [18], all functions can either be used in prefix or in postfix applications, but identifiers that are used for postfix application have to be written in back-ticks, while graphemes that are used in prefix applications have to be put in a parenthesis.*

Precedence Restrictions

In some languages that do allow to add infix operators or even mixfix operators, the user has to restrict their use by giving precedences between them if they shall be mixable in one expression. These precedences then have to be adhered to in *all* those expressions.

Example 58 *The language SDF2*[1]*, defined in [46], allows the introduction of arbitrary mixfix operators together with precedences and associativities between these operators.*

Example 59 *The language OBJ3 ([14]) also allows the introduction of operator patterns. Associativity and precedence level can be given for each operator and for every operand it can be defined whether it must have a lower precedence level, or can have an equal precedence level to the operator or any precedence level.*

Unfortunately, even the least restrictive of these purely syntactic approaches have huge drawbacks. They forbid some semantically unambiguous expressions while allowing semantically nonsensical expressions that have to be filtered out by the semantic analysis.

Example 60 *Consider the operator signature in figure 5.3. if we assign the precedences* $\_\,! > \_ + \_ > \_ :: \_ > \#\ \_$*, then*

- $\#\ 1\ !\ +\ 1\ ::\ \texttt{<>}$ *is allowed as* $\#(((1!) + 1) :: \texttt{<>})$*, but*
- $1\ +\ \#\ 1\ !\ ::\ \texttt{<>}$ *is forbidden, even though there is only one type correct possibility:* $1\ +\ (\#((1\ !) ::\ \texttt{<>}))$*, and*
- $\#\ 1\ +\ 1\ ::\ \texttt{<>}\ !\ +\ \texttt{<>}$ *is allowed as* $\#\ ((1\ +\ 1) :: ((\texttt{<>}\ !) +\ \texttt{<>}))$ *which is clearly not type-correct.*

Any other arbitrarily chosen operator precedence would cause similar problems.

5.2.2 Semantic Language Restrictions

Other approaches to avoid ambiguities in the first place are restrictions of the power of the language itself by not allowing the use of polymorphism, overloading or higher-order functions.

This is clearly undesirable in the field of modern high-level functional languages which derive their power of expressiveness from these constructions.

[1]Syntax Definition Formalism 2

```
FUN 1       :  nat
FUN _ !     : [nat] →  nat
FUN _ + _ : [nat ,  nat] →  nat
FUN <>      :  seq nat
FUN _ ::    : [nat , seq nat] →  seq nat
FUN # _     : [seq nat] →  nat
```

Figure 5.3: Signature of example 60

Example 61 *In [1], the author defines* distfix grammars *as something very similar to our* mixfix grammars*, i.e. as describing operator instantiations where the* operator words *are distributed amongst the operands. However, the operator set of the distfix grammar is restricted in several ways:*

- *All operands of any one operator are separated by at least one non-empty operator word, i.e. no adjacent operands are allowed.*
- *An initial operator word cannot be used as a subsequent operator word.*
- *No whole sequence of operator words of one operator is allowed to be an initial sequence of operator words of another operator.*

This can be seen both as a syntactic restriction as well as a restriction on the overloading of operator words. It is – in part – akin to our properties to avoid operator overlap, as described in section 5.4.2 on page 82.

5.2.3 Semantic Filters

If the syntax is not restricted and the grammar is ambiguous, the set of possible parse trees resulting from the parsing of an expression can be filtered to allow only the ones that are type-correct. If exactly one such parse tree remains at the end of the filtering process, the expression was unambiguous.

However, if the grammar is ambiguous, every expression has potentially an exponential (if not infinite[2]) number of parse trees assigned by the parser. Thus the filtering algorithm would be of exponential (or non-terminating) worst-case-complexity dependent on the length of the expression.

In case that the ambiguity is only a syntactic one, such behavior is clearly very undesirable.

Example 62 *In [46], the language SDF allows the definition of operators with arbitrary patterns where the types of the operands are used as grammar nonterminals. These patterns are not otherwise restricted and every expression in the induced language is interpreted as having all possible type-correct interpretations. Therefore, a semantic filtering process must be employed to filter out type-incorrect and also type-correct but otherwise undesired interpretations.*

Example 63 *In OBJ3, the first parse tree found by the parser is selected, even though there might be several type-correct interpretations of an expression. Indeed, sometimes the parser fails, according to [14], to find any type-correct parse tree because of initial assumptions, even though one exists.*

[2]because of invisible operators

5.2.4 Conclusion

Concluding our survey of the standard approaches, we can say that in all of them it is often necessary for the programmer to use a lot of parentheses or split up the expressions into very small syntactically unambiguous chunks to get unambiguous, efficiently parsable expressions.

They must also be aware of the parsing technique of the language, even if they have possibilities to influence the disambiguation process. Thus, the parsing technique influences the specification technique.

Otherwise, the programmer may need to wait a long time for the parser to parse even correct programs.

All these approaches are thus – in our view – in some way inappropriate for solving the ambiguity problem for freely definable mixfix operators.

5.3 Causes of Ambiguity

In our mixfix expression language, there are four causes of ambiguity, three of which are more syntactic in nature while the last one can only be dealt with semantically. We will show in this chapter that elimination and proper restriction of these causes yields an unambiguous subset of our expression language.

cause	possible effect
shared separator tokens	backbone ambiguity
adjacent operands vs. non-unary invisible operators	adjacent-operand ambiguity
left-open vs. right-open operators	precedence-related ambiguity
polymphism or converter operators	type ambiguity

Table 5.1: causes for different kinds of ambiguity

As we have already motivated in section 2.5.2, if none of the first three kinds of ambiguity arises, we can determine at most one type-correct syntactic interpretation via backbone parsing, adjacent-operand-restricted left-weighted parsing and precedence reordering of the resulting left-weighted interpretation. Thus, the list of causes is closely related to our disambiguation process where any of the causes can have the effect of letting this process fail.

In the rest of this chapter, we shall explore each of these formally, showing in what way they actually do cause ambiguity and giving solutions as to how this can be prevented in the least restrictive ways.

Since all of these causes are more or less separate from each other, they can be surveyed and dealt with independently, leaving room for different unambiguous sub-languages of our mixfix expression language.

Example 64 *A mixfix expression language*

- *where every separator token occurs only once in the whole operator set,*
- *that has no invisible operators,*
- *which allows only left-open, but no right-open operators and*
- *where also no polymorphism is allowed*

would automatically be unambiguous.

However, such an approach is far from being the *least restrictive.* We have found much better solutions to the problems above that are still efficiently computable.

5.4 Shared Separator Tokens

In general, if a separator token can be shared by[3] two different separators[4] or if it can appear in the same separator multiple times, it can cause a syntactic ambiguity that we call *backbone ambiguity*.

Of course, this is just a necessary condition for that kind of ambiguity to occur. Most situations are much more complicated than that and there can be languages with shared separator tokens which do not have backbone ambiguity.

Definition 48 *A mixfix expression has a* backbone ambiguity *if there exists more than one derivation for it with the backbone grammar.*

It is not possible in general to determine for an arbitrary grammar whether any of the words belonging to the language of that grammar have a backbone ambiguity, as that is equivalent to the problem of determining ambiguity of the backbone grammar. This problem is undecidable for arbitrary grammars [4]. It is necessary to find one word in the language that contains a backbone ambiguity. If the set of expressions in the language is infinite, the searching process could be non-terminating.

However, we have found some satisfying conditions that imply that no backbone ambiguity can occur in the mixfix expression language induced by a given operator set. But in our approach *it is not really necessary* to find out whether the whole language induced by an operator set contains no backbone ambiguity, but only whether or not *each expression that actually occurs in a program* does. This approach is described in the following section.

5.4.1 Avoiding Backbone Ambiguity by Backbone Parsing

Lemma 2 *An expression contains no backbone ambiguity, if the backbone parser finds only one derivation for that expression using the backbone grammar of the operator set of the given scope.*

The backbone parser must be a generalized parser which can deal with left-recursion (since all rules of the backbone grammar are left-recursive) and finds all parse possible backbone parse trees for a given token sequence. Since the backbone grammar contains no right-recursion, it can, for instance, be dealt with by an Earley parser or a generalized LR parser (see [29]).

The proof of this is self-evident. If more than one derivation is found by backbone parsing, there is an ambiguity.

Efficiency

The effort to build a backbone parser out of a given operator set is linear to the sum of the lengths of the operators, since every operator must be mapped into a backbone grammar rule of maximally the same length plus one.

Parsing an arbitrary grammar with a generalized parser that recognizes multiple derivations for the same expression, e.g. an Earley parser, takes in the worst case cubic effort dependent on the length of the expression to be parsed.

[3]i.e. appear in

[4]in either the same or a different operator pattern

To make the resulting parser more efficient, it is advisable to filter the operator set before building the parser by using the tokens actually occurring in the expression. This filtering process needs to count the occurrence of each token in the expression and in each operator and leaves only those operators in the operator set that contain no extra tokens.

Example 65 *Take the operator patterns* | _ | *and* _ | _. *Even though they are overlapping in general, the backbone parser would not find a backbone ambiguity for expression* **a** | **b**, *if* **a** *and* **b** *have no backbone ambiguity and do not contain the token* |.

This is because the operator | _ | *causing the potential backbone ambiguity could be filtered out of the operator set given to the backbone parser. It has more occurrences of the token* | *than the expression itself.*

We need the following definitions to describe the subset **operators(OP , E)**.

Definition 49 *Let* **bag A** *be the type of* finite multisets $\{\mathbf{e_1} , \dots , \mathbf{e_n}\}$ *of elements* $\mathbf{e_i} : \mathbf{A}$*, i.e. sets where the number of occurrences of an element is significant, but not its position. That means, for instance,* $\{1 , 2 , 2\} = \{2 , 1 , 2\} \neq \{2 , 1\}$.

Every such multiset can be seen as a function of type **A** → **nat**.

```
VAR A     : SORT
FUN ∅     : bag A
FUN { _ } : [A] → bag A
FUN _ ∪ _ : [bag A , bag A] → bag A
FUN _     : [bag A] → (A → nat)
```

Definition 50 **count(s)(e)** *is the number of occurrences of an element* **e** *in the sequence* **s**. *The function* **count** *forgets the position of each element, thus yielding a bag.*

```
VAR e             : A
VAR s             : seq A
FUN count         : seq A → bag A
DEF count(<>)     ==∅
DEF count(e :: s) =={e} ∪ count(s)
```

Definition 51 *Now, we can be define the subset of occurring operators* **operators(Op , E)** *of the given operator set* **Op** *for the expression* **E***:*

```
DEF operators(Op , E)=={op ∈ Op|count(tokens(op)) ⊆ count(E)}
```

The effort to compute this subset is linear to the sum of the lengths of the operator patterns to be filtered and the length of the expression.

This effort can be worth spending when the operator set in a given scope is very large. Both the construction of the generalized parser as well as its computation are more efficient if only a potentially very small subset of rules has to be considered.

5.4.2 Avoiding Backbone Ambiguity by Separator Analysis

A different approach for avoiding backbone ambiguity is analyzing properties of the separators in the operators possibly occurring in the expression to determine if overlap of operators can actually take place.

Proposition 3 *If it can be determined for every pair of operators in* **operators(OP , E)** *that they cannot possibly overlap then an expression made up of these operators can contain no backbone ambiguity.*

Unfortunately, while we have found several restrictions to be imposed on the operator set which we believe to be sufficient to entail its overlap-freeness, we have found no proof of this.

The idea is to impose restrictions like unique identifiers (identifiers which occur only once in only one operator pattern) for every operator pattern or unique prefixes or suffixes of operator patterns which can be used to check whether or not an overlap between different operators could be possible.

However, since this is only conjecture, we leave the specifics of these restrictions out of this thesis.

5.5 Adjacent Operands vs. Invisible Operators

The next problem we are faced with in our mixfix expression language is ambiguity caused by adjacent operands. In conjunction with invisible operators, we can have situations where it is not exactly clear where the left one of the adjacent operand expressions ends and where the right one starts.

However, it is possible to find restrictions that allow us to *prefer* some syntactic interpretations for expressions which are ambiguous because of adjacent operands. These preferred interpretations are then seen as unambiguous in our mixfix expression language while the other interpretations are rejected. Although this is mainly a matter of taste, and it could be argued that such expressions should also be rejected, this would mean that only closed operators could be used for adjacent operand instantiations which we deem too much of a restriction.

We will show that these preferred syntactic interpretations are syntactically unambiguous, thus enlarging our class of unambiguously parsable mixfix expressions.

In this context, we will abstract both from typing of operators as well as their natural or ad-hoc precedences. We will deal with the real precedences amongst expressions with such unambiguous interpretations in the next section.

We also assume that no backbone ambiguity exists in the expressions to be restricted that way.

5.5.1 Left-Weighted Interpretations

To describe ambiguity that is dependent only on adjacent operands, we introduce another syntactic interpretation of mixfix expressions which abstracts from natural precedence and type-correctness, called *left-weighted expression interpretation* , by supposing a fixed left-precedence of all operators to each other and ignoring type-correctness, thereby gaining the possibility to reason about the adjacent operand parts without interference of precedence- or type-related ambiguity issues.

The idea behind this concept is to take every visible or empty operator as being left-precedent towards every left-open operator and no left-open operator as being right-precedent towards any left-open operator. The binary concatenation operator (which is sufficient to derive all possible concatenation expressions) is taken as only left-precedent only towards itself, while all other operators are left- and right-precedent to it.

Basically, we impose a pair of precedence relations $(\mathbf{Left}_L, \mathbf{Right}_L)$ with $\mathbf{Left}_L = (\mathbf{Visible} \cup \{\epsilon\}) \times \mathbf{LeftOpen} \cup \{(\_\,\_, \_\,\_)\}$ and $\mathbf{Right}_L = \mathbf{RightOpen} \times (\mathbf{Visible} \cup \{\epsilon\} \setminus \mathbf{LeftOpen})$ where **Visible** is the set of visible operators.

Simply put, this precedence relation ignores all other natural or ad-hoc precedence relations by always assuming left-precedence. All ambiguity that occurs in such left-weighted interpretations cannot be precedence related. Since we assume that also no backbone ambiguity is present, ambiguity can only occur in the mapping of sub-expressions to adjacent operands. By imposing several restrictions on expressions allowed as adjacent operands, we arrive at an unambiguous expression language, i.e. where all expressions have at most one acceptable syntactic interpretation.

Reordering that single parse tree according to the actually present natural and ad-hoc precedence relations, we will later arrive at a single type-correct parse tree, if one exists and also no precedence conflicts occur in the expression.

Basic and Concatenation Expression Interpretations

To become able to partition expression concatenations into adjacent operands, we want to avoid problems caused by concatenation operators. We do this by restricting adjacent operand instantiations to non-concatenation expressions. Therefore, we need a characterization of such expressions.

Definition 52 *A* basic expression interpretation *is an operator instantiation of a visible or empty operator.*

A concatenation expression interpretation *is an operator instantiation of a concatenation operator.*

Lemma 3 *Every mixfix expression can be interpreted as a basic expression or a concatenation expression interpretation.*

Proof 4 *Since every expression interpretation is always a mixfix operator instantiation, it can be either*

- *an instantiation of a visible or empty operator, which makes it a basic expression interpretation*
- *an instantiation of a concatenation operator, which makes it a concatenation expression interpretation*
- *an instantiation of a converter operator, which can also be interpreted as its operand expression, which, by induction, must be either a basic or a concatenation expression interpretation.*

Because there do not exist other operators, thus, all mixfix expressions have either a basic expression interpretation or a concatenation expression interpretation.

Left-Weighted Expression Interpretations

Using this further characterization of expressions, we can now introduce the concept of left-weighted expression interpretations formally.

Definition 53 *A* left-weighted expression interpretation *is either*

- *a* basic *expression interpretation where all operands are instantiated with left-weighted expression interpretations, or*
- *a* concatenation *expression interpretation where all but the leftmost operand must be instantiated with basic left-weighted expression interpretations, while the leftmost operand can be instantiated with any left-weighted expression interpretation.*

If the root operator of a left-weighted expression interpretation is right-open, its rightmost operand must be instantiated either with an empty or a prefix operator instantiation.

For left-weighted expression interpretations, we assume that in our mixfix expression language, all right-open operators can be seen as left-precedent towards all left-open operators (by allowing only prefix and empty operators as instantiations of right-open operands) with the exception of concatenation operators which are defined as left-precedent to each other and which all other operators are defined as both left and right-precedent to.

We need this interpretation to prove that the restrictions imposed on instantiations of adjacent operands lead to unambiguity in regard to these operands which is independent of the ambiguity issues induced by precedence and typing.

We can describe the language of left-weighted expressions also with a special context-free mixfix grammar. To arrive at this grammar, the usual grammar transformations for resolving operator precedence ambiguity must be applied to the normal context-free mixfix grammar[5].

Definition 54 *The* left-weighted mixfix grammar *is a context-free grammar where the set of rules is constructed as follows:*

- *take the rules of the meta-grammar*
- *add rule* **E** ::= **B**
- *add rule* **E** ::= **E B**
- *add rule* **B** ::= **P**
- *map every visible or empty operator pattern to a mixfix rule with the following properties:*
  - *all separator tokens are mapped to their respective terminal symbols inside the right-hand-side*
  - *if the operator is left-open, the left-hand side of the rule is* **B**
  - *if the operator is a prefix or empty operator, the left-hand side of the rule is* **P**
  - *map all right-open placeholders to nonterminal* **P** *inside the right-hand side*
  - *map all enclosed placeholders to nonterminal* **COMMA** *inside the right-hand side*
  - *map all left-open or adjacent placeholders to nonterminal* **B** *inside the right-hand side*

Example 66 *Take the signature of example 49 on page 64, the induced left-weighted mixfix grammar (apart from the meta-grammar) is the one in figure 5.4.*

Since we abstract from typing, all concatenation operators are described by the same rule **E** ::= **E B** to avoid ambiguity introduced by mixing concatenation operators of different arities. This can safely be done because every concatenation instantiation of a concatenation operator of arity greater than two can also be interpreted as multiple applications of a binary concatenation operator.

Now, we still have to make sure that the language described by the left-weighted expression interpretations include all unambiguous expressions of the mixfix expression language.

[5]see [3]

$\mathbf{E} ::= \mathbf{B}$
$\mathbf{E} ::= \mathbf{E}\ \mathbf{B}$
$\mathbf{B} ::= \mathbf{P}$
$\mathbf{P} ::= \underline{\texttt{if}}\ \mathbf{COMMA}\ \underline{\texttt{then}}\ \mathbf{COMMA}\ \underline{\texttt{else}}\ \mathbf{P}$
$\mathbf{B} ::= \mathbf{B}\ \underline{=}\ \mathbf{P}$
$\mathbf{P} ::= \underline{\texttt{0}}$
$\mathbf{P} ::= \underline{\texttt{1}}$
$\mathbf{B} ::= \mathbf{B}\ \underline{-}\ \mathbf{P}$
$\mathbf{B} ::= \mathbf{B}\ \underline{*}\ \mathbf{P}$
$\mathbf{B} ::= \mathbf{B}\ \underline{!}$
$\mathbf{P} ::= \underline{\texttt{x}}$
$\mathbf{P} ::= \underline{\texttt{fac}}\ \mathbf{P}$

Figure 5.4: Grammar of example 66

Proposition 4 *Every mixfix expression has a left-weighted interpretation.*

Proof 5

1. *Instantiations of nullary operators are trivially left-weighted.*

2. *By induction, instantiations of postfix operators have a left-weighted interpretation, if all their operands have a left-weighted interpretation.*

3. *By induction, instantiations of right-open operators where the rightmost operand is empty or a prefix operator instantiation, have a left-weighted interpretation, if all their operands have a left-weighted interpretation.*

4. *Suppose we have an expression* $\mathbf{s}_0\ \mathbf{E}_1\ \mathbf{s}_1 \ldots \mathbf{E_n}\ \mathbf{s_n}\ \mathbf{E_{n+1}}\ \mathbf{s_{n+1}} \ldots \mathbf{E_{n+m}}\ \mathbf{s_{n+m}}$ *where all* $\mathbf{E_i}$ *have left-weighted interpretations and where* $\mathbf{s}_0\ \_\ \mathbf{s}_1\ \ldots\ \_\ \mathbf{s_{n-1}}\ \_$ *is the pattern of a right-open operator and* $\_\ \mathbf{s_n}\ \_\ \mathbf{s_{n+1}} \ldots \_\ \mathbf{s_{n+m}}$ *is the pattern of a left-open operator.*

   This expression can be seen as an instantiation of operator $\_\ \mathbf{s_n}\ \_\ \mathbf{s_{n+1}} \ldots \_\ \mathbf{s_{n+m}}$. *Since by induction* $\mathbf{s}_0\ \mathbf{E}_1\ \mathbf{s}_1 \ldots \mathbf{E_n}$ *has a left-weighted interpretation,* $\mathbf{s}_0\ \mathbf{E}_1\ \mathbf{s}_1 \ldots \mathbf{E_n}\ \mathbf{s_n}\ \mathbf{E_{n+1}}\ \mathbf{s_{n+1}} \ldots \mathbf{E_{n+m}}\ \mathbf{s_{n+m}}$ *also has a left-weighted interpretation where either* $\_\ \mathbf{s_n}\ \_\ \mathbf{s_{n+1}} \ldots \_\ \mathbf{s_{n+m}}$ *is the topmost instantiated operator or* $\mathbf{s_{n+m}} = \epsilon$ *and the expression* $\mathbf{E_{n+m}}$ *is also of the form* $\mathbf{E}'_{\mathbf{n+m}}\ \mathbf{s_{n+m+1}} \ldots \mathbf{E}'_{\mathbf{n+m+k}}\ \mathbf{s_{n+m+k}}$, *which is the same case as above.*

Prefix and Postfix Expressions

We need an additional characterization of non-empty expressions into prefix and postfix expressions to describe, what kind of expressions can be adjacent to each other without ambiguity.

Definition 55 *A* prefix expression *is an operator instantiation of either a prefix operator or of a left-open operator where the leftmost operand is instantiated with a prefix expression.*

Basically, this characterization ensures that such an expression does not have an empty operator instantiation at the very beginning.

Definition 56 *A* postfix expression *is an operator instantiation of either a postfix operator or of a right-open operator where the rightmost operand is instantiated with a postfix expression.*

Analogously, this characterization ensures that such an expression does not have an empty operator instantiation at the very end.

The rest of this section will describe specific causes for ambiguities caused by adjacent operands and introduce restrictions that must be hence-with imposed on instantiations of such operands.

Following that, we will use these restrictions to prove the unambiguity of the language of left-weighted interpretations.

5.5.2 Adjacent Empty Operands

Obviously, the instantiation of an empty operator which just constitutes the empty token sequence has only one left-weighted interpretation, since it is a nullary operator. So, for every two adjacent operands that are instantiated with one empty and one non-empty expression, it could be syntactically ambiguous whether the non-empty expression instantiates the left or the right operand. In both cases, we could still get left-weighted interpretations for the expression.

Example 67 *In the signature in figure 5.5, the expression* **a b c** *as an instantiation of the operator* **a _ _ c** *is ambiguous as it is not clear which of the operands is instantiated with* **b***.*

On the other hand, the expression **a c** *is unambiguous.*

```
FUN  A      : SORT
FUN  B      : SORT
FUN  a _ _ c : [A , A] →  B
FUN  b      : A
FUN  ϵ      : A
FUN  E_1    : B
DEF  E_1 ==  a b c
               -- a b ϵ c
               -- a ϵ b c
FUN  E_2    : B
DEF  E_2 ==  a c
               -- a ϵ ϵ c
```

Figure 5.5: Signature of example 67

This leads us to our first restriction regarding adjacent operands:

Restriction 1 *In an unambiguous expression, one operand that is adjacent to another operand can be instantiated with an empty expression if and only if the adjacent operand is also instantiated with an empty expression.*

5.5.3 Adjacent Concatenation Operands

A similar problem arises for concatenation operators , i.e. expressions that have a syntactic interpretation as the instantiation of an invisible operator of arity greater than 1.

A concatenation expression is one which can always be partitioned into more than one expression. Thus, for adjacent operands, if one of them is instantiated with a concatenation expression, part of that concatenation could also belong to a concatenation of the other adjacent operand.

Example 68 *In the signature in figure 5.6, the expression* **a b b c** *as an instantiation of the operator* **a** _ _ **c** *is ambiguous as it is not clear how the token list* **b b** *is to be distributed between the adjacent operands.*

```
FUN A       : SORT
FUN B       : SORT
FUN _ _     : [A , A] → A
FUN a _ _ c : [A , A] → B
FUN b       : A
FUN ϵ       : A
FUN E1      : B
DEF E1 ==  a b b c
               -- a (b) (b) c
               -- a (b b) ϵ c
               -- a ϵ (b b) c
```

Figure 5.6: Signature of example 68

This leads us to our second restriction regarding adjacent operands:

Restriction 2 *In an unambiguous expression, no operand that is adjacent to another operand (except the leftmost operand of a concatenation operator) can be instantiated with a concatenation expression.*

In concatenation operators, all operands are adjacent to each other. If we would not allow concatenation operands in any of these, it would not be possible to have nested concatenation expressions. Since relaxing the overall restriction in that instance does not introduce ambiguity, we do it to be able to define the application operator as a concatenation operator with the usual left-associativity.

Of course, we could relax the condition in such a way that the leftmost of any list of adjacent operands can contain a concatenation expression, but we think that would be more confusing than beneficial. Whatever restriction is chosen, it must ensure that it is clear which part of which expression belongs to which adjacent operand.

5.5.4 Adjacent Postfix and Prefix Operands

Empty operators can cause another ambiguity in conjunction with left-open or right-open operators.

If an operand that is left-adjacent to another operand is instantiated with a non-postfix expression, and the right-adjacent operand is instantiated with a left-open operator instantiation, the leftmost operand of that expression could possibly also be seen as the rightmost operand of a rightmost sub-expression of the left-adjacent operand expression.

If an operand that is left-adjacent to another operand is instantiated with a non-postfix expression which is a right-open operator instantiation where the rightmost operand is again a right-open operator instantiation, it is unclear, where the left-adjacent operand expression ends and the right-adjacent operand expression starts.

Example 69 *In the signature in figure 5.7, the expression*
begin pre closed post end *is ambiguous as it is not clear whether* **closed** *is the right operand of* **pre** _ *or the left operand of* _ **post**.

Likewise, the expression **begin pre pre closed end** *still is ambiguous, even when we take restriction 1 into account.*

The expression **begin pre closed end** *would be unambiguous, though.*

```
FUN  begin _ _ end : [A , A] →  A
FUN  pre _         : [A] →  A
FUN  _ post        : [A] →  A
FUN  closed        :  A
FUN  ε             :  A

FUN  E1            :  A
DEF  E1 ==           begin pre closed post end
                        -- begin (pre ε) (closed post) end
                        -- begin (pre closed) (ε post) end

FUN  E2            :  A
DEF  E2 ==           begin pre pre closed end
                        -- begin (pre ε) (pre closed) end
                        -- begin (pre pre ε) closed end
FUN  E3            :  A
DEF  E3 ==           begin pre closed end
                        -- begin (pre ε) closed end
```

Figure 5.7: Signature of example 69

These reasonings lead us to our third restriction regarding adjacent operands:

Restriction 3 *In an unambiguous expression, if two adjacent operands are instantiated with non-empty expressions, the left operand must be instantiated with a postfix expression and the right operand must be instantiated with a prefix expression.*

Closed Expressions as Adjacent Operands

Definition 57 *A* closed expression *is an operator instantiation of a closed operator, i.e. an operator that is both prefix and postfix .*

As can be seen in example 69, there is a special case where a non-postfix expression as left-adjacent operand is not syntactically ambiguous (if restrictions 1 and 2 are adhered to).

When the left operand is the instantiation of a right-open operator with the empty expression as its rightmost operand, there is no ambiguity towards the right-adjacent operand if that is instantiated with a *closed* operator instantiation.

The analogous case for non-prefix expressions as right-adjacent operands exists, of course, as well.

However, we are not sure whether or not relaxing our restriction 3 towards these unambiguous expressions would not be too confusing for the user of our language and thus have refrained from doing so.

5.5.5 Empty Operands in Concatenations

One more special case deserves our attention. If we were to allow adjacent operands in concatenation operators to be instantiated with empty expressions (which because of restriction 1 would mean that all operands or none would have to be instantiated with empty expressions, we arrive at infinitely many left-weighted interpretations for the empty expression.

Therefore, we introduce our fourth restriction regarding adjacent operands:

Restriction 4 *In unambiguous expressions, operands of concatenation operators shall not be instantiated with empty expressions.*

5.5.6 Unambiguity

With the help of these restrictions, we can now prove that adjacent operands do not cause ambiguity in left-weighted interpretations.

Lemma 4 *If the restrictions regarding adjacent operands are adhered to, every expression has at most one partition e_1e_2 so that e_1 and e_2 are not empty, have exactly one left-weighted interpretation where e_1 is a postfix expression while e_2 is a basic prefix expression.*

Proof 6 *Let us assume, there exists another partition $e'_1e'_2$ for expression e_1e_2 with $e'_1 = e_1v$ and $e_2 = ve'_2$ (with $v \neq \epsilon$) with the same restrictions as those for e_1 and e_2.*

Because there is no backbone ambiguity, v must be a mixfix expression, too. This expression either excludes the root of the left-weighted interpretation of e_2 (i.e. is part of the left subtrees) or includes it (i.e. e'_2 is part of the right subtrees).

- *If the root of e'_2 is the same as the root of e_2, one of its left subtrees now misses at least one leftmost operand, which would make e'_2 a non-prefix expression, contradicting that e'_2 is supposed to be a prefix expression.*
- *If the root of e'_2 is different from the root of e_2, then e'_2 must be a rightmost operand of a rightmost subtree of e_2 to be a basic prefix expression.*

  But in that case v now misses at least one rightmost operand. Since e_1 is a postfix expression, all its operators have all their rightmost operands instantiated with non-empty expressions. Also, in v, all operators have their leftmost operands instantiated with non-empty expressions, because e_2 was a prefix expression and there is no other left-weight interpretation possible for v than that for the left part of e_2.

  Additionally, since e_1 was a postfix expression, v must also be a postfix expression, i.e. all its rightmost operators are not empty. Therefore, ve'_2 must be a concatenation expression which contradicts e_2 being a basic expression.

Proposition 5 *The left-weighted interpretations of expressions in which no backbone ambiguity occurs are syntactically unambiguous, if the restriction properties 1, 2, 3 and 4 are met for all expressions.*

Proof 7 *Because of lemma 4, there is at most one partition into basic expressions possible for every non-empty concatenation of expressions.*

Thus, for every list of adjacent operands, the unambiguous partition of the concatenation expression instantiating these operands can be found and then each of the resulting sub-expressions disambiguated separately using precedence. Since we allow only basic expressions as instantiations for adjacent operands, we only accept expressions where the operation-concatenation can be partitioned into exactly the same number of basic expressions as there are adjacent operands.

We still must distinguish two cases for each adjacent operand list:

1. *If the list of adjacent operands occurs in front of a terminal symbol in the instantiated operator, the rest of the operator cannot overlap the operands, therefore it can be unambiguously determined, whether or not all operands are instantiated with the empty expression.*

2. *If the list of adjacent operands is not in front of a terminal symbol of the instantiated operator, then the whole expression either ends after these operands (in which case we can determine whether they must all be empty or not) or there is an outer operator which had this operator as an operand. Again, we have to distinguish the two different cases for the operand list containing* that *operand.*

If the root of the left-weighted interpretation of an expression is a concatenation operator, it can still be unambiguously partitioned into the rightmost operands (which are all basic expressions) and the leftmost operand. If the leftmost operand is again a concatenation expression, it can be handled the same way.

Therefore, all expressions containing operators with adjacent operands have at most one left-weighted interpretation adhering to the restrictions governing adjacent operands.

5.6 Left-Open vs. Right-Open Operators

In the section about shared separator tokens, we only explored expression overlap caused by overlapping *separators*. However, it is also possible that expressions only overlap in their *operand parts* when left-open operator instantiations are mixed with right-open operator instantiations to their right in one mixfix expression.

The restrictions in the previous section about adjacent operands ensure that such operand overlap can only occur between the left-open operand of one operator and the right-open operand of an operator.

But even with these restrictions expressions can have different syntactic interpretations. To disambiguate such expressions, a unique topological order between the backbones of the operators of that expression must be found which orders the operators into a parse tree.

Precedence relations between operators can help us find such a topological order on operators. Thus, we want to develop some precedence relations in this section that allow no syntactic ambiguity inside mixfix expressions, as long as no backbone ambiguity occurs and the adjacent operand restrictions are adhered to.

The two kinds of precedence we want to describe we will call *natural* precedence and *ad-hoc* precedence [6].

The first of these precedence kinds determines precedence between operators depending on their types, while the second precedence uses precedence *preferences* given by the user.

Both kinds of precedence take the fixities of the operators and the presence of other operators in the mixfix expression into account to determine unambiguity. We only must consider left-precedence towards left-open operators and right-precedence towards right-open operators. During the actual precedence test, it is also only necessary to test precedence between operator pairs where the precedent operator can be an operand to the dominant one according to typing.

[6]which could also be called *unnatural* precedence, because it does not follow from the *nature* of the operators involved

5.6.1 Natural Precedence

As was already mentioned, naturally syntactically unambiguous expressions have at most one syntactic interpretation in every type. But this can lead to situations where, dependent on the type of the expression, operator pairs can be naturally right-precedent and/or naturally left-precedent for the same expression (in *different* types).

Example 70 *Under three different types, the operator pair* (# _ , _ :: _) *in the signature 5.8 can be naturally left-precedent, right-precedent or both. In the first two cases, the resulting expression is unambiguous, in the latter, it is ambiguous.*

```
FUN <>       : seq A
FUN _ :: _   : [A , seq A] → seq A
FUN # _      : [seq A] → nat

FUN E1       : nat
DEF E1 == # <> :: <>
              -- #(<> :: <>)
FUN E2       : seq A
DEF E2 == # <> :: <>
              -- ((#<>) :: <>) : seq nat
FUN E3       : A
DEF E3 == # <> :: <>
              -- (#(<> :: <>)) : nat
              -- ((#<>) :: <>) : seq nat
```

Figure 5.8: Signature of example 70

Weak Natural Precedence Relations

To be able to describe situations where this natural ambiguity cannot occur, we define the natural, type-dependent, left- and right-precedence relations $\mathbf{Left}_T(S)$ and $\mathbf{Right}_T(S)$[7] in definition 58.

Now, these relations describe for all operator pattern pairs (op_1, op_2), whether op_1 *may be* the root of the leftmost left-open operand of op_2 (i.e. $(op_1, op_2) \in \mathbf{Left}_T(S)$) or whether op_2 may be the root of the rightmost right-open operand of op_1 according to scope S by simple type compatibility (i.e. if the types are unifiable).

If the result type of the left operator is compatible with the leftmost left-open type of the right operator, then there could be expressions where the right operator *dominates* the left operator directly.

Since this relation only describes the *possibility* of precedence between operators which could be ambiguous, we call it the *weak natural precedence relation.*

If $(op_1, op_2) \in \mathbf{Left}_T(S)$ and $(op_1, op_2) \in \mathbf{Right}_T(S)$, there is obviously potential for ambiguity. But disjointness of $\mathbf{Left}_T(S)$ and $\mathbf{Right}_T(S)$ is a too strong criterion to avoid this ambiguity as too many naturally unambiguous expressions would then be rejected if both precedences were excluded in such a case from the grammar. Also, this criterion would only imply unambiguity, if both $\mathbf{Left}_T(S)$ and $\mathbf{Right}_T(S)$ are transitive which is seldom the case.

[7]The subscript T here stands for *type-dependent*, while the operand S is a signature.

But the weak natural precedence relations will help us in describing the strong natural precedence relations we aim for.

Definition 58 *We define* $\mathbf{Left}_T(S)$ *as those pairs of operator patterns from signature* S*, where the type of leftmost operand of the dominant left-open operator can be returned by the possibly precedent operator. The analogous construction determines the relation* $\mathbf{Right}_T(S)$*.*

```
    TYPE  pattern == seq[token]
    TYPE  operator == (_ : _)(pattern : pattern , type : type)
    TYPE  scope == set[operator]


  VAR  S : scope
  VAR  $p_1$ , $p_2$ : pattern
    FUN  $\mathbf{Left_T(S)} : \mathbb{P}(\mathbf{Op(S)} \times \mathbf{LeftOpen(S)})$
    FUN  $\mathbf{Right_T(S)} : \mathbb{P}(\mathbf{RightOpen(S)} \times \mathbf{Op(S)})$
    DEF  $\mathbf{Left_T(S)}$=={($p_1$ , $p_2$)|$p_1$ returns$_S$ leftmost$_S$ ($p_2$)}
    DEF  $\mathbf{Right_T(S)}$=={($p_1$ , $p_2$)|$p_2$ returns$_S$ rightmost$_S$ ($p_1$)}


  VAR  op : operator
  VAR  p : pattern
  VAR  T , $T_0$ , $T_1$ , $T_2$ , $T_3$ ,..., $T_n$ : type
    FUN  _ compatible _ : [type , type] → bool
    DEF  $T_1$ compatible ($T_2 \cup T_3$)==
           ($T_1$ compatible $T_2$)or($T_1$ compatible $T_3$)
    DEF  ($T_1 \cup T_2$) compatible $T_3$==
           ($T_1$ compatible $T_3$)or($T_2$ compatible $T_3$)
    DEF  $T_1$ compatible $T_2$==$T[T1 := T_2] \neq \bot$
           WHERE T==copy ($T_1$ , $T_2$)
    FUN  _ returns_ _ : [pattern , scope , type] → bool
    DEF  p returns$_S$ T==
           IF (p : $T_1 \rightarrow T_2$) ∈ S and
               ($T_2$ compatible T)
           THEN true
           IF (T = LIFT $T_0$) and
               (separators(p) = [$s_0$ , . . . , $s_n$]) and
               (p : $T_1 \rightarrow T_2$) ∈ S and
               ($T_0$ compatible $s_0$ $T_1$ $s_1$ . . . $T_n$ $s_n$)
           THEN true
           ELSE false
           FI
    FUN  leftmost_ : [scope] → pattern → type
    DEF  leftmost$_S$ (p)==
           $\bigcup\{ T_1 | (p : [T_1 , \ldots , T_n] \rightarrow T_0) \in S\}$
    FUN  rightmost_ : [scope] → pattern → type
    DEF  rightmost$_S$ (p)==
           $\bigcup\{ T_n | (p : [T_1 , \ldots , T_n] \rightarrow T_0) \in S\}$
```

The following definitions of `leftmost`$_S$ **(op , T)** and `rightmost`$_S$ **(op , T)** are used in definitions 67 (on page 96) and 66 (on page 96) . They compute the leftmost or rightmost operand type for operator **op**, if the demanded type for the instantiation of that operator is **T**.

Definition 59

```
FUN leftmost_ : [scope] → pattern × type → type
DEF leftmost_S (p , T_1 ∪ T_2)==
    leftmost_S (p , T_1) ∪ leftmost_S (p , T_2)
DEF leftmost_S (p , T)==
    ⋃{ T_1[T0 := T]|
        op ∈ S ,
         pattern(op) = p ,
        [T_1 , . . . , T_n] → T_0 = copy(type(op) , T) }
FUN rightmost_ : [scope] → pattern × type → type
DEF rightmost_S (p , T_1 ∪ T_2)==
    rightmost_S (p , T_1) ∪ rightmost_S (p , T_2)
DEF rightmost_S (p , T)==
    ⋃{ T_n[T0 := T]|
        op ∈ S ,
         pattern(op) = p ,
        [T_1 , . . . , T_n] → T_0 = copy(type(op) , T) }
```

For the above definitions, we need an auxiliary function `copy`. The function `copy` creates a fresh copy of its first argument, not containing variables from its first or second argument. It is used for compatibility checks. This copying is necessary since the variables used in types of declarations can be the same, although all these variables are free for every declaration.

The auxiliary function `Free` computes all free type variable of a type.

The function `rename` computes a substitution function for types function which renames all variables in its first arguments injectively to variables which are not in its first and not in its second argument.

Definition 60

```
FUN copy : type × type → type
DEF copy (T_1 , T_2)== rename ( Free (T_1) , Free (T_2))(T_1)
FUN Free : type → set[type]
FUN rename : set[type] × set[type] → (type → type)
```

Natural Precedence Ambiguity

Let us now look at the different scenarios where natural syntactic ambiguities in operator expressions can occur.

Basically, this is the case if at least two operators could be the root operator of the same operator expression with the same type.

Let τ be an arbitrary given type, $\mathbf{op}_1$ a right-open operator and $\mathbf{op}_2$ a left-open operator in $\mathbf{Op}$ with $\mathbf{op}_1 = \mathbf{s}_0\ \_\ \dots\ \mathbf{s}_{\mathbf{n}-1}\ \_$, $\mathbf{op}_2 = \_\ \mathbf{s}_{\mathbf{n}+2}\ \dots\ \_\ \mathbf{s}_{\mathbf{m}}$.

Suppose further, we have an operator expression
$\mathbf{e} = \{\mathbf{s_i}\ \mathbf{e_i}\}_{\mathbf{i}=0}^{\mathbf{n}-2}\ \mathbf{s}_{\mathbf{n}-1}\ \mathbf{v}\ \mathbf{s}_{\mathbf{n}+2}\ \{\mathbf{e}_{\mathbf{i}-1}\ \mathbf{s_i}\}_{\mathbf{i}=\mathbf{n}+3}^{\mathbf{m}}$ with $\mathbf{e} \in \mathcal{L}(\Sigma)$, $\mathbf{e_i} \in \mathcal{L}(\Sigma)$, and both $\mathbf{e_l} = \{\mathbf{s_i}\ \mathbf{e_i}\}_{\mathbf{i}=0}^{\mathbf{n}-2}\ \mathbf{s}_{\mathbf{n}-1}\ \mathbf{v} \in \mathcal{L}(\Sigma)$ and
$\mathbf{e_r} = \mathbf{v}\ \mathbf{s}_{\mathbf{n}+2}\ \{\mathbf{e}_{\mathbf{i}-1}\ \mathbf{s_i}\}_{\mathbf{i}=\mathbf{n}+3}^{\mathbf{m}} \in \mathcal{L}(\Sigma)$ are also operator expressions.

Obviously, $\mathbf{e}$ could be ambiguous for a type τ, if both $\mathbf{op}_1$ and $\mathbf{op}_2$ can return something of type τ and $\mathbf{e_r}$ has at least one type correct interpretation for the type of the rightmost operand of $\mathbf{op}_1$ and also $\mathbf{e_l}$ has at least one semantically correct interpretation of the type of the leftmost operand of $\mathbf{op}_2$.

Now, let us have a look at the different scenarios where it is possible for $\mathbf{e_r}$ to have a syntactically and semantically correct interpretation for the type of the rightmost operand of $\mathbf{op}_1$. The analogous scenarios are possibilities for $\mathbf{e_l}$.

1. $\mathbf{v}$ is also an expression in $\mathcal{L}(\Sigma)$ and $\mathbf{op}_2$ may be the root operator of $\mathbf{e_r}$.
2. $\mathbf{op}_2$ is a postfix operator (i.e. $\mathbf{m} > \mathbf{n} \wedge \ \mathbf{s_m} \neq \epsilon$). Then, there must be an operator $\mathbf{op}_3 \in \mathbf{Op}$ which can be the root operator of $\mathbf{e_r}$ and $\mathbf{op}_2$ can be the root of a rightmost subtree of $\mathbf{op}_3$.
3. $\mathbf{op}_2$ is also a right-open operator (i.e. an infix operator with $\mathbf{m} = \mathbf{n} \vee \mathbf{s_m} = \epsilon$). Then, there must be an operator $\mathbf{op}_3 \in \mathbf{Op}$ which may be the root operator of $\mathbf{e_r}$ and $\mathbf{op}_2$ can be the root of any subtree of $\mathbf{op}_3$.

This leads us to the following definitions that will help us formalize the above conditions. They characterize different relations between operator, namely whether an operator can appear as a direct or indirect rightmost or leftmost child of another right- or left-open operator or can be a descendant reachable only over leftmost left-open or rightmost right-open operands.

They are used to describe possibly precedent situations between operators which again helps describing conflicts between such situations.

Definition 61 *An operator op_2 may appear as the* rightmost child *of a right-open root operator op_1 in an expression of type τ according to the right-precedence relation* **Right** *if* $(op_1, op_2) \in \texttt{DirectlyRight}_\tau(S, \mathbf{Right})$.

$$\texttt{DirectlyRight}_\tau(S, \mathbf{Right}) == \\ \mathbf{Right} \cap \{(op_1, op_2) \mid op_2\, \texttt{returns}_S\ \texttt{rightmost}_S(op_1, \tau)\}$$

Definition 62 *An operator op_1 may appear as the* leftmost child *of a left-open root operator op_2 in an expression of type τ according to the left-precedence relation* **Left** *if* $(op_1, op_2) \in \texttt{DirectlyLeft}_\tau(S, \mathbf{Left})$.

$$\texttt{DirectlyLeft}_\tau(S, \mathbf{Left}) == \\ \mathbf{Left} \cap \{(op_1, op_2) \mid op_1\, \texttt{returns}_S\ \texttt{leftmost}_S(op_2, \tau)\}$$

Definition 63 *The postfix operator op_2 may appear as a* rightmost descendant *of the right-open root operator op_1 in an expression of type τ according to the right-precedence* **Right**, *if* $(op_1, op_2) \in \texttt{RightmostDescendant}_\tau(S, \mathbf{Right})$.

```
$\texttt{RightmostDescendant}_\tau(S, \mathbf{Right}) ==$
  $\mathbf{RightOpen}(S) \times \mathbf{PostfixOp}(S) \cap$
  $\{ \ (op_1, op_2) \mid$
    $\exists \ op_3 :$
      $(op_1, op_3) \in \texttt{DirectlyRight}_\tau(S, \mathbf{Right}) \wedge$
      $(op_3, op_2) \in \mathbf{Right}^\star$
  $\}$
```

Definition 64 *The prefix operator op_1 may appear as a* leftmost descendant *of the left-open root operator op_2 in an expression of type τ according to the left-precedence relation* **Left**, *if* $(op_1, op_2) \in \texttt{LeftmostDescendant}_\tau(S, \mathbf{Left})$.

```
$\texttt{LeftmostDescendant}_\tau(S, \mathbf{Left}) ==$
  $\mathbf{PrefixOp}(S) \times \mathbf{LeftOpen}(S) \cap$
  $\{\ (op_1, op_2) \mid$
    $\exists\ op_3 :$
      $(op_3, op_2) \in \texttt{DirectlyLeft}_\tau(S, \mathbf{Left}) \wedge$
      $(op_1, op_3) \in \mathbf{Left}^\star$
  $\}$
```

Definition 65 *The infix operator op_1 may appear as a* descendant *of the left or right-open root operator op_2 in an expression of type τ according to the pair of precedence-relations* $(\mathbf{Left}, \mathbf{Right})$*, if* $(op_1, op_2) \in \texttt{Descendant}_\tau(S, \mathbf{Left}, \mathbf{Right})$*.*

```
$\texttt{Descendant}_\tau(S, \mathbf{Left}, \mathbf{Right}) ==$
  $\mathbf{InfixOp}(S) \times (\mathbf{LeftOpen}(S) \cup \mathbf{RightOpen}(S)) \cap$
  $\{\ (op_1, op_2) \mid$
    $\exists\ op_3 :$
      $op_2 \in \mathbf{LeftOpen}(S)\ \wedge$
      $(op_3, op_2) \in \texttt{DirectlyLeft}_\tau(S, \mathbf{Left}) \wedge$
      $(op_1, op_3) \in (\mathbf{Left} \cup \mathbf{Right}^{-1})^\star$
      $\vee$
      $op_2 \in \mathbf{RightOpen}(S)\ \wedge$
      $(op_2, op_3) \in \texttt{DirectlyRight}_\tau(S, \mathbf{Right}) \wedge$
      $(op_1, op_3) \in (\mathbf{Left} \cup \mathbf{Right}^{-1})^\star$
  $\}$
```

Now, we are able to define the possibly ambiguous situations between left-open operators to the right of right-open operators (as described above) formally:

Definition 66 *If* $(op_1, op_2, op_3) \in \texttt{PossiblyRightPrecedent}_\tau(S, \mathbf{Left}, \mathbf{Right})$*, then op_2 may be the right child of the root op_1 in expressions of type τ (according to the pair of precedence-relations* $(\mathbf{Left}, \mathbf{Right})$*) where op_3 might both be in the the right subtree of op_2 or the root operator of the whole expression.*

We call this a possibly right-precedent situation .

```
$\texttt{PossiblyRightPrecedent}_\tau(S, \mathbf{Left}, \mathbf{Right}) ==$
  $\mathbf{RightOpen}(S) \times \mathbf{Op}(S) \times \mathbf{LeftOpen}(S) \cap$
  $\{\ (op_1, op_2, op_3) \mid$
    $(op_1, op_2) \in \texttt{DirectlyRight}_\tau(S, \mathbf{Right})\ \wedge$
    $op_3\ \texttt{returns}_S\ \tau\ \wedge$
    $($
      $op_3 = op_2\ \vee$
      $(op_2, op_3) \in \texttt{RightmostDescendant}_{\texttt{rightmost}_S(op_1, \tau)}(S, \mathbf{Right})\ \vee$
      $(op_3, op_2) \in \texttt{Descendant}_{\texttt{rightmost}_S(op_1, \tau)}(S, \mathbf{Left}, \mathbf{Right})$
    $)$
  $\}$
```

Definition 67 *If* $(op_1, op_2, op_3) \in \texttt{PossiblyLeftPrecedent}_\tau(S, \mathbf{Left}, \mathbf{Right})$*, then op_2 may be the left child of the root op_3 in expressions of type τ (according to the pair of precedence-relations* $(\mathbf{Left}, \mathbf{Right})$*) where op_1 might both be in the the left subtree of op_2 or the root operator of the whole expression.*

We call this a possibly left-precedent situation .

```
$\texttt{PossiblyLeftPrecedent}_\tau(S, \mathbf{Left}, \mathbf{Right}) ==$
  $\mathbf{RightOpen}(S) \times \mathbf{Op}(S) \times \mathbf{LeftOpen}(S) \cap$
  $\{\ (op_1, op_2, op_3)\ |$
    $(op_2, op_3) \in \texttt{DirectlyLeft}_\tau(S, \mathbf{Left}) \wedge$
    $op_1\ \texttt{returns}_S\ \tau \wedge$
    $($
      $op_1 = op_2 \vee$
      $(op_1, op_2) \in \texttt{LeftmostDescendant}_{\texttt{leftmost}_S(op_3,\tau)}(S, \mathbf{Left}) \vee$
      $(op_1, op_2) \in \texttt{Descendant}_{\texttt{leftmost}_S(op_3,\tau)}(S, \mathbf{Left}, \mathbf{Right})$
    $)$
  $\}$
```

Naturally Unambiguous Precedence

With the help of these relations, we now are able to define two *strong* natural precedence relations $\mathbf{LeftPrec}_\tau$ and $\mathbf{RightPrec}_\tau$ according to the pair of precedence-relations $(\mathbf{Left}_T(S), \mathbf{Right}_T(S))$ which are unambiguous.

Definition 68

```
$\mathbf{RightPrec}_\tau(S, \mathbf{Left}, \mathbf{Right}) ==$
  $\texttt{DirectlyRight}_\tau(S, \mathbf{Right}) \cap$
  $\{\ (op_1, op_2)\ |$
    $\forall\ op_3, op_4 :$
      $(op_1, op_2, op_3) \in \texttt{PossiblyRightPrecedent}_\tau(S, \mathbf{Left}, \mathbf{Right}) \implies$
      $(op_1, op_4, op_3) \notin \texttt{PossiblyLeftPrecedent}_\tau(S, \mathbf{Left}, \mathbf{Right})$
  $\}$

$\mathbf{LeftPrec}_\tau(S, \mathbf{Left}, \mathbf{Right}) ==$
  $\texttt{DirectlyLeft}_\tau(S, \mathbf{Left}) \cap$
  $\{\ (op_1, op_2)\ |$
    $\forall\ op_3, op_4 :$
      $(op_3, op_1, op_2) \in \texttt{PossiblyLeftPrecedent}_\tau(S, \mathbf{Left}, \mathbf{Right}) \implies$
      $(op_3, op_4, op_2) \notin \texttt{PossiblyRightPrecedent}_\tau(S, \mathbf{Left}, \mathbf{Right})$
  $\}$
```

If $(op_1, op_2) \in \mathbf{LeftPrec}_\tau(S, \mathbf{Left}_T(S), \mathbf{Right}_T(S))$, we can safely – without fear of loss of ambiguity – allow op_1 as the leftmost child of the root operator op_2 of an expression of type τ.

Likewise, if $(op_1, op_2) \in \mathbf{RightPrec}_\tau(S, \mathbf{Left}_T(S), \mathbf{Right}_T(S))$, we can safely allow op_2 as the rightmost child of the root operator op_1 of an expression of type τ.

This pair of precedence relations is used in constraints of our two level grammar in section 6.2.2.

Efficiency

The most inefficient part of computing these strong precedence relations is computing the transitive closures of the given weak precedence relations which is a cubic algorithm in its worst case.

It depends on the number of declared operators occurring in the backbone list of the expression multiplied by the number of declared converter operators in the scope of the expression.

But again, the number of *different* operators occurring in an expression is usually very small, so that the computation remains efficient in practice.

If backbone parsing is actually used to determine the exact set of the operators involved, this could very well be the dominant effort to be taken for long expressions since it depends on the length of the expression instead of the size of the operator set.

Restrictiveness

The set of expressions allowed by these natural precedence relations is a subset of the syntactically unambiguous, type-correct expressions.

However, while the natural precedence approach finds a lot of precedent situations that are usually ignored in most programming languages, it does not help in most of the standard operator scenarios like the arithmetic operators, since they are naturally ambiguous precedence-wise. Here, we will have to use ad-hoc precedences to resolve the syntactic ambiguity inherent in these operators.

5.6.2 Ad-Hoc Precedence

We want to allow the user to give *preferred* precedences for operator pairs that can have different *natural* precedences in different contexts, so that the parser can choose this preferred precedence whenever it finds a possible natural ambiguity.

In this section, we want to explore the restrictions that have to be imposed on such user-given ad-hoc precedence relations so that they agree with the natural precedence relations in such a way that no syntactic ambiguity is introduced into the language.

In general, the set of trees should be only enlarged by given ad-hoc precedence relationships, so that all expressions that were accepted as unambiguous without them are still accepted and some additional expressions (that are only ambiguous in regard to the given ad-hoc precedence) are accepted, as well.

Conflicting Operators

First of all, we have to define what possibly conflicting operator pairs are and how they can be found.

Definition 69 *The* possibly conflicting operator pairs *that can cause ambiguity are those which can occur both in a possibly left precedent situation and a possibly right precedent situation according to a given precedence-relation pair* $(\mathbf{Left}, \mathbf{Right})$.

```
$\texttt{Conflicting}_\tau(\mathbf{Left}, \mathbf{Right}) ==$
  $\mathbf{Left} \times \mathbf{Right} \cap$
  $\{\ ((op'_1, op_2), (op_1, op'_2))\ |$
     $(op_1, op'_1, op_2) \in \texttt{PossiblyLeftPrecedent}_\tau(\mathbf{Left}, \mathbf{Right})\ \wedge$
     $(op_1, op'_2, op_2) \in \texttt{PossiblyRightPrecedent}_\tau(\mathbf{Left}, \mathbf{Right})$
  $\}$
```

Thus, according to the weak precedence relations $\mathbf{Left}_T(S)$ and $\mathbf{Right}_T(S)$, an operator pair $(op'_1, op_2) \in \mathbf{Left}_T(S)$ is in conflict with the operator pair $(op_1, op'_2) \in \mathbf{Right}_T(S)$ when there is a possibly left precedent situation between op_1, op'_1, and op_2 (i.e. $(op_1, op'_1, op_2) \in \texttt{PossiblyLeftPrecedent}_\tau(\mathbf{Left}_T(S), \mathbf{Right}_T(S))$) and also a possibly right precedent situation between op_1, op'_2, and op_2, (i.e. $(op_1, op'_2, op_2) \in \texttt{PossiblyRightPrecedent}_\tau(\mathbf{Left}_T(S), \mathbf{Right}_T(S))$).

But this categorization does not help us much since for instance all operator pairs of the arithmetic operators are conflicting according to the natural weak precedence relations. Thus, we can not exclude these pairs from our ad-hoc precedence relations because there would be no gain in the parsable expressions.

What we definitely need is to find those operator pairs which are in conflict *according to the ad-hoc precedences*, because these would be the naturally conflicting ones *not* rejected by the parser. Thus, these definitely conflicting operator pairs can be excluded from the ad-hoc precedence relations by ensuring the following restriction:

Restriction 5

$$\mathbf{Left}_A \times \mathbf{Right}_A \ \cap \ \texttt{Conflicting}_\tau(\mathbf{Left}_A, \mathbf{Right}_A) = \emptyset$$

It is obvious that conflicts between such situations are real conflicts and must be forbidden. Unfortunately, it is not so easy to see whether generally conflicting operator pairs that are not in $\texttt{Conflicting}_\tau(\mathbf{Left}_A, \mathbf{Right}_A)$ introduce an actual ambiguity in an expression.

Example 71 *In the signature in figure 5.9, the expression* **a b c d** *could be seen as having two* preferred *interpretations because of the given ad-hoc precedence pairs, even though they are not definitely in conflict. If operator* **a _** *is chosen as the root, the rest expression is syntactically unambiguous because it contains no prefix operators. If, on the other hand* **_ d** *is chosen as the root, the rest expression is naturally unambiguous for type* $\mathbf{T}_1$*.*

```
FUN   a _     : [T2] → T2
FUN   _ d     : [T1] → T2
FUN   _ c     : [T2] → T1
FUN   b       : [T1] → T2
PREC  a (_ d)
PREC  (_ c) d
FUN   E       :  T2
DEF   E ==      a b c d
                   -- ((ab) c) d
                   -- a((b c) d)
```

Figure 5.9: Signature of example 71

Thus, we need to find some additional restrictions that prevent conflicts of natural precedence inside partial expressions of formerly ambiguous trees to conflict with the ad-hoc precedences.

Hierarchical Ad-Hoc Precedence Relations

Definition 70 *We call a pair of ad-hoc precedences* $(\mathbf{Left}_A, \mathbf{Right}_A)$ hierarchical*, if the following condition holds:*

Property 2

$$\mathbf{Left}_A{}^+ \ \cap \ \mathbf{Right}_A{}^+ = \emptyset$$

In a hierarchical pair of precedence relations, no pair of operators can both be in the left precedence hierarchy and the right precedence hierarchy.

Fully Hierarchical Ad-Hoc Precedence Relations

Definition 71 *We call these precedences* fully hierarchical *if they are hierarchical and also the following additional conditions hold:*

Property 3

$$
\begin{aligned}
&\forall op_1, op_2 \in (\mathbf{LeftOpen} \cup \mathbf{RightOpen}) : \\
&\qquad (op_1, op_2) \in (\mathbf{Left}_A \cup {\mathbf{Right}_A}^{-1})^+ \lor \\
&\qquad (op_2, op_1) \in (\mathbf{Left}_A \cup {\mathbf{Right}_A}^{-1})^+
\end{aligned}
$$

$$
\begin{aligned}
&\forall op_1, op_2 \in (\mathbf{LeftOpen} \cup \mathbf{RightOpen}) : \\
&\qquad (op_1, op_2) \in (\mathbf{Left}_A \cup {\mathbf{Right}_A}^{-1})^+ \land \\
&\qquad (op_2, op_1) \in (\mathbf{Left}_A \cup {\mathbf{Right}_A}^{-1})^+ \implies \\
&\qquad\qquad (op_1, op_2) \in {\mathbf{Left}_A}^+ \lor \\
&\qquad\qquad (op_2, op_1) \in {\mathbf{Right}_A}^+
\end{aligned}
$$

This means that for every pair of operators, one must be reachable by the other via the ad-hoc precedence relations, i.e. there must be a precedence defined (directly or indirectly) between every two operators. Also, if each operator can reach the other one, i.e. they have the same precedence level, then they should be reachable only using one of the left or the right ad-hoc precedence relations, meaning they should only be left- or right-associative.

We say that op_1 has a higher precedence level *than op_2, if $(op_1, op_2) \in (\mathbf{Left}_A \cup {\mathbf{Right}_A}^{-1})^+$ and $(op_2, op_1) \notin (\mathbf{Left}_A \cup {\mathbf{Right}_A}^{-1})^+$.*

We say that op_1 has the same precedence level *as op_2, if $(op_1, op_2) \in (\mathbf{Left}_A \cup {\mathbf{Right}_A}^{-1})^+$ and $(op_2, op_1) \in (\mathbf{Left}_A \cup {\mathbf{Right}_A}^{-1})^+$.*

Classical precedence level hierarchies which additionally assign either left- or right-*associativity* among the operators of equal precedence level are examples of fully hierarchical ad-hoc precedence relations.

Conflict-Freeness in Fully Hierarchical Ad-Hoc Precedence Relations

Using the above definitions, we can now proceed to prove that fully hierarchical ad-hoch precedence relation pairs cannot have conflicting operator pairs.

Lemma 5 *For fully hierarchical ad-hoc precedence relations, there can only be a definite conflict between the left-precedent operator pair (op_1, op_2) and the right-precedent operator pair (op_3, op_4), if op_2 and op_3 have the same precedence level.*

Proof 8 *If op_3 has a higher precedence level than op_2, then there cannot exist an op_4 so that $(op_3, op_4, op_2) \in \texttt{PossiblyRightPrecedent}_\tau(\mathbf{Left}_A, \mathbf{Right}_A)$. The analogous is true if only op_2 has a higher precedence level than op_3.*

Lemma 6 *For fully hierarchical ad-hoc precedence relations, all operators on the path between op_2 and op_3 in every possibly left-precedent situation (op_3, op_1, op_2) have the same precedence level, if op_2 and op_3 have the same precedence level.*

Proof 9 *If (op_3, op_1, op_2) is a possibly left-precedent situation, then op_3 can be a descendant of op_1 and op_1 can be a direct descendant of op_2. If op_2 and op_3 have the same precedence level, op_2 can also be a descendant of op_3. Therefore, op_1 can be a descendant of op_3 and also op_2 can be a descendant of op_1.*

The analogous is true for possibly right-precedent situations.

If $op_3 \neq op_1$, there must either be a possibly left-precedent situation (op_3, op_4, op_1) or a possibly right-precedent situation (op_1, op_4, op_3), if (op_3, op_1, op_2) is a possibly left-precedent situation.

It follows that (since there can be no infinite paths between op_2 and op_3) every operator on the path from op_2 to op_3 must have the same precedence level.

Lemma 7 *For fully hierarchical ad-hoc precedence relations, every possibly left-precedent situation* (op_3, op_1, op_2) *where* op_2 *and* op_3 *have the same precedence level involves only left-precedent operator pairs on the path from* op_2 *to* op_3.

The analogous is true for possibly right-precedent situations (op_3, op_4, op_2).

Proof 10 *Suppose there is a right-precedent operator pair* (op_4, op_5) *on the path from* op_2 *to* op_3. *Let us further assume that there are no right-precedent operator pairs between* op_2 *and* op_4, *so that* $(op_4, op_2) \in \mathbf{Left}_A{}^+$ *and* (op_4, op_5) *is the outermost right-precedent operator pair (i.e.* $(op_4, op_5) \in \mathbf{Right}_A$*).*

Because op_2 *and* op_3 *have the same precedence level,* op_4 *and* op_5 *also have the same precedence level as these.*

Now, because the precedence relations are fully hierarchical, there are two possibilities:

1. $(op_5, op_2) \in \mathbf{Right}_A{}^+$, *which implies that also* $(op_4, op_2) \in \mathbf{Right}_A{}^+$. *This contradicts the relations being hierarchical, because obviously* $(op_4, op_2) \in \mathbf{Left}_A{}^+$.

2. $(op_2, op_5) \in \mathbf{Left}_A{}^+$, *which implies that also* $(op_4, op_5) \in \mathbf{Left}_A{}^+$. *This also contradicts the relations being hierarchical, because also* $(op_4, op_5) \in \mathbf{Right}_A{}^+$.

Thus, there can be no such right-precedent operator pair which implies that all operator pairs involved must be left-precedent.

Proposition 6 *Fully hierarchical ad-hoc precedence relations have no definitely conflicting operator pairs.*

Proof 11 *If fully hierarchical ad-hoc precedence relations had definitely conflicting operator pairs* (op_1, op_2) *and* (op_3, op_4) *there should exist an expression which has a possibly left-precedent situation* (op_3, op_1, op_2) *and a possibly right-precedent situation* (op_3, op_4, op_2) *according to the ad-hoc precedence relations.*

Because of lemma 6, op_1, op_2, op_3 *and* op_4 *must all have the same precedence level.*

Because of lemma 7, $(op_3, op_2) \in \mathbf{Left}_A{}^+$ *in the possibly left-precedent situation and* $(op_3, op_2) \in \mathbf{Right}_A{}^+$ *in the possibly right-precedent situation. This contradicts the precedence relations being hierarchical.*

It could be argued that users that introduce naturally ambiguous operators always (should) have a hierarchy between these operators in mind (i.e. their *preferred* precedence for expressions involving these operators) if they shall be used in combination without parentheses. Following that, we should forbid those natural precedences inside normally ambiguous expressions that are included because of ad-hoc precedence which contradict that given hierarchy.

This means that as soon as a precedence of an operator pair is chosen during syntactic disambiguation according to ad-hoc precedence which is not the natural precedence between these operators, in all left- or right-open sub-expressions of the chosen dominated operator, only ad-hoc precedence should be respected. (This is implemented in the **FollowsHierarchy** constraint in section 6.2.2).

Example 72 *In the above example 71 on page 99, the naturally unambiguous expression* **(a b) c** *would be such a case where the natural precedence of* **a** _ *towards* _ **c** *contradicts the hierarchy, because* **a** _ *is above* _ **d** *which is above* _ **c**.

Using only fully hierarchical ad-hoc precedence relations, it is easy to find a restriction for the natural precedences inside naturally ambiguous expressions. We simply remove all naturally precedent operator pairs where the dominant operator has a higher ad-hoc precedence than the naturally precedent operator.

Unfortunately, this approach can be seen as too restrictive because it demands fully hierarchical ad-hoc precedence relations. These forbid operators being non-associative towards each other and it forces the user to add precedences for operator pairs that are naturally precedent everywhere, just in case that they occur inside an otherwise naturally ambiguous expression.

But we can also just use simple hierarchical precedence relations (i.e. relations that can have several unrelated hierarchies) and restrict those natural precedences that involve operators from the same hierarchy. Additionally, we restrict natural precedence of operators that are ad-hoc dominant to at least one operator because that could be enough to cause an ambiguity.

Example 73 *In the signature in figure 5.10, the natural precedence of* _ **c** *towards* _ **d** *is still valid, while the natural precedence of* **a** _ *towards* _ **c** *is invalid because the operator* _ **e** *is precedent to* **a** _.

```
FUN   a _     : [T1] → T1
FUN   b       : [T1]
FUN   _ c     : [T1] → T2
FUN   _ d     : [T2] → T3
FUN   _ e     : [T3] → T1
PREC  a (_ e)
PREC  (_ d) e
FUN   E       :  T2
DEF   E ==     a b c d e
                  -- (((a b) c) d) e would be rejected
                  -- a (((b c) d) e) would be allowed
```

Figure 5.10: Signature of example 73

Now, we can safely use all operator pairs as left-precedent which are given as left-precedent by the user or which are naturally left-precedent and not contradicting the ad-hoc hierarchies.

Natural vs. Ad-Hoc Precedence

Sometimes, this still might lead to situations where the given ad-hoc precedence contradicts the natural precedence. In that case, only one of the two possible parse trees can be type-correct, so at most one of them can be derived.

This follows from the conflict-freeness of both kinds of precedence-relations.

Example 74 *In the signature in figure 5.11, the precedence of operator* _ :: _ *has been chosen as left-precedent, even though in the example it is only used as naturally right-precedent. Since there is no type given for* **E**, *we assume that the expression would be rejected as possibly ambiguous without the ad-hoc precedence between* # _ *and* _ :: _.

Because there exists no left-precedent derivation of type **nat** *for* 0 :: 0 :: <>, *the natural right-precedent derivation is chosen. It does not contradict hierarchy since the operator* _ :: _ *has the same precedence level as itself.*

Thus, if using the natural precedence relations does not yield a definite answer, then the ad-hoc precedence, if it exists, is taken as the *preferred* precedence.

We find it prudent to warn the user whenever there might be a conflict between the chosen precedence and a given ad-hoc precedence.

```
FUN  0            : nat
FUN  <>           : seq nat
FUN  _ :: _       : [nat , seq nat] → seq nat
FUN  # _          : [seq nat] → nat
PREC # (_ :: _)
PREC (_ :: _) :: _
DEF   E ==        # 0 :: 0 :: <>
                      -- # (0 :: (0 :: <>))
```

Figure 5.11: Signature of example 74

Obviously, it can be very complex to define ad-hoc precedence relations and therefore, lots of mistakes can be made.

Also, the user should be warned about conflicting operator pairs in the given ad-hoc precedences.

Efficiency

Computing the hierarchies of the operators from the given ad-hoc precedences is a cubic algorithm dependent on the number of operator patterns used in these precedences.

Checking the hierarchy and full hierarchy conditions takes quadratic time dependent on the number of open operators in the worst case.

Finally, the computation of definitely conflicting operator pairs is quadratic and depends on the number of given ad-hoc precedence operator pairs. We can re-use the descendant relationship between operators already computed for the hierarchy tests.

Computing the precedence level relations between operators has a very great additional benefit. It can be used as a heuristic by the parser to determine which of the possible operators should be tried first in the matching process for a given subexpression.

This can greatly reduce the effort to find the first type-correct derivation for an expression, which should be the only one if the expression is unambiguous.

How this heuristic can benefit the parser depends a lot on the parsing algorithm used. In our experiments, we used a top-down matching approach on a parse-tree-representation produced by an Earley parser.

Here, we identified the possible top-items for each subexpression to be matched and first selected those that had the lowest precedence level according to the given ad-hoc precedences. We then sorted the possible rightmost children of this item according to their associativity, taking longest items for right-associative operators and shortest items for left-associative operators first.

This resulted in a linear number of matching operations for arithmetic expressions involving mixes of the usual arithmetic operators $\_ + \_$, $\_ - \_$, $\_ * \_$ and $\_/\_$, dependent on the number of operator instantiations in the expression. Since the actual number of possible syntactic derivations is exponential depending on the number of operator instantiations, this is a very good result.

Restrictiveness

The ad-hoc precedence approach does not restrict the mixfix language recognized by only using natural precedence, but only enlarges it.

Some unambiguous expressions which are rejected because of *possible* natural precedence ambiguity are accepted by use of ad-hoc precedences.

Also, some actually *really* naturally ambiguous expressions are accepted as unambiguous *according to the ad-hoc precedence* by singling out the parse-tree most preferable to the user.

Even our most restrictive approaches to ad-hoc precedences yield at least as good results as the classical precedence level method.

5.6.3 Ad-Hoc Precedence of Concatenation Operators

We have already claimed that not allowing concatenation operators to be precedent to each other would make them next to useless. Therefore, we have relaxed our restriction that adjacent operands may not contain concatenation expressions in case of concatenation operators.

We assume that concatenations of the same arity are always left-precedent to each other and that all visible operators are both left and right precedent to the concatenation operators. This assumption saves the user from having to declare these precedences. It also saves us from including these operators into our precedence analyses.

However, there is an ambiguity problem which arises between concatenation operators of different arity which is similar to the backbone ambiguity problem.

Example 75 *In the signature in figure 5.12, the binary and the ternary concatenation operators have the same precedence level, i.e. they are left-precedent towards each other. This causes the expression* **t t t t** *to be ambiguous because of a precedence conflict as it is unclear which of the concatenations is the dominant one.*

Even when the operators do not share the same precedence level, both possible precedences (_ _) _ _ *as well as* (_ _ _) _ *would still leave the expression ambiguous.*

```
FUN     _ _       : [T , T] → T
FUN     _ _ _     : [T , T , T] → T
EQPREC  (_ _ _)(_ _)
FUN     t         : T
FUN     E         : T
DEF     E ==        t t t t
                       -- (tt) t t
                       -- (tt t) t
                       -- ((tt) t) t
```

Figure 5.12: Signature of example 75

Thus, we cannot allow concatenation operators of different arity in the same scope, since no ad-hoc precedence could possibly remove all possible ambiguities between them.

Efficiency

Checking for the presence of concatenation operators of different arities takes linear time dependent on the number of declared operators.

Restrictiveness

Though forbidding concatenations of different arities sounds like a severe restriction, concatenations of arity greater than 2 are pretty pathological. This is probably due to the ambiguity described.

Also, since we allow the overloading of the useful binary concatenation operator, it is possible to simulate operators of greater arity with its help, so that we do not really lose any amount of expressiveness.

Therefore, it is not a great loss to forbid concatenation operators of arity greater than 2 in the presence of the built-in application operator.

5.7 Polymorphism and Converter Operators

5.7.1 Semantic Ambiguity caused by Polymorphism

Polymorphism is the possibility for the programmer to introduce the same operator pattern with generic or different types. This can lead to semantic ambiguity of the expressions to be parsed.

However, if we can find a syntactically unambiguous interpretation for every type which might still be semantically ambiguous, this ambiguity can be dealt with by normal semantic analysis *after the parsing process* in the common way. Thus, we do not have to deal with such semantic ambiguity during parsing.

But since the precedence disambiguation takes type information into account we have to deal with polymorphism already on that level to find out which operators are type-compatible[8]. This is necessary for the weak precedence relations $\mathbf{Left}_T(S)$ and $\mathbf{Right}_T(S)$ as described in section 5.6.1.

5.7.2 Converter Operators

There is one syntactic problem related to polymorphism, though: converter operators.

Converter operators are unary invisible operators and as such have several interesting properties that make them unique in our mixfix operator framework.

Lack of Fixity

In our normal fixity categorization, converter operators would be left-open, right-open and non-empty, i.e. infix. However, it is better to treat them differently from other operators by saying that they have no fixity whatsoever and that an instantiation of a converter operator with an expression inherits all fixity categorization from its operand, since the syntactic properties of the expression do not change by converter instantiation.

Since converter operators are only relevant for typing anyway, and because they are invisible, they can be ignored for backbone ambiguity analysis and they also do not play a role for the restrictions pertaining to adjacent operands.

But they have a big effect on compatibility between non-converter operator instantiations.

Lack of Precedence

If we were to treat converter operators like other left-open or right-open operators, we would be faced with difficult problems.

Converter operators defy hierarchicality. To be practical, they should be applicable everywhere where necessary, both left- and right-precedent to the same operator. Also, all operators which would be left-precedent to a converter operator would automatically also be right-precedent to it because the leftmost left-open operand is also the rightmost right-open operand.

[8]See sections 6.1.4 and 6.2.2

All this leads us to the conclusion that there should be no precedence restrictions on converter operators towards other non-converter operators.

Another rationale to support this notion is the following. If the user wants to be able to control conversion of expressions *syntactically*, they can choose a *syntactic* means to do so, i.e. a *visible* converter operator for which they can define ad-hoc precedences, if necessary.

Converter Instantiation Ambiguity

Even though a converter instantiation looks the same as its operand, it is still a different syntactic interpretation of the same expression. In general, it would thus be possible to have multiple, stacked converter instantiations to the same expression.

If such converter compositions of different length can yield the same type, we could thus have multiple syntactic interpretations for the same expression type, i.e. a syntactic ambiguity for that type. If such a converter composition has the same result type as its operand type, this could even lead to infinitely many syntactic interpretations.

To prevent this ambiguity, we restrict our mixfix expression language so that no converter operator can be instantiated with an expression that is already a converter operator instantiation, i.e. converter instantiation can be done at most once to an expression.

Remains the question of whether or not a converter operator should be applied *once* to an expression or not. There could be expressions that have one converted and one unconverted syntactic interpretation, again yielding a possible syntactic ambiguity.

We deal with this problem the following way. We declare that in every context, an identity-converter operator of type $[\mathbf{A}] \rightarrow \mathbf{A}$ (whose semantic is the identity operation, $\mathbf{A}$ being a type variable) is present and that *every expression* has only syntactic interpretations which are converter operator instantiations. Thus, if no *other* converter operator is applicable because of the demanded type for the expression, but there existed a syntactic interpretation that is not an instantiation of a user-defined converter operator, the identity-converter can be applied as a default to arrive at a converted interpretation of the same type.

This way, the question of whether or not a single converter-operator is applied to an expression never arises because *it always is.*

To avoid semantic ambiguity with the identity-converter, converters where the operand type is equal or compatible to the result type should be disallowed as they would also be applicable everywhere.

With these restrictions in mind, we can pair off all non-converter operators with all converter-operators (including the identity-converter) inside each scope and combine them to a single *converted* operator . The set of converted operators is then treated as our operator set for the precedence analysis, treating the operators as if they were simply overloaded and no converter operators existed.

For this purpose, we replace the relation _ `returns_` _ used by our natural precedence relations by the following definition:

Definition 72

```
FUN _ returns_ _ : [pattern , scope , type] → bool
DEF p returns$_S$ T==
    IF (_ : [A] → B ∈ S) and
        (B compatible T) and
        (p : [$T_1$ , . . . , $T_n$] → $T_0$) ∈ S and
        ($T_0$ compatible A[B := T])
    THEN true
    IF (T = LIFT $T_0$) and
        (separators(p) = [$s_0$ , . . . , $s_n$]) and
        (p : [$T_1$ , . . . , $T_n$] → $T_0$) ∈ S and
        ($T_0$ compatible $s_0$ $T_1$ $s_1$ . . . $T_n$ $s_n$)
    THEN true
    ELSE false
    FI
```

Efficiency

Building all converted operators is in the order of the product of the number of non-converter operators with the number of converter operators in any context. Thus, the actual size of the operator set to be dealt with by precedence analysis is in the worst case quadratic to the number of declarations given by the user.

Restrictiveness

Allowing converter operators gives us a powerful language with a coercion semantics, if so desired by the user.

Multiple coercion between different types for one expression is not allowed in most languages that have coercion semantics for their built-in types, either, probably because of the same ambiguity reasons that we stated here. But the user can feel free to define converter operators that are implemented as *compositions* of other converter operators.

Disallowing converters where the operand type is compatible to the result type is prudent as such operators that actually carry a semantics (i.e. *do something*) would have a detrimental effect on the understandability of the programs written with the help of the mixfix expression language.

Since such implicitly present operators as the converter operators can lead easily to semantic ambiguity, and even in unambiguous cases also can lead to confusing programs, they should be used with caution. Also, the more converter operators are defined, i.e. the more compatibility exists between operators, the fewer natural precedences can be found which means that the user has to declare more ad-hoc precedences.

Finally, converter operators can easily lead to an exponential number of semantic interpretations of an expression for the same type, presenting a similar problem as that of mixfix parsing to the latter semantic analysis phase.

Example 76 *In the signature in figure 5.13, the converter operators result in an exponential number of semantic interpretations for expressions of the form* **id . . . id x** *for type* **nat** *dependent on the length of the expression.*

```
FUN nat  : SORT
FUN int  : SORT
FUN id _ : [nat] → nat
FUN id _ : [int] → int
FUN _    : [nat] → int
FUN _    : [int] → nat
FUN x    : nat
FUN E    : nat
DEF E == id id x
             -- id(id(x : nat) : nat) : nat
             -- id(id(x : nat) : int) : nat
             -- id(id(x : int) : nat) : nat
             -- id(id(x : int) : int) : nat
```

Figure 5.13: Signature of example 76

This problem could probably be solved by having the user give preferences between different overloaded versions of the same operator pattern. Using the means of our expression language, it can only be overcome syntactically by type annotation for the inner parts of the expression.

All in all, converter operators seem dangerous in multiple ways in regard to semantic ambiguity. However, forbidding them is too restrictive and also would not solve such semantic problems in general as the user could decide to declare a multitude of overloaded non-invisible converter operators including an identity operator and get the same disastrous effects if these operators are used everywhere.

5.8 Proof of Unambiguity

We will now prove that, given the introduced restrictions regarding adjacent operands, precedence relations and invisible operators, the resulting accepted expression language is unambiguous, i.e. that at most one type-correct parse tree for every mixfix expression for every demanded type is accepted by the mixfix parser.

We assume that there is no backbone ambiguity for the expression to be parsed, meaning that the expression can unambiguously be partitioned into inner operator parts, which in turn can be partitioned into separator and operand parts according to the backbone grammar.

We further assume that only one acceptable left-weighted interpretation exists for the expression, meaning that all token sequences of adjacent operand parts can be partitioned into basic expressions.

Now, it only needs to be shown that only one type-correct parse tree can be derived for every sentence established as a basic expression for every given demanded type for the expression.

Proposition 7 *Given a mixfix operator signature* $\mathbf{S}$ *and a fully hierarchical pair of ad-hoc precedence relations* $\mathbf{Left_A(S)}$ *and* $\mathbf{Right_A(S)}$*, for every type-correct mixfix operator instantiation* $\mathbf{E}$ *involving only operators from* $\mathbf{S}$ *where no backbone or adjacent-operator ambiguity occurs in* $\mathbf{E}$*, there exists at most one type-correct parse tree for any given demanded type* τ *which respects all adjacent-operand restrictions, restrictions on converter operators and both the ad-hoc and natural precedence relations.*

Proof 12 *We prove this by induction on the maximal depth of the parse tree for expression* $\mathbf{E}$.

induction begin *For nullary operators, i.e. operators which have no operands, their operator instantiation is the same token sequence as the operator pattern which is the first (and last) separator of the operator.*

Therefore, if there is no backbone ambiguity in $\mathbf{E}$*, there can only be one syntactical interpretation of E, regardless of the demanded type.*

If the result type of the nullary operator is not compatible with the demanded type, no type-correct interpretation of $\mathbf{E}$ *for that type can be found.*

induction step *Suppose we have expressions* $\mathbf{E}_1 , \dots , \mathbf{E_n}$ *which are all unambiguous for any given demanded types* $\tau_1 , \dots , \tau_\mathbf{n}$ *and whose parse trees have a maximal depth of* n.

Let $\mathbf{E}$ *be the instantiation of a non-visible non-nullary operator* $\mathbf{op} = \mathbf{s}_0 \; \_ \; \dots \; \_ \; \mathbf{s_n} : \bigcup\{\mathbf{T} | \mathbf{T} = [\mathbf{T}_1 , \dots , \mathbf{T_n}] \rightarrow \mathbf{T}_0 \; , \mathbf{s}_0 \; \_ \; \dots \; \_ \; \mathbf{s_n} : \mathbf{T} \in \; \mathbf{S}\}$ *instantiated with the operands* $\mathbf{E}_1 , \dots , \mathbf{E_n}$*, i.e.* $\mathbf{E} = \mathbf{s}_0 \; \mathbf{E}_1 \; \mathbf{s}_1 \; \dots \; \mathbf{s_{n-1}} \; \mathbf{E_n} \; \mathbf{s_n}$.

The demanded type $\tau_\mathbf{i}$ *for every expression* $\mathbf{E_i}$ *can be determined from the demanded type* τ *and the declared types of* $\mathbf{op}$.

If there exists one type-correct interpretation $\mathbf{P_i}$ *for every* $\mathbf{E_i}$ *in type* $\tau_\mathbf{i}$ *with maximal depth* n*, then clearly* $\mathbf{P} = \mathbf{op}(\mathbf{P}_1 \; , \; \dots \; , \; \mathbf{P_n}) : \tau$ *is a type-correct interpretation of* $\mathbf{E}$ *for type* τ *of maximal depth* $n + 1$.

According to our precondition, we assume that there is no backbone or adjacent-operand ambiguity in $\mathbf{E}$*, so that all* $\mathbf{E_i}$ *which instantiate open or adjacent operands of* $\mathbf{op}$ *must be basic expressions. Thus,* $\mathbf{E}$ *is a basic expression as well.*

If $\mathbf{op}$ *is a closed operator, then it cannot be involved in any precedence-related ambiguity.*

Let us assume that $\mathbf{op}$ *is a right-open operator*[9] *and* $\mathbf{P}$ *is an* acceptable *interpretation of* $\mathbf{E}$ *for type* τ *according to the ad-hoc and natural precedence relations.*

Let the root operand of the rightmost operand expression $\mathbf{P_n}$ *be* $\mathbf{op_n}$*. Then* $(\mathbf{op} \; , \mathbf{op_n}) \in \mathbf{Right_A(S)} \cup \mathbf{RightPrec}_\tau \; (\mathbf{Left_T(S)} \; , \mathbf{Right_T(S)})$ *because* $\mathbf{P}$ *is acceptable according to precedence.*

Because of the definition of $\mathbf{RightPrec}$[10] *and proposition 6*[11] *and the given full hierarchicality of* $\mathbf{Left_A(S)}$ *and* $\mathbf{Right_A(S)}$*, there can be no left-precedent situation possible between a left-open operator* $\mathbf{op'}$ *which is either* $\mathbf{op_n}$ *or a left-most or right-most descendant thereof and* $\mathbf{op}$*, since this would make another parse tree* $\mathbf{P'}$ *for* $\mathbf{E}$ *with* $\mathbf{op'}$ *as its root and* $\mathbf{op}$ *as the root of a subtree of its leftmost operand also acceptable.*

Therefore, $\mathbf{P}$ *must be the only acceptable syntactical interpretation of* $\mathbf{E}$.

If at most one acceptable parse tree exists for every basic expression for every type and all non-basic expressions can be unambiguously partitioned into separators and basic expressions, then no further syntactic ambiguity exists in the mixfix expression language.

[9]The following reasoning can be done analogously for left-open operators.

[10]see definition 68 on page 97

[11]see page 101

Chapter 6

Two-Level Mixfix Grammars

As a description tool, two-level grammars have been used successfully both in the field of linguistics and artificial intelligence, as they bridge the gap between syntax and semantics while allowing to use arbitrary context-free grammars, as in those fields, ambiguity must be allowed and thus has to be dealt with differently than in the field of computer languages.

Although we want to define an *unambiguous* mixfix expression language, we *still* think that the two-level grammar formalism is also very useful for incorporating our restrictions on the language into the grammar, thereby yielding an unambiguous two-level grammar for a the underlying context-free grammar.

In this chapter, we will introduce the two-level grammars for our expression language derived from the given mixfix operators and parameterized with the ad-hoc precedence relations. As can be seen, it is simply an annotated version of the possibly ambiguous context-free grammar describing our mixfix expression language. No other grammar transformation needs to take place.

Most definitions in this chapter are simply implementations of the restrictions and properties described in the previous chapter.

6.1 From Context-Free Mixfix-Grammar to Two-Level Mixfix-Grammar

Each nonterminal symbol in the grammars (i.e. **COMMA**, **COLON**, **ARROW**, **DOT**, **MIXFIX**, **E** and **GROUP** is annotated with a variable representing its attributes.

With the help of these attributes, we can describe constraints for the expressions to arrive at an unambiguous grammar. Every rule is annotated with a such a constraint and quantified with both the attribute variables and the other variables in the respective constraint[1].

The following definition yields a shorthand-notation for the variable-quantification.

Definition 73 $\Gamma_{\mathbf{k}}$ *is the short-hand notation for the variable set* $\{\mathbf{A_i}\}_{\mathbf{i=0}}^{\mathbf{k}}$.

[1]Ideally, all equations we give here in constraint-form should be implicit, hidden in the annotations of the symbols with their attribute structures, so later on, during derivation, only unification needs to take place. However, this notation would make even the most simple grammar rules too long to remain readable. But it can be derived by unfolding the different constraints.

6.1.1 Two-Level Meta-Operator Grammar

The meta-operator grammar is the annotated version of the grammar in section 4.2.8 on page 63.

$$\begin{array}{lll}
\forall\Gamma_2 \mid \mathbf{Expr}([\mathbf{A_i}]_{\mathbf{i=0}}^{2}\ ,\ \_\ ,\ \_) \ . & \mathbf{COMMA(A_0)} & ::= \mathbf{COLON(A_1)}\ \underline{,}\ \mathbf{COMMA(A_2)} \\
\forall\Gamma_0\ . & \mathbf{COMMA(A_0)} & ::= \mathbf{COLON(A_0)} \\
\forall\Gamma_2 \mid \mathbf{Expr}([\mathbf{A_i}]_{\mathbf{i=0}}^{2}\ ,\ \_\ :\ \_) \ . & \mathbf{COLON(A_0)} & ::= \mathbf{ARROW(A_1)}\ \underline{:}\ \mathbf{COLON(A_2)} \\
\forall\Gamma_0\ . & \mathbf{COLON(A_0)} & ::= \mathbf{ARROW(A_0)} \\
\forall\Gamma_2 \mid \mathbf{Expr}([\mathbf{A_i}]_{\mathbf{i=0}}^{2}\ ,\ \_\ \rightarrow\ \_) \ . & \mathbf{ARROW(A_0)} & ::= \mathbf{DOT(A_1)}\ \underline{\rightarrow}\ \mathbf{ARROW(A_2)} \\
\forall\Gamma_0\ . & \mathbf{ARROW(A_0)} & ::= \mathbf{DOT(A_0)} \\
\forall\Gamma_2 \mid \mathbf{Expr}([\mathbf{A_i}]_{\mathbf{i=0}}^{2}\ ,\ \_\ .\ \_) \ . & \mathbf{DOT(A_0)} & ::= \mathbf{DOT(A_1)}\ \underline{.}\ \mathbf{MIXFIX(A_2)} \\
\forall\Gamma_0\ . & \mathbf{DOT(A_0)} & ::= \mathbf{E(A_0)} \\
\forall\Gamma_0\ . & \mathbf{E(A_0)} & ::= \mathbf{GROUP(A_0)}
\end{array}$$

6.1.2 Two-Level Mixfix-Operator Grammar

Finally, we modify the derivation of rules from mixfix patterns of the operators signature Σ described in section 4.3.2 in the following way:

- For every operator pattern *op* inside Σ, we derive one rule with the following properties:
  - it is quantified over the set Γ_k where k is the arity of the operator pattern *op*.
  - it has the constraint $\mathbf{Expr}([\mathbf{A_i}]_{\mathbf{i=0}}^{\mathbf{k}}\ ,\ \mathbf{op})$
  - it has the left-hand-side nonterminal $\mathbf{E(A_0)}$
  - The right-hand side for operator pattern *op* is derived as follows:
    - map every separator token to its respective terminal symbol
    - map very enclosed placeholder token[2] to nonterminal $\mathbf{COMMA(A_i)}$, where i is the index of placeholder symbol inside *op* (counting from 1).
    - map every left-open, right-open or adjacent placeholder token to nonterminal $\mathbf{E(A_i)}$, where i is the index of placeholder symbol inside *op* (counting from 1).

Example 77 *The context-free mixfix grammar for the operator signature from example 49 on page 64 would look like the one in figure 6.1. As can be seen by comparison with example 50 on page 65, every nonterminal in the context-free mixfix grammar is simply annotated with an attributes variable and a constraint over these variables is added to each rule.*

[2]a placeholder symbol which appears between two non-empty separators

$\forall\Gamma_3 \mid \mathbf{Expr}([\mathbf{A_i}]^3_{\mathbf{i=0}}\ ,\ \mathbf{if}\ \_\ \mathbf{then}\ \_\ \mathbf{else}\ \_)\ .$
$\quad \mathbf{E}(\mathbf{A}_0) ::= \underline{\texttt{if}}\ \mathbf{COMMA}(\mathbf{A}_1)\ \underline{\texttt{then}}\ \mathbf{COMMA}(\mathbf{A}_2)\ \underline{\texttt{else}}\ \mathbf{E}(\mathbf{A}_3)$
$\forall\Gamma_2 \mid \mathbf{Expr}([\mathbf{A_i}]^2_{\mathbf{i=0}}\ ,\ \_\ =\ \_)\ .$
$\quad \mathbf{E}(\mathbf{A}_0) ::= \mathbf{E}(\mathbf{A}_1)\ \underline{=}\ \mathbf{E}(\mathbf{A}_2)$
$\forall\Gamma_0 \mid \mathbf{Expr}([\mathbf{A_i}]^0_{\mathbf{i=0}}\ ,\ 0)\ .$
$\quad \mathbf{E}(\mathbf{A}_0) ::= \underline{\texttt{0}}$
$\forall\Gamma_0 \mid \mathbf{Expr}([\mathbf{A_i}]^0_{\mathbf{i=0}}\ ,\ 1)\ .$
$\quad \mathbf{E}(\mathbf{A}_0) ::= \underline{\texttt{1}}$
$\forall\Gamma_2 \mid \mathbf{Expr}([\mathbf{A_i}]^2_{\mathbf{i=0}}\ ,\ \_\ -\ \_)$
$\quad \mathbf{E}(\mathbf{A}_0) ::= \mathbf{E}(\mathbf{A}_1)\ \underline{-}\ \mathbf{E}(\mathbf{A}_2)$
$\forall\Gamma_2 \mid \mathbf{Expr}([\mathbf{A_i}]^2_{\mathbf{i=0}}\ ,\ \_\ *\ \_).$
$\quad \mathbf{E}(\mathbf{A}_0) ::= \mathbf{E}(\mathbf{A}_1)\ \underline{*}\ \mathbf{E}(\mathbf{A}_2)$
$\forall\Gamma_1 \mid \mathbf{Expr}([\mathbf{A_i}]^1_{\mathbf{i=0}}\ ,\ \_\ !)\ .$
$\quad \mathbf{E}(\mathbf{A}_0) ::= \mathbf{E}(\mathbf{A}_1)\ \underline{!}$
$\forall\Gamma_0 \mid \mathbf{Expr}([\mathbf{A_i}]^0_{\mathbf{i=0}}\ ,\ \mathbf{x})\ .$
$\quad \mathbf{E}(\mathbf{A}_0) ::= \underline{\texttt{x}}$
$\forall\Gamma_1 \mid \mathbf{Expr}([\mathbf{A_i}]^1_{\mathbf{i=0}}\ ,\ \mathbf{fac}\ \_)\ .$
$\quad \mathbf{E}(\mathbf{A}_0) ::= \underline{\texttt{fac}}\ \mathbf{E}(\mathbf{A}_1)$

Figure 6.1: Grammar of example 77

We will cover the constraints in greater detail in this section. The operator rules are added for every *converted operator*, not for the converter operator, so there is no rule $\forall\Gamma_1 \mid \mathbf{Expr}([\mathbf{A_i}]^1_{\mathbf{i=0}}\ ,\ \_)\ .\ \ \mathbf{E}(\mathbf{A}_0) ::= \mathbf{E}(\mathbf{A}_1)$ in the grammar.

We will describe the constraints as Prolog-like clauses over the expression attributes.

6.1.3 Expression Attributes

What attributes do we need to describe the constraints on the expressions?

We can divide the attributes into two general categories:

- input-attributes, i.e. those attributes the parser needs to parse and disambiguate the expression, and
- output-attributes, i.e. the parsing results for each expression, parts of which can also be used as input-attributes for other expressions.

Input Attributes

For every rule derivation, the parser needs to know several things about the expression to be derived for disambiguation:

- the operator pattern to be instantiated,
- the demanded result type,
- the token sequence to be parsed,
- the scope in which the expression is parsed,
- the ad-hoc precedence relations in that scope, and
- the precedence status of the expression, i.e. whether anti-hierarchical natural precedences shall be rejected.

From these input-attributes, we can compute:

- the declared types of the operator pattern to be instantiated,
- the declared converter types,
- the subset of visible operator patterns that can be present in the expression.
- the backbone list of the expression, i.e. the operators that potentially have to be ordered via precedence relations,
- the set of those concurrent operators which could also be the root of the expression.

Output Attributes - The Annotated Parse Tree

The output of the parse process is an annotated parse tree.

This tree is a conglomerate of several inferred attributes for the parsed operator instantiation:

- the pattern of the root operator,
- the list of operand parse trees,
- the inferred type,
- the scope in which the parse tree was inferred.

Thus, if the operator pattern $\mathbf{op} = \mathbf{s_0}\_\ \mathbf{s_1} \ldots \_\ \mathbf{s_n}$ and the inferred type $\mathbf{T} = \bigcup\{\mathbf{L_i} \to \tau_\mathbf{i}\}_{\mathbf{i} \in \mathbf{I}}$, we represent an operator instantiation of **op** with the operands $\mathbf{t_1}, \ldots, \mathbf{t_n}$ in scope **S** as $\mathbf{S}\ .\ \mathbf{op}[\mathbf{t_1}, \ldots, \mathbf{t_n}] : \mathbf{T}$ with the help of the following type definition.

```
TYPE  parseTree == (_ . _ _ _ : _)
                   (scope : scope ,
                    operator : pattern ,
                    operands : seq[parseTree] ,
                    type : type)
```

From these attributes, it is easy to infer, whether or not an expression is:

- empty,
- a concatenation,
- prefix,
- postfix,
- a variable, or
- type-erroneous.

These secondary attributes are mainly needed to implement the restrictions concerning adjacent operands, but also to mark expressions as variable so that this can be taken into account in the unification process involving parsed expressions.

An expression is empty only if it is an instantiation of the empty operator (covering the empty sentence, obviously). Likewise, an expression is a concatenation, if the root operator is a concatenation operator. An expression is a variable if the root operator is declared variable in the scope of the expression.

The only one of these secondary output attributes where it makes sense to store them additionally to the result are the prefix and postfix characteristics which might depend on the leftmost or rightmost operands of instantiations of left-open or right-open operators und thus need to be inferred bottom-up.

Finally, an expression is type-erroneous when the inferred type is the empty union, as then there is no type-correct interpretation of that sentence for the given demanded result type.

Attribute Representation

```
TYPE attributes ==
     attributes(scope : scope ,
                checkHierarchy : bool ,
                sentence : seq[phrase] ,
                operator : pattern ,
                demanded : type ,
                result : parseTree ,
                prefix : bool ,
                postfix : bool)
```

6.1.4 Type Inference

Up to now, we have described the type inference only in terms of single demanded and single inferred types.

However, in the face of overloading and converter operators, we have to generalize the algorithm from section 4.4 in such a way that it can deal also with a union of demanded types and can infer a corresponding union of inferred types for an operator instantiation, as soon as there are overloaded operators present in the scope of the expression. This allows us to delay treatment of any semantic ambiguity until after the parsing phase.

To that end, we define functions that compute the set of demanded types of each operand from the set of demanded type for an operator instantiation. Likewise, we define functions that take the demanded types of the instantiation, the sets of inferred types of the operands and the declared types of the instantiated operator and infer the possible types of the instantiation. As can be easily verified, these functions are in accord with the top-down bottom-up single type approach given in chapter 4.

The following definitions are used in section 6.2.

Normal Operator Instantiation

The demanded types for an operand at the *i*-th position in a normal operator instantiation depend on the demanded type of that instantiation and the declared types of the instantiated operator.

The inferred types for such an instantiation additionally depend on the inferred types of all operand expressions.

Definition 74 *The function* **demand** *computes the demanded types for the* **i***-th operand of an operator instantiation which has the demanded result types* **Demanded** *and the declared types* **Declared***.*

```
FUN  $\mathbf{demand}: \ \mathbf{set[type]**nat**set[type]} \to \ \mathbf{set[type]}$
DEF  $\mathbf{demand(Demanded\ ,\ i\ ,\ Declared)}$
     $\{\ \mathbf{T_i[T_0} := \tau_0]|$
     $\tau_0 \in \mathbf{Demanded}\ ,$
     $[\mathbf{T}_1\ , \ldots , \mathbf{T_n}] \to \mathbf{T}_0 \in \mathbf{Declared}\}$
```

Definition 75 *The function* **infer** *computes the inferred result types of an operator instantiation from the demanded result types* **Demanded***, the declared types of the operator* **Declared** *and the inferred types of the operands* $[\mathbf{Inferred_i}]_{\mathbf{i=1}}^{\mathbf{n}}$.

```
FUN  $\mathbf{infer}: \ \mathbf{set[type]**seq[set[type]]} \to \ \mathbf{set[type]}$
DEF  $\mathbf{infer(Demanded\ ,\ Declared\ ,\ [Inferred_i]_{i=1}^{n})==}$
     $\{([\mathbf{L}_1] \ldots [\mathbf{L_n}] \to \mathbf{T}_0)[\mathbf{T_i} := \tau_\mathbf{i}]_{\mathbf{i=0}}^{\mathbf{n}}|$
     $\tau_0 \in \mathbf{D}\ ,$
     $[\mathbf{T}_1\ , \ldots , \mathbf{T_n}] \to \mathbf{T}_0 \in \mathbf{Declared}\ ,$
     $\forall\ \mathbf{i} \in 1 \ ..\ \ \mathbf{n} : [\mathbf{L_i}] \to \tau_\mathbf{i} \in \mathbf{Inferred_i}\}$
```

Nullary Operator Inference

For nullary operators, the nullary section unlifting operation is also invoked which is similar to the normal inference operation, but takes only those declarations into account which result in a section type.

Definition 76 *The function* **inferNullary** *infers the result types of a nullary operator instantiation where the demanded result type is* **Demanded** *and the declared types of the operator are* **Declared***.*

```
FUN  $\mathbf{inferNullary}: \ \mathbf{set[type]**set[type]} \to \ \mathbf{set[type]}$
DEF  $\mathbf{inferNullary(Demanded\ ,\ Declared)==}$
     $\{([\mathbf{L}] \to \ \mathbf{T})[\mathbf{T} := \tau]|$
     $\tau \in \mathbf{Demanded}\ ,$
     $[] \to [\mathbf{L}] \to \ \mathbf{T} \in \ \mathbf{Declared}\}$
```

Section Lifting

For lifting section expressions in parentheses to fully instantiated function expressions, the following two functions are employed.

Only the result type of the demanded section type is taken as demanded type for the expression. Later on, the inferred section operand types are lifted and unified with the actually demanded result type.

Definition 77 *The function* **demandSection** *computes the demanded section result types from a set of demanded section types.*

```
FUN  $\mathbf{demandSection}: \ \mathbf{set[type]} \to \ \mathbf{set[type]}$
DEF  $\mathbf{demandSection(Demanded)==}$
     $\{\ \mathbf{T}_0|[\mathbf{T}_1] \to \mathbf{T}_0 \in \mathbf{Demanded}\}$
```

Definition 78 *The function* **inferSection** *computes the inferred empty section types from a set of demanded result types* **Demanded** *and a set of inferred types* **Inferred**. *In essence, each inferred section type is lifted to a function type.*

```
FUN inferSection : set[type]**set[type] → set[type]
DEF inferSection(Demanded , Inferred)==
    {(([] → T)[T := $\tau$]|
    $\tau \in$ Demanded ,
    [L] → $T_0 \in$ Inferred ,
     T = (L) → $T_0$}
```

LIFT conversion

Definition 79 *The function* **demandLIFT** *computes a set of demanded result types from demanded* **LIFT** *types.*

```
DEF demandLIFT(Demanded)==
    { T|LIFT E ∈ Demanded , [] → T ∈ type(E)}
```

Definition 80 *The function* **inferLIFT** *takes a set of demanded* **LIFT** *types and an inferred result type and infers the* **LIFT** *type for that result type.*

```
DEF inferLIFT(Demanded , T)==
    {[] → LIFT E[E := T]|LIFT E ∈ Demanded}
```

Type Annotation

Definition 81 *The function* **demandTypedRight** *computes the demanded type (which is simply the* **LIFT** *type) for the right-hand side of the type annotation expression.*

```
FUN demandTypedRight : set[type] → set[type]
DEF demandTypedRight(Demanded)==
    { LIFT T|T ∈ Demanded}
```

Definition 82 *The function* **demandTypedLeft** *computes the demanded type for the left-hand side of the type annotation, using the inferred types of the right-hand side* **Inferred**.

```
FUN demandTypedLeft : set[type] → set[type]
DEF demandTypedLeft(Inferred)==
    {$\tau$|
    [] → LIFT([L] → $\tau$) ∈ Inferred}∪
    {$\tau$|
    [] → LIFT $\tau$ ∈ Inferred ,
    $\tau \neq$ [] → $\tau'$}
```

Definition 83 *The function* **inferTyped** *infers the type of the type annotation expression, using the inferred types of both the left-hand and the right-hand sides.*

```
FUN  inferTyped : set[type]**set[type] → set[type]
DEF  inferTyped(Inferred_1 , Inferred_2)==
      {$\tau_1[\tau_1 := \tau_2]$|
      $\tau_1 \in$ Inferred_1 ,
      [] → LIFT $\tau_2 \in$ Inferred_2}∪
      {$\tau_1[\tau_1 := [] \to \tau_2]$|
      $\tau_1 \in$ Inferred_1 ,
      [] → LIFT $\tau_2 \in$ Inferred_2 ,
      $\tau_2 \neq [] \to \tau'$}
```

Binary Concatenation — Apply Operator

Definition 84 *The function* **demandConcatLeft** *computes the demanded type of the left operand of a concatenation expression by using the demanded result types of the concatenation expression* **Demanded** *and the declared types of the concatenation operator* **Declared**. *The built-in application functionality is also taken into account in that respect.*

```
FUN  demandConcatLeft : set[type]**set[type] → set[type]
DEF  demandConcatLeft(Demanded , Declared)==
     demand(Demanded , 1 , Declared ∪ { ApplyDecl})
     WHERE ApplyDecl==[T_2 → T_0 , T_2] → T_0
```

Definition 85 *The function* **demandConcatRight** *computes the demanded type for the right operand of a concatenation expression from the demanded result types of the concatenation expression* **Demanded**, *the declared types of the concatenation operator* **Declared** *as well as the inferred types of the left operand of the concatenation operator. The latter is only used for the built-in application functionality.*

```
FUN  demandConcatRight : set[type]**set[type] → set[type]
DEF  demandConcatRight(Demanded , Declared , InferredLeft)==
     demand(Demanded , 2 , Declared)∪
    { $T_2[T_0 := \tau_0]$|
    $\tau_0 \in$ Demanded ,
    [L] → T_2 → T_0 ∈ InferredLeft}
```

6.2 Mixfix-Grammar Constraints

In this section, we will describe the constraints on the attributes of the rules in our two-level grammar, using the definition from the previous section.

6.2.1 Operator Kinds

We have six kinds of different operators and for each we have different constraints, some of which of course overlap in part:

- normal, simple operator instantiations, including
  - all visible non-meta operators,
  - the operator _ , _,
  - the operator _ → _, as well as
  - the empty operator,

- the type annotation operator $\_ \; : \; \_$,
- the scope annotation operator $\_ \; . \; \_$,
- the parenthesis operator $(\; \_ \;)$,
- the binary concatenation operator $\_ \; \_$, and
- the concatenation operators of arity greater than 2.

The only missing operator is the converter operator whose restricted behavior is integrated into the constraints of the other operators.

Definition 86

$$
\begin{array}{l}
\mathbf{TypeAnnotation}(\_ \; \underline{:} \; \_) \; . \\
\mathbf{ScopeAnnotation}(\_ \; \underline{.} \; \_) \; . \\
\mathbf{Paren}(\underline{(} \; \_ \; \underline{)}) \; . \\
\mathbf{Normal}(\mathbf{Op}) \Leftarrow \\
\quad \mathbf{visible}(\mathbf{Op}) \; , \\
\quad \neg\mathbf{TypeAnnotation}(\mathbf{Op}) \; , \\
\quad \neg\mathbf{ScopeAnnotation}(\mathbf{Op}) \; , \\
\quad \neg\mathbf{Paren}(\mathbf{Op}) \; . \\
\mathbf{Normal}(\epsilon) \; . \\
\mathbf{BinaryConcat}(\_ \; \_) \; . \\
\mathbf{MultiConcat}(\mathbf{Op}) \Leftarrow \\
\quad \neg\mathbf{visible}(\mathbf{Op}) \; , \\
\quad \mathbf{arity}(\mathbf{Op}) > 2 \; .
\end{array}
$$

6.2.2 Normal Operator Instantiations

For a normal operator instantiation to be a valid mixfix expression, its type and fixity must be inferrable from the demanded result types and the inferred types and fixities of the operand expressions. Also, for all adjacent operands it must hold that they are either both empty or both non-empty.

Definition 87

$$
\begin{array}{l}
\mathbf{Expr}([\mathbf{A_i}]_{\mathbf{i=0}}^{\mathbf{n}} \; , \mathbf{Op}) \Leftarrow \\
\quad \mathbf{Normal}(\mathbf{Op}) \; , \\
\quad \mathbf{Op} = \mathbf{s}_0 \; \_ \; \mathbf{s}_1 \; \ldots \; \_ \; \mathbf{s_n} \; , \\
\quad \mathbf{sentence}(\mathbf{A}_0) = \mathbf{s}_0 \; \mathbf{sentence}(\mathbf{A}_1) \; \mathbf{s}_1 \; \ldots \; \mathbf{sentence}(\mathbf{A_n}) \; \mathbf{s_n} \; , \\
\quad \textit{– type inference and precedence checks } \mathbf{Inference}([\mathbf{A_i}]_{\mathbf{i=0}}^{\mathbf{n}} \; , \mathbf{Op}) \; , \\
\quad \textit{– check for restriction 1 } \mathbf{AdjacentEmptyOperands}([\mathbf{s_i}]_{\mathbf{i=1}}^{\mathbf{n-1}} \; , [\mathbf{A_i}]_{\mathbf{i=1}}^{\mathbf{n}}) \; .
\end{array}
$$

Top-Down Bottom-Up Inference

According to our top-down bottom-up type inference algorithm, the declared types of the operator to be matched are combined with the demanded result types to infer the demanded result types of the operand expressions.

The inferred types of the operand expressions are then used to infer the actual type of the overall expression.

Also, fixities are computed from the operator shape and for open operators from the fixities of the leftmost left-open or rightmost right-open operand expressions.

Definition 88

```
$\mathbf{Inference}([\mathbf{A_i}]_{\mathbf{i}=0}^{\mathbf{n}}, \mathbf{Op}) \Leftarrow$
  $\mathbf{operator}(\mathbf{result}(\mathbf{A}_0)) = \mathbf{Op}$ ,
  $\mathbf{operator}(\mathbf{A}_0) = \mathbf{Op}$ ,
  $\mathbf{ComputePrefix}(\mathbf{Op}, [\mathbf{A_i}]_{\mathbf{i}=1}^{\mathbf{n}}, \mathbf{prefix}(\mathbf{A}_0))$ ,
  $\mathbf{ComputePostfix}(\mathbf{Op}, [\mathbf{A_i}]_{\mathbf{i}=1}^{\mathbf{n}}, \mathbf{postfix}(\mathbf{A}_0))$ ,
  $\mathbf{demanded}(\mathbf{A}_0) = \bigcup \mathbf{Dem}_0$ ,
  – *get the converter operator types*
  $\mathbf{types}(\mathbf{scope}(\mathbf{A}_0))(\_) = \bigcup \mathbf{CDecl}$ ,
  – *compute the demanded types with converters*
  $\mathbf{CDem} = \mathbf{Dem}_0 \cup \mathbf{demand}(\mathbf{Dem}_0, 1, \mathbf{CDecl}) \cup \mathbf{demandLIFT}(\mathbf{Dem}_0)$ ,
  – *get the declared types of Op*
  $\mathbf{types}(\mathbf{scope}(\mathbf{A}_0))(\mathbf{Op}) = \bigcup \mathbf{Decl}$ ,
  – *check restrictions for operands and precedences*
  $\mathbf{InferenceForAllOperands}(\mathbf{A}_0, \mathbf{Decl}, \mathbf{CDem}, [\mathbf{A_i}]_{\mathbf{i}=1}^{\mathbf{n}}, [\mathbf{Inf_i}]_{\mathbf{i}=1}^{\mathbf{n}})$ ,
  – *infer the unconverted types from the inferred operand types*
  $\mathbf{CInf} = \mathbf{infer}(\mathbf{CDem}, \mathbf{Decl}, [\mathbf{Inf_i}]_{\mathbf{i}=1}^{\mathbf{n}}) \cup \mathbf{inferNullary}(\mathbf{CDem}, \mathbf{Decl})$ ,
  $\mathbf{S} = \mathbf{scope}(\mathbf{A}_0)$ ,
  $\mathbf{Operands} = [\mathbf{result}(\mathbf{A_i})]_{\mathbf{i}=1}^{\mathbf{n}}$ ,
  $\mathbf{CResult} = \mathbf{S}\ .\ \mathbf{Op}\ \mathbf{Operands} : \bigcup \mathbf{CInf}$ ,
  – *infer the converted types*
  $\mathbf{Inf}_0 = \mathbf{CInf} \cup \mathbf{infer}(\mathbf{Dem}_0, \mathbf{CDecl}, [\mathbf{CInf}]) \cup \mathbf{inferLIFT}(\mathbf{Dem}_0, \mathbf{CResult})$ ,
  $\mathbf{result}(\mathbf{A}_0) = \mathbf{S}\ .\ \mathbf{Op}\ \mathbf{Operands} : \bigcup \mathbf{Inf}_0$ .
$\mathbf{InferenceForAllOperands}(\mathbf{A}_0, \mathbf{Decl}, \mathbf{Dem}, [\mathbf{A_i}]\mathbf{A}, [\mathbf{Inf_i}]\mathbf{Inf}) \Leftarrow$
  $\mathbf{InferenceForOperands}(\mathbf{A}_0, \mathbf{Decl}, \mathbf{Dem}, 1, [\mathbf{A_i}]\mathbf{A}, [\mathbf{Inf_i}]\mathbf{Inf})$ .
$\mathbf{InferenceForOperands}(\mathbf{A}_0, \mathbf{Decl}, \mathbf{Dem}, \mathbf{i}, [\mathbf{A_i}]\mathbf{A}, [\mathbf{Inf_i}]\mathbf{Inf}) \Leftarrow$
  – *compute the demanded type of the i-th operand*
  $\mathbf{demanded}(\mathbf{A_i}) = \bigcup \mathbf{demand}(\mathbf{Dem}, \mathbf{i}, \mathbf{Decl})$ ,
  – *check the i-th operand's fixity/precedence restrictions*
  $\mathbf{CheckOperand}(\mathbf{A}_0, \mathbf{A_i}, \mathbf{Op}, \mathbf{i})$ ,
  $\mathbf{type}(\mathbf{result}(\mathbf{A_i})) = \bigcup \mathbf{Inf_i}$ ,
  $\mathbf{CheckOperands}(\mathbf{A}_0, \mathbf{Decl}, \mathbf{Dem}, \mathbf{i}+1, \mathbf{A}, \mathbf{Inf})$ .
$\mathbf{InferenceForOperands}(\mathbf{A}_0, \mathbf{Decl}, \mathbf{Dem}, \mathbf{i}, [\,], [\,])$ .
```

Computing Fixity

The following constraints describe prefix and postfix expressions according to restriction 3.

An empty expression is neither prefix nor postfix.

An non-empty expression is prefix if its first separator is non-empty or if its leftmost operand expression is prefix.

An non-empty expression is postfix if its last separator is non-empty or if its rightmost operand expression is postfix.

Definition 89

```
$\mathbf{ComputePrefix}(\epsilon\ ,\ []\ ,\ \mathbf{false})\ .$
$\mathbf{ComputePrefix}(\mathbf{s}_0\ \_\ \ldots\ \_\ \mathbf{s}_\mathbf{n}\ ,\ [\mathbf{A}_\mathbf{i}]_{\mathbf{i}=1}^{\mathbf{n}}\ ,\ \mathbf{p}) \Leftarrow$
    – *not left-open?*
    $(\mathbf{s}_0 = \epsilon) = \mathbf{false}\ ,$
    $\mathbf{p} = \mathbf{true}\ .$
$\mathbf{ComputePrefix}(\mathbf{s}_0\ \_\ \ldots\ \_\ \mathbf{s}_\mathbf{n}\ ,\ [\mathbf{A}_\mathbf{i}]_{\mathbf{i}=1}^{\mathbf{n}}\ ,\ \mathbf{p}) \Leftarrow$
    $(\mathbf{s}_0 = \epsilon) = \mathbf{true}\ ,$
    $\mathbf{p} = \mathbf{prefix}(\mathbf{A}_1)\ .$
$\mathbf{ComputePostfix}(\epsilon\ ,\ []\ ,\ \mathbf{false})\ .$
$\mathbf{ComputePostfix}(\mathbf{s}_0\ \_\ \ldots\ \_\ \mathbf{s}_\mathbf{n}\ ,\ [\mathbf{A}_\mathbf{i}]_{\mathbf{i}=0}^{\mathbf{n}}\ ,\ \mathbf{p}) \Leftarrow$
    – *not right-open?*
    $(\mathbf{s}_\mathbf{n} = \epsilon) = \mathbf{false}\ ,$
    $\mathbf{p} = \mathbf{true}\ .$
$\mathbf{ComputePostfix}(\mathbf{s}_0\ \_\ \ldots\ \_\ \mathbf{s}_\mathbf{n}\ ,\ [\mathbf{A}_\mathbf{i}]_{\mathbf{i}=0}^{\mathbf{n}}\ ,\ \mathbf{p}) \Leftarrow$
    $(\mathbf{s}_\mathbf{n} = \epsilon) = \mathbf{true}\ ,$
    $\mathbf{p} = \mathbf{postfix}(\mathbf{A}_\mathbf{n})\ .$
```

Adjacent Empty Operands

All adjacent operands must either both be empty or both be non-empty. To operands are adjacent if the separator between them is empty.

Definition 90

```
$\mathbf{AdjacentEmptyOps}([\mathbf{s}_\mathbf{i}]\mathbf{S}\ ,\ [\mathbf{A}_\mathbf{i}\ ,\ \mathbf{A}_{\mathbf{i}+1}]\mathbf{A}) \Leftarrow$
    – *are* $\mathbf{A}_\mathbf{i}$ *and* $\mathbf{A}_{\mathbf{i}+1}$ *adjacent?*
    $(\mathbf{s}_\mathbf{i} = \epsilon) = \mathbf{true}\ ,$
    – *both must be empty or both non-empty*
    $(\mathbf{operator}(\mathbf{A}_\mathbf{i}) = \epsilon) = (\mathbf{operator}(\mathbf{A}_{\mathbf{i}+1}) = \epsilon)\ ,$
    $\mathbf{AdjacentEmptyOps}(\mathbf{S}\ ,\ [\mathbf{A}_{\mathbf{i}+1}]\mathbf{A})\ .$
$\mathbf{AdjacentEmptyOps}([\mathbf{s}_\mathbf{i}]\mathbf{S}\ ,\ [\mathbf{A}_\mathbf{i}\ ,\ \mathbf{A}_{\mathbf{i}+1}]\mathbf{A}) \Leftarrow$
    – *not adjacent, then no restriction* $(\mathbf{s}_\mathbf{i} = \epsilon) = \mathbf{false}\ ,$
    $\mathbf{AdjacentEmptyOps}(\mathbf{S}\ ,\ [\mathbf{A}_{\mathbf{i}+1}]\mathbf{A})\ .$
$\mathbf{AdjacentEmptyOps}([]\ ,\ \mathbf{A})\ .$
```

Operand Restrictions

To be able to express the different constraints on operand expressions, we have to distinguish between the different possible situations an operand expression can occur in. For that we need to know what the corresponding placeholder has directly to its left and right. Each placeholder can have either another placeholder, a terminal symbol or the end of the operator there.

Definition 91

```
TYPE situation==
      open      -- end of operator
      closed     -- terminal symbol
      ph     -- placeholder
FUN leftSituation :
    pattern**nat → situation
FUN rightSituation :
    pattern**nat → situation
```

The constraint **PH** describes the situation of the **I**-th placeholder in an operator **Op**. This information can then be used for further restrictions on the operand expression.

Definition 92

```
PH(Op , I , L , R) ⇐
   L = leftSituation(Op , I) ,
   R = rightSituation(Op , I) .
```

We introduce some constraints to describe the kind and fixity of an operand in a short fashion.

Definition 93

```
Pre(A)     ⇐ prefix(A) = true .
Post(A)    ⇐ postfix(A) = true .
Closed(A)  ⇐ Pre(A) ,  Post(A) .
Basic(A)   ⇐ visible(operator(A)) = true .
Empty(A)   ⇐ (operator(A) = ϵ) = true .
```

The actual constraints on an operand depend on its actual situation of its placeholder in the operator.

- Left-open operands shall be left-precedent
- Right-open operands shall be right-precedent
- Left-Adjacent operands shall be empty or postfix.
- Right-adjacent operands shall be empty or prefix.
- Adjacent operands shall either both be empty or both be basic.
- Operands enclosed between terminal symbols are allowed to be any kind of expression, while other operands must be basic.

These conditions derive from the restrictions imposed in sections 5.5 and 5.6.

Definition 94

$\mathbf{Operand(A_0\ ,A_i\ ,N\ ,\ I)} \Leftarrow$
  $\mathbf{PH(N\ ,I\ ,open\ ,R)\ ,\ LP(A_0\ ,A_i)}$.
$\mathbf{Operand(A_0\ ,A_i\ ,N\ ,\ I)} \Leftarrow$
  $\mathbf{PH(N\ ,I\ ,L\ ,open)\ ,\ RP(A_0\ ,A_i)}$.
$\mathbf{Operand(A_0\ ,A_i\ ,N\ ,\ I)} \Leftarrow$
  $\mathbf{PH(N\ ,I\ ,ph\ ,ph)\ ,\ Basic(A_i)\ ,Closed(A_i)\ ,CH(A_0\ ,A_i)}$.
$\mathbf{Operand(A_0\ ,A_i\ ,N\ ,\ I)} \Leftarrow$
  $\mathbf{PH(N\ ,I\ ,ph\ ,ph)\ ,\ Empty(A_i)\ ,CH(A_0\ ,A_i)}$.
$\mathbf{Operand(A_0\ ,A_i\ ,N\ ,\ I)} \Leftarrow$
  $\mathbf{PH(N\ ,I\ ,closed\ ,ph)\ ,\ Basic(A_i)\ ,Post(A_i)\ ,CH(A_0\ ,A_i)}$.
$\mathbf{Operand(A_0\ ,A_i\ ,N\ ,\ I)} \Leftarrow$
  $\mathbf{PH(N\ ,I\ ,closed\ ,ph)\ ,\ Empty(A_i)\ ,CH(A_0\ ,A_i)}$.
$\mathbf{Operand(A_0\ ,A_i\ ,N\ ,\ I)} \Leftarrow$
  $\mathbf{PH(N\ ,I\ ,ph\ ,closed)\ ,\ Basic(A_i)\ ,Pre(A_i)\ ,CH(A_0\ ,A_i)}$.
$\mathbf{Operand(A_0\ ,A_i\ ,N\ ,\ I)} \Leftarrow$
  $\mathbf{PH(N\ ,I\ ,ph\ ,closed)\ ,\ Empty(A_i)\ ,CH(A_0\ ,A_i)}$.
$\mathbf{Operand(A_0\ ,A_i\ ,N\ ,\ I)} \Leftarrow$
  $\mathbf{PH(N\ ,I\ ,closed\ ,closed)\ ,\ CH(A_0\ ,A_i)}$.

Finally, we have to introduce the constraints **LP** and **RP** describing left- and right-precedence of operands, as well as the constraint **CH** which passes down the **checkHierarchy** flag.

A left-open operand will be allowed as left-precedent **LP** if one of the following conditions is met:

- It is ad-hoc left precedent, but not naturally left precedent. If so, the ad-hoc precedence hierarchies must be checked for natural precedences in the sub-expression.
- It is ad-hoc left precedent and also naturally left precedent.
- It is not ad-hoc left precedent, but naturally left precedent and the ad-hoc precedence hierarchy must not be checked.
- It is not ad-hoc left precedent, but naturally left precedent, the ad-hoc precedence hierarchy must be checked and is followed by the two operators.

The analogous constraints can be defined for right precedence **RP**.

These conditions ensure that there is never any doubt as towards the precedence of the different involved operators in an expression.

Definition 95

$\mathbf{PRECHELP(A_0\ ,A_i\ ,Op\ ,\ S\ ,\ L\ ,\ R\ ,\ T)} \Leftarrow$
 $\mathbf{Op = operator(A_i)\ ,S = scope(A_0)\ ,T = demanded(A_0)}$
 $\mathbf{L = Left_T(S)\ ,R = Right_T(S)}$.
$\mathbf{CH(A_0\ ,A_i)} \Leftarrow$
 $\mathbf{checkHierarchy(A_0) = checkHierarchy(A_i)}$.
$\mathbf{FollowsHierarchy(Op_1\ ,Op_2\ ,S)} \Leftarrow$
 – $\mathbf{Op_1}$ *must be reachable from* $\mathbf{Op_2}$ *in the ad-hoc precedence hierarchy*
 $\mathbf{(Op_1\ ,Op_2) \in (Left_A(S) \cup Right_A(S)^{-1})^+}$.

Definition 96

$\mathbf{LP(A_0\ ,\ A_i) \Leftarrow}$
$\quad\mathbf{PRECHELP(A_0\ ,\ A_i\ ,\ Op\ ,\ \ S\ ,\ \ L\ ,\ \ R\ ,\ \ T)}$,
$\quad\mathbf{(Op\ ,\ operator(A_0)) \in Left_A(S)}$,
$\quad\mathbf{(Op\ ,\ operator(A_0)) \notin LeftPrec_T\ (L\ ,\ R)}$,
$\quad\mathbf{checkHierarchy(A_i) = true}$.
$\mathbf{LP(A_0\ ,\ A_i) \Leftarrow}$
$\quad\mathbf{PRECHELP(A_0\ ,\ A_i\ ,\ Op\ ,\ \ S\ ,\ \ L\ ,\ \ R\ ,\ \ T)}$,
$\quad\mathbf{(Op\ ,\ operator(A_0)) \in Left_A(S)}$,
$\quad\mathbf{(Op\ ,\ operator(A_0)) \in LeftPrec_T\ (L\ ,\ R)}$,
$\quad\mathbf{CH(A_0\ ,\ A_i)}$.
$\mathbf{LP(A_0\ ,\ A_i) \Leftarrow}$
$\quad\mathbf{PRECHELP(A_0\ ,\ A_i\ ,\ Op\ ,\ \ S\ ,\ \ L\ ,\ \ R\ ,\ \ T)}$,
$\quad\mathbf{checkHierarchy(A_0) = false}$,
$\quad\mathbf{(Op\ ,\ operator(A_0)) \notin Left_A(S)}$,
$\quad\mathbf{(Op\ ,\ operator(A_0)) \in LeftPrec_T\ (L\ ,\ R)}$,
$\quad\mathbf{CH(A_0\ ,\ A_i)}$.
$\mathbf{LP(A_0\ ,\ A_i) \Leftarrow}$
$\quad\mathbf{PRECHELP(A_0\ ,\ A_i\ ,\ Op\ ,\ \ S\ ,\ \ L\ ,\ \ R\ ,\ \ T)}$,
$\quad\mathbf{checkHierarchy(A_0) = true}$,
$\quad\mathbf{FollowsHierarchy(Op\ ,\ \ operator(A_0)\ ,\ S)}$,
$\quad\mathbf{(Op\ ,\ operator(A_0)) \notin Left_A(S)}$,
$\quad\mathbf{(Op\ ,\ operator(A_0)) \in LeftPrec_T\ (L\ ,\ R)}$,
$\quad\mathbf{CH(A_0\ ,\ A_i)}$.
$\mathbf{RP(A_0\ ,\ A_i) \Leftarrow}$
$\quad\mathbf{PRECHELP(A_0\ ,\ A_i\ ,\ Op\ ,\ \ S\ ,\ \ L\ ,\ \ R\ ,\ \ T)}$,
$\quad\mathbf{(Op\ ,\ operator(A_0)) \in Right_A(S)}$,
$\quad\mathbf{(Op\ ,\ operator(A_0)) \notin RightPrec_T\ (L\ ,\ R)}$,
$\quad\mathbf{checkHierarchy(A_i) = true}$.
$\mathbf{RP(A_0\ ,\ A_i) \Leftarrow}$
$\quad\mathbf{PRECHELP(A_0\ ,\ A_i\ ,\ Op\ ,\ \ S\ ,\ \ L\ ,\ \ R\ ,\ \ T)}$,
$\quad\mathbf{(Op\ ,\ operator(A_0)) \in Right_A(S)}$,
$\quad\mathbf{(Op\ ,\ operator(A_0)) \in RightPrec_T\ (L\ ,\ R)}$,
$\quad\mathbf{CH(A_0\ ,\ A_i)}$.
$\mathbf{RP(A_0\ ,\ A_i) \Leftarrow}$
$\quad\mathbf{PRECHELP(A_0\ ,\ A_i\ ,\ Op\ ,\ \ S\ ,\ \ L\ ,\ \ R\ ,\ \ T)}$,
$\quad\mathbf{checkHierarchy(A_0) = false}$,
$\quad\mathbf{(Op\ ,\ operator(A_0)) \notin Right_A(S)}$,
$\quad\mathbf{(Op\ ,\ operator(A_0)) \in RightPrec_T\ (L\ ,\ R)}$,
$\quad\mathbf{CH(A_0\ ,\ A_i)}$.
$\mathbf{RP(A_0\ ,\ A_i) \Leftarrow}$
$\quad\mathbf{PRECHELP(A_0\ ,\ A_i\ ,\ Op\ ,\ \ S\ ,\ \ L\ ,\ \ R\ ,\ \ T)}$,
$\quad\mathbf{checkHierarchy(A_0) = true}$,
$\quad\mathbf{FollowsHierarchy(Op\ ,\ \ operator(A_0)\ ,\ S)}$,
$\quad\mathbf{(Op\ ,\ operator(A_0)) \notin Right_A(S)}$,
$\quad\mathbf{(Op\ ,\ operator(A_0)) \in RightPrec_T\ (L\ ,\ R)}$,
$\quad\mathbf{CH(A_0\ ,\ A_i)}$.

6.2.3 Type Annotation

We use the type inference rules introduced for type-annotation expression for the operator _ : _.

Also, the resulting parse tree is the parse tree for the left operand expression, annotated with the type inferred for the whole expression.

Definition 97

$$\begin{aligned}
&\mathbf{Expr([A_0\ ,\ A_1\ ,\ A_2]\ ,\ Op)} \Leftarrow \\
&\quad \mathbf{TypeAnnotation(Op)}\ , \\
&\quad \mathbf{sentence(A_0) = sentence(A_1)}\ \underline{:}\ \mathbf{sentence(A_2)}\ , \\
&\quad \mathbf{demanded(A_0)} = \bigcup \mathbf{Dem_0}\ , \\
&\quad \mathbf{demanded(A_2)} = \bigcup \mathbf{demandTypedRight(Dem_0)}\ , \\
&\quad \mathbf{type(result(A_2))} = \bigcup \mathbf{Inf_2}\ , \\
&\quad \mathbf{demanded(A_1)} = \bigcup \mathbf{demandTypedLeft(Inf_2)}\ , \\
&\quad \mathbf{result(A_1) = S\ .\ Op'Ops} : \bigcup\ \mathbf{Inf_1}\ , \\
&\quad \mathbf{Inf_0 = inferTyped(Inf_1\ ,\ Inf_2)}\ , \\
&\quad \mathbf{result(A_0) = S\ .\ Op'Ops} : \bigcup\ \mathbf{Inf_0}\ .
\end{aligned}$$

6.2.4 Scope Annotation

For a scope annotation expression, the expression to the left side of the dot is parsed in the same scope as the whole expression. The parse result is then added to that scope to yield the scope in which the expression to the right side of the dot is to be parsed.

The demanded type of the annotated expression is equal to the demanded type of the whole expression and the parse result of the whole expression is equal to the parse result of the annotated expression.

Definition 98

$$\begin{aligned}
&\mathbf{Expr([A_0\ ,\ A_1\ ,\ A_2]\ ,\ Op)} \Leftarrow \\
&\quad \mathbf{ScopeAnnotation(Op)}\ , \\
&\quad \mathbf{sentence(A_0) = sentence(A_1)}\ \underline{.}\ \mathbf{sentence(A_2)}\ , \\
&\quad \mathbf{scope(A_1) = scope(A_0)}\ , \\
&\quad \mathbf{scope(A_2) = scope(A_0) \oplus result(A_1)}\ , \\
&\quad \mathbf{demanded(A_2) = demanded(A_0)}\ , \\
&\quad \mathbf{result(A_0) = result(A_2)}\ .
\end{aligned}$$

6.2.5 Parenthesis Operator

The parenthesis operator has two built-in functionalities, identity and section lifting, but could also be overloaded to possess additional ones. All these must be taken into account for the type inference of instantiations of that operator.

Definition 99

$$
\begin{array}{l}
\mathbf{Expr([A_0\ ,\ A_1]\ ,\ Op)} \Leftarrow \\
\quad \mathbf{Paren(Op)}\ , \\
\quad \mathbf{sentence(A_0)} = \underline{(}\mathbf{sentence(A_1)}\underline{)}\ , \\
\quad \mathbf{demanded(A_0)} = \bigcup \mathbf{Dem_0}\ , \\
\quad \mathbf{types(scope(A_0))(\_)} = \bigcup \mathbf{CDecl)}\ , \\
\quad \mathbf{CDem} = \mathbf{Dem_0} \cup \mathbf{demand(Dem_0}\ ,\ 1\ ,\ \mathbf{CDecl)} \cup\ \mathbf{demandLIFT(Dem_0)}\ , \\
\quad \mathbf{types(scope(A_0))(Op)} = \bigcup \mathbf{Decl)}\ , \\
\quad \mathbf{Dem_1} = \mathbf{demand(CDem}\ ,\ 1\ ,\ \mathbf{Decl)} \cup\ \mathbf{demandSection(CDem)}\ , \\
\quad \mathbf{demanded(A_1)} = \bigcup \mathbf{Dem_1}\ , \\
\quad \mathbf{type(result(A_1))} = \bigcup \mathbf{Inf_1}\ , \\
\quad \mathbf{CInf} = \mathbf{infer(CDem\ ,\ Decl\ ,\ [Inf_1])} \cup \mathbf{inferSection(CDem\ ,\ \ Inf_1)}\ , \\
\quad \mathbf{S} = \mathbf{scope(A_0)}\ , \\
\quad \mathbf{Ops} = \mathbf{[result(A_1)]}\ , \\
\quad \mathbf{R} = \mathbf{S\ .\ OpOps} : \bigcup\ \mathbf{CInf}\ , \\
\quad \mathbf{Inf_0} = \mathbf{CInf} \cup \mathbf{infer(Dem_0\ ,\ CDecl\ ,\ [CInf])} \cup\ \mathbf{inferLIFT(Dem_0\ ,\ R)}\ , \\
\quad \mathbf{result(A_0)} = \mathbf{S\ .\ OpOps} : \bigcup\ \mathbf{Inf_0}\ .
\end{array}
$$

6.2.6 Binary Concatenation Operator

Like with adjacent operands of an operator, we impose some restrictions on all built-in binary invisible (concatenation) operators. They shall not be empty (restriction 4), so both their operands shall not be empty. They shall be left-precedent to each other. Their left operand shall be a non-empty postfix expression, and their right operand shall be a non-empty non-concatenation prefix expression (restriction 3).

Definition 100

$$
\begin{array}{l}
\mathbf{Expr([A_0\ ,\ A_1\ ,\ A_2]\ ,\ Op)} \Leftarrow \\
\quad \mathbf{BinaryConcat(Op)}\ , \\
\quad \mathbf{prefix(A_0)} = \mathbf{prefix(A_1)}\ , \\
\quad \mathbf{postfix(A_0)} = \mathbf{postfix(A_2)}\ , \\
\quad \mathbf{operator(A_0)} = \_\ \_\ , \\
\quad \mathbf{Post(A_1)}\ , \\
\quad \mathbf{Pre(A_2)}\ ,\ \mathbf{Basic(A_2)}\ , \\
\quad \mathbf{types(scope(A_0))(\_\ \_)} = \bigcup \mathbf{Decl)}\ , \\
\quad \mathbf{demanded(A_0)} = \bigcup \mathbf{Dem_0}\ , \\
\quad \mathbf{types(scope(A_0))(\_)} = \bigcup \mathbf{CDecl)}\ , \\
\quad \mathbf{CDem} = \mathbf{Dem_0} \cup \mathbf{demand(Dem_0}\ ,\ 1\ ,\ \mathbf{CDecl)} \cup\ \mathbf{demandLIFT(Dem_0)}\ , \\
\quad \mathbf{Dem_1} = \mathbf{demandConcatLeft(CDem\ ,\ Decl)}\ , \\
\quad \mathbf{demanded(A_1)} = \bigcup \mathbf{Dem_1}\ , \\
\quad \mathbf{type(result(A_1))} = \bigcup \mathbf{Inf_1}\ , \\
\quad \mathbf{Dem_2} = \mathbf{demandConcatRight(CDem\ ,\ Decl\ ,\ \ Inf_1)}\ ,\ \mathbf{demanded(A_2)} = \bigcup \mathbf{Dem_2}\ , \\
\quad \mathbf{type(result(A_2))} = \bigcup \mathbf{Inf_2}\ , \\
\quad \mathbf{S} = \mathbf{scope(A_0)}\ , \\
\quad \mathbf{Op} = \mathbf{operator(A_0)}\ , \\
\quad \mathbf{Ops} = \mathbf{[result(A_1)\ ,\ result(A_2)]}\ , \\
\quad \mathbf{CInf} = \mathbf{infer(CDem\ ,\ Decl\ ,\ [Inf_1\ ,\ Inf_2])}\ , \\
\quad \mathbf{R} = \mathbf{S\ .\ Op\ Ops} : \bigcup \mathbf{CInf}\ , \\
\quad \mathbf{Inf_0} = \mathbf{CInf} \cup \mathbf{infer(Dem_0\ ,\ CDecl\ ,\ [CInf])} \cup\ \mathbf{inferLIFT(Dem_0\ ,\ R)}\ , \\
\quad \mathbf{result(A_0)} = \mathbf{S\ .\ Op\ Ops} : \bigcup \mathbf{Inf_0}\ , \\
\quad \mathbf{AllNotEmpty([A_i]_{i=1}^{n})}\ .
\end{array}
$$

6.2.7 Multi Concatenation Operators

For concatenation operators with greater arity than 2, if they are allowed (i.e. no concatenation operators of other arities are allowed), the inference is done as normal, but additionally the restriction that all operands are not empty is enforced.

Definition 101

$$
\begin{aligned}
&\mathbf{Expr}([\mathbf{A_i}]_{\mathbf{i}=0}^{\mathbf{n}}\ ,\mathbf{Op}) \Leftarrow\\
&\quad \mathbf{MultiConcat(Op)}\ ,\\
&\quad \mathbf{sentence}(\mathbf{A}_0) = \mathbf{sentence}(\mathbf{A}_1)\ldots\mathbf{sentence}(\mathbf{A_n})\ ,\\
&\quad \mathbf{Inference}([\mathbf{A_i}]_{\mathbf{i}=0}^{\mathbf{n}}\ ,\mathbf{Op})\ ,\\
&\quad \mathbf{AllNotEmpty}([\mathbf{A_i}]_{\mathbf{i}=1}^{\mathbf{n}})\ .\\
&\mathbf{AllNotEmpty}([\mathbf{A_i}]\mathbf{A}) \Leftarrow\\
&\quad (\mathbf{operator}(\mathbf{A_i}) = \epsilon) = \mathbf{false}\ ,\\
&\quad \mathbf{AllNotEmpty(A)}\ .\\
&\mathbf{AllNotEmpty}([])\ .
\end{aligned}
$$

Chapter 7

Two-Level Grammar Transformations

Commonly used language-preserving grammar transformation schemes [3] exist to remove left-recursion from context-free grammars (CFGs). This makes straightforward automatic derivation of (efficient) top-down parsers or transducers for these languages possible. However, the transformations are normally not applied to the two-level grammars [21] used in the context of parsing natural languages, but only to the underlying CFG, leaving the integration with the second level to the implementor of the actual transducer.

To remedy this, we generalize the common CFG transformations of *left-recursion elimination*, *left factorization*, *unfolding* and also the *left corner transformation* [28] to the concept of two-level grammars with semantic actions. We do this in a very simple way by using the two-level annotation mechanism itself, an approach which we have not found in such a straightforward way in existing literature in regard to the above transformations. For grammars that contain semantic actions [20] [5] at the beginning of right-hand sides of rules we introduce the language-preserving grammar transformation of *action shifting* which is not possible in CFGs without changing the semantic actions. We also show further uses of this transformation in regard to avoidance of backtracking in ambiguous grammars. By canonically introducing only auxiliary nonterminal symbols which are built out of the symbols of the original grammar, all the grammar transformations can be related to each other and possibly even reversed. We hope that this will help readability of the transformed grammar and thus make it better manageable.

With the generalized transformations, we can automatically derive top-down transducers for originally left-recursive two-level grammars. This can be useful both for parsing arbitrary user-defined mixfix operator expressions in programming languages as well as for parsing natural languages with left-recursive constructs efficiently.

7.1 Motivation for Generalizing Grammar Transformations

7.1.1 Why Grammar Transformations?

Grammar transformations for context-free grammars (CFGs) are a well-known and accepted tool in the field of compiler construction for artificial languages as well as natural language recognition in the field of linguistics. They can be used to automatically derive more efficient top-down parsers or transducers for grammars that are designed with the semantics of the language and not so much the parsing paradigm in mind, and as such may sport such features as left-recursion.

7.1.2 Why Top-Down Transducers?

The generalization of top-down parsers to two-level grammars is straightforward. But a preliminary grammar transformation approach to make an arbitrary grammar top-down parsable is often warranted or even needed. Efficiently top-down parsable grammars are for the most part unwieldy, very large, and hard to understand in regard to their semantics, while semantics oriented grammars tend to be much smaller and much easier to understand by the programmer. To work with understandable grammars is important from a software engineering point of view. It allows easier further development of the grammars and keeps them manageable.

Because of their straightforward implementation and in many settings better average performance as opposed to more general parsing algorithms, top-down parsers or transducers derived from grammars can be very useful for prototyping. Also, if the derivation is automatic from the original grammar, only this grammar has to be maintained when the language is developed further.

7.1.3 Why Two-Level Grammars?

In the field of linguistics and artificial intelligence, the concept of two-level grammars ([23], [33]) have had great success for both description purposes as well as for disambiguation purposes in natural languages.

This approach was deemed unnecessary for programming languages as they tend to be designed with unambiguity in mind which makes disambiguation unnecessary as opposed to the situation with natural languages.

Because parsing of natural languages becomes ever more important, the different deterministic parsing approaches have been generalized to deal with ambiguities as efficiently as possible[1], but only in regard to CFGs.

Thus, two-level grammars are mostly only used as a *description* tool for artificial languages [48]. For the actual parsing, other approaches are taken, leaving open the question of equivalence of the implementation with the originally given two-level grammar.

Where two-level grammars are actually used for generation of parsers or programming environments, the grammar writer, though with the help of tools, has to provide the already *transformed* grammar instead of the original (e.g. [25]). Thus, they are highly dependent of the parsing paradigm of the tool. This, of course, makes it harder to work with two-level grammars and hence makes them less acceptable.

7.1.4 What about Semantic Actions?

The same approach is taken for the semantic actions in attribute grammars used by parser generator tools like `yacc` [5].[2]. The attributation is done only *after the actual grammar transformation process* on the CFG. Again, the grammar transformations don't have to deal with the semantic actions, but the programmer.

The equivalent grammar transformations of *left-recursion elimination*, *unfolding* and *left factorization* can deal with semantic actions in a large class of context-free

[1]An interesting survey over generalizations of parsing techniques to ambiguous grammars is given by Nederhof [29].

[2]The relation between attribute grammars and affix grammars is explored by [23]

grammars without change of the input/output semantics of the grammar[3]. But there are also grammars, for instance containing hidden left recursion, where such transformations are simply not possible without changing the original semantic actions or the parsing algorithm. Because of this, such constructions are normally avoided in language design by restricting the languages to those syntactic constructs that can be dealt with automatically.

For all cycle-free CFGs, where this problem cannot be solved statically, this flaw can be remedied by using a two-level grammar approach and the concept of *spine*-annotations together with the additional *action shifting* transformation which requires a second parsing phase for the transduction.

We generalize all needed transformations to pay respect to the two-level aspect of the grammars. In doing so, we become able to deal with almost all kinds of two-level grammars, not only those attributed with spines.

7.1.5 Usability

As has been shown in the field of parsing of natural languages, by interweaving parsing and type checking through the annotation mechanism, disambiguation and rejection of type-incorrect or ambiguous parse trees can often be done *during parsing* locally, as soon as possible. If the grammar is known to be unambiguous (in the sense that it yields at most one type correct parse tree for every token stream), backtracking can be kept to a minimum. This is even more so the case, since the type attributes often can be used for directing the parsing process when a mere fixed look-ahead is not sufficient.

The transformations shown here, though motivated by the need to parse mixfix expressions, could of course also benefit the prototyping of parsers for natural languages derived from left-recursive two-level grammars given for those.

Additionally, since LR-parsers or other tabular parsing algorithms cannot deal well with hidden left-recursion, our given transformations could help to remove hidden left-recursion from grammars intended for such algorithms.

7.2 Generalized Transformations

In the following section, we will first show the usual CFG variant of each transformation and then generalize it to be usable for our two-level grammars.

To be equivalent transformations, the generalized transformation schemes have to take the variable sets, as well as the annotations and constraints of the production rules to be transformed into account.

7.2.1 Unfolding

Unfolding of nonterminals is sometimes needed both for elimination of indirect and hidden left-recursion and left factorization.

The transformation is a derivation for a nonterminal on the right-hand side of a rule with all its right-hand sides, respectively, yielding a new set of unfolded rules.

Say, $u, v, w_i \in (V \times T_\Sigma(V_V)^\star)^\star$, $N_1, N_2 \in V_N$. Then, the unfolding transformation scheme for unfolding the nonterminal N_2 with rules $N_2 ::= w_i$ in rule $N_1 ::= u\ N_2\ v$ works as follows in the underlying one-level CFG:

[3]This is a fact often used in combinator parsing where the original grammar is automatically decorated with semantic actions that build the parse trees.

$$\begin{array}{lll} N_1 & ::= & u\ N_2\ v \\ N_2 & ::= & w_i \\ & \Longrightarrow & \\ N_1 & ::= & u\ w_i\ v \\ N_2 & ::= & w_i \end{array}$$

By carefully annotating the above scheme with the needed corresponding quantifiers, annotations and constraints from the definition of derivation, we arrive at the following (where $\Gamma'_i = \Gamma \cup \sigma_i(\Gamma_i)$ and $C'_i \equiv C \wedge \overline{\sigma_i}(C_i) \wedge \overline{\sigma_i}(T_i) = T'$):

$$\begin{array}{llll} \forall\Gamma|C. & N_1(T) & ::= & u\ N_2(T')\ v \\ \forall\Gamma_i|C_i. & N_2(T_i) & ::= & w_i \\ & \Longrightarrow & & \\ \forall\Gamma'_i|C'_i. & N_1(T) & ::= & u\ \overline{\sigma_i}(w_i)\ v \\ \forall\Gamma_i|C_i. & N_2(T_i) & ::= & w_i \end{array}$$

7.2.2 Direct Left-Recursion Elimination

Left-recursion ($N \Rightarrow^+ N\ w$) should be eliminated from grammars to make the language top-down parsable and avoid termination problems. All indirectly or hidden left-recursive rules can be transformed into directly left-recursive rules by unfolding and action shifting, so it suffices to give only a transformation for direct left-recursion elimination.

As a side-note, it must be taken care that the unfolding process to arrive at only directly left-recursive rules can terminate. To achieve this, the rules must be unfolded in topological order, meaning that a rule for nonterminal N_1 must be unfolded before N_2, if $N_2 \Rightarrow^+ N_1\ w$. If also $N_1 \Rightarrow^+ N_2\ w'$, the two nonterminals are cyclically dependent and the respective order of their unfoldinig is irrelevant.[4]

Suppose we have a set of production rules of nonterminal N (where all v_i are not of the form $N\ w$). Then, the following transformation scheme eliminates the directly left-recursive rules for N from the grammar. Towards this end, we have to introduce a new auxiliary nonterminal symbol $[N, N]$[5] (representing the rest of an N after an N at the beginning):

$$\begin{array}{lll} N & ::= & v_i \\ N & ::= & N\ w_j \\ & \Longrightarrow & \\ N & ::= & v_i\ [N,N] \\ [N,N] & ::= & w_j\ [N,N] \\ [N,N] & ::= & \epsilon \end{array}$$

In our two-level grammar setting, the annotations seem to pose a difficult problem as the first derivation step which also takes care of the first variable substitution on the substituted nonterminal, now becomes the last derivation step and vice versa. Thus, the annotations inside the v_i *at the beginning* is determined by the occurrences of w_j *at the end.*

Example 78 *Take the following simple grammar:*

$$\begin{array}{lllll} \forall\{\alpha\} & . & E(\mathbf{seq}\,\alpha) & ::= & \underline{\mathtt{s}} & \mathbf{0}(\mathbf{seq}\,\alpha) \\ \forall\{\alpha\} & . & E(\alpha) & ::= & E(\mathbf{seq}\,\alpha)\ \underline{\mathtt{ft}} & \mathbf{1}(\alpha) \end{array}$$

[4]Basically, we need to compute the strongly-connected-components and sort these topologically according to the above relation between nonterminals.

[5]which is traditionally used in the description of left corner parsing [38]

This makes the following derivation possible:

$$\forall\{\alpha\}.E(\alpha) \Rightarrow^\star \forall\{\alpha\}.\underline{\mathtt{s}}\ \mathbf{0}(\mathbf{seq}^n\alpha)\ (\underline{\mathtt{ft}}\ \mathbf{1}(\mathbf{seq}^{n-i}\alpha))_{i=1}^n$$

So, the number of occurrences of $\underline{\mathtt{ft}}$ at the end *decides about the annotation of the action* $\mathbf{0}$ at the beginning.

We achieve this by annotating the auxiliary nonterminal symbol $[E, E]$ with the two annotations corresponding to the two Es in it [6]. In the ϵ-production of $[E, E]$, these two annotations must of course be equal as the left corner is already the whole tree.

This generalizes the CFG transformation to the following (where $\Gamma'_i = \Gamma_i \uplus \{\tau_i\}$ and $\Gamma'_j = \Gamma_j \uplus \{\tau_j\}$):

$$\begin{array}{lll}
\forall\Gamma_i|C_i. & N(T_i) & ::= \quad v_i \\
\forall\Gamma_j|C_j. & N(T_j) & ::= \quad N(T'_j)\ w_j \\
 & \Longrightarrow & \\
\forall\Gamma'_i|C_i. & N(\tau_i) & ::= \quad v_i\ [N,N](\tau_i, T_i) \\
\forall\Gamma'_j|C_j. & [N,N](\tau_j, T'_j) & ::= \quad w_j\ [N,N](\tau_j, T_j) \\
\forall\{\tau\}. & [N,N](\tau,\tau) & ::= \quad \epsilon
\end{array}$$

As we can see, the transformation is done for each rule separately without taking annotation information from other rules into account.

Example 79 *In our example grammar, this transformation looks as follows:*

$$\begin{array}{llll}
\forall\{\alpha\}. & E(\mathbf{seq}\,\alpha) & ::= & \underline{\mathtt{s}}\ \mathbf{0}(\mathbf{seq}\,\alpha) \\
\forall\{\alpha\}. & E(\alpha) & ::= & E(\mathbf{seq}\,\alpha)\ \underline{\mathtt{ft}}\ \mathbf{1}(\alpha) \\
 & & & \\
 & & \Longrightarrow & \\
\forall\{\tau,\alpha\}. & E(\tau) & ::= & \underline{\mathtt{s}}\ \mathbf{0}(\mathbf{seq}\,\alpha)\ [E,E](\tau, \mathbf{seq}\,\alpha) \\
\forall\{\tau,\alpha\}. & [E,E](\tau, \mathbf{seq}\,\alpha) & ::= & \underline{\mathtt{ft}}\ \mathbf{1}(\alpha)\ [E,E](\tau,\alpha) \\
\forall\{\tau\}. & [E,E](\tau,\tau) & ::= & \epsilon
\end{array}$$

Since combinator parsers will normally implement each of the rules as a sequence of function calls, the resulting right-recursive calls can be optimized with tail-recursion, minimizing call-stack-depth.

7.2.3 Left Factorization

The most efficient top-down parsers are those that are deterministic for a minimal look-ahead, so the look-ahead necessary to decide which production rule to use for the derivation of next nonterminal should be minimized which can be done via left factorization.

Unfolding, left-recursion elimination and also action shifting might be necessary before the left factorization is applicable on rules that derive strings with the same prefix.

Also, the iterative application of left factorization and unfolding can lead to non-termination if $N \Rightarrow^+ v\ N\ w$. Therefore, no further unfolding (and subsequent left factorization) should be done for the nonterminal N in derived rules of the form $[N, s_1, \ldots, s_n] ::= N\ w$.

[6] We give the annotations the same order as the symbols in the composite nonterminal symbol – this will make sense when looking at the other transformations. Basically, $[N, s](T_1, T_2)$ should be read as $[N(T_1), s(T_2)]$, i.e. the rest of an N with annotation T_1 after the beginning s with annotation T_2.

Again, we orient ourselves towards the normal left factorization transformation scheme on CFGs (which introduces the new auxiliary nonterminal $[N,s]$ for symbols N and s):

$$\begin{array}{ll} N & ::= s\ v_i \\ & \Longrightarrow \\ N & ::= s\ [N,s] \\ [N,s] & ::= v_i \end{array}$$

We can generalize this scheme to:

$$\begin{array}{llcl} \forall \Gamma_i | C_i. & N(T_i) & ::= & s(T_i')\ v_i \\ & & \Longrightarrow & \\ \forall \{\tau, \tau'\}. & N(\tau) & ::= & s(\tau')\ [N,s](\tau,\tau') \\ \forall \Gamma_i | C_i. & [N,s](T_i, T_i') & ::= & v_i \end{array}$$

In case of not annotated factorized symbols, we *can* use the following simplification:

$$\begin{array}{llcl} \forall \Gamma_i | C_i. & N(T_i) & ::= & s\ v_i \\ & & \Longrightarrow & \\ \forall \{\tau\}. & N(\tau) & ::= & s\ [N,s](\tau) \\ \forall \Gamma_i | C_i. & [N,s](T_i) & ::= & v_i \end{array}$$

However, this simplification might make the grammar less readable as then it is not so obvious anymore, how the annotations relate to the symbols.

7.2.4 Action Shifting

Because semantic actions are allowed anywhere in our grammar formalism, it is possible that there are rules, either already in the original or in the transformed grammar, that have leading semantic actions before any other symbol.

This, obviously, hinders the further necessary transformation process both for left-recursion elimination as well as left factorization.

Example 80 *Consider the following, albeit artificial, one-level grammar, containing hidden left-recursion:*

$$\begin{array}{lcl} A & ::= & \mathbf{0} \\ B & ::= & \underline{\mathtt{a}}\ \mathbf{1} \\ B & ::= & A\ B\ \underline{\mathtt{b}}\ \mathbf{2} \\ B & ::= & A\ B\ \underline{\mathtt{c}}\ \mathbf{3} \end{array}$$

Because A is basically an epsilon-production, B is hidden left-recursive which is yielded by unfolding A in the productions of B.

$$\begin{array}{lcl} A & ::= & \mathbf{0} \\ B & ::= & \underline{\mathtt{a}}\ \mathbf{1} \\ B & ::= & \mathbf{0}\ B\ \underline{\mathtt{b}}\ \mathbf{2} \\ B & ::= & \mathbf{0}\ B\ \underline{\mathtt{c}}\ \mathbf{3} \end{array}$$

Clearly, if B is the starting symbol, action **0** *must occur as many times at the beginning (without consuming any terminal symbols) as there are terminal symbols $\underline{\mathtt{b}}$ and $\underline{\mathtt{c}}$ occurring at the end.*

Simply removing the leading actions or shifting them elsewhere would change the order in which the actions would appear, thereby destroying the input/output semantics of the grammar.

To cope with this situation, we can introduce an attribute for all the nonterminals and actions: the position where the corresponding node will appear in the parse tree, called its *spine*. It can be encoded as the path from the root of the tree to the respective node as a list of numbers, each number representing the child-position in regard to its parent node. This attribute must be introduced into the original grammar *before all other transformations take place* and can then be treated just like an additional, normal attribute alongside others, if any.

By merit of this attribute, we can safely change the position of semantic actions (without further need of changing any of their attributes) inside the rules, as they carry their final position information with them.

Although the semantic actions would not appear in their proper order during transduction, if the resulting grammar is interpreted by a simple top-down transducer, they can be accumulated from the result and sorted lexically by their position attribute, making it possible to reconstruct the order intended by the original grammar.

Example 81 *In our example, we would first introduce the position annotation by numbering the nonterminal symbols in the right-hand-side of each rule from left to right:*

$$\begin{array}{lrcl}
\forall\{\pi\}. & A(\pi) & ::= & \mathbf{0}(\pi) \\
\forall\{\pi\}. & B(\pi) & ::= & \underline{\mathtt{a}}\ \mathbf{1}(\pi) \\
\forall\{\pi\}. & B(\pi) & ::= & A(\pi.1)\ B(\pi.2)\ \underline{\mathtt{b}}\ \mathbf{2}(\pi) \\
\forall\{\pi\}. & B(\pi) & ::= & A(\pi.1)\ B(\pi.2)\ \underline{\mathtt{c}}\ \mathbf{3}(\pi)
\end{array}$$

Unfolding yields the following grammar:

$$\begin{array}{lrcl}
\forall\{\pi\}. & A(\pi) & ::= & \mathbf{0}(\pi) \\
\forall\{\pi\}. & B(\pi) & ::= & \underline{\mathtt{a}}\ \mathbf{1}(\pi) \\
\forall\{\pi,\pi'\}|\pi' = \pi.1. & B(\pi) & ::= & \mathbf{0}(\pi')\ B(\pi.2)\ \underline{\mathtt{b}}\ \mathbf{2}(\pi) \\
\forall\{\pi,\pi'\}|\pi' = \pi.1. & B(\pi) & ::= & \mathbf{0}(\pi')\ B(\pi.2)\ \underline{\mathtt{c}}\ \mathbf{3}(\pi)
\end{array}$$

Whenever the annotations of the nonterminals to be unified don't yield mutually recursive equations all new constraints can be removed by using variable substitution according to these constraints. [7]

$$\begin{array}{lrcl}
\forall\{\pi\}. & A(\pi) & ::= & \mathbf{0}(\pi) \\
\forall\{\pi\}. & B(\pi) & ::= & \underline{\mathtt{a}}\ \mathbf{1}(\pi) \\
\forall\{\pi\}. & B(\pi) & ::= & \mathbf{0}(\pi.1)\ B(\pi.2)\ \underline{\mathtt{b}}\ \mathbf{2}(\pi) \\
\forall\{\pi\}. & B(\pi) & ::= & \mathbf{0}(\pi.1)\ B(\pi.2)\ \underline{\mathtt{c}}\ \mathbf{3}(\pi)
\end{array}$$

Shifting these leading actions behind the terminal symbol in the left-recursive rules, we get:

$$\begin{array}{lrcl}
\forall\{\pi\}. & A(\pi) & ::= & \mathbf{0}(\pi) \\
\forall\{\pi\}. & B(\pi) & ::= & \underline{\mathtt{a}}\ \mathbf{1}(\pi) \\
\forall\{\pi\}. & B(\pi) & ::= & B(\pi.2)\ \underline{\mathtt{b}}\ \mathbf{0}(\pi.1)\ \mathbf{2}(\pi) \\
\forall\{\pi\}. & B(\pi) & ::= & B(\pi.2)\ \underline{\mathtt{c}}\ \mathbf{0}(\pi.1)\ \mathbf{3}(\pi)
\end{array}$$

Now, the normal left-recursion elimination scheme can be applied again.

$$\begin{array}{llcl}
\forall\{\pi\}. & A(\pi) & ::= & \mathbf{0}(\pi) \\
\forall\{\pi,\pi'\}. & B(\pi') & ::= & \underline{\mathtt{a}}\ \mathbf{1}(\pi)\ [B,B](\pi',\pi) \\
\forall\{\pi,\pi'\}. & [B,B](\pi',\pi.2) & ::= & \underline{\mathtt{b}}\ \mathbf{0}(\pi.1)\ \mathbf{2}(\pi)\ [B,B](\pi',\pi) \\
\forall\{\pi,\pi'\}. & [B,B](\pi',\pi.2) & ::= & \underline{\mathtt{c}}\ \mathbf{0}(\pi.1)\ \mathbf{3}(\pi)\ [B,B](\pi',\pi) \\
\forall\{\pi\}. & [B,B](\pi,\pi) & ::= & \epsilon
\end{array}$$

It is interesting to note that the spine-attribute is actually not necessary if the semantic actions explicitly refer to their input and output states (as would be the

[7]So, when this condition holds, we always remain with constraint-free production rules if we started with them in the first place.

case in DCGs of PROLOG, for instance) as then the input/output semantics of the grammar is immune to action shifting already. In our grammar formalism, we also could use constraints on the rules (which have no position) to formulate such relationships between the different constituents of a rule.

7.2.5 Left Corner Transformation

As noted by Moore [28], the normal left-recursion elimination transformation (and in conclusion our generalized version of it) does not always yield optimal (or even very acceptable) grammars in case of very large grammars for natural languages. Instead, he proposes the use of the left corner transformation on the left-recursive rules of the grammar which, in combination with left factorization yields much more acceptable grammars.

To understand the transformation, we need the following definitions.

Definition 102 *A* retained nonterminal symbol *is a nonterminal symbol reachable from the nonterminals of the original grammar, i.e.* $\{N | N' \in V_N \wedge N' \Rightarrow^+ v\ N\ w\}$ *(where* V_N *is the set of nonterminal symbols in the original grammar).*

Definition 103 *A symbol* s *is a* left corner *of a nonterminal symbol* N *if* $N \Rightarrow^+ s\ v$.

Thus, not hidden left-recursive nonterminals are always left corners of themselves. Hidden left-recursive nonterminals are only left corners of themselves, if they are not hidden by semantic actions.

The transformation scheme given by Moore, can also easily and naturally be generalized to apply to our generic grammars, yielding the following scheme:

1. If a terminal, action or non-left-recursive nonterminal symbol s is a left corner of a retained left-recursive nonterminal symbol N in the original grammar, add:

   $\forall\{\tau, \tau'\}.N(\tau) ::= s(\tau')\ [N, s](\tau, \tau')$

2. If N_2 is a left-recursive left corner of a retained left-recursive nonterminal symbol N_1, production $\forall\Gamma|C.N_2(T) ::= s(T')\ v$ is in the original grammar, add (where $\tau \notin \Gamma$):

   $\forall\{\tau\} \cup \Gamma|C.[N_1, s](\tau, T') ::= v\ [N_1, N_2](\tau, T)$

3. If a symbol s is a left corner of a retained left-recursive nonterminal symbol N and $\forall\Gamma|C.N(T) ::= s(T')\ v$ is a production of the original grammar, add:

   $\forall\Gamma|C.[N, s](T, T') ::= v$

4. Keep all productions of non-left-recursive nonterminals of the original grammar.

Example 82 *We will show this alternative approach on (part of) our example grammar (where the necessary action shifting has already taken place):*

$$
\begin{array}{llll}
\forall\{\pi\}. & A(\pi) & ::= & \mathbf{0}(\pi) \\
\forall\{\pi\}. & B(\pi) & ::= & \underline{\mathsf{a}}\ \mathbf{1}(\pi) \\
\forall\{\pi\}. & B(\pi) & ::= & B(\pi.2)\ \underline{\mathsf{b}}\ \mathbf{0}(\pi.1)\ \mathbf{2}(\pi) \\
 & \Longrightarrow & & \\
1. & & & \\
\forall\{\tau,\tau'\}. & B(\tau) & ::= & \underline{\mathsf{a}}\ [B,\underline{\mathsf{a}}](\tau,\tau') \\
2. & & & \\
\forall\{\pi,\tau\}. & [B,\underline{\mathsf{a}}](\tau,()) & ::= & \mathbf{1}(\pi)\ [B,B](\tau,\pi) \\
\forall\{\pi,\tau\}. & [B,B](\tau,\pi.2) & ::= & \underline{\mathsf{b}}\ \mathbf{0}(\pi.1)\ \mathbf{2}(\pi) \\
 & & & [B,B](\tau,\pi) \\
3. & & & \\
\forall\{\pi\}. & [B,\underline{\mathsf{a}}](\pi,()) & ::= & \mathbf{1}(\pi) \\
\forall\{\pi\}. & [B,B](\pi,\pi.2) & ::= & \underline{\mathsf{b}}\ \mathbf{0}(\pi.1)\ \mathbf{2}(\pi) \\
4. & & & \\
\forall\{\pi\}. & A(\pi) & ::= & \mathbf{0}(\pi)
\end{array}
$$

Obviously, this grammar transformation also is combinable with the other transformations presented in this paper, and, thus can also deal with hidden left-recursion by use of action shifting.

It is interesting to note, that the left corner transformation combines left factorizations, unfoldings, foldings and left recursion elimination into one complex transformation by using the left-corner relationship.

For directly recursive nonterminal symbols, almost the same grammar is produced (where, e.g. $[B, B]$ plays even the same role in both), but the left corner transform handles mutually left-recursive nonterminals better.

7.3 Optimizations

Using the above generalized transformation schemes, some optimizations come to mind.

For instance, we can avoid some backtracking in the top-down parsing process by delaying semantic actions through unfolding of mutually recursive rules and shifting.

Because unfolding generally can lead to an exponential explosion of the grammar (and often at least to a quadratic amount of rules), a grammar transformation which we would call *partial unfolding* can be beneficial.

7.3.1 Partial Unfolding

Using nonterminal disjunctions $(N_1|\cdots|N_n)$, a construct which is a subset of EBNF, we can express partial unfolding of nonterminals. Nonterminal disjunctions are treated like nonterminals. They have implicit rules of the form $(\cdots|N_i|\cdots) ::= N_i$.

Partial unfolding of the jth rule of a nonterminal E looks like this:

$$\begin{array}{ll}
E & ::= v_i \\
N & ::= u\ E\ w \\
 & \Longrightarrow \\
E_i & ::= v_i \\
E & ::= (E_1|\cdots|E_n) \\
N & ::= u\ E\ w \\
 & \Longrightarrow \\
E_i & ::= v_i \\
E & ::= (E_1|\cdots|E_n) \\
N & ::= u\ (E_1|\cdots|E_{j-1}|E_{j+1}|\cdots|E_n)\ w \\
N & ::= u\ E_j\ w
\end{array}$$

First, the rules of the partially to be unfolded nonterminal each get their own new specialized nonterminal and we add a rule for the original nonterminal with a disjunction of the specialized nonterminals as its right-hand side.

Then we can replace the nonterminal to be unfolded by rules that instead contain only partial disjunctions of these specializations. If every one of the specializations appears in exactly one of the resulting rules, we have a language-preserving transformation. [8]

Generalizing this scheme is a simple replacement task. Whenever an annotated nonterminal is replaced by a disjunction, this disjunction is instead annotated with the same annotation the replaced nonterminal had.

7.3.2 Avoiding Backtracking by Action Shifting

Whenever the underlying CFG of the original grammar has a nonterminal N that is both left-recursive ($N \Rightarrow^+ N\ w\ \mathbf{1}$) and right-recursive ($N \Rightarrow^+ v\ N\ \mathbf{2}$), the CFG is ambiguous ($N \Rightarrow^+ N\ w\ \mathbf{1} \Rightarrow^+ v\ N\ \mathbf{2}\ w\ \mathbf{1}$ and $N \Rightarrow^+ v\ N\ \mathbf{2} \Rightarrow^+ v\ N\ w\ \mathbf{1}\ \mathbf{2}$) so the two-level grammar might be ambiguous as well if the annotations do not resolve the ambiguity.

Shifting the semantic actions in the above case to the end and ordering them the same way during parsing we can avoid backtracking over w by factoring out the common part $v\ N\ w\ \mathbf{1}\ \mathbf{2}$.

Without action shifting, we obviously could only have factored out the common part $v\ N$.

7.4 Cyclic Grammars

Left-recursion elimination can only be successfully applied to grammars that do not contain a cycle ($N(T_1) \Rightarrow^+ N(T_2)$). Otherwise, the algorithm would not terminate. If a grammar contains a cycle, infinite ambiguities might be possible. This is why the converter operators in our mixfix expression language are restricted in such a way that they do not cause such cyclic rules to be introduced into the grammar.

Even though rules that contain a cycle are nonsensical in CFGs (as they don't add any information) and thus can be safely removed from the grammar, unfortunately, this is not necessarily true for general two-level grammars as the left-hand side can carry a different annotation from the right-hand side and also the right-hand side may contain semantic actions.

Though top-down backtrack parsers can cope with such grammars, too, their termination cannot in general be guaranteed unless such rules are not present in the grammar.

[8]This approach is similar to character class grammars of [44].

Simple cyclic rules of the above form, can, if the annotations are the same, of course, still be safely removed from our two-level grammars.

7.5 Conclusion

We have generalized the common grammar transformations of unfolding, left-recursion elimination, and left factorization on CFGs, as well as the left corner transformation [28] towards two-level grammars [48], [21], [22], [23].

This allows the automatic generation of top-down parsers for languages defined by arbitrary two-level grammars (e.g. for natural languages or left-associative mixfix operators).

We dealt with the problem of leading semantic actions of otherwise (hidden) left-recursive rules by using spine attributes and action shifting, thereby abolishing the need of changing the semantic actions.

We only introduce equations as constraints which can be solved by use of unification. Hence, the DCGs [33], [34], of PROLOG [7] would be the natural candidate for the efficient generation of parsers from such two-level grammars. Using combinator parsers in a functional language environment would be just as easy.

The transformations introduce canonically derivable nonterminal symbols from the symbols of the original grammar, thereby ensuring disjointness with the existing set (or, if the same symbol is used, automatic merging), allowing for some modularity in the transformation approach.

Since our auxiliary nonterminal symbols carry the semantics of the original grammar, even the transformed grammar will probably remain humanly readable and thus better manageable.

7.6 Related Work

One prominent approach to tackling two-level grammars from a parsing point of view is AGFL (Affix Grammars over Finite Lattices) [2]. There, left-recursion is dealt with by using left corner or bottom-up parsers [30]. Grammar transformations are only performed on the underlying CFG and the integration with the second level is left to the imagination of the reader.

If at all, even if annotation of the underlying grammar is to take place, the problem of left-recursion removal is mostly solved by referring to the standard approach, letting the programmer first transform the underlying CFG and then worry about the combination with the annotation techniques. How this combination has to be carried out is left to the imagination of the reader by the respective authors.

Nederhof [29] solves the left-recursion problem for top-down parsers of attributed DCGs dynamically by enhancing the given DCG with checks for cyclic calls and continuations, deriving a so called *cancellation parser*. This approach is only aimed at (and suited for) top-down parsers, and so can't be seen as a generalization of standard grammar transformations[9].

Ridoux [36] has formally generalized grammar transformations for two-level grammars by transforming them into λ-Prolog clauses, splitting the two-level rules into their context-free part and a constraint on the attributes of the constituents. The result seems much more complicated than ours. The problem of leading semantic actions is not explicitly addressed by him.

Dymetman [9] generalizes the transformation of a CFG into the Greibach Normal Form for DGCs. There, the term-annotations have to be changed into variables

[9]Our transformation schemes can be employed in any kind of parser environment.

and the original terms must be transferred into the definition of their respective semantic predicates. In [10], Dymetman proposes grammar transformations for DCGs closer to ours, introducing meta-nonterminal symbols that implement a general goal-corner transformation and transforming the DCG into a generic grammar where the former nonterminals become the attributes of the meta-nonterminals. However, this approach for some reason makes epsilon removal necessary which is not true in our approach. The leading semantic action problem is also ignored.

Grootjen [15] maps the AGFL formalism into ATNs (Augmented Transition Networks), but uses grammar transformations on the corresponding affix grammar as an explanatory device, using the same left-recursion elimination scheme as Dymetman without formalization and also ignoring the leading semantic action problem.

Finally, Lohmann et. al [24] consider automatic semantic action migration for left recursion removal in attribute grammars. Since the semantic actions are treated separate from the context-free part[10], their position is irrelevant and no action shifting has to take place. The transformation process only uses the normal left recursion elimination approach and focusses more on the change of the semantic actions in a seemingly more complicated way. The finally resulting grammars look very similar, though.

[10]like our constraints

Chapter 8

Implementation Results

8.1 Mixfix Parsing Algorithm

We have implemented the parsing process for mixfix expressions according to the two-level grammars described in chapter 6 with the help of an Earley parser. For each expression to be parsed, such a parser is instantiated with the underlying possibly ambiguous context-free grammar. This grammar corresponds to the operator-set of the expression, determined from the scope of the expression and the backbones occurring in the expression.

8.1.1 Earley Backbone Parsing

Thus, before the actual parsing of a sub-expression, we invoke a backbone parser which is also implemented via an Earley parser with the appropriate backbone grammar on that sub-expression.

This is also done to determine whether or not there are possible backbone ambiguities and to find out which operators have to be ordered topologically for that sub-expression. These so-called *sibling* operator backbones are used to restrict the operator set in all precedence related constraints.

8.1.2 Top-Down Backtrack Parsing

Derivation

The result of the Earley mixfix parser is a representation of all possible derivations of the start symbol given to that parser to the sentence to be parsed. This representation allows us for each sub-expression to iterate over all operators the expression can be an instantiation of. For every such possible instantiation, all segmentations of the sentence of the sub-expression that match the corresponding rule of the operator can be iterated over. Every such segmentation corresponds to a node in a possible parse tree.

We start to iterate over possible root-nodes for the mixfix expression (using start symbol **COLON**), working our way down to their children until finally we find nullary operator nodes or group phrases which have no children. We reject all nodes that do not fulfil the constraints of the corresponding two-level grammar rule and those which have no acceptable children for at least one nonterminal.

Look-Ahead

This algorithm implements in essence a recursive descent backtrack parser. We traverse the node-space in the same fashion as such a parser would. But the look-ahead is not made of tokens. Instead we determine the possible sub-nodes which are consistent both type-wise and precedence-wise and do not violate the adjacent-operand restrictions.

We do not have the same termination problems as a normal recursive descent parser would have with our grammars, because our restrictions ensure that the sentence of every sub-node is always at least one symbol shorter than the sentence of its parent node or it is the sub-node of an acyclic chain-production rule. All segmentation possibilities are already available from the Earley mixfix parse tree representation.

Groups and Scoping

Whenever a scope-annotation node is matched, first the scope subexpression is parsed and evaluated. From that, the scope in which the annotated expression is to be parsed is computed. The parsing of the annotated expression is done with a new mixfix parser in that scope with start symbol **E**.

Whenever a group phrase is to be parsed, a *group parser* is invoked which in turn invokes mixfix parsers for all the declarations inside the group. We will not go into further detail how this is achieved as this is beyond the scope of this thesis. Suffice it to say that a powerful scope inference algorithm including group import has been successfully integrated with our mixfix parser. For further details, see [37].

Performance of the Earley Parser

It is a well-known fact that Earley's parsing algorithm has a worst case performance that needs a cubic amount of computation steps and a quadratic amount of space dependent on the length of the expression to be parsed.

Thus, our use of such a parser limits us in such a way that our parsing algorithm in general can never be better (in the worst case) than the Earley algorithm. Our choice of parser thus might appear questionable, but all generalized parsing algorithms that can deal with arbitrary context-free grammars like the ones we have to deal with suffer from the same defects as the Earley parser (see [29]), though they might be more efficient in some cases.

In accordance with this, we have found that – using all the optimizations covered in the remainder of this chapter – the Earley parsing phase to find the possible matches takes up the bulk amount of computation time for longer expressions.

In our implementation, we used a very simple, straightforward version of the Earley parser although there are surely heuristic optimizations available for the algorithm in general the addition of which our general parsing algorithm could benefit from.

Also, by integrating the parsing and the matching phase more tightly, using type and precedence information already to restrict possible top-down restrictions of the Earley parser itself, we could probably gain another significant performance boost.

Using memoization of the Earley parsing results for sub-expressions is an optimization that has been implemented successfully, so that this parsing effort at least only occurs at most once.

8.2 Mixfix Parsing Optimizations

8.2.1 Top-Down Restrictions for Adjacent Operators

Whenever we try to find a match for an operator instantiation that has adjacent operands, we can exclude all matches delivered by the Earley parser which violate the restrictions on adjacent operands, i.e.

- all matches where an empty operand expression would be adjacent to a non-empty one,
- all matches of a concatenation operator where one of the operand expressions would be the empty expression.

This way, we do not only restrict the matches to be tried by a significant number, but also avoid a termination problem that we would be faced with if such restrictions were not applied.

Consider concatenation expressions where the left operand expression would be allowed to be empty. Then, the right operand expression could again be a concatenation expression, spanning the whole expression, and could again be split into an empty and a non-empty one, ad infinitum.

8.2.2 Allowed Operand Operators

Because of our top-down bottom-up type inference algorithm, we always know the possible demanded result types of the expression to be matched. If we try to match an instantiation of an operator, we can thus compute the possible demanded result types for all operand expressions to be matched.

This allows us to filter the set of known operands occurring as possible top-level operator of every operand tree in question (which we have determined via backbone parsing) so that only those operators remain that have the respective demanded result type.

Once we have entered a subtree where precedence is only determined via ad-hoc precedence (because some operator on the path to the root was chosen by ad-hoc precedence because natural precedence couldn't be established), we can also filter the operator set of leftmost and rightmost operands by consulting the ad-hoc precedence relations. Only those operators allowed by these relations need to be tried as roots of the respective operand expressions.

If for *one* operand this filtered set of allowed operand operators is empty, the *whole* match for that operator instantiation is impossible and we do not need to try to derive *any* of the operand trees.

8.2.3 Operator Hierarchies

In any normal mixfix operator environment, there is a high probability that ad-hoc precedences are given by the user for most the operators. As we know from section 5.6.2, such ad-hoc precedence relations should be hierarchical to avoid ambiguities with the natural precedence relations.

We assume that the information given by the user describes the expressions that will occur in his program in such a way that the operators that have a high precedence level *normally* occur higher in any given parse tree than the ones with a lower precedence level.

Therefore, it makes sense, whenever we have to choose between different operators as roots of a (sub)expression, to try them in order of their precedence level (from highest to lowest).

Once we have chosen an infix operator as the root of the expression, we can also order the possible matches of instantiations of that operator using the operator hierarchy information.

- If the operator is left-precedent towards itself, we will start with the match which has the longest leftmost operand expression and the shortest rightmost operand expression.

- If the operator is right-precedent towards itself, we will start with the match which has the shortest leftmost operand expression and the longest rightmost operand expression.

Good Results

The unoptimized algorithm, where operator matches are tried in an arbitrary order needs in the order of n^k matching operations in the worst case for some expressions that contain n infix operator applications with operators from k different operator precedence hierarchy levels.

However, with the optimization the number of present operator precedence hierarchy levels loses its significance and the number of matches becomes linear to the number of present operators if all operator pairs that occur in the expression have ad-hoc precedences.

Example 83 *Assuming the operator set and precedences defined in figure 2.7 on page 18, we have measured the amount of top-down matching operations for the different expressions in figure 8.1 both with this optimization and without it. The measurements can be found in table 8.1.*

From example 83, it can be seen that this optimization yields very good results. In the optimized version, the amount of matching operations where only ad-hoc precedences are involved is exactly the number of instantiated operators in the expression (keeping in mind that for every visible operator instantiation, there is exactly one instantiation of the invisible conversion operator present).

```
FUN x  : int
FUN E1: x  *x*x*x*x*x*y*y*y*y*y
           *x*x*x*x*x*y*y*y*y*y
           *x*x*x*x*x*y*y*y*y*y
           *x*x*x*x*x*y*y*y*y*y
FUN E2: x  *x*x*x*x*x*x*x*x*x*x
           /x/x/x/x/x/x/x/x/x/x
           +y+ y+ y+ y+ y+ y+ y+ y+ y+ y
           −y−y−y−y−y−y−y−y−y−y
FUN E3: x  +y+ y+ y+ y+ y+ y+ y+ y+ y+ y
           −y−y−y−y−y−y−y−y−y−y
           *x*x*x*x*x*x*x*x*x*x
           /x/x/x/x/x/x/x/x/x/x
FUN E4: x  *x*x*x*x*x*x*x*x*x*x
           +y+ y+ y+ y+ y+ y+ y+ y+ y+ y
           /x/x/x/x/x/x/x/x/x/x
           −y−y−y−y−y−y−y−y−y−y
FUN E5: x  *x+y*x+y*x*x+y*x+y*x
           /x−y/x−y/x/x−y/x−y/x
           *x+y*x+y*x*x+y*x+y*x
           /x−y/x−y/x/x−y/x−y/x
FUN E6: x  *x+y*x+y*x*x∧y*x∧y*x
           /x−y/x−y/x/x∧y/x∧y/x
           *x+y*x+y*x*x∧y*x∧y*x
           /x−y/x−y/x/x∧y/x∧y/x
```

Figure 8.1: Signature of example 83

expr	optimized	unoptimized
E1	162	569
E2	162	922
E3	162	1647
E4	162	17081
E5	162	57920
E6	162	194917

Table 8.1: Matching Operations with and without Optimization

Bad Results

Unfortunately, the optimization can also have a detrimental effect in the pathological case that the user has given ad-hoc precedences for operators which are used in expressions where they are naturally precedent, but differently so from their given ad-hoc precedences. Here, the precedence information which is supposed to guide the backtrack parser into the right direction is instead misleading it, thereby causing a lot of backtracking.

We have found no expressions, yet, though, where this increased the amount of matching operations to more than the square of the number of occurring infix operator instantiations.

Example 84 *In the signature in 8.2, we have introduced two list construction operators, _ :: _ and _ ::: _ the first with a given right-precedence and the second with a given left-precedence. We have measured the amount of matching operations for two long list expressions* $\mathbf{E}_1$ *and* $\mathbf{E}_2$*, both with and without the optimization. Both expressions have only one right-precedent interpretation. The measurements can be found in table 8.2.*

From example 84, we can see that the hierarchy optimization is still an optimization for the case that the user has given the proper ad-hoc precedence for the concatenation operator, but the performance worsens if the wrong ad-hoc precedence is given in comparison with the unoptimized case.

```
FUN   nat         : SORT
FUN   seq _       :[SORT] → SORT
VAR   A           : SORT
FUN   <>          : seq A
FUN   _ :: _      :[nat , seq nat] → seq nat
FUN   _ ::: _     :[nat , seq nat] → seq nat
PREC _ :: (_ :: _)
PREC (_ ::: _) ::: _
FUN   a           : nat
FUN   _ + _       :[nat , nat] → nat
FUN   _ * _       :[nat , nat] → nat
FUN   _ !         :[nat] → nat
FUN   E1          : a + a :: a + a :: a * a :: a * a :: a + a :: a! :: a! ::
                    a + a :: a + a :: a * a :: a * a :: a + a :: a! :: a! :: <>
FUN   E2          : a + a ::: a + a ::: a * a ::: a * a ::: a + a ::: a! ::: a! :::
                    a + a ::: a + a ::: a * a ::: a * a ::: a + a ::: a! ::: a! ::: <>
```

Figure 8.2: Signature of example 84

expr	optimized	unoptimized
E1	118	225
E2	1391	286

Table 8.2: Matching Operations with and without Optimization

Solution: Adaptive Parsing with Implicit Ad-Hoc Precedences

The bad effect could probably be countermanded by the following further optimization.

If the precedences that are actually *chosen* by the mixfix parser for an expression are not the ones given by the user, the parser records the number of occurrences of each precedence.

If the non-ad-hoc precedence occurs consistently, then instead of using the ad-hoc precedence, the parser shall use this other precedence instead to build an *implicit ad-hoc precedence hierarchy* to use as a guideline for matching.

This approach could also be beneficial in scenarios where only naturally precedent operators exist and thus the user does not actually *need* to give ad-hoc precedences other than to guide the parser faster to the sole interpretation.

Since users tend to have their own *style* of writing down expressions, this could lead to a behavior of the parser which adapts to this style, so the parser becomes better at recognizing expressions written in that style.

Of course, for the human reader, ad-hoc precedences given also for naturally precedent non-standard operators can have a documentary effect and thus heighten readability.

Obviously, such an optimization enters the realm of artificial intelligence and thus we have chosen not to pursue it in depth.

Concatenations

In our experiments, we found that the concatenation operator again must be treated differently from the other operators.

We remember that it is defined as being ad-hoc left precedent because of the built-in function application operator. It also has the highest precedence level since it cannot occur as a rightmost or leftmost operand to other left open or right open operators.

Hence, we would assume that we should always try to match the concatenation operator applications first before trying to match any other operators.

However, this seems to be a *pessimization* because a concatenation could occur potentially *everywhere*, but in reality does not occur that often.

Trying to match the concatenation operator as the root of an expression only after finding no other matching root operator yields much better results in the mixfix operator scenarios we have experimented with.

8.2.4 Maximal Application Depth

We have found another worrying effect that ties in with the same problems encountered for concatenation operators mentioned in section 8.2.3, especially the built-in application operator.

Even adhering to the top-down restrictions for concatenations, there are still a quadratic amount of interpretations to be considered for matching.

To decrease that amount dramatically, we consider the operators that occur in an expression before the matching process starts.

We look at the leftmost operator backbone to see whether it can be a higher-order function and how many arguments it can be applied to via the application operator maximally. If this number is smaller than the length of the expression, then we have found the *maximal application depth* which we can use to further restrict our concatenation matching in a top-down fashion, decreasing the allowed depth for every layer of application until it drops to zero which is when no more application instantiations are allowed.

Example 85 *In the signature in figure 8.3 on page 146, we have introduced three kinds of prefix operators on natural numbers, namely* **f**, **g** *and* **h**. *All of these operators can be applied to two natural numbers, either by operand instantiation or by using the built-in application operator.*

We have measured the parsing time for the differently parenthesized expressions involving these operators in figure 8.3, both with the maximal application depth optimization and without it. The results of these measurements can be seen in table 8.3.

The third row of this table are the parsing time measurements for the same kind of expressions, but where each operand $\mathbf{x}+\ \mathbf{x}+\ \mathbf{x}$ *is replaced with* $\mathbf{x}+\ \mathbf{x}+\ \mathbf{x}+\ \mathbf{x}+\ \mathbf{x}+\ \mathbf{x}+\ \mathbf{x}+\ \mathbf{x}+\ \mathbf{x}+\ \mathbf{x}+\ \mathbf{x}$. *We give only the results for the optimized version since the unoptimized version took too long to parse in our experiments.*

expr	unoptimized short	optimized short	optimized long
f1	0.8 s	0.8 s	3.6 s
f2	0.2 s	0.2 s	0.3 s
f3	0.1 s	0.1 s	0.5 s
f4	0.3 s	0.1 s	0.6 s
g1	38.7 s	0.7 s	9.3 s
g2	0.7 s	0.2 s	1.7 s
g3	90.6 s	0.5 s	4.9 s
g4	0.1 s	0.1 s	0.3 s
g5	0.1 s	0.1 s	0.3 s
g6	0.1 s	0.1 s	0.4 s
h1	33.0 s	0.4 s	6.5 s
h2	0.5 s	0.2 s	1.3 s
h3	76.9 s	0.6 s	6.0 s
h4	0.1 s	0.1 s	0.3 s
h5	0.3 s	0.2 s	1.0 s
h6	0.6 s	0.1 s	0.6 s

Table 8.3: Parsing Time Measurements

The results of the previous example show that this optimization is sensible and very necessary. Unfortunately, it only remains valid as long as no other higher-order function application concatenation operator (like, e.g. the currying application operator) is introduced into the operator set by the user.

Therefore, the user should not be allowed to introduce generic higher-order function concatenation operators.

The most interesting result of this experiment, though, is probably the fact that in the unoptimized case, it takes longer to parse the parenthesized expressions $\mathbf{g}_3$ and $\mathbf{h}_3$ than their unparenthesized counterparts $\mathbf{g}_1$ and $\mathbf{h}_1$, respectively. This is of course a very confusing behavior as one would expect the parentheses to help the parser instead of hampering its performance. Fortunately, as we also can see in table

```
FUN  f _ _        : [nat , nat] → nat
FUN  g _          : [nat] → nat → nat
FUN  h            : nat → nat → nat
FUN  x            : nat
FUN  _ + _        : [nat , nat] → nat
PREC (_ + _) + _
PREC f _ (_ + _)
PREC g (_ + _)
FUN  f1           :  f x+ x+ x+ x   x+ x+ x+ x  : nat
FUN  f2           :  f (x+ x+ x+ x) x+ x+ x+ x  : nat
FUN  f3           :  f (x+ x+ x+ x) (x+ x+ x+ x): nat
FUN  f4           :  f  x+ x+ x+ x  (x+ x+ x+ x): nat
FUN  g1           :  g  x+ x+ x+ x   x+ x+ x+ x : nat
FUN  g2           :  g (x+ x+ x+ x) x+ x+ x+ x  : nat
FUN  g3           : (g  x+ x+ x+ x) x+ x+ x+ x  : nat
FUN  g4           :  g (x+ x+ x+ x)(x+ x+ x+ x) : nat
FUN  g5           :  g  x+ x+ x+ x  (x+ x+ x+ x): nat
FUN  g6           : (g  x+ x+ x+ x)(x+ x+ x+ x) : nat
FUN  h1           :  h  x+ x+ x+ x   x+ x+ x+ x : nat
FUN  h2           :  h (x+ x+ x+ x) x+ x+ x+ x  : nat
FUN  h3           : (h  x+ x+ x+ x) x+ x+ x+ x  : nat
FUN  h4           :  h (x+ x+ x+ x)(x+ x+ x+ x) : nat
FUN  h5           :  h  x+ x+ x+ x  (x+ x+ x+ x): nat
FUN  h6           : (h  x+ x+ x+ x)(x+ x+ x+ x) : nat
```

Figure 8.3: Signature of example 85

8.3 the optimization also corrects this behavior so that parenthesized expressions are parsed faster than their unparenthesized counterparts.

However, we can still see that mixing the concatenation application operator with other operator instantiations does not yield as good results as when only concatenation or only operator instantiations are used.

This optimization works especially well if no generic operators are present in the scope that are not fully instantiated in the result-type, i.e. all operators with a type like **T** → **A** where **A** is a type variable countermand this optimization.

8.2.5 Conclusions

The experimental results in this chapter lead us to the following conclusions.

We think that we have found optimizations for the mixfix parsing process that make mixfix parsing usable, but there are some recommendations that the user should adhere to when writing down programs with long mixfix expressions with as few as possible parentheses.

- **Import only what you need!** That is, import only those operators that are actually used in the program.
- **Import generic operators fully instantiated!** If you introduce your own generic operators, do so in a separate module and use fully instantiated import for them, as well.
- **Give ad-hoc precedences for all pairs of open operators!** This helps the parser assign a hierarchy to the operators which it can make use of in

the hierarchy optimization. For different operator sets like the arithmetic operators, you will probably import the precedences, already, so it is only necessary to add their precedences to other imported operators they should be combined with.

- **Give ad-hoc precedences that reflect the usage of the operators in the program!** If an imported precedence does not reflect your usage, either change this precedence or use parentheses.
- **Introduce higher-order functions only when absolutely necessary!** Use prefix operators with multiple adjacent right-open operands, instead. Through section lifting and empty placeholders, these can be used as higher-order functions, if necessary.
- **Refrain from overloading!** Especially the invisible operators should not be overloaded too often.
- **Don't mix concatenation with normal operator instantiation too often!**

It should be noted that these are only guidelines that make mixfix expressions more easily parsable. However, it is clear that following these recommendations makes the programs more human-readable, as well.

Chapter 9

Conclusion

The aim of this thesis was to show that it is possible to integrate almost arbitrary user-defined mixfix operators into programming languages without losing the ability to efficiently parse programs written in such languages.

Such a language has the power to be more readable than common programming languages, as it allows the introduction of constructs which have a more natural feel to the user. It also becomes possible to customize the programming language to specific domains of users which have common nomenclatures and other common ways of writing things down formally or semi-formally, but are not programming languages experts. Thereby, the usability and understandability of the language can be heightened, leading to higher productivity and better maintainability of software, which in turn could reduce production costs for large-scale software architectures.

Of course, we are aware that there is potential for obfuscation of programs written in this language, but this cannot be denied for any programming language known to us, so we are not worse off, but still have the potential for being better.

9.1 Approach

By identifying the natural causes of ambiguity in mixfix expressions posed by user-defined mixfix operators, we have been able to introduce natural restrictions regarding the mixfix operators to be defined, as well as their combined usage. These restrictions are specifically designed to counteract the causes of ambiguity, allowing us to define a two-level grammar which describes an unambiguous mixfix expression language. They are not implied by the algorithm to be used to parse the language.

While this language does not include *all* unambiguous expressions, it includes many expressions which are unambiguous which nevertheless would be rejected by most other languages. This can only be possible by using type and user-preferred precedence information in the parsing phase as described in the second-level part of the two-level grammar.

To remain efficient in parsing, we have to integrate the parsing phase with (parts of) the semantic analysis phase to allow for early semantics-driven syntactic disambiguation.

9.2 The Functional Mixfix Language

We have taken a functional OPAL-like language with features like overloading, polymorphism, higher-order functions and operator sections, where every expression can also be annotated with a demanded type and a declaration scope for semantic disambiguation purposes, and added the possibility to declare all operators using mixfix patterns. Also, it is possible to declare precedence relation preferences between all declared operators. The types of the language are again mixfix expressions, creating a multi-level type-universe.

9.3 Language Restrictions

The operator patterns are described by a sequence of separator tokens and operand placeholders. We have refrained from allowing more complicated constructions like optional parts or repetitions because they do not add to the expressivity of the language, but can easily be simulated using the simple operator patterns. Therefore, we considered them more as "syntactic sugar", which means that such pattern descriptions could also easily be added to the language.

Although it is allowed for two different operator patterns to share the same separator tokens, this can only be allowed in a restricted manner because otherwise, the so-called backbone ambiguity can occur in mixfix expressions containing the tokens of these operators.

Special care must also be taken with invisible operators, especially in conjunction with adjacent operands.

The conversion operators can create ambiguity in conjunction with operator overloading which also has to be taken care of.

Finally, the user-defined preferred precedence relations must be conflict-free for every expression to be parsed.

9.4 Parsing Efficiency

All these restrictions must be ensured to allow for efficient parsing and disallow ambiguity of the mixfix expressions to be parsed. We have given efficient algorithms to check for these restrictions and for parsing mixfix expressions.

Unfortunately, only the parsing of type-correct expressions is guaranteed to be efficient, while for type-incorrect expressions, such efficiency cannot be guaranteed by our algorithm. However, by introducing a computation threshold (dependent on time or number of computation steps) after which the parser rejects an expression as too complicated to be parsed and giving the user hints as to the possible problems encountered, we could still remain efficient in parsing in general while also maintaining usability.

9.5 Implementation

We implemented the algorithms described in this thesis in Java. A simple unoptimized implementation of an Earley parser, used both for backbone parsing and context-free parsing of the grammars induced by the mixfix operator sets was implemented, as well. Its result is fed into a backtracking matching algorithm which implements the top-down bottom-up type inference and precedence checks. Several optimizations have been added to make this algorithm more efficient by reduction of backtracking. Some more optimizations have been envisioned.

Experiments using this implementation have shown some interesting results in regard to the combinations of language features that should be used.

9.6 Related Work

9.6.1 Inspiration

The idea for this thesis, i.e. of embedding mixfix expressions with user-defined mixfix operators into another programming language, was conceived from a sketch of such ann application in [32]. There, the idea of splitting the parse phase into two parts, the coarse parsing and the mixfix parsing is already considered, but only as a matter for further research.

9.6.2 Comparison with Other Approaches to Mixfix Operators

While our approach can never be as efficient as the state-of-the-art parsing methods for programming languages using LALR-parsers with unambiguous context-free grammars, due to the fact that it is much more powerful and also does more work during the parsing phase, we are still confident that it is efficient enough for most common programming purposes. Expressions written by hand are seldom very large in practice, and if so, have a very repetitive character, i.e. use a very small number of operators.

We are not as restrictive as Annika Aasa [1] in our possibilities to define operator patterns. Also, while she introduces ways of ensuring the absence of so-called *R-Ambiguity* in grammars derived from more general mixfix operators, other ambiguity issues are ignored, or at least, not treated.

We also are more free in letting the user define precedences between operators as Visser [46], leaving the possibility open to have no ad-hoc precedences between operators.

The method of grammatical disambiguation filters that are applied to all parse trees only on a syntactic level as Visser [45] or Thorup [40], [39] propose is not sufficient to accept all unambiguous expressions that are possible in our mixfix expression language, because they are working on the context-free grammar level, while we remain on the declarative level and use type-information for disambiguation. Of course, it could be possible to add declarative parts to our language such that similar filtering mechanisms could also be added to our language. This could still heighten the efficiency of our approach.

The mixfix parsing approach in CASL [47] uses the general ASF+SDF [41] approach, allowing the definition of arbitrary mixfix operator patterns. They introduce a mixfix disambiguation algorithm which uses precedences based on the length (i.e. number of tokens and placeholders) of the different user-defined mixfix operators. In essence, they describe a disambiguation filtering mechanism on the set of all possible parse trees of an expression, which of course, leads to inefficiencies in general, because all the parse trees have to be enumerated to avoid ambiguities. The ambiguity caused by invisble converter operators is dealt with by introducing the same restriction on it (namely that it can only applied at most once to any expression). But, as far as we can see, similar ambiguity situations that can be caused by empty and concatenation operators are not treated at all. Also, the parsing and disambiguation algorithm does not make use of the type information, leading to the common unsatisfactory results.

Finally, we are also less restrictive in letting the user define multi-level signatures as Visser [44], by not explicitly categorizing the operators into different levels, instead inferring this information. Thus, in our approach it is not necessary to lift any signature onto a different level if need be.

9.6.3 Programming Languages

Since we have designed a functional programming language, we have included a lot of features which can be found in other such programming languages.

The basic syntax and structure of our language is owed to the language OPAL [31], since it was designed as a draft for a possible new version of that language.

But, the ideas of type-classes and generic polymorphism of Haskell are also present in our language, although in a slightly generalized way.

9.6.4 Two-Level Grammars

An overview over different sorts of Two-Level Grammars can be found in [23].

9.6.5 Type System

Our type-system is basically a Hindler/Milner [27] type system. Our type-inference algorithm is closely related to that introduced by Damas and Milner in [6].

9.6.6 Parsing

Both, the formalisms of LR(k) and LL(k) parsing ([5], [3]) of context-free grammars are not powerful enough for the problems we needed to solve for parsing our mixfix expression language. These parsing algorithms need unambiguous grammars to succeed in deriving an unambiguous parse tree. Their generalized counterparts via generalized LR parsing [29], or grammar transformations and LL parsing with backtracking [32] which can deal with arbitrary context free grammars, have the general problem of ignoring the typing information of the operators involved or being only applicable to languages without generic types. If generic types are present, only parsers which can deal with two-level grammars can use the type-information during parsing, as unification on the type annotations needs to take place for restricting the applicable rules for the backtrack parser.

Because of its simplicity, we have chosen Earley's bottom-up parsing algorithm [11] for generating the parse tree representation which is then again parsed by a top-down backtrack parsing algorithm for the two-level part of the grammar.

We could also try to adapt the approach described by Pepper [32], using our generalizations on grammar transformations to two-level grammars. Because our two-level grammars are unambiguous by design, the LL backtrack parser would need to find only one parse tree. Unfortunately, the transformation of the optimizations in our present algorithm is still a matter for further research.

9.7 Outlook

In our opinion, following this thesis, user-definable mixfix operators could very well be integrated into new or even existing programming languages, not only functional, but also imperative ones and derivatives thereof.

Of course, all keywords and predefined graphemes must disallowed as valid separator tokens in the operator pattern declarations, but the same restriction already applies to all other identifiers that can be declared in programming languages.

If some features like higher-order functions, overloading, polymorphism, or multi-level type-systems do not exist in the host language, it is no problem to remove those concepts from the mixfix expression language, as well.

Also, it is very probable that the expression constraints on our two-level grammar could still be enlarged so that the mixfix expression language can contain even more unambiguous expressions.

Bibliography

[1] Annika Aasa. *User Defined Syntax.* PhD thesis, Chalmers University, Dept of Computer Science, Chalmers University, Sweden, 1992.

[2] AGFL. www.cs.kun.nl/agfl/, 2004.

[3] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers - Principles, Techniques and Tools.* Addison-Wesley, 1986.

[4] Alfred V. Aho and Jeffrey D. Ullman. *The Theory of Parsing, Translation, and Compiling Volume 1: Parsing.* Prentice-Hall, 1972.

[5] Stephen C. Johnson Alfred V. Aho. Lr parsing. *ACM Computing Surveys*, 6(2):99–124, June 1974.

[6] Luis Damas and Robin Milner. Principal Type-Schemes for Functional Programs. In *Proceedings of POPL '82*, pages 207–212. ACM, 1982.

[7] Luis M. Pereira David H. D. Warren and Fernando Pereira. Prolog – the Language and its Implementation Compared with Lisp. In *Proceedings of the Symposium on Artificial Intelligence and Programming Languages*, pages 109–115, Rochester, New York, August 1977. Association for Computing Machinery.

[8] Gilles Dowek. Higher-Order Unification and Matching. pages 1009 – 1062, 2001.

[9] Marc Dymetman. A Generalized Greibach Normal Form for Definite Clause Grammars. In *Proceedings of the 14th Conference on Computational Linguistics*, pages 366–372, 1992.

[10] Marc Dymetman. A Simple Transformation for Offline-Parsable Grammars and its Termination Properties, 1994.

[11] J. Earley. Ambiguity and Precedence in Syntax Description. In *Acta Informatica*, volume 4 (1), pages 183 – 192, 1975.

[12] Methods for Testing and Specification (MTS); The Testing and Test Control Notation Version 3; Part 1: TTCN-3 Core Language. www.etsi.org, 2004.

[13] Etsi. www.etsi.org.

[14] Joseph A. Goguen, Timothy Winkler, Jos Meseguer, Kokichi Futatsugi, and Jean-Pierre Jouannaud. Introducing OBJ. Technical report, 1993.

[15] F.A. Grootjen. Efficient Recursive Backup Parsing. Master's thesis, University of Nijmegen, Nijmegen, The Netherlands, 1992.

[16] Cordelia V. Hall, Kevin Hammond, Simon L. Peyton Jones, and Philip Wadler. Type Classes in Haskell. In *ESOP*, Jan 1994.

[17] Jan Heering, Paul Hendriks, Paul Klint, and Jan Rekers. The Syntax Definition Formalism SDF – Reference Manual. In *SIGPLAN Notices*, volume 24, 11, pages 43 – 75, 1989.

[18] Simon Peyton Jones, Lennart Augustsson, Dave Barton, Brian Boutel, Warren Burton, Joseph Fasel, Kevin Hammond, Ralf Hinze, Paul Hudak, John Hughes, Thomas Johnsson, Mark Jones, John Launchbury, Erik Meijer, John Peterson, Alastair Reid, Colin Runciman, and Philip Wadler. Haskell 98 Language and Libraries, The Revised Report. Technical report, 2002.

[19] P. Klint and E. Visser. Using Filters for the Disambiguation of Context-Free Grammars. In G. Pighizzini and P. San Pietro, editors, *Proc. ASMICS Workshop on Parsing Theory*, pages 1 – 20, Milano, Italy, October 1994. Dipartimento di Scienze dell'Informazione, Universit di Milano.

[20] D.E. Knuth. Semantics of Context-Free Languages. *Mathematical Systems Theory*, 2(2):127–145, 1968.

[21] C.H.A. Koster. Affix Grammars for Natural Languages. In H. Alblas and B. Melichar, editors, *Attribute Grammars, Applications and Systems*, volume 545 of *LNCS*, pages 469–484, Heidelberg, 1991. Springer-Verlag.

[22] C.H.A. Koster. Affix Grammars for Programming Languages. In H. Alblas and B. Melichar, editors, *Attribute Grammars, Applications and Systems*, volume 545 of *LNCS*, pages 358–373, Heidelberg, 1991. Springer-Verlag.

[23] C.H.A. Koster. The Family of Affix Grammars. In C.H.A. Koster and E. Oltmans, editors, *Proceedings of the First Workshop on AGFL*, Nihmegen University, 1996.

[24] Wolfgang Lohmann, Günter Riedewald, and Markus Stoy. Semantics-Perserving Migration of Semantic Rules after Left Recursion Removal in Attribute Grammars. In *Proceedings of LDTA 2004*, Barcelona, 2004.

[25] C. Dekkers M.-J. Nederhof, C.H.A. Koster and A. van Zwol. The Grammar Workbench: A First Step towards Lingware Engineering. In A. Nijholt W. ter Stal and H.J. op den Akker, editors, *Linguistic Engineering: Tools and Products, Proc. of the second Twente Workshop on Language Technology*, volume 29 of *Memoranda Informatica 92*, pages 103–115, University of Twente, April 1992.

[26] R. Milner, M. Tofke, and R. Harper. *The Definition of Standard ML.* Mass. Institute of Technology Press, Cambridge, Mass., 1990.

[27] Robin Milner. A Theory of Type Polymorphism in Programming. *Journal of Computer and System Sciences*, 17:348–375, 1978.

[28] Robert C. Moore. Removing Left Recursion from Context-Free Grammars. In *Proceedings, 1st Meeting of the North American Chapter of the Association for Computational Linguistics*, pages 249–255, Seattle, Washington, 2000.

[29] M.-J. Nederhof. *Linguistic Parsing and Program Transformations.* PhD thesis, University of Nijmegen, Nijmegen, The Netherlands, 1994.

[30] Erik Oltmans, Caspar Derksen, and et al. Efficiency and Robustness in AGFL. citeseer.nj.nec.com/oltmans97efficiency.html, 1997.

[31] P. Pepper. The Programming Language OPAL (5th corrected edition). Technical Report 91–10, TU Berlin, June 1991.

[32] Peter Pepper. LR Parsing = Grammar Transformation + LL Parsing. technical report 99-05, TU Berlin, April 1999. to appear.

[33] Fernando Pereira and David H. D. Warren. Definite Clause Grammars for Language Analysis–A Survey of the Formalism and A Comparison with Augmented Transition Networks. *Artificial Intelligence*, 13:231–278, 1980.

[34] Fernando Pereira and David H. D. Warren. Parsing as Deduction. In *21st Annual Meeting of the Association for Computational Linguistics*, pages 137–144, Cambridge, Massachuetts, 1983. Association for Computational Linguistics.

[35] Simon L. Peyton Jones. Parsing Distfix Operators. In *Communcations of the ACM (CACM)*, volume 29, 2, pages 118 – 122, February 1986.

[36] Olivier Ridoux. Engineering Transformations of Attributed Grammars in LambdaProlog. In Micheal J. Maher, editor, *Logic Programming, Proceedings of the 1996 Joint International Conference and Symposium on Logic Programming, September 2-6*, pages 244–258. MIT Press, 1996.

[37] Diez B. Roggisch. Typinferenz in Multi-Level Spezifikation. Master's thesis, Technische Universität Berlin, Berlin, Germany, 2006.

[38] D.J. Rosenkrantz and P.M. Lewis. Deterministic Left Corner Parsing. In *Proc. of 11th Annual Symposium on Switching and Automata Theory*, pages 139–152, 1970.

[39] Mikkel Thorup. Controlled Grammatic Ambiguity. In *TOPLAS*, volume 16, 3, pages 1024 – 1050, 1994.

[40] Mikkel Thorup. Disambiguating Grammars by Exclusion of Sub-Parse Trees. In *Acta Informatica*, volume 33, 6, pages 511 – 522, 1996.

[41] M.G.J. van den Brand, A. van Deursen, T.B. Dinesh, J.F.Th. Kamperman, and E. Visser (eds.). ASF+SDF'95. Technical Report P9504, University of Amsterdam, May 1995. workshop proceedings.

[42] E. Visser. Multi-Level Specifications. In A. J. Heering van Deursen and P. Klint, editors, *Language Prototyping. An Algebraic Specification Approach*, volume 5 of *AMAST Series in Computing*, pages 105 – 196, Singapore, September 1996. World Scientific.

[43] E. Visser. Solving Type Equations in Multi-Level Specifications. 1996.

[44] E. Visser. From Context-Free Grammars with Priorities to Character Class Grammars. In *Liber Amicorum Paul Klint*, Amsterdam, July 1997. CWI.

[45] E. Visser. *Syntax Definition for Language Prototyping.* PhD thesis, University of Amsterdam, September 1997.

[46] E. Visser. Polymorphic Syntax Definition. In *Theoretical Computer Science*, volume 199, pages 57 – 86, 1998.

[47] Bjarke Wedemeijer. Introduction & basic tooling for casl using asf+sdf. Technical report, University of Amsterdam, October 1998.

[48] A.v. Wijngaarden, B.J. Mailloux, J.E.C. Peck, C.H.A. Koster, H. Sintzhoff, C.H. Lindsey, C.G.C.T. Meertens, and C.G. Fischer. Revised Report on the Algorithmic Language ALGOL-68. *Acta Informatica*, 5(1-3), 1975.

Index

Zusammenfassung

Die vorliegende Arbeit beschäftigt sich mit dem Thema Mixfix Operatoren, einem Konzept, welches eine Verallgemeinerung der in gängigen Programmiersprachen üblichen Operatoren, Funktionen, Prozeduren und sonstigen Konstrukte darstellt. Insbesondere geht es darum, zu zeigen, daß es möglich ist, dem Nutzer von Programmiersprachen zu erlauben, solche Operatoren in fast beliebiger Form selbst inneralb dieser Sprachen zu definieren, um diese dann in sogenannten Mixfix Ausdrücken benutzen zu können. Dies soll zur besseren Schreib- und Lesbarkeit von Programmiersprachen beitragen und damit zur ihrer besseren Benutzbarkeit sowie erhöhter Wartbarkeit von darin geschriebenen Programmen aus softwaretechnischer Sicht. Sprachen mit solchen Ausdrücken können außerdem besser an die Bedürfnisse der sie benutzenden Kreise angepasst werden.

Es werden notwendige Einschränkungen der definierbaren Operatoren vorgestellt, so daß die Sprache der Mixfix Ausdrücke syntaktisch eindeutig wird und effiziente Verfahren der syntaktischen Analyse solcher Ausdrücke möglich werden, welche ebenfalls erläutert werden.

Die Gründe für diese Einschränkungen werden motiviert durch die möglichen Ursachen für syntaktische Mehrdeutigkeit in Mixfix Ausdrücken, namentlich

- in mehreren Operatoren vorkommende Separator Teile,
- aneinander angrenzende Operanden und unsichtbare Operatoren,
- natürliche und benutzerdefinierte Präzendenz von Operatoren,
- Konverter Operatoren und Polymorphie von Operatoren.

Es werden bewußt keine Einschränkungen auf die Sprache der Mixfix Ausdrücke eingeführt, die auf das Verfahren der Parsierung zurückzuführen sind und für den Nutzer, der von diesen Verfahren nichts weiß, verwirrend sein könnten, sondern nur solche, die aus den oben genannten Gründen zwingend notwendig sind.

Die Arbeit diskutiert die bisherigen etablierten Ansätze, mit Mixfix Operatoren in Programmiersprachen umzugehen, kritisch und legt Gründe offen, warum diese Ansätze für eine so mächtige Sprache von Mixfix Ausdrücken wie der hier vorgestellten nicht angemessen sind.

Als Gegenvorschlag wird ein anderer Ansatz diskutiert, der von der klassischen Vorgehensweise abweicht und basierend ist auf aus den vom Nutzer definierten Mixfix Operatoren hergeleiteten Zwei-Stufen-Grammatiken sowie einem Verfahren, diese algorithmisch zur Implementation des Parsierungsvorgangs zu verwenden, indem die syntaktische Analyse mit Teilen der semantischen Analyse derart verquickt wird, daß die Typ-Inferenz auf die Parsierung Einfluß nehmen kann.

Das ganze Verfahren wird am Beispiel einer modernen funktionalen Programmiersprache geschildert, da solche nach Meinung des Autors die meisten und ausdrucksmächtigsten Programmierkonzepte besitzen. Es sollte gezeigt werden, dass

so gut wie keine Einschränkungen dieser Konzepte notwendig sind, wenn benutzerdefinierte Mixfix Operatoren in eine solche Sprache eingeführt werden. Es wäre dementsprechend ebenso möglich, solche Operatoren auch in weniger ausdrucksmächtigen Sprachen einzuführen.

Die Arbeit ist wie folgt untergliedert.

In Kapitel 1 wird die Motivation und das Problem der Arbeit sowie deren Lösungsansatz kurz umrissen.

Kapitel 2 gibt einen Überblick über Mixfix Ausdrücken und Operatoren ähnliche Konstruktionen in gängigen Programmiersprachen und motiviert deren Verallgemeinerung zum Begriff der Mixfix Operatoren. Anhand vieler Beispiele werden bekannte Konstruktionen mit Hilfe von Mixfix Operatoren beschrieben. Weiterhin wird informell auf das Problem der Mehrdeutigkeit von Mixfix Ausdrücken eingegangen.

In Kapitel 3 werden einige Begriffe formal eingeführt, die als Grundlage der formalen Behandlung der Probleme in den darauffolgenden Kapiteln dienen sollen.

Die funktionale Mixfix Ausdruck Sprache mitsamt ihrem Typsystem wird in Kapitel 4 eingeführt.

Kapitel 5 behandelt ausführlich und formal die verschiedenen Ursachen von Mehrdeutigkeit und begründet die daraus folgenden Einschränkungen der vom Benutzer definierbaren Operatoren sowie der Sprache der Mixfix Ausdrücke, um sowohl deren Eindeutigkeit als auch die Möglichkeit der effizienten Parsierung zu erhalten.

Die aus den vom Nutzer definierten Operator-Mengen abzuleitenden Zwei-Stufen Grammatiken einschließlich der aus Kapitel 5 folgenden Einschränkungen werden in Kapitel 6 vorgestellt.

Kapitel 7 verallgemeinert die bekannten Grammatik-Transformationen für kontextfreie Grammatiken auf kontextfreie Zwei-Stufen Grammatiken, welches die Möglichkeiten für die automatische Generierung von Parsern für solche Grammatiken eröffnet.

Die Implementierung unseres Parsierungsverfahrens wird in Kapitel 8 beschrieben. Zudem werden Ergebnisse von Experimenten mit dieser Implementierung sowie daraus motivierte Optimierungen und ihre Auswirkungen auf die Effizienz vorgestellt. Andere mögliche Optimierungen werden angedacht.

In Kapitel 9 finden sich einige abschließende Bemerkungen.