Diss. ETH No. 12108

Higher-Order Mixfix Syntax for Representing Mathematical Notation and its Parsing

A dissertation submitted to the
SWISS FEDERAL INSTITUTE OF TECHNOLOGY (ETH)
ZURICH

for the degree of
Doctor of Technical Sciences

presented by
STEPHAN ALBERT MISSURA
Dipl. Informatik-Ing. ETH
born February 20, 1968
citizen of Bettlach, Solothurn

accepted on the recommendation of
Dr. Roman Mäder, examiner
Prof. Dr. Erwin Engeler, co-examiner
Dr. Peter Baumann, co-examiner

1997

Für Silvia

Acknowledgments

I would like to thank my adviser, Dr. Roman Mäder, for his guidance and confidence in my work. He introduced me to the fields of designing symbolic computation systems and functional programming and gave me the freedom to develop my own ideas. I am grateful to Prof. E. Engeler and Dr. Peter Baumann for agreeing to be co-examiners of this thesis. A special source of inspiration was Peter's "Theory Club" at the University of Zurich. I am indebted to all three examiners for their expert knowledge in computer science and mathematics.

I want to thank Andreas Weber for sharing his ideas on various computer algebra topics with me and for proof-reading this thesis. It was always a pleasure to work with him.

I am thankful to my colleagues from the Symbolic Computation group, Giovanni Cesari, Oliver Gloor, Georg Grivas, and Gérard Milmeister, for insightful discussions, including real-world problems. I shall never forget the group members of the Logics and Computer Science group headed by Prof. E. Engeler who provided me with a pleasant working environment during the final phase of my work.

Christoph Denzler and Niklaus Mannhart helped me a lot with proof-reading and offered support on LaTeX questions. We had many stimulating discussions on computer science in general. Thanks also go to Volker Strumpen for interesting discussions about functional and distributed programming and to my student, Gilles Everling, who raised some challenging questions in his diploma thesis.

My special thanks go to Angela Rast. Being of English mother tongue, she helped me to improve this work language-wise.

Last but not least, I thank my parents for their moral support throughout the years it has taken me to conclude my studies and for making them possible in the first place.

Contents

Abstract

This dissertation deals with the features of mathematical notation, their consequences for a fundamental formalism to describe mathematical notation, and the parsing of the induced languages.

The main features of ordinary mathematical notation are the huge alphabet of ground symbols, different operator fixities for denoting function application, short and undeclared identifiers, juxtaposition, various variable binding constructs, the use of few whitespaces, coercions, two-dimensional layouts, incremental extensions by new operators, etc. These features lead to a very compact style of writing mathematical sentences. They become very context-dependent through the reuse of characters in different identifiers. Both the syntactical form of identifiers and their types are important for parsing mathematical notation.

We present higher-order mixfix syntax in this dissertation. It is a metaformalism to describe the linear part of mathematical notation, and to deal with the various ambiguities and context sensitivities being present in mathematical syntax. It includes arbitrary binding operators, finite sequence constructions, arbitrary syntax also for type constructors, and a new operator attribute for refined control of syntactical ambiguities. We show that higher-order mixfix syntax is better suited as a metaformalism for mathematical notation than context-free grammars and other formalisms found in today's programming languages.

On the algorithmic side, we integrated Hindley–Milner style type inference into the parsing process of higher-order mixfix syntax. Parsing is fully parameterized by an arbitrary context. A bracketing scanner helps to reduce the input size by prestructuring the input. These ideas are put into practice in a prototypical implementation written in the functional language Standard ML.

Kurzfassung

Die vorliegende Arbeit beschäftigt sich mit den Eigenschaften der mathematischen Notation und deren Folgen für einen geeigneten Basisformalismus, der es erlauben soll, die mathematische Notation angemessen zu beschreiben. Zudem ist auch das Parsen der durch diesen Formalismus induzierten Sprachen von Interesse.

Die übliche mathematische Notation besitzt Eigenschaften, die zu einer kompakten Schreibweise für mathematische Formeln führen. Diese speziellen Eigenschaften sind die grosse Anzahl verwendeter Zeichen, unterschiedliche Schreibweisen für die Applikation von Operatoren, die Verwendung von unsichtbaren Operatoren, Konstrukte, welche Variablen binden, Zweidimensionalität usw. Die mehrfache Verwendung von Zeichen in Bezeichnern führt zudem zu einer starken Kontextabhängigkeit von Formeln. Damit ist nicht nur die syntaktische Form eines Bezeichners, sondern auch sein Typ für das Parsen von mathematischer Notation relevant.

In dieser Arbeit wird nun eine "higher-order mixfix syntax" genannte Metanotation vorgeschlagen, die sich für die Beschreibung von linearer mathematischer Notation mit all ihren Mehrdeutigkeiten und Kontextabhängigkeiten eignet. Die Metanotation unterstützt beliebige Bindeoperatoren und endliche Sequenzen in Infixschreibweise. Der gleiche Syntaxdefinitionsmechanismus kann auch uneingeschränkt für Typkonstruktoren verwendet werden. Zudem wird ein neues Operatorenattribut eingeführt, das eine feinere Kontrolle von Mehrdeutigkeiten erlaubt. Es wird gezeigt, dass sich dieser Metaformalismus besser eignet, die mathematische Notation zu beschreiben, als kontextfreie Grammatiken und ähnliche Sprachbeschreibungsmechanismen, die in heutigen Programmiersprachen Verwendung finden.

Ein spezieller Algorithmus wurde für das Parsen von "higher-order mix-fix syntax" entworfen, in welchen die Typinferenz im Stile von Hindley–Milner integriert wurde und welcher mit einem beliebigen Kontext parametrisierbar ist. Ein spezieller Scanner, der die Eingabe bereits vorstrukturiert, reduziert die Grösse der zu parsenden Tokenliste. Diese Ideen wurden in einen Prototyp in der funktionalen Sprache Standard ML umgesetzt.

Chapter 1

Introduction

Figure 1.1 shows abstractly the internal data flow of a typical interactive framework that handles formalized mathematics, such as a computer-algebra system or a theorem-proving assistant. A parsing system processes the input relative to some kind of grammar, which describes the language. This grammar is either fixed or can be influenced by the user. The output of the parser is an abstract syntax tree (AST), which is subsequently evaluated in some environment. The result is then passed to the typesetter, which needs the grammar to generate a readable output. In some modern systems parsing and typesetting is carried out straightaway, i.e. after each keystroke, thus providing the user with an already formatted presentation of the input.

The dotted box of the figure shows where our work is located. We are interested in parsers for handling mathematical notation, including suitable grammars and their meta-formalisms. Typesetting mathematical formulas and evaluating them lies beyond the scope of this thesis, and it is not our intention to improve mathematical notation itself for the sake of easier parsing or understanding. Our aim is to support the given properties of mathematical notation with all its shortcomings and difficulties.

After describing the motivating forces for this work, we summarize the results and give an overall view of the remaining chapters.

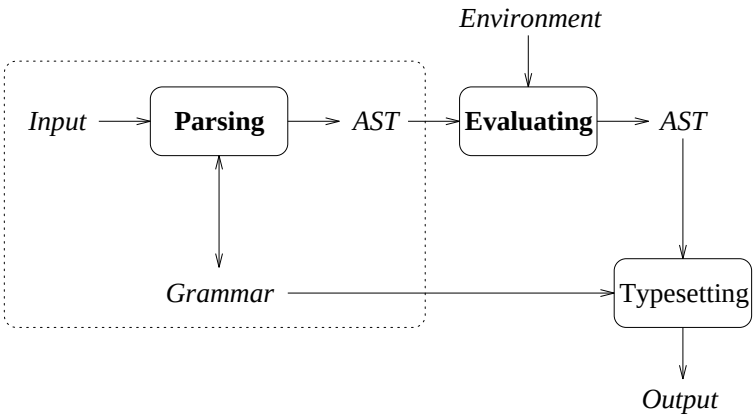

Figure 1.1: Data flow in an interactive mathematics system.

1.1 Motivation

Mathematical notation uses many features that lead to a very compact and context-dependent style of writing mathematical sentences. Their compactness is mainly due to short (and usually undeclared) identifiers, the use of juxtaposition, invisible variable binding constructs and few whitespaces. Through the reuse of characters in different operators and constant symbols — overloading of identifiers is a special case — the sentences become strongly context-dependent. Therefore, not only the syntactical form of variables and operators, including their attributes such as priority or implicit associativity annotations, are required to get unambiguous meaning of a given sentence; the *types* of identifiers are equally important.

For example, the input string "$xsinyz$" can be unambiguously read as "$x \cdot (sin(y \cdot z))$" owing to knowledge of the shape and types of the occurring identifiers and knowledge of the use of juxtaposition for denoting multiplication and function application. Without types, the string could be interpreted for example as "$((x \cdot sin)\ y)\ z)$" or as "$(x \cdot sin(y))(z)$". Consequently, even "$(((x \cdot s) \cdot i) \cdot n) \cdot yz$" is possible!

Or, take a further example: can we cancel the fraction x^2/x by x? The answer depends on the type of x. If x denotes an arbitrary rational number, then we cannot apply this simplification (x could be zero). But

if x denotes an indeterminate, we find ourselves in the field of rational functions and can, therefore, simplify.

For this reason, we have to use a surrounding context for reading mathematical sentences. In fact, operator attributes are also context-dependent. For example, in a context where the multiplication operator denotes an associative operation, the operator is declared to be left associative, and in a different context, where the same operator is used for a non-associative operation, it is not necessarily equipped with any implicit associativity.

Further characteristics of mathematical notation are the huge alphabet of ground symbols, different fixities for denoting application, two-dimensional layouts, various variable binding constructs, coercions, incremental extensions by new operators and so on.

Nevertheless, generally speaking, mathematical notation is easily understood by trained people, although the underlying semantics are usually more difficult to grasp. But all the mentioned properties make parsing by a machine a challenging problem! Due to the compact notation, occurring words are very seldom atomic but have to be decomposed further as shown in the above example. For this reason, the parsing system cannot rely on static assumptions but has to be very dynamic in nature. In addition, type checking and inference have to be an integral part of the parsing process to get unique abstract syntax trees.

Mathematical notation is weakly supported by today's programming languages, making the task of a parser simpler. But with the advance of computer-algebra systems and theorem-proving environments, the parsing of mathematical notation as part of the user interface can no longer be neglected.

For describing and extending mathematical syntax on the computer with all its features, a suitable formalism underlying mathematical notation is needed (a *meta-notation*). Context-free grammars — the most common tool for describing the syntax of ordinary programming languages — are not suitable because they cannot describe notations with higher-order polymorphic type systems and binding constructs in a satisfactory way. Neither extensions of context-free grammars, such as context-sensitive grammars, nor typesetting languages such as TEX, which can produce formatted two-dimensional mathematical formulas, are suitable. TEX can be used as an input language for representing two-dimensional notation in linear form within the ASCII character set, but it cannot describe

what is valid input and what is not.

Mixfix syntax is a formalism which covers a broad spectrum of mathematical notation. It is widespread in the algebraic-specification community, where it is used exclusively with first-order monomorphic type systems. The distinguishing feature of mixfix formalisms is the combination of the declaration of an operator's source and target type with the definition of its syntactical shape, including the positions of the individual arguments. This emphasizes the need for types, which can in fact be higher-order and polymorphic, for the parsing process. Prefix-, postfix-, infix-operators and mixed forms are supported. Juxtaposition and coercion operators fit very well in this scheme, too. Unfortunately, there is no support for binding operators, infix-style sequences and non-linear notation and no mixfix formalism currently allows the syntax of occurring types to be described by mixfix declarations.

Algorithms for recognizing mixfix syntax can be classified into the following three categories:

- algorithms which use no type information at all,
- algorithms which use type inference in a second phase after translating the input to an AST,
- or algorithms which integrate the inference of types into the parsing process itself.

Only the second and third categories handle context-dependence.

Conventional algorithms for parsing arbitrary or restricted context-free grammars, such as Earley's algorithm or LR-parsing, share those disadvantages of context-free grammars mentioned above. They can, at most, describe first-order type systems. Depending on the formulation of the underlying grammar, they belong to the first or third category.

Algorithms in the second category handle arbitrary type systems, because the parsing process does not take types into account. They can therefore be based on a conventional recognizer for arbitrary context-free grammars (including ambiguous grammars!), but have to produce unnecessary intermediate abstract syntax trees which are eliminated later during the type inference step. The logical framework Isabelle works in this way.

One interesting example of the third category is the interactive grammar-construction and expression-parsing tool Cigale. This uses a trie (dig-

ital search tree) — a classical data structure for handling dictionaries — for representing grammars induced by mixfix declarations. Parsing is done by a trie traversal, where the current trie represents the surrounding context. Due to the use of heuristics for choosing trie paths, CIGALE's parser does not cover every possible path and therefore fails on some input. Furthermore, no priorities and associativities are handled and the underlying type system is first-order.

We do not know of any parsing algorithm that takes advantage of the integration of higher-order polymorphic type inference into the parsing process itself.

1.2 Contributions

We present mixfix syntax independently of a concrete type system, not excluding higher-orderness and polymorphism. In fact, several of the provided examples expect higher-order and polymorphic types. We discuss different forms of ambiguities that arise in this general syntax definition mechanism, especially those that cannot be solved by the traditional methods of priority and implicit associativity annotations. We show that if a higher-order polymorphic type system is available, mixfix syntax is strictly more powerful to describe languages than positive (ε-free) context-free grammars.

We give a thorough survey of the features of mathematical notation and point especially to the use of juxtaposition, and the important difference between variables introduced by binding constructs and those introduced by definitions. Using these insights, we extend mixfix syntax with the following features:

- Type constructors are also supported by arbitrary mixfix syntax through a syntactical "type of types".
- Support of binding operators with arbitrary notation, using a higher-order abstract syntax approach. This extension implies a higher-order type system.
- The possibility of defining sequence operators for denoting functions of an arbitrary number of arguments written in infix style.
- Coherent use of mixfix declarations: local declarations can also be expressed using mixfix syntax.

- A new operator attribute called *positional exclusion*, which is used for excluding the construction of certain abstract syntax trees. With the help of this attribute, some of the ambiguities which occur in mathematical notation can be prevented.
- Relative priorities as opposed to absolute priorities or precedence grammars used by today's mixfix formalisms.
- All operator attributes are context-dependent.

We call this resulting formalism *higher-order mixfix syntax* and show its usefulness as a meta-notation for describing and extending the linear part of mathematical notation. It is especially useful for computer scientists and mathematicians who still want to use ordinary mathematical notation when working with computer-based systems.

To show the feasibility of the parsing of higher-order mixfix syntax, we designed and implemented a suitable scanner and parser in the functional language Standard ML. The starting point is Cigale's idea of representing the grammar by a trie. We translated its parsing method into a functional setting, and added the following extensions:

- We use a higher-order polymorphic type system in the style of modern functional languages, taking advantage of Cigale's independence of a specific type system (this fact is not exploited by Cigale's author).
- Hindley–Milner style type inference of undeclared bound variables integrated into the parsing process.
- The resulting algorithm follows all possible paths in a trie. Therefore, it is more general than Cigale's method.
- Use of a dynamic and bracketing scanner, which is also capable of character-based parsing.

The idea behind returning possibly more than one abstract syntax tree retains the possibility of deducing equalities between them (semantical uniqueness of different abstract syntax trees).

As partial compensation for the performance loss caused by traversing all possible trie paths, we introduce the idea of a dynamic and bracketing scanner.

Dynamic in the sense that the scanner, which translates substrings into tokens, has access to the syntactical forms of identifiers currently used. It extends this set itself in the case of occurring bound variables.

The method of bracketing takes advantage of the fact that certain operators, such as parentheses, bracket their argument(s). The scanner recognizes these and returns a corresponding *structured* token list. This bracketing results in a reduction of the recursion depth in the subsequent parsing phase.

The scanner supports character-based parsing. This means that it detects separators, as well as bound and free variables inside arbitrary strings.

1.3 Overview

The organization of this thesis is as follows.

In Chapter 2, we give some notational remarks and define the terminology.

Chapter 3 introduces the formalism of mixfix syntax, including a discussion of different sources of ambiguities. We compare the power of mixfix syntax to describe languages with that of context-free grammars. Thereafter, some systems are presented that support mixfix syntax.

Chapter 4 investigates the features and difficulties of mathematical notation as it is commonly used today and characterizes the requirements for a suitable meta-notation underlying mathematical notation. A survey on the support for mathematical notation in various programming languages is given as well.

In Chapter 5, we introduce higher-order mixfix syntax. We discuss the conditions for its underlying type system and its operator attributes, and introduce sequence and binding operators as new kinds of defining operators. Furthermore, we interpret it in a Curry-style second-order polymorphic λ-calculus.

Chapter 6 discusses the design of a parsing system that is able to handle higher-order mixfix syntax. A survey of existing parsing methods is also included.

Some notes on the prototype implementation, which was done using Standard ML in its newest revision, are presented in Chapter 7.

We finish this thesis with some conclusions and an outlook for possible future work. References to related work are given at the appropriate places in the individual chapters throughout this thesis.

Chapter 2

Preliminaries

We first fix some typographical and mathematical notation. Then we introduce some terminology used throughout this thesis. Generally, we assume knowledge of the most important λ-calculi [Bar84, BH90, Sch94], Hindley–Milner style type inference, including Milner's algorithm W [Hin69, Mil78, Sch94], and the theory of context-free grammars [EL92, DSW94]. Algebraic structures such as monoids, fields or modules over rings are used in various examples without introducing them formally. Their definitions can be found in algebra textbooks [Lan84, GCL92].

2.1 Notation

`Typewriter` font is used for source code. Meta-variables, bound variables, and string constants are written in *italic*, and sans-serif is mainly used for variables being introduced by definitions (e.g. sin, i, map, IN, Ring).

2.1.1 Sets and structures

The natural numbers including 0 are denoted by $\mathbb{N}$, the integers by $\mathbb{Z}$, the rationals by $\mathbb{Q}$ and the reals by $\mathbb{R}$. The identifier $\mathbb{B}$ denotes booleans with values true and false. If the context is clear, we use the same (!) letters $\mathbb{Z}$, $\mathbb{Q}$ and $\mathbb{R}$, respectively, for the corresponding algebraic structures as for the underlying carriers (namely, the euclidean domain of

integers and the fields of the rational and the real numbers, respectively). Analogously, if $n \in \mathbb{Z}$ then $\mathbb{Z}_n$ means either the residue class ring (residue class field if n is a prime) $\mathbb{Z}/(n)$ or its underlying carrier set. In the case of the natural numbers, $\mathbb{N}^+$ denotes the additive monoid and $\mathbb{N}^*$ the multiplicative one.

The cartesian product (or direct product) of the sets $A_1, \ldots, A_n$ is denoted by $A_1 \times \cdots \times A_n$ or by $\prod_{i=1}^{n} A_i$. Its values are called n-tuples and written $(a_1, \ldots, a_n)$ where $a_i \in A_i$. By () we denote the set with the 0-tuple as the only element. The 0-tuple is written (), too. $A^n = A \times \cdots \times A$ denotes the set of homogeneous n-tuples and its elements are called vectors (of length n). Given an n-tuple v then the i-th component $(1 \leq i \leq n)$ is accessed by v_i.

The cartesian product over an arbitrary index set I is denoted by $\prod_{i \in I} A_i$. Its elements v — "infinite" tuples or families [Lan84] — have the property $\forall i \in I. v_i \in A_i$.

The disjoint union (also called coproduct or sum) of the sets $A_1, \ldots, A_n$ is denoted by $A_1 + \cdots + A_n$ or by $\sum_{i=1}^{n} A_i$. The injections, which construct elements of the sum, are written inj_i, $1 \leq i \leq n$. We often omit them if they are clear from context. The sum over an arbitrary index set is written $\sum_{i \in I} A_i$.

We treat both the cartesian product and the disjoint union as associative with respect to isomorphism but not with respect to equality ($A \times B \times C$ contains 3-tuples but $(A \times B) \times C$ pairs).

The power set of A is denoted by 2^A. By $\mathsf{Set}(A)$ we mean the set of all *finite* subsets of A. The set $A_\perp$ is the set A with the element $\perp$ adjoint, i.e. $A_\perp = A + \{\perp\}$. This construction is used to create additional error or nil values for A. We sometimes call an element of $A_\perp$ an optional element. By $|A|$ we denote the cardinality of a A. If M is a set of sets then $\bigcup M$ denotes the union over all its member sets.

2.1.2 Functions

$A \to B$ (or equivalently B^A) denotes the set of partial functions with source A and target B. Domain and range of a function (operation) are defined as usual. $\perp \in A \to B$ is the nowhere defined function (the *empty function*). $A \xrightarrow{f} B$ is the subset of those $f \in A \to B$ having a finite domain. $A \hookrightarrow B$ is the set of all total injective functions between A and B (also called *embeddings*). By $f|A'$ with $A' \subseteq A$, we denote the

restriction of f to source A'.

As usual the function-space operators $\to$, $\xrightarrow{f}$, and $\hookrightarrow$ associate implicitly to the right and have a lower priority than the cartesian-product operator.

We use the following functions: If $f \in A \to B$ then $\mathsf{source}(f) = A$ and $\mathsf{target}(f) = B$. The *arity* of an operation is n if the argument is a n-tuple. Otherwise, the arity is 1. That is, if $\mathsf{source}(f) = A_1 \times \cdots \times A_n$ then $\mathsf{arity}(f) = n$. If $\mathsf{arity}(f) = 1$ we call f a unary operation, if $\mathsf{arity}(f) = 2$ then f is called a binary operation, and generally, we speak of an n-ary operation.

Application of a function f to an argument x is written $f(x)$ or without parenthesis $f\ x$. This is the usual prefix notation for application.

If we define a function by a finite list of conditional equations — possibly with patterns on the left-hand side — and more than one of the equations' conditions is fulfilled, then the first of these equations is taken operationally, i.e. the subsequent equations can be thought to be equipped implicitly with the negated conditions of the preceding equations. Thus, the equations have to be worked through sequentially as is done in most functional languages with pattern-matching.

2.1.3 Strings and lists

An alphabet is a finite or denumerable set. The elements of the alphabet are called characters or symbols. Let A be an alphabet then A^* is the set of all finite strings (or words) over the alphabet A and A^+ is the set of non-empty strings over A. ε is the empty word and if $w, w' \in A^*$ then ww' means the concatenation of both words. If $c \in A$ then $c \in A^*$, too. Thus, every character is also a string. Quotation marks are used for denoting string constants: "$\cdots$". Given $w \in A, n \in \mathbb{N}$ then $w^0 = \varepsilon$ and $w^n = ww^{n-1}$. Let $w \in A^*$, $U, V \subseteq A^*$ then $|w|$ denotes the length of w. Furthermore, we have $wU = \{ww' \mid w' \in U\}$, $Uw = \{w'w \mid w' \in U\}$, $UV = \{ww' \mid w \in U, w' \in V\}$. $U^0 = \{\varepsilon\}$, $U^n = UU^{n-1}$, and $U^* = U^0 \cup U^1 \cup U^2 \cup \ldots$, $U^+ = UU^*$ (hence, vectors are *not* lists!).

Semantically, lists are identical to strings. The difference lies purely in the different syntax. Given $a_1, \ldots, a_n \in A$ then the list $[a_1, \ldots, a_n]$ denotes the string $a_1 \cdots a_n$. The empty list [] stands for ε. We write prepending and appending of an element to a list, as well as concatenation of lists juxtaposed, too. Hence, prepending is denoted by al,

appending by la, and concatenation by ll', where $a \in A$ and $l, l' \in A^*$. The set A^* is sometimes denoted by $\mathsf{list}(A)$.

2.2 Terminology

2.2.1 Types

A *type* is an element of some set T of terms [BH90]. T is usually inductively defined with some type constants and by forming new types out of old ones with type constructors. The base type constants are also called *sorts*, especially in first-order algebraic specification [EGL89, G+92, Wir90]. In case of a polymorphic type system (cf Section 2.2.4) type variables also belong to the base case. Important type constructors are function-type, tuple-type and list-type constructors.

Types are a pure syntactical vehicle to restrict a too general term language (usually induced from a context-free grammar) to a subset of *well-formed* or *well-typed* terms [Gun92]. To quote Goguen [G+92]:

> Ordinary unsorted logic offers the dubious advantage that anything can be applied to anything; for example,
>
> ```
> first-name(not(age(3 * false))) iff 2 birth-place(temperature(329))
> ```
>
> is a well-formed expression. Although beloved by Lisp and Prolog hackers, unsorted logic is too permissive.

A *type system* defines now those terms that are well-formed. The connection of well-formed terms with types is usually done by some set of inference rules, the *typing rules* [Gun92, p. 38ff.]. Formation rules are needed to form types, introduction rules for constructing values of a particular type, and elimination rules are needed to "get rid" of the constructed values.

We call a language *typed* if it possesses a type system that belongs to the language specification. Otherwise we call it *untyped*. *Strongly-typed* is the property of a language that there are no type violations during evaluation [Tho91, p. 30], expressed by Milner's slogan "well-typed programs cannot go wrong" [Mil78].

If the set of types T is closed under all occurring type constructors, we speak of a *higher-order* type system. If it is closed under a particular

type constructor C we call the type system higher-order with respect to C.

Given a type and a term, *type checking* decides if the term has indeed that type. *Type inference* tries to infer a type for a given term [CW85, Rea89]. In any case, one needs an equality relation on the set of types for type checking and type inference, respectively.

We use the same syntax for types (type constructors) and for sets (set constructors) as introduced in Section 2.1. But one should carefully distinguish them on the semantic level! For example, $\mathbb{Q}$ denotes the *set* of rationals and the *type* of rationals. The term $1/0$ has type $\mathbb{Q}$ in most type systems, but $1/0 \notin \mathbb{Q}$. It is a well-formed term by the usual typing rules but does not denote a value in the set $\mathbb{Q}$.

Dependent product and sum types are generalizations of function space types and cartesian products [Sch94, Tur91]. They are written $\prod_{i\in I} T$ and $\sum_{i\in I} T$, where in T the bound variable i can appear free, analogously to generalized cartesian products and disjoint unions of sets. They possess the property that their elimination rules need β-reduction. Hence, in general type inference can no longer be decided [Sch94].

2.2.2 Identifiers and operators

An *identifier* is a name standing for an arbitrary value (as usual, we do not use any whitespace characters within the name). We use the term *variable* synonymously with identifier. This is the mathematical use of the word variable. It is a syntactical entity [Bar84] in contrast to imperative languages where variables are memory locations.

A (normal) *declaration* is an assignment of a type to an identifier. Declarations do not have any computational content; they constrain the syntactical use of identifiers. We use the syntax "$x : T$" for a declaration of the identifier x with type T.

A *type context* is a list of declarations, written $x_1 : T_1, \ldots, x_n : T_n$ (the common syntax is to omit the brackets of the list constructor). If Σ denotes a type context then $(x, T) \in \Sigma$ means that the declaration $x\!:\!T$ occurs in Σ. By $\Sigma.x$ we denote the set $\{\, T \mid (x, T) \in \Sigma \,\}$, all types of a possibly overloaded identifier x. The domain of a type context Σ is its set of occurring identifiers: $\mathsf{dom}\,(x_1 : T_1, \ldots, x_n : T_n) = \{x_1, \ldots, x_n\}$. Given a set $A = \{x_1, \ldots, x_n\}$, $\Sigma|A$ denotes the restriction of the type context Σ to identifiers occurring in A: $\mathsf{dom}(\Sigma|A) \subseteq A$, $(x, T) \in (\Sigma|A) \to (x, T) \in \Sigma$.

With $\Sigma, x:T$ the identifier x with type T is added to Σ. We can concatenate two type contexts Σ_1 and Σ_2 by Σ_1, Σ_2. Multiple occurrences of the same declaration in a type context do not matter. This means that the type context $\Sigma_1, x:T, \Sigma_2, x:T, \Sigma_3$ is equal to $\Sigma_1, x:T, \Sigma_2, \Sigma_3$. *Signatures* are type contexts made available in the user-language.

Identifiers are introduced by definitions or binders:

- A *definition* is an association of an identifier x with a value v [Rea89]. We use the syntax "$x := e$" where e is a term denoting the value v (this syntax should not be confused with the assignment statement found in imperative languages). We also allow type definitions by letting e denoting a type. Definitions do not denote any values but are a mechanism to make subsequent expressions more readable.
- A *binder* is a variable-binding construct [Pau94, p. 135]: it introduces a new variable which can be used in the binder's body. Hence, a binder contains at least two components: a declaration of the new variable and an expression — the body (being the scope of the variable) — which may contain this variable. We call the introduced variable the *bound variable* [Pau91, p. 333]. If its type is clear from context we may omit the type of the variable, simplifying the declaration to an introduction of an identifier. In this case, we call it an *undeclared variable*.

Typical examples of binders are the logical quantifiers $\forall$ and $\exists$, the abstraction construct λ of the λ-calculus or the $\sum$- and $\prod$-binders for sums or products over expressions denoting numbers. There are also *implicit* binders which are syntactically invisible, embracing the whole expression implicitly. Say in the axiom $x + y = y + x$ for expressing commutativity of addition, the variables x and y are *implicitly bound* by an invisible outer $\forall$-quantifier: $\forall\, x, y\,.\, x + y = y + x$. Or sometimes one speaks of the function "$\mathsf{sin}(x)$" and means in fact $x \mapsto \mathsf{sin}(x)$.

Note that we do not treat definitions or declarations as binders. As opposed to the bound variable of a binder whose name does not matter — α-conversion allows a switch to other names without changing meaning —, the identifier introduced by a definition cannot be changed by α-conversion because a definition has no body!

The terms *bound occurrence* and *free occurrence* are defined as usual [Bar89, Pau91, Sch84]: The occurrence of a variable x in an expression

is called bound if it is within the scope of a binder with bound variable x. Otherwise, the occurrence is called free. A variable x is called free in an expression E if there is at least one free occurrence of x in E. Else, if all occurrences of x are bound, x is bound.

An *operator* (*function symbol*) is an identifier denoting a function, whereas a *constant symbol* is an identifier that denotes something other than a function (namely a *constant*).

Source, target and arity are defined analogously as in Section 2.1.2: If the operator f is declared as $f : A \to B$ then $\mathsf{source}(f) = A$ and $\mathsf{target}(f) = B$ give the source and target type, respectively, of f. If $\mathsf{source}(f) = A_1 \times \cdots \times A_n$ then $\mathsf{arity}(f) = n$ else $\mathsf{arity}(f) = 1$. f is an n-ary operator if $\mathsf{arity}(f) = n$. We call f a unary or a binary operator if $\mathsf{arity}(f) = 1$ and 2, respectively. The term *arguments* or *operands* means that $\mathsf{arity}(f) >= 2$ and the individual arguments are the components in the argument tuple (which is *the* argument of the function; in fact every function takes exactly one argument and produces one result in an application).

The *fixity* of an operator indicates the visual relation between the operator and its argument(s) in an application. Common fixities are prefix, postfix and infix: If the operator f is written before (after) the argument, f is a prefix (postfix) operator; if f is a binary operator and is written between the two arguments, it is an infix operator. We will see further fixities in Chapter 3.

Besides fixities there are further *operator attributes*: *Priorities* (*precedences*) and *implicit associativities* of operators are used to give a unique meaning to the linearized form of a term [ASU86].

We do not treat operators differently from constant symbols. Hence, in the expression $f(x, y)$ not only x and y but also the operator f is free! This seems to be different to the mathematical usage as quoted by Stoy in [Sto77, p. 38–38]:

> Even mathematicians are accustomed to treating functions as underprivileged objects. This attitude starts young. In an expression like
>
> $$(x + y) \times (x - y)$$
>
> the "x" and "y" are regarded as variables, but not "+", "−", or "×". Similarly, in the expression
>
> $$f(x)$$

> f is usually thought of as a constant.

This un-orthogonality comes from using first-order languages [Bar89] where operators do not have the same status as variables (similarly to first-order algebraic-specification systems [Wir90]). It is interesting to note that in texts dealing with functional programming this special status of operators can also be seen [Pau91, p. 33]:

> In the integral $\int_a^b f(x)dx$, the variables a and b are free while x is bound. In the product $\prod_{k=0}^{n} p(k)$, the variable n is free while k is bound.

As already mentioned, we treat f and p as free variables in the above examples. But $\int$ and $\prod$ are also free if treated as higher-order operators! (cf Section 2.2.6 and Chapter 5).

2.2.3 First-class values and higher-order functions

An entity in a programming language is called a *first-class* value [Sto77] if it can be

- associated with an identifier,
- passed as an argument to functions,
- returned as a result of functions, and can be
- an element in an aggregating data structure (e.g. a list or a record).

The possibility of constructing a first-class value anonymously (expressing it without the need to bind it to an identifier) is not mandatory but often very useful.

A function is *higher-order* or a *functional* if it takes functions as arguments and/or returns functions [Pau91, Rea89]. We call a programming language *higher-order* or a *functional* language if functions are first-class values. If in addition the language has a type system, then this has to be closed under the function-type constructor.

2.2.4 Overloading and polymorphism

An identifier is *overloaded* if it occurs with different types more than once in a type context. For example, the operator $<$ is used simultaneously for denoting the "less" relation on integers and on strings. Or the identifier $\mathbb{Z}$ denotes the *set* and the *ring* of integers. Overloading is sometimes called *ad-hoc polymorphism*.

Polymorphism means that the type system admits substitutable type variables in type expressions. Often, Greek letters α, β, ... are used for such variables. Types containing variables are called polymorphic, otherwise monomorphic. A value is called polymorphic if its type is polymorphic. A polymorphic function can be applied to values of *different* types [Gun92].

We distinguish between *implicit* and *explicit* polymorphism, depending on the implicit or explicit quantification of the type variables in types. Implicit polymorphism is widely used in typed functional programming languages, usually in conjunction with type inference in the style of Milner [Mil78]. Explicit polymorphism arises for example in Girard's and Reynolds second order λ-calculus [Rey74, GLT89] or in systems with the possibility of forming dependent product types and a kind of types [BH90], such as the calculus of constructions [H$^+$95].

Not to be confused with implicit and explicit polymorphism are *implicit* and *explicit typing* [BH90]. The latter — the Church style — annotates the bound variables of λ-abstractions with types, whereas the former — the Curry style — writes bound variables without type annotations, using in fact the untyped λ-calculus as term language. Therefore, it needs more powerful type-inference algorithms.

An important property of polymorphism in Curry style systems is, that an identifier can have more than one type! In conjunction with a higher-order type system, we get in fact infinite possible types that can be derived. Note that this does not mean that the identifier is overloaded.

Typical examples of polymorphic functions are identity, function composition or reversing a list. Their types — written in an implicit-polymorphic style — are

- $\alpha \to \alpha$
- $(\beta \to \gamma) \times (\alpha \to \beta) \to (\alpha \to \gamma)$
- $\mathsf{list}(\alpha) \to \mathsf{list}(\alpha)$

Using explicit polymorphism, one would write

- $\forall\,\alpha\;.\;\alpha\to\alpha$
- $\forall\,\alpha,\beta,\gamma\;.\;(\beta\to\gamma)\times(\alpha\to\beta)\to(\alpha\to\gamma)$
- $\forall\,\alpha\;.\;\mathsf{list}(\alpha)\to\mathsf{list}(\alpha)$

Note that functions taking *polymorphic* arguments cannot be typed by implicit polymorphism. The type variables of the argument type requires explicit quantification. For example, the type of functions which take a binary polymorphic function and produce a unary polymorphic function [Tur91]:

- $(\forall\,\alpha\;.\;\alpha\times\alpha\to\alpha)\;\to\;(\forall\,\alpha\;.\;\alpha\to\alpha)$

We will usually omit the surrounding $\forall$-quantifiers.

2.2.5 Coercions

Coercions are *implicit type conversion* operations for converting elements of one type to elements of another type [ASU86, Rey80, Web93]. A coercion $\phi : A \to B$ between types A and B implies that every expression with type A can also be viewed as an expression having type B, because the coercion can be implicitly applied. This property is usually called *subsumption*.

Coercions are commonly used by interpreters or compilers to lift arguments in an application — without explicit user request — so that they are compatible with the operator's source type. Note that therefore a coercion of type $A \to A$ is not very useful.

The term "lifting" used in the preceding paragraph does not mean that coercions have to be injective. The following example demonstrates this. Using the canonical ring homomorphism $\phi : \mathbb{Z} \to \mathbb{Z}_6$ between the rings $\mathbb{Z}$ and $\mathbb{Z}_6$ as a coercion, every integer constant symbol can also be used as a constant symbol in $\mathbb{Z}_6$. Thus, they allow a restricted style of overloading. But note the difference to real overloading where an identifier can have completely unrelated types.

If there is a coercion $\phi : A \to B$ then the type A becomes a *subtype* of type B. This is the so-called "subtypes as coercion semantics" [Gun92, Sch94].

2.2.6 Concrete and abstract syntax

Given an alphabet A then a *language* is a subset of A^*. A *grammar* of a language is some rule set that describes how words of the language are built.

The *concrete syntax* of a programming language is a description how words — the programs — of the language look. This is usually done with context-free grammars [ASU86, DSW94].

The language defined by a concrete syntax usually contains a lot of syntactical details, such as whitespaces, comments or priorities between operators, which are not relevant for the further manipulation of programs. Hence, one can abstract from those details and use a simpler domain for program manipulations, namely *syntax trees* or *parse trees*. The valid syntax trees are described by an *abstract syntax*, usually a context-free grammar, too, but a simpler one than that used for the concrete syntax.

In fact, syntax trees are terms. Therefore, an abstract syntax can also be given by a signature whose induced term algebra [Rea89, Wir90] consists of the syntax trees as carrier and the functions to build them as operations. Syntax trees are often written linearly as strings augmented with parenthesis to make the inductive structure of the abstract syntax visible. But one has to take care not to confuse parenthesis as meta-symbols with parenthesis in the object syntax!

We call an abstract syntax a *higher-order abstract syntax* if it embodies some kind of λ-calculus. This allows us to treat the various binding constructs found in most languages in a uniform way [PE88], reducing name-clash problems of variables to the *lambda*-calculus. Additionally, with the use of some typed λ-calculus the binders in fact get a (higher-order) type. Hence, we call them *binding operators*. This idea goes back to Church [Chu40] and is widely used nowadays in theorem-proving environments [Pau94] and program transformation systems [PE88].

The task of a parsing system, usually separated into scanner and parser, is to recognize if a given word in A^* is in the language defined by the concrete syntax and if so, to translate it into a unique tree representation defined by the corresponding abstract syntax [ASU86].

Chapter 3

Mixfix Syntax

High-level languages usually allow us to define more general application syntax for operators than the conventional prefix notation found in most programming languages. This leads to programs which are more readable and which acquire a "mathematical flavor". But this possibility is often restricted to the declaration of infix operators.

The possibility of mixfix declarations for operators — especially popular in algebraic-specification environments — permits a much more flexible application syntax by supporting various fixities. This yields a more general concrete syntax which is therefore called *mixfix syntax*. In the next two sections, we formally introduce mixfix declarations and give various examples. Thereafter, we investigate different forms of ambiguity, naturally arising in such a general syntax definition mechanism, and we see how these ambiguities can be resolved. Finally, we compare the strength of mixfix declarations to express languages with that of context-free grammars and have a look at some systems supporting mixfix syntax.

We assume in this chapter an alphabet C of characters, a set W of whitespace characters, and a set of ordinary characters S, where $W \subseteq C$, $S \subseteq C$, and $W \cap S = \{\}$.

3.1 Mixfix declaration

A mixfix declaration is an extended declaration for operators: In addition to declaring the operator's source and target, we also define its fixity. This is done by using a special symbol — the *placeholder* — to express the positions of the arguments in future applications of this operator. Optionally, further parsing information can be stated, such as priorities or implicit associativities. The distinguishing feature of mixfix declarations is that *at the same time* as the operator's source and target are declared, its syntactic form is defined [G+92, p. 6].

With the exception of the underlying type system, all languages supporting mixfix declarations use more or less the following syntactical scheme, consisting of the operator's *syntactical shape*, its *type* and *attributes* (with slight changes to the underlying alphabet and the operator attributes).

We call the words of the set S^* *separators*. Given a placeholder character $p \in C - W - S$, then a mixfix declaration is defined as

$$lp^{m_0}s_1p^{m_1}s_2 \cdots s_kp^{m_k}r \quad : \quad A_1 \times \cdots \times A_n \to B \qquad attrib$$

where $l, r \in S^* \cup \{\varepsilon\}$ are optional separators, the $s_i \in S^*$ are separators (where $1 \leq i \leq k$), and the $m_i > 0$ count the number of consecutive placeholders ($0 \leq i \leq k$). The cartesian product $A_1 \times \cdots \times A_n$ represents the source type, where the individual A_i are arbitrary types, and B the target type of the operator. To have a meaningful declaration, the arity n of the declared operator must be equal to the number of placeholders, $n = \sum_{i=0}^{k} m_i$.

The non-terminal *attrib*, possibly empty, is used for declaring further properties of the operator, such as its priority. We will define it formally in a later chapter.

The string $lp^{m_0}s_1p^{m_1}s_2 \cdots s_kp^{m_k}r$ is called the *form* of the operator because it defines its syntactical shape. According to the above definition the form cannot be the empty string. This representation is unique in the following sense: Given a $w \in (S^* \cup \{p\})^+$ then l, r, the numbers n, m_i and the s_i are uniquely determined.

In Table 3.1 we introduce a classification of operators defined by mixfix declarations. An open operator is left-open and right-open, and a closed operator is left-closed and right-closed. A bracketing operator surrounds

Classification	Condition
left-open operator	$l = \varepsilon$
right-open operator	$r = \varepsilon$
left-closed operator	$l \neq \varepsilon$
right-close operator	$r \neq \varepsilon$
open operator	$l = \varepsilon \wedge r = \varepsilon$
closed operator	$l \neq \varepsilon \wedge r \neq \varepsilon$
bracketing operator	$(k = 0 \wedge l \neq \varepsilon \wedge r \neq \varepsilon) \vee$ $(k = 1 \wedge (l \neq \varepsilon \vee r \neq \varepsilon)) \vee$ $k \geq 2$
invisible operator	$l = \varepsilon \wedge r = \varepsilon \wedge k = 0$
coercion operator	$l = \varepsilon \wedge r = \varepsilon \wedge k = 0 \wedge m_0 = 1$

Table 3.1: Classification of mixfix operators.

at least one argument by separators, and we also call its separators *delimiters*, because they isolate or bracket intermediate arguments in an application. The property of an invisible operator is the lack of any separator. Additionally, a coercion operator is invisible and takes exactly one argument.

3.2 Example fixities expressed by mixfix declarations

From now on, instead of the meta-variable p we use the concrete placeholder character "_". Most systems supporting mixfix declarations use this character (cf Section 3.5). In some mathematical textbooks a centralized dot "·" can be found for denoting the position of the argument in a bracketing operator of arity one [Tri80, Sch94]. In [Cro93], Crole uses parallel "+" and "−" for the same purpose. In addition, he uses them for partial evaluation, but does not explain why he uses two different placeholder symbols.

The examples show the variety of fixities which can be achieved by mixfix declarations. Some of them require a type system that supports polymorphic and higher-order types. We begin with some simple and well-known fixities:

- *Prefix* operators are the simplest examples of left-closed and right-open operators with the condition $k=0$. We declare the unary plus operator on the integers and the operator denoting logical negation on the booleans:

$$\begin{array}{rcl} +\_ & : & \mathbb{Z} \to \mathbb{Z} \\ \neg\_ & : & \mathbb{B} \to \mathbb{B} \end{array}$$

- Analogously, *postfix* operators are the simplest left-open and right-closed operators (also $k = 0$). A typical example is the factorial operator or the primality predicate:

$$\begin{array}{rcl} \_! & : & \mathbb{N} \to \mathbb{N} \\ \_\,\mathsf{prime} & : & \mathbb{N} \to \mathbb{B} \end{array}$$

- *Infix* operators are open and possess one separator (i.e., $k = 1$). Usually graphical characters are used for the separator. Examples are integer binary plus, the logical "and" operator for booleans and the divisibility relation on the integers:

$$\begin{array}{rcl} \_ + \_ & : & \mathbb{Z} \times \mathbb{Z} \to \mathbb{Z} \\ \_ \wedge \_ & : & \mathbb{B} \times \mathbb{B} \to \mathbb{B} \\ \_ \mid \_ & : & \mathbb{Z} \times \mathbb{Z} \to \mathbb{B} \end{array}$$

- *Roundfix* operators delimit their argument(s) by two separators, that is, they are closed and $k = 0$. They are simple examples of bracketing operators. Their fixity is also called *matchfix* [Soi95] or *outfix* [G+92]. An example is the operator for denoting the absolute value of an integer:

$$|\_| \quad : \quad \mathbb{Z} \to \mathbb{N}$$

  A surprising example of a roundfix operator is the following. We declare parenthesis, which are usually used for grouping subexpressions within expressions to be an operator on an arbitrary type, expressed by the use of the type variable α:

$$(\_) \quad : \quad \alpha \to \alpha$$

  If we define the identity operation as the denoted function then we get the same effect of grouping, i.e., overriding default priorities and associativities, as in languages where parenthesis are hard-coded and cannot be used for other purposes! Note the need for polymorphism.

The power of mixfix declarations can be seen by declaring operators which do not fit into the above well-known fixity schemes:

- *Juxtaposition* operators expect consecutive arguments without separators between them. Thus, their classifying property is $\exists i.m_i{>}1$. We do not require a complete absence of separators, but only that at least two arguments are written without separator between them. Typical examples are multiplication of integers and function application, where the identifier denoting the function is written prefixed with respect to the operands:

$$\begin{array}{rcl} \_\,\_ & : & \mathbb{Z} \times \mathbb{Z} \to \mathbb{Z} \\ \_\,\_ & : & (\alpha \to \beta) \times \alpha \to \beta \end{array}$$

  The latter example requires a higher-order polymorphic type system.

- Mixfix syntax allow an elegant solution for the declaration of *coercion* operators. Because coercion operators are applied to an argument without explicit user request they are in fact *invisible*. An example is the declaration of the coercion between integers and rationals:

$$\_ \;:\; \mathbb{Z} \to \mathbb{Q}$$

  The following coercion suggests as semantics the extraction of the abelian group being present in every ring:

$$\_ \;:\; \mathsf{Ring} \to \mathsf{AbelianGroup}$$

- Some fixities are even more complicated than the above ones but can nevertheless be treated by mixfix declarations (hence, the whole declaration formalism is called *mixfix*). The functional if–then–else construction or the addition modulo an integer are such examples:

$$\begin{array}{rcl} \mathsf{if}\_\mathsf{then}\_\mathsf{else}\_ & : & \mathbb{B} \times \alpha \times \alpha \to \alpha \\ \_ + \_ \,\mathsf{mod}\, \_ & : & \mathbb{Z} \times \mathbb{Z} \times \mathbb{Z} \to \mathbb{Z} \end{array}$$

  But also the constructor for univariate polynomial rings in an indeterminate (having type Symbol) is an example of a mixfix operator:

$$\_[\_] \;:\; \mathsf{Ring} \times \mathsf{Symbol} \to \mathsf{Ring}$$

Such constructions are rarely supported by simpler formalisms!

Apart from defining a general syntax for operator applications, mixfix declarations still have the task of assigning a type to an identifier. But what is the identifier in a mixfix declarations? We introduce the following convention: The *whole form* in a declaration is used as the identifier that denotes the underlying operation, i.e., we identify form and identifier. Thus, we can distinguish between a *unary* plus operation, whose corresponding identifier is written "+_", and a *binary* one, written "_+_", both using the same separator. This distinction would not be possible if only "+" were used as the identifier. Also, operators without any separators in their form, such as "__", can be identified with this method. That is why we excluded whitespace characters within separators, because this would contradict our assumption that identifiers do not contain any whitespaces.

Note that this problem does not arise in systems possessing first-order type systems, because there, functions themselves cannot be accessed in a first-class manner!

From now on a declaration is either a mixfix declaration, including possible operator attributes, or a *normal declaration* as encountered in Section 2.2.2. We call a list of such declarations a *context*. We still need normal declarations within contexts, at least for the declaration of constant symbols. By omitting the operator attributes, a context becomes a type context. A context represents simultaneously a grammar for some language, as well as it declares identifiers.

We adopt the convention that identifiers declared by normal declarations contain no placeholders. This makes the distinction between mixfix operators and other identifiers simple: only the former contain placeholders.

3.3 Ambiguities and their resolving

As we saw in the last section, mixfix declarations are a very general tool for defining various application syntax for operators. Thus, they can be used for inducing languages on C^*, similarly to the languages induced by context-free grammars. With their help we define several notions of *ambiguity*, an important concept in the theory of grammars and parsers. Unambiguity is essential for parsing because otherwise it is not possible to generate a unique abstract syntax tree for an input

string. We investigate how ambiguities can appear and how they can be resolved, while classifying them.

3.3.1 Definitions

We assume in this section an arbitrary type system with the following minimal conditions:

- We can form types of the shape $A_1 \times \cdots \times A_n \to B$, a function type with a possible tuple type as source. Depending on the specific system the types $A_1, \ldots, A_n, B$ have to be sorts or can be arbitrary monomorphic or polymorphic types.
- We admit a reflexive — but not necessarily transitive — *subtype* relation, which may depend on a type context. The relation is denoted by $A \preceq_\Sigma B$, where A, B are types and Σ is a type context. For any identifier id and type A with $(id, A) \in \Sigma$, the type A is called a *principal* type of id. Any type B with $A \preceq_\Sigma B$ is called a *derived* type of id. If we have $A \preceq_\Sigma B$, every identifier with principal type A has also B as a (derived) type by the principle of subsumption.

  This subtyping relation can be, for example, generic instantiation between types in case of a polymorphic type system [BH90], corresponding to Mitchell's containment relation [Mit90]. Say, we would have

  $$\mathsf{list}(\alpha) \to \mathsf{list}(\alpha) \;\preceq_\Sigma\; \mathsf{list}(\mathbb{Z}) \to \mathsf{list}(\mathbb{Z})$$

  for an arbitrary type context Σ.

  A system supporting coercions would permit us to derive $\mathbb{N} \preceq_\Sigma \mathbb{Q}$ by transitivity of coercions, where Σ contains at least the declarations $\_ : \mathbb{N} \to \mathbb{Z}$ and $\_ : \mathbb{Z} \to \mathbb{Q}$. This example also shows the possible context-dependence of $\preceq_\Sigma$.

We define the set M relative to a given type context Σ. M contains all mixfix operators — except coercion operators — with all possible derived source and target types, and is used in the definition of ambiguity later on:

$$M = \{\ (op,\ A',\ B') \ \mid\ (op,\ A \to B) \in \Sigma,\ A \to B \preceq_\Sigma A' \to B',\ op \text{ is a mixfix operator but not a coercion}\ \}$$

Depending on $\preceq_\Sigma$, the set M can be infinite.

We exclude coercion operators from the set M because they do not have any syntactic appearance but influence only the derivation of types from principal types.

Using their separators, all mixfix operators in M naturally induce a "print" function on C^*. For each $(op, \underbrace{A_1 \times \cdots \times A_n}_{=A}, B) \in M$ we get

$$\begin{aligned} d_{op,A,B} &: (C^*)^n \to 2^{C^*} \\ d_{op,A,B}\,(w_1, \ldots, w_n) &= lW^*w_1W^*w_2 \cdots w_{m_0}W^*s_1 \cdots s_k \cdots w_nW^*r \end{aligned}$$

where $op = l\_^{m_0}s_1\_^{m_1}s_2 \ldots s_k\_^{m_k}r$.

The application $d_{op,A,B}(w_1,\ldots,w_n)$ denotes the set of those strings which represent syntactically the application of op to the operands $w_1,\ldots,w_n$ with optional whitespaces between the individual arguments and separators. For example, $d_{\_+\_,\mathbf{Z},\mathbf{Z}}(\text{“1”}, \text{“23”}) = \{$ “1+23”, “1 + 23”, “1 + 23”, $\ldots\}$.

Note that coercions would simply induce the function $w \mapsto \{w\}$ if they were in M. Furthermore, we have a trivial condition on the lengths of the words, not being valid if coercions were included in M:

$$\forall\, w \in d_{op,A,B}\,(w_1, \ldots, w_n) \;.\; |w| > \sum_{i=1}^{n} |w_i| \tag{3.1}$$

The language $L'_\Sigma \subseteq C^*$ defined below is a coarse approximation of the set of strings that can be produced by applying the functions $d_{op,A,B}$ repeatedly, where the identifiers in Σ are the base case. L'_Σ is the least set fulfilling the following two conditions:

- $\forall\, (id, T) \in \Sigma \;.\; id \in L'_\Sigma$
- $\forall\, (op, \overbrace{A_1 \times \cdots \times A_n}^{=A}, T) \in M \;.\; \forall\, w_1, \ldots, w_n \in L'_\Sigma \;.\; d_{op,A,B}(w_1, \ldots, w_n) \subseteq L'_\Sigma$.

Every word in L'_Σ is either an identifier or can be represented by a finite number of applications of the functions $d_{op,A,B}$, using condition 3.1.

A word $w \in L'_\Sigma$ is called *unambiguous* if it is either a non-overloaded identifier contained in the type context Σ or if it can be generated in only one way by the above functions. Formally this means: If w is an identifier then it is not overloaded

$$\forall\ (id_1, T_1),\ (id_2, T_2) \in \Sigma\ \ .\ \ w = id_1 = id_2\ \ \rightarrow\ \ T_1 = T_2$$

and if w is contained in the resulting set of applying the function $d_{op,A,B}$ to some words $w_1, \ldots, w_n \in L'_\Sigma$ and of $d_{op',A',B'}$ applied to some $v_1, \ldots, v_m \in L'_\Sigma$

$$d_{op,A,B}(w_1, \ldots, w_n)\ \ni\ w\ \in\ d_{op',A',B'}(v_1, \ldots, v_m)$$

then it is the only possible way

- $\forall\ (id, T) \in \Sigma\ \ .\ \ id \neq w$
- $op = op'\ \ \wedge$
  $\Big|\ \{\ T\ \ |\ \ T \in \Sigma.op\ \wedge\ (T \preceq_\Sigma A \rightarrow B\ \ \vee\ \ T \preceq_\Sigma A' \rightarrow B')\ \}\ \Big|\ =\ 1$
- $n = m\ \ \wedge\ \ \forall\ 1 \leq i \leq n\ .\ (w_i = v_i\ \wedge\ w_i$ unambiguous).

That is to say that w is not an identifier, the types $A \rightarrow B$ and $A' \rightarrow B'$ are derived from exactly one and the same principal type, and the arguments themselves are unambiguous.

We call a word $w \in L'_\Sigma$ *ambiguous* if w is not unambiguous.

Γ serves as an exemplary context in the rest of this section:

$$\begin{array}{rcl@{\qquad}rcl}
\Gamma\ = & x & : & \mathbb{Z}\,, & y & : & \mathbb{Z}\,, \\
 & z & : & \mathbb{Z}\,, & f & : & \mathbb{Z} \rightarrow \mathbb{Z}\,, \\
 & -_ & : & \mathbb{Z} \rightarrow \mathbb{Z}\,, & _-_ & : & \mathbb{Z} \times \mathbb{Z} \rightarrow \mathbb{Z}\,, \\
 & __ & : & \mathbb{Z} \times \mathbb{Z} \rightarrow \mathbb{Z}\,, & __ & : & (\mathbb{Z} \rightarrow \mathbb{Z}) \times \mathbb{Z} \rightarrow \mathbb{Z}, \\
 & |_| & : & \mathbb{Z} \rightarrow \mathbb{Z}\,, & _|_ & : & \mathbb{Z} \times \mathbb{Z} \rightarrow \mathbb{B}\,, \\
 & _|_|_ & : & \mathbb{Z} \times \mathbb{Z} \times \mathbb{Z} \rightarrow \mathbb{B}\,, & _=_ & : & \mathbb{Z} \times \mathbb{Z} \rightarrow \mathbb{B}\,, \\
 & _=_ & : & \mathbb{B} \times \mathbb{B} \rightarrow \mathbb{B}\,, & _\wedge_ & : & \mathbb{B} \times \mathbb{B} \rightarrow \mathbb{B}
\end{array}$$

Apart from the normal identifiers x, y, z and f, the type context Γ declares a unary and a binary minus operator, juxtaposed multiplication and application, absolute value, the divisibility relation for 2 and 3 integers, equality for integers and booleans, and the logical $\wedge$-operator. Note the use of a higher-order type for the application operator.

Using Γ, the words "x" and "$-y$" are unambiguous. The words "$f\,x$", "$x\,y$", "$x - y\,z$", "$x - y - z$" and "$x - y$" are all ambiguous.

Usually, most words in L'_Σ are ambiguous for an arbitrary type context Σ! Hence, we define a more refined version of L'_Σ taking the types of the identifiers in Σ into account.

For each type T we define simultaneously the sets $L_\Sigma(T) \subseteq C^*$, which are the least sets fulfilling

- $\forall\ (id, T) \in \Sigma\ .\ T \preceq_\Sigma T' \ \rightarrow\ id \in L_\Sigma(T')$
- $\forall\ (op, \overbrace{A_1 \times \cdots \times A_n}^{A}, B) \in M\ .\ \forall\, w_1 \in L_\Sigma(A_1), \ldots,\, w_n \in L_\Sigma(A_n)\ .$
  $d_{op,A,B}(w_1, \ldots, w_n) \subseteq L_\Sigma(B).$

For $w \in L_\Sigma(T)$, we write $\Sigma \vdash w : T$, pronouncing "the word w has type T in type context Σ".

Let $L_\Sigma = \bigcup_T L_\Sigma(T)$, where T ranges over all types. We have $L_\Sigma \subseteq L'_\Sigma$ but not necessarily $L_\Sigma = L'_\Sigma$. Also $L_\Sigma(T) \cap L_\Sigma(T') \neq \{\}$ is possible for arbitrary types T and T'.

The notion of unambiguity also needs a refinement taking types into account. A word $w \in L(\Sigma)$ is called *unambiguous in the type context* Σ if it can be generated in only one way:

If w is a normal identifier then it is not overloaded (which is the same condition as before)

$$\forall\ (id_1, T_1),\ (id_2, T_2) \in \Sigma\ .\ w = id_1 = id_2 \rightarrow T_1 = T_2$$

and otherwise, w is in exactly one set generated by some $d_{op,A,B}$, where $A = A_1 \times \ldots \times A_n$ and $A' = A'_1 \times \ldots \times A'_m$:

If

- $\Sigma \vdash w_1 : A_1, \ldots,\ \Sigma \vdash w_n : A_n,\ \Sigma \vdash v_1 : A'_1, \ldots,\ \Sigma \vdash v_m : A'_m$

- $d_{op,A,B}(w_1, \ldots, w_n) \ni w \in d_{op',A',B'}(v_1, \ldots, v_m)$

then

- $\forall\ (id, T) \in N \ .\ id \neq w$
- $op = op' \ \wedge$
  $\left| \{\ T \mid T \in \Sigma.op \ \wedge \ (T \preceq_\Sigma A \to B \ \vee \ T \preceq_\Sigma A' \to B')\ \} \right| = 1$
- $n = m \ \wedge$
  $\forall\ 1 \leq i \leq n\ .\ (w_i = v_i \ \wedge \ w_i$ unambiguous in the type context $\Sigma)$.

This definition is almost the same as before. The difference lies in the strengthened assumption, which takes the types of the argument words into account, according to the operators' sources.

As above, $w \in L_\Sigma$ is called *ambiguous in the context* Σ if it is not unambiguous in Σ.

Taking the same example strings as above we now get: "x" and "$-y$" are still unambiguous in the context Γ. But "$f\,x$" and "$x\,y$" are now unambiguous in Γ, too. The other words "$x - y\,z$", "$x - y - z$" and "$x - y$" stay still ambiguous in Γ. In general, an unambiguous word is also unambiguous using an arbitrary type context, but not necessarily vice versa.

The next sections give a classification of various ambiguities and their method of solution. They are presented in an increasing order in the following sense: If a method n disambiguates a possible ambiguity then we are done. Otherwise, we try method $n + 1$ and so on.

3.3.2 Use of the type context

The step from the language L'_Σ to L_Σ is nothing other than the use of the identifiers' type information — contained in the type context — for disambiguating words. Révész calls this need of the context *active semantic ambiguity* [RL91]. A typical example is application vs. multiplication, which can both be written juxtaposed and hence, the words "$f\,x$" and "$x\,y$" become unambiguous using Γ. Note that untyped languages cannot eliminate such ambiguities.

Another example is multiple consecutive applications. Using a suitable context, the word "*sin cos x*" can be unambiguously read by anyone

with a knowledge of trigonometric functions, as applying the sine function to $cos(x)$ getting "$sin(cos(x))$" and not "$(sin(cos))(x)$". The word "$map\ succ\ l$" means "$(map\ succ)(l)$", the curried functional map applied to the successor function and the resulting function applied to list l, as every functional programmer will be familiar with. Thus, thanks to using appropriate contexts, application does not need any implicit associativity.

In fact, this problem of needing the context during parsing also arises in general purpose languages. For example in the programming language OBERON [Wir88a] where a function application and a type cast have the same syntactic appearance and can only be distinguished by using the surrounding context. Analogously for record and module component access.

But not all ambiguities can be solved by using the type context as the next sections show.

3.3.3 Priority conflicts

Given a context Σ that contains a right-open operator id and a left-open operator id' with the following forms and types:

$$\underbrace{l \cdots \_}_{id} : A_1 \times \cdots \times A_n \to B, \quad \underbrace{\_ \cdots r}_{id'} : A'_1 \times \cdots \times A'_{n'} \to B'$$

Furthermore, we can deduce for a string w and for id and id'

$$\begin{aligned} \Sigma &\vdash w : C \\ \Sigma &\vdash id : A_1 \times \cdots \times A_{n-1} \times C \to C \\ \Sigma &\vdash id' : C \times A'_2 \times \cdots \times A'_{n'} \to C\ . \end{aligned}$$

Assume two applications of the operators id and id' written side by side and with w standing "between" them as an argument of type C. Then we do not know if the resulting function composition has to be read from left to right or from right to left; due to the deduced types, w can be an argument of either of the operators:

$$(l \cdots w) \cdots r \quad \stackrel{?}{=} \quad l \cdots w \cdots r \quad \stackrel{?}{=} \quad l \cdots (w \cdots r)$$

Associativity of string concatenation leads to precisely this problem. For example, using the context Γ, the string "$x - y\,z$" can be read as "$(x - y)\,z$" or as "$x - (y\,z)$".

Priorities (or *precedences*) between the two operators can be used in these situations for resolving the ambiguity, a well-known technique in ordinary parsing theory [ASU86]. Priorities can be given by associating natural numbers to the operators and using the ordinary $<$-relation on $\mathbb{N}$ — resulting in *absolute* priorities — or by directly relating two operators, giving *relative* priorities. The former has the disadvantage that all operators get a priority due to the totalness of the used order, which need not necessarily be the case when using relative priorities.

Note that priorities also depend on the types of operators. The *same* operator form can have different priorities according to the types of the overloaded operator! For example, we want to express that within the context Γ, juxtaposed multiplication should have a higher priority than function application. So the string "$f\,x\,y$" should be read as "$f\,(x\,y)$" and not as "$(f\,x)\,y$". But both operators have the same form! Hence, we have to take the types of the two operators into account for stating the precedence. Note that the word "$x\,f\,y$" can only be understood as "$x\,(f\,y)$" due to the use of the context, and therefore, no priority is needed to disambiguate.

A similar example can be found in [vG90]. Given "$a = b \wedge c$" and a boolean variable c. According to the context that declares a and b, we read the string as "$a = (b \wedge c)$" if the two variables are declared as booleans but as "$(a = b) \wedge c$" if they denote integers. The latter reading is the only possible one within the integer context. But the former needs a priority between the boolean equality operator $\_ = \_$ and $\_ \wedge \_$ to resolve the ambiguity. Hence, the priorities are once again type dependent.

Priorities only help in situations where two applications with a right-open and a left-open operator are written consecutively. But they are not restricted to binary infix operators! They can relate arbitrary right-open and left-open operators. For example, the word "if b then 2! else 3!" has to be disambiguated by stating priorities between the operator if $\_$ then $\_$ else $\_$ and the factorial operator $\_!$. Note the difference to "if b then 2! else 3", which does not need any priority for disambiguation, because the first and second argument of the operator are fully bracketed.

3.3.4 Implicit associativities

Implicit associativities are a refinement of priorities. An operator is either implicit *left*, *right*, or *not* associative. These operator attributes are needed if in the preceding section's setting both operators have the *same* priority. Then again, we do not know if grouping should be from the left or from the right. If both operators are implicit left associative then grouping is to the left, and if both are right associative then grouping is to the right. In case of mixed or no implicit associativities, *both* readings are possible, resulting in an ambiguity [App92, p. 15].

An implicit associativity is always required in case of an open operator if we are able to write several consecutive applications with it. For example, in the case of the operator $\_ - \_$, where its (implicit) left associativity allows it to disambiguate the word "$x-y-z$" being read as "$(x-y)-z$".

Note that implicit associativities are a pure syntactical property of an operator and should not be confused with the semantical notion of associativity as the preceding example shows, too.

3.3.5 Explicit type information

Because mixfix declarations allow overloaded identifiers we always get ambiguous words. Using an additional type annotation — a (type) *qualification* — these can nevertheless be resolved in a straightforward manner. But there are also other cases where explicit information of the expected type can help as the following example, found in [Soi95], shows: Using the context Γ, the word "$x|y|z$" can only be read unambiguously if the type of the whole expression is provided. It can be

- "$\big(x\,|y|\big)\,z$" (multiplication of x with the absolute value of y and with z, left-associativity of multiplication assumed) if we expect an integer result, or
- "$\big(x|y\big) \wedge \big(y|z\big)$" ($x$ divides y and y divides z) expecting a boolean result.

Using the type context for x, y, and z does not help because their types are the same in both possible readings!

3.3.6 Other ambiguities

Besides the listed possibilities there are still further ambiguities possible. Unfortunate choice of variable names and separators in operator forms can lead to ambiguous readings.

The standard example is "$x - y$", also contained in Γ. Does it denote subtraction or multiplication written juxtaposed, "$x\,(-y)$" ? Neither the context, nor an explicit type annotation help us here; the result type is $\mathbb{Z}$ in both cases.

Some ambiguities cannot be resolved with the above methods; in Chapter 5 we present a method for handling some of them.

In fact, some ambiguities can only be resolved with semantical knowledge. Take the following part of a type context covering simultaneously integer and rational arithmetic. Due to associativity of taking greatest common divisors (gcd), the gcd operator is written in a rather unusual infix style [vG90, p. 146].

$$
\begin{array}{rcl@{\qquad}rcl}
 & \ldots & & & & \\
x & : & \mathbb{Z}\,, & y & : & \mathbb{Z}\,, \\
\_+\_ & : & \mathbb{Z}\times\mathbb{Z}\to\mathbb{Z}\,, & \_\mathsf{gcd}\_ & : & \mathbb{Z}\times\mathbb{Z}\to\mathbb{Z}\,, \\
\_ & : & \mathbb{Z}\to\mathbb{Q}\,, & \mathsf{num}\_ & : & \mathbb{Q}\to\mathbb{Z}\,, \\
\_+\_ & : & \mathbb{Q}\times\mathbb{Q}\to\mathbb{Q}\,, & \_\mathsf{gcd}\_ & : & \mathbb{Q}\times\mathbb{Q}\to\mathbb{Q}\,, \\
 & \ldots & & & &
\end{array}
$$

Both the strings "$\mathsf{num}(x + y)$" and "$\mathsf{num}(x\ \mathsf{gcd}\ y)$" are ambiguous because the coercion — the canonical embedding of $\mathbb{Z}$ into $\mathbb{Q}$ — can be applied either *before* or *after* adding and taking the gcd, respectively (the operator num forces both arguments to have type $\mathbb{Q}$).

The first ambiguity can be abolished on the semantical level, say by theorem proving, because addition commutes with the canonical embedding. Hence, it does not matter in which order the coercion is applied. In contrast, the second word stays ambiguous because the gcd operation does not commute with the embedding.

Analogously with "$x\ \mathsf{gcd}\ y\ \mathsf{gcd}\ x$". This string becomes ambiguous if the gcd operator is not equipped with an implicit associativity. Because the underlying operation is in fact associative, this ambiguity is resolvable on a semantical level [vG90, p. 146].

We do not pursue these semantical issues further but refer to [MW94],

a joint work with Andreas Weber, which deals with some of them.

3.4 The relation to context-free grammars

The language L'_Σ of the preceding section can be described very easily by a context-free grammar with a single non-terminal symbol S. Each $(id, T) \in \Sigma$ induces the production

$$S \implies id$$

and each mixfix operator op in M with $op = lp^{m_0}s_1p^{m_1}s_2 \cdots s_k p^{m_k} r$ induces the production

$$S \implies l \underbrace{S \cdots S}_{m_0} s_1 \underbrace{S \cdots S}_{m_1} s_2 \cdots s_k \underbrace{S \cdots S}_{m_k} r$$

Note that ambiguity as defined for context-free grammars does not correspond to either of the two ambiguities defined in the preceding chapter, because overloading of identifiers is not reflected in the above grammar.

In general, the language L_Σ cannot be described by a context-free grammar, because an infinite amount of productions would be needed to simulate the infinite number of occurring types in case of polymorphism and higher-orderness. Take the following type context, containing a single juxtaposed identifier with a polymorphic higher-order type:

$$\Gamma \quad = \quad \_\_ \; : \; \alpha \times \alpha \to (\alpha \to \alpha)$$

With $v =$ "$\_\_$" we get

$$L_\Gamma \; = \; \{\, v,\; vW^*v,\; vW^*vW^*vW^*v,\; \ldots \,\} \; = \; \{\, v(W^*v)^{2^i-1} \;\mid\; i \geq 0 \,\}$$

But L_Γ is not context-free, which can be shown by using Bar-Hillel's pumping lemma [DSW94, p. 287].

On the other hand, mixfix declarations are powerful enough to describe every positive context-free grammar[1]. The non-terminal symbols are treated as sorts and a production of the form

[1] Positive means that the grammar does not contain productions with the empty word ε on the right side. Every context-free language not containing ε can be described by a positive context-free grammar [DSW94].

$$B \implies lA_0A_1 \cdots A_{m_0}s_1A_{m_0+1} \cdots s_k \cdots A_nr$$

with $l, r, s_1, \ldots, s_n$ terminal symbols, l and r possibly empty, and $A_1, \ldots, A_n$ non-terminal symbols, is translated into a mixfix declaration by

$$lp^{m_0}s_1p^{m_1}s_2 \cdots s_kp^{m_k}r \quad : \quad A_1 \times \cdots \times A_n \to B \ .$$

Because the occurring argument and result types are all sorts, a first-order monomorphic type system is in fact enough for the encoding.

3.5 Systems supporting mixfix syntax

This section discusses systems that support some kind of mixfix syntax. Furthermore, they are compared with two parser generators based on context-free grammars. We briefly present the individual systems.

The interactive grammar construction and expression parsing tool CIGALE [Voi86] was specially designed for the incremental construction of grammars and the parsing of expressions. Grammars are treated as modular units that can be combined together and extended incrementally by new mixfix declarations with arbitrary fixities. CIGALE is used as front-end in the algebraic-specification system ASSPEGIQUE [BCC+87]. Due to its origin in algebraic specification, CIGALE supports only first-order monomorphic types. The abstract data type of the natural numbers serves us as an example of a specification written in a CIGALE:

```
SPEC: nat;
  OPERATIONS
  0:    -> nat;
  succ: nat -> nat;
  _+_:  nat nat -> nat;
  _*_:  nat nat -> nat;
VARIABLES:
  x,y: nat;
AXIOMS:
  x + 0      == x;
  x + succ y == succ (x + y);
  x * 0      == 0;
  x * succ y == x + (x * y);
```

```
END nat;
```

The placeholder-less declaration for the identifier `succ` is a shorthand notation for prefix operators. The use of juxtaposition for expressing the cartesian product of sorts is typical for first-order type systems where cartesian products are not first class.

ISAR [Bau92, Bau93] and the LARCH SHARED LANGUAGE (LSL) [GH93] are both algebraic-specification systems based on a traditional many-sorted algebra approach with a first-order monomorphic type system [EGL89, Wir90]. Their syntax definitional mechanism uses a restricted form of mixfix syntax.

In contrast, OBJ3 [G$^+$92] and its descendents such as FOOPS [RS92] uses a more general order-sorted approach for specifying and defining algebras [GM92]. OBJ3's power lies in the presence of theories, objects and views. Objects denote order-sorted algebraic structures, which are the models of theories. Views are used for transforming objects of one theory into models of other theories. They can be compared with the logical notion of a theory interpretation. Both theories and objects are parameterizable by objects, leading to a functional style for programming in the large, called parameterized programming. OBJ3's type system offers subsorts, which are denotationally restricted to model subset relations. As in the already mentioned systems, it does not provide higher-order or polymorphic types. The OBJ family of algebraic-specification environments was one of the first to exploit the power of mixfix declarations.

An example using OBJ3: The parameterized structure of lists, where the axiomatic part of the specification is left out for reasons of simplicity:

```
obj LIST[X :: TRIV] is
  protecting NAT .
  sorts List NeList .
  op nil : -> List .
  subsorts Elt < NeList < List .
  op __ : List List -> List   [assoc id: nil]
  op |_| : List -> Nat
  op tail_ : NeList -> List   [prec 120]
  ...
endo
```

`TRIV` is the trivial theory of structures with the only sort `Elt` and without

any operations and axioms. This theory can be used as interface theory because lists do not pose any syntactical and semantical requirement but work for arbitrary sorts. `LIST` introduces the sorts `List` and `NeList` for arbitrary and non-empty lists, respectively. Later on, `Elt` is declared as a subsort of `NeList` and the latter as a subsort of `List` The sort `Nat` used for the length operator `|_|` is made available while importing `NAT`, the structure of the natural numbers. The operator attributes within square brackets declare associativity and an identity element for the concatenation operator `__` and a precedence for operator `tail_`.

All the above algebraic-specification formalisms provide only monomorphic type systems, where the argument types of operators have to be sorts. Thus, apart from new sorts, no new type constructors can be defined. Interestingly, algebraic-specification environments with advanced higher-order and polymorphic type systems such as EXTENDED ML [San89] — an extension of SML— and SPECTRUM [BFG+91] do not currently offer any kind of mixfix syntax. In fact, ISABELLE— presented in the next paragraph — is the only system we are aware of which offers mixfix syntax *and* higher-order polymorphic types.

ISABELLE [Pau94] is a generic theorem prover supporting various logics, which are grouped into theories. ISABELLE's meta-logic is based on the simply-typed λ-calculus and uses implicitly bound type variables. Hence, it is higher-order and polymorphic. Through the use of mixfix declarations in theories, which have a slightly different syntactical appearance than our presentation, the concrete syntax of newly defined theories can be very general.

The two parser generator tools presented for comparison purposes are SDF and YACC. Based on context-free grammars, they cannot simulate the languages induced by higher-order polymorphic types as seen in Section 3.4.

SDF is the syntax definition formalism of the algebraic-specification environment ASF+SDF [K+93]. Arbitrary context-free grammars are treated. Besides the possibility of defining implicit associativities and relative priorities, SDF provides a finite sequence construct with a possible separator as an infix symbol (cf Sections 4.4 and 5.3). Note the difference to the sequence construct available in EBNF-like grammar descriptions where one cannot define such infix separators.

YACC [MB91] is a widely-used parser generator for a proper subset of

	CIGALE	ISABELLE	ISAR	LSL	OBJ3	SDF	YACC
Placeholder symbol	_	_	·	__	_		
Juxtaposition	√	√			√	√	√
Coercions	√	√				√	√
Priorities		(√)	√		√	√	√
Implicit associativities		(√)	√		(√)	√	√
Arbitrary overloading	√	√		√		√	
Sequence construct						√	
Higher-order polymorphic		√					

Table 3.2: Features of syntax-description formalisms.

context-free grammars [ASU86]. For example, ambiguous grammars are not in this subset, and hence, overloading is restricted.

We now highlight some features and deficiencies of the systems with respect to mixfix declarations. We investigate the following properties, which are summarized in Table 3.2 for the individual systems.

- Syntactical form of the placeholder symbol (this point is not applicable for the two parser generators).
- Are juxtaposition and coercion operators allowed?
- Can priorities and implicit associativities be declared?
- Can identifiers be overloaded arbitrarily?
- Is there a sequence construct with a possible infix separator available?
- Is the underlying type system higher-order and polymorphic?

CIGALE's main deficiencies are the lack of priority and implicit associativity attributes. They have to be coded into the grammar itself, leading to unnecessarily large grammars and to intermediate types without any semantical meaning. The default behavior is equal precedence between operators and left associativity. Furthermore, CIGALE does not always recognize ambiguous input.

Priority and associativity declarations are missing in LSL, too. Furthermore, neither LSL nor ISAR permit coercion operators and disallow juxtaposition. The individual arguments have to be separated by non-empty strings, hence some interesting fixities cannot be represented by these systems.

OBJ3 does not support arbitrary coercions either [G$^+$92, p. 7]. They are implicitly derived from subsorts. As opposed to all other systems, OBJ3 also permits semantical annotations for operator attributes. One can declare associativity, commutativity, and identity element properties for operators as seen in the preceding list example. On the other hand, implicit associativities for non-associative operators are not supported directly but only through so-called gathering patterns [G$^+$92, p. 22], which are rather complicated. Furthermore, arbitrary overloading is restricted by requiring regularity and coherence conditions for identifiers and their types [G$^+$92, p. 9]. As CIGALE, OBJ3 does not find all ambiguities [G$^+$92, p. 22].

Note that SDF supports all features in the table except higher-orderness and polymorphism. The latter is an implication of Section 3.4.

ISABELLE supports ordinary priority and associativity declarations only for binary infix operators. The offered precedence grammars for arbitrary mixfix declarations, where each each argument position can be weighted by a natural number, are a more general mechanism for declaring priority and associativity but have the disadvantage that they are not very readable. Overloading is restricted by the use of type classes.

Furthermore, ISABELLE provides mixfix declarations only on the global level of theories and not in local declarations (say in λ-abstractions).

None of the above systems supports mixfix syntax on the level of types. This is not surprising in the case of the algebraic-specification environments because they do not provide the possibility of defining new type constructors (except sorts). But in the case of ISABELLE this is a severe restriction because type constructors are a rich source of interesting syntactical constructions.

There is no general support for binding constructs in these mixfix systems. Once again, this is not surprising because one needs a higher-order type system for the clean handling of binders [PE88]. There, one can reduce arbitrary binding constructs to λ-abstractions (cf Section 5.4). ISABELLE supports only prefix notation for binders.[2]

[2] ISABELLE offers the possibility to hook functions written in SML into the parsing

Furthermore, none of the mixfix systems supports *relative* priorities and sequence constructs in the way that SDF does.

process; but this is beyond mixfix syntax.

Chapter 4

Mathematical Notation

Mathematical notation is a common language for describing mathematical ideas and concepts used by scientists and engineers. It evolved over centuries to get to today's alphabet of symbols and their combinations. The development of new mathematical disciplines will lead to further new notation. Hence, mathematical notation is an evolving notation. To quote Révész [Rév91]:

> Symbolic notations used in mathematics have a long history of evolution. They underwent various changes and adaptions until they reached their present state and become part of our culture. Their form, meaning, and usage is governed only by the consensus of their users without any codified standards, and anyone is free to use any notation as long as he explains it. Nevertheless, engineers and scientists from all over the world usually feel quite comfortable with these notations, which they use extensively in their daily work.

This suggests that arbitrary notation is allowed. Indeed, the resulting languages are very rich. But there are nevertheless some conventions — usually implicitly used — on how to use and extend mathematical syntax. We analyze them and investigate the properties that make mathematical notation so different to ordinary programming languages. Examples of its diversity are given, using various mathematical and computer-science textbooks as references. The references and their classifying fields are:

- algebra [Lan84]
- category theory [Pie91, AL91, Cro93]
- computer algebra [GCL92]
- discrete mathematics [GS94]
- functional analysis [Tri80, Rud88]
- logic [Bar89]
- theoretical computer science [Sch84, EL92, DSW94]
- type theory [C$^+$86, NPS90, Tho91]

We then have a look at the support of mathematical notation in various existing programming languages and systems.

We summarize some relevant related work which deals with the syntactic features and shortcomings of today's mathematical syntax.

Driscoll describes in his article some characteristics of ordinary mathematical notation and their implications for programming [Dri90]. Révész and Lynch describe mathematical notation by a context-free grammar and present a lexical and a syntactical analyzer, pointing out the difficulties of parsing the various arising ambiguities [RL91, Rév91].

In his dissertation, Zhao describes a formalization method based on keyboard and mouse input to support two-dimensional notation. Furthermore, he extends context-free grammars with context-sensitive features to handle most of the specialties of mathematical notation [Zha96].

The second part of van Gasteren's book about the shape of mathematical arguments discusses disadvantages of current mathematical notation and presents improvements [vG90]. Interesting notes on various shortcomings can be found in Backhouse's publications, too [Bac88, Bac89].

Some articles dealing with typesetting and user-interfaces for mathematical environments also contain remarks about the syntactical aspects of mathematical notation [Soi95, KS]. In the context of structured search in mathematical documents, similar remarks and examples can be found [CG96].

Cajori describes the evolution of most symbols from the beginning of mathematics until the early years of the 20th century [Caj93]. Hence, there is no account for types, higher-orderness and similar topics developed later.

4.1 The shape of identifiers and separators

The base alphabet of symbols used for forming mathematical sentences is huge [Dri90]. In addition to whitespace characters there are

- letters of different alphabets: A, x, Γ, δ, $\aleph$, ...
- digits: 0,1,2,...,9
- numerous graphical symbols: $\oplus$, $\langle$, $\lfloor$, $\downarrow$, $\uplus$, $\circ$, $\emptyset$, $\{$, $\cap$, $\diamond$, ∂, $\angle$, $\pm$, $\div$, $\cong$, $\Rightarrow$, $\perp$, $\gg$, ∇, $=$, $\wedge$, $\vdash$, $\exists$, ∞, $\|$, ...

Additionally, the alphabet contains different

- fonts: x, sin, $\mathbb{Z}$, $\mathbf{1}$, $\mathcal{F}$, $\wp$, ...
- font sizes: $_{i+j}$, ^{2n}, $\bigcup$, ...
- embellishments: $\vec{x}$, $\tilde{v}$, $\underline{\mathbb{Z}}$, ...

We can partition the mathematical alphabet into 4 classes:

- whitespaces
- letters
- digits
- graphical symbols

This partition is not different from the situation found in most programming languages, except that the last three classes are much larger — counting all the different fonts, font sizes and embellishments — than their counterparts found in ordinary programming languages.

A classification of the words generated by the mathematical alphabet into syntactic categories would be the next logical step. But this seems impossible because of the multiple use of most symbols in different separators and identifiers. The only syntactic category we can identify are bound variables, which have a rather specific shape.

Due to the plethora of available letters, bound variables occurring in mathematical statements can be (and are in fact) very short. They

consist of a single letter, optionally suffixed with digits or some special graphical characters, such as a prime. Using more than one extending digit or graphical character or even a letter is rare. The digits are often subscripted (but seldom superscripted) and the graphical characters are sometimes superscripted. Examples are f, B, α, η, m_1, x', y^*, $\tilde{z}$. Using digits or graphical characters as bound variables looks very strange. For example, the neutral element property of the empty word and the doubling function on integers expressed by using a digit and a graphical character as bound variables:

$$\forall\, 1 \in A^* \;.\; \varepsilon 1 = 1 = 1\varepsilon$$

$$\lambda\, | : \mathbb{Z} \;.\; |+|$$

The fact that bound variables are not mnemonic is not disturbing because they have a very local nature, and hence, are often undeclared. Their use is near to the place of their introduction, because the binder's scope is usually small (also in the case where the binder is implicit).

This is quite different to identifiers introduced by definitions. Their place of introduction can be at a completely different place than their use (say in a different chapter or even a different book!). Hence, they have a much more mnemonic task to fulfill than bound variables and also arise in greater syntactical diversity. There are constant symbols built by

- several letters: Monoid, Ring
- single letters: i, e, π, $\mathbb{N}$
- digits: 0, 123
- graphical characters: (), {}, [] (empty tuple, set and list)

and operators built by

- several letters: tanh, max, log
- graphical characters (most examples of operators in Section 3.2)
- mixing character classes:

_–Module	theory of R-modules over a given ring R
$\int \_ \,\mathsf{d}\, \_$	indefinite integration
$\_^{-1}$	inverse of an element in a group
$\sin^{-1}\_$	arc sine

Except for whitespaces, identifiers use characters from all 3 classes! Rare exceptions of identifiers using whitespaces are the trigonometric functions which are sometimes denoted by arc sin, arc tan, etc. But they all have well-known counterparts, namely $\sin^{-1}$ or simply arcsin.

Often, the same characters are also used in different contexts for different purposes and fixities, leading to possible context-dependencies. We give a few examples:

- Digits are used as digit sequences or in bound variables as suffixes.

- The symbol "+" is used as a separator within prefix, infix and postfix operators:

$+2$	unary plus
$n+m$	addition
$f+$	mapping the function f to a non-empty list [Bac89].

- The prime symbol $'$ is used for differentiating a function and appears within bound variables, too.

- The main purpose of parenthesis — "(" and ")" — is for grouping expressions. But they are also used for

$()$	0-tuple and its set
$(a_1, \ldots, a_n)$	the ideal in a ring generated by $a_1, \ldots, a_n$ or an n-tuple
$x \cong y\,(m)$	the integer x is equal to y modulo m
$(a_1 \cdots a_n)$	cycle representation of a permutation
$D(x)$	the field of rational functions over an integral domain D and the indeterminate x
$(D, \circ)$	denoting algebraic structures
(a, b)	open intervals (a and b elements of some totally ordered ring)
$(a, b\,]$ and $[a, b)$	half-open intervals

  The last example shows that parenthesis do not have to appear pairwise!

- Similarly for brackets "["and "]":

$[\,]$	the empty list
$[\,a_1, \ldots, a_n\,]$	the list of the elements $a_1, \ldots, a_n$
$[\,f(x) \mid \cdots]$	list comprehension [PJ87]
$M\,[\,N \,/\, x\,]$	substitution of free occurrences of the variable x in M with the term N
$R\,[\,x_1, \ldots, x_n\,]$	the ring of polynomials over the ring R and the indeterminates $x_1, \ldots, x_n$
$R\,[\,[\,x\,]\,]$	the formal power series over the ring R in x
$[\,a, b\,]$	closed intervals

- Braces are used primarily for describing sets:

$\{\}$	the empty set
$\{a_1, \ldots, a_n\}$	the set of the elements $a_1, \ldots, a_n$
$\{\,f(x) \mid \cdots\}$	set comprehension notation

  But not exclusively:

  $$|x| = \begin{cases} x & \text{if } x > 0 \\ -x & \text{otherwise} \end{cases}$$

  denotes the definition of the absolute value using a case distinction.

- Besides denoting set and list comprehensions, the bar symbol "|" is also used for:

$n \,|\, m$	n divides m
$|\, z \,|$	the radius of the complex number z
$f \,||\, g$	the lines f and g are parallel
$||\, v \,||$	norm of v

- The characters "<" and ">" are primarily used for denoting various relations. But they also find use in:

$< a_1, \ldots, a_n >$	finitely generated ideals
$< D, \circ >$	denoting algebraic structures
$< a_1, \ldots, a_n \mid$	row vector

- The comma character "," is used for constructing finite sequences resulting in tuples, lists, sets, structures, type contexts and so on. They are also used as a synonym for the logical $\wedge$-operation — sometimes in strange positions: $0 < a, b < 1$ means in fact $0 < a < 1 \wedge 0 < b < 1$. Commas are also used within Mc-Carthy's "if" operator [AIB+92], written "$p \to r, s$", and for the construction of intervals.

Given the above remarks and examples a general categorization of all words seems impossible, because within identifiers and separators various combinations of letters, digits and graphical characters appear. The only distinguishable syntactic category is that of bound variables.

It is interesting to note that none of the authors of the articles mentioned makes this important distinction. They only state that most variables use one or few letters.

4.2 Whitespaces and character-based parsing

As we saw in the preceding section, graphical characters are mainly used as separators in an operator's form. Hence, besides identifying the operator they separate the operator's arguments. This leads to words with very few whitespaces when applying such operators, especially if the arguments are bound variables. The remaining whitespaces are often not even mandatory but only used for better readability! A very compact style of writing results; we have to look "inside" words to understand their meaning. We call this decomposition of words into their building blocks — something which both humans and machines have to do — *character-based parsing*.

For example in the equation $-x/y + z! = x$ every character is either an identifier, probably a bound variable due to its shortness, or a one character separator (using a suitable context for numbers). Nevertheless, its meaning is clear without the use of any whitespace!

4.3 Juxtaposition

Mathematical operators use all the fixities presented in chapter 3. In fact, most examples shown there were from the mathematical world. One fixity almost exclusively used by mathematicians is juxtaposition, the writing of the arguments of an operator without separating them by separators.

The most frequent case of juxtaposition are invisible binary operators possessing type $A \times A \to A$ and being associative. Hence, no parenthesis need be written for consecutive applications. Therefore, juxtaposition

can be found in various semi-groups such as multiplication of numbers, concatenation of strings [Sch84, DSW94] or in rings where the multiplicative operation is written juxtaposed [GCL92, p. 26]. Composition of terminal and non-terminal symbols in context-sensitive grammars are written juxtaposed, too.

Some authors also use juxtaposition for non-associative operators of the above type [Lan84, p. 3]. Hence, the binary operation in an arbitrary groupoid can be written "multiplicative". An important instance is multiplication in R-algebras.

Juxtaposition is also used for binary operators not having type $A \times A \to A$. The scalar multiplication in a R-modules M, being of type $R \times M \to M$, is often written juxtaposed [Lan84]. Other examples are function application (already encountered in section 3.1) and powering of an element x in a monoid by a (superscripted) natural number n, written x^n. In the case of an abelian monoid, where the binary operation is written additively with a "+"-operator, this operation of taking n times an element x is written nx.

Further examples of juxtaposition are various indexing operations. The indexing of a vector v by a positive integer v_i to get the i-th element. In the case of matrices we get n-ary juxtaposed operations where $n \geq 3$ and usually $n \leq 4$. Given a two-dimensional matrix M, we denote by M_{ij} the element (i, j) of M and given a three-dimensional matrix M then M_{ijk} means the element (i, j, k).

Even composition of functions is sometimes written without the usual separator "$\circ$". This usage is seen in formalisms where composition is more important than application, such as in some versions of the Bird-Meertens formalism [Mee86] or in category theory [Cro93]. As Backhouse notes, writing function application *and* composition juxtaposed can quickly lead to ambiguities [Bac89].

In conjunction with single letter identifiers juxtaposition leads to an additional compactness of mathematical sentences. Polynomial domains are an example where 4 (!) different juxtaposed operators are used if they are viewed as the free R-algebra over the free abelian monoid of monomials [Lan84]. The result is a very compact notation for polynomials, which nevertheless remains readable. An example in $\mathbb{Z}[x, y]$:

$$(xy + 10x)(x - y^2)$$

The 4 juxtaposed binary operations are

xy	multiplication of monomials,
$10x$	scalar multiplication of 10 with the monomial x,
y^2	abbreviating operation for yy and
$(xy + 10x)(x - y^2)$	multiplication of two polynomials.

Note that the number 10 is written juxtaposed, too. Hence, we get in fact a fifth operation using juxtaposition in the case of integers!

As noted by van Gasteren, the use of juxtaposition in the case of integers and rationals is rather confusing [vG90]: compare $31x$, $3\frac{1}{2}$, $\frac{1}{2}x$ and 312 with $x \in \mathbb{Q}$. The first denotes $31x$, the second means $3+\frac{1}{2}$, the third $\frac{x}{2}$ and the last $3 \cdot 100 + 1 \cdot 10 + 2 \cdot 1$. Interestingly, although these conventions lead to various ambiguities, they do not cause problems for human readers.

A final example for juxtaposition is the already-mentioned cycle representation for permutations. Over $\mathbb{N}$, $(1\,2)(3\,5)$ denotes the permutation which swaps the numbers $1, 2$ and $3, 5$, respectively, and behaves as identity on the other values.

Whitespaces between the arguments of juxtaposed operators are sometimes completely omitted. This is often done in the case of operators denoting some form of multiplication, which, in general, need not to be commutative or associative. We call this form of juxtaposition *whitespace-less juxtaposition*. To stay readable, the omission is usually only done in conjunction with single letter identifiers and requires character-based parsing, too. Writing and omitting whitespaces can be used for expressing the precedence between two juxtaposed operators. For example, $\sin xy$ means $\sin(xy)$ and not $(\sin x)y$ because juxtaposed multiplication has a higher priority than application. The latter is written with whitespaces between the two arguments, the former without.

4.4 Finite sequences

Finite sequences are an often-used tool in mathematical formulas. They are needed for denoting values that are expressed by a varying number of elements, where the elements are all members of a fixed set. Examples are vector, list and set constants, where the elements are enumerated individually and separated by commas:

$(1, 2, 3, 4)$
$[\,\text{“}a\text{”}, \text{“}b\text{”}, \text{“}c\text{”}, \text{“}d\text{”}\,]$
$\{\, x, x^2, x^3, x^4 \,\}$

Ellipses are used within sequences as an abbreviating tool:

$(1, \ldots, 9)$
$[\,\text{“}a\text{”}, \ldots, \text{“}z\text{”}\,]$
$\{\, x, x^2, \ldots, x^{10} \,\}$

But there are also examples of sequences that are used for non-collectional values such as permutations, relational expressions or polynomial ring constructors (cf Section 5.3).

4.5 Binding constructs

Mathematical notation uses various binding constructs with explicit bound variables. Among others, there are Σ and Π for denoting sums and products of numbers, $\forall$ and $\exists$ as quantifiers used in logic, and λ and Λ as variable-binding constructs used in various λ-calculi.

Interestingly, some of the binding constructs are used differently according to the actual context, i.e. they are overloaded themselves. Σ and Π may instead denote sums and products of sets and types, respectively. Or they can denote coproducts and products in category theory. $\forall$ and $\exists$ are also used in conjunction with polymorphic and existential types.

Note that indefinite integration is usually *not* a binding construct, because $\int x\,\mathsf{d}x$ and $\int y\,\mathsf{d}y$ are not equal if treated as polynomials in $\mathbb{Z}[x]$ and $\mathbb{Z}[y]$, respectively, where x and y denote symbols (indeterminates). Therefore, α-conversion is not valid, which is a property of binders. The same applies to differentiation, too: $\mathsf{d}x^2/\mathsf{d}x \neq \mathsf{d}y^2/\mathsf{d}y$. Using mixfix declarations, the two operators can be expressed as

$$\begin{array}{lcl} \int \_\,\mathsf{d}\_ & : & \mathsf{Expr} \times \mathsf{Symbol} \to \mathsf{Expr} \\ \mathsf{d}\_/\mathsf{d}\_ & : & \mathsf{Expr} \times \mathsf{Symbol} \to \mathsf{Expr} \end{array}$$

where Expr is a suitable type of expressions, containing symbols.

Note that *definite* integration is invariant with respect to α-conversion, because the expressions to be integrated are used as functions. Hence, it can be expressed as a binder.

4.6 Non-linearity

One distinguishing feature of mathematical notation is its use of two-dimensional operators. Hence, mathematical statements are non-linear and cannot be read from left to right in a linear fashion.

There are examples where the argument(s) are below the operator

$\sqrt{x-1}$	square root
$\dot{s}$	derivative of the function s with respect to time
$\overrightarrow{AB}$	vector from point A to B (note the juxtaposition!)
$\overline{AB}$	line segment between points A and B
$\overline{z}$	conjugation of the complex number z

examples where they are above and below

$\frac{x}{y}$	fraction with numerator x and denominator y
$\frac{A_1 \ \ldots \ A_n}{B}$	inference rule with premises A_i and conclusion B

and mixed forms

$\int_1^2 x \, \mathsf{d}x$	definite integration
$\sum_{i=1}^{n} i$	summation
$\begin{pmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{pmatrix}$	matrix
$\binom{x}{y}$	binomial coefficients
$\left(\frac{n}{m}\right)$	Jacobi symbol

Note the use of parenthesis for various further operators and the ambiguity between the Jacobi symbol and a parenthesized fraction in the last example!

Powering, vector and matrix access encountered in the preceding section on juxtaposition could be treated as two-dimensional operators as well. A different view is treating them as linear notation in conjunction with a smaller font. The latter view leads to problems if the operators have to be composed.

4.7 Types in mathematics

Today's mathematics is usually done within the context of set theory where all arising object such as natural numbers, relations, functions and so on are coded as sets. This suggests that there are no types available. But in everyday mathematics there seems to be some notion of type, at least implicitly. We do not, for example, treat natural numbers and functions as sets, but abstract from such "implementation details". We also take care to apply functions only to meaningful arguments by respecting their sources and targets. Often, the terms denoting sets are implicitly used as types, restricting the possibility of "applying everything to everything" (but set theory admits very general constructions of sets so that we quickly lose decidability and meaningful type checking). With the words of Cardelli and Wegner [CW85]:

> As soon as we start working in an untyped universe, we begin to organize it in different ways for different purposes. Types arise informally in any domain to categorize objects according to their usage and behavior. The classification of objects in terms of the purposes for which they are used eventually results in a more or less well-defined type system. Types arise naturally, even starting from untyped universes.

Really untyped are those mathematical subjects which deal *explicitly* with untyped objects such as axiomatic set theory [TZ82], the untyped λ-calculus or combinatory logic [Bar89, EAG+94, Hin86]. There, everything can be applied to everything.

Also implicit polymorphism arises naturally in mathematics. If some of the variables used in source and target of a function declaration do not

denote some specific sets, but are implicitly bound variables, we can use any function whose type matches source and target. This corresponds to implicit polymorphism. For example in the definition of function composition. Given functions $f : A \to B$ and $g : B \to C$ then $g \circ f$ denotes the function $x \mapsto g(f(x))$. Hence, composition can be applied to *every* pair of functions that matches the prerequisites. Explicit polymorphism can be partly simulated by means of cartesian products.

Mathematics is higher-order with respect to tuples and to functions, too. There are a lot of operations on functions, such as differentiation or integration. Fourier and Laplace transformations or arbitrary operators applied to elements of function spaces in functional analysis are higher-order, too. Similarly for tuples, which are treated as first-class values for example in vector analysis and differential geometry.

4.8 Overloading

Overloading of constant symbols and operators is an extensively-used tool in mathematical notation. Overloading is used for

- unrelated values,
- restricting or extending given operations, and inducing new ones while retaining the original syntax,
- related values. They are often components in algebras that are in the same class of algebraic structures.

Examples of unrelated values:

- The postfix operator "$\_'$" is used for the derivative of a function and for denoting the successor of a natural number.
- The operator "$\_^*$" is used for A^* set of words over the alphabet A and for R^*, the reflexive transitive closure of the relation R.
- The operator "$\_/\_$" is used for the usual division of numbers, for "G/H", the cosets of a group G modulo a subgroup H [Lan84], for "$R/\sim$", the residue classes modulo the equivalence relation $\sim$, and as the fold functional on lists in the Bird-Meertens formalism [Bac88].

- $\_ \to \_$ is used for denoting the function type constructor, logical implication, rewrite rules, and the class of functors and natural transformations in category theory.
- Besides denoting the absolute value of a number, the operator $|\_|$ is used for the cardinality of a set, the length of a vector in a normed space, and the number of elements in a list.
- The operator $\_ \models \_$ is used in logic to denote that the structure M is a model for the theory T (written $M \models T$) but also that the sentence E is a logical consequence of the sentences in S ($S \models E$).

An example of extending operations is the quotient-field construction over an integral domain. All its operations are extended to operations on fractions while retaining their conventional syntax.

If the identifier 1 is used as the neutral element with respect to the binary associative operation while defining the class of monoids, most algebras being monoids will also use 1 as their neutral element. Analogously for the binary operation, usually denoted multiplicatively by "$\_*\_$". Hence, we get a whole bunch of overloaded identifiers 1 and operators "$\_*\_$". Examples are the natural number with respect to multiplication, the identity function with respect to composition, and the integer function that is constantly 1 with respect to pointwise multiplication of integer functions.

This overloading can even happen in the same algebraic structure: 0 in an R-module can denote the scalar 0 of the ring R or the zero of the underlying abelian group.

Overloading is a special case of reusing characters in different contexts as encountered in Section 4.1. Hence, it also leads — in any case — to context-dependence!

4.9 Coercions

As Weber notes in [Web93], mathematical objects are frequently identified with their images under embeddings or arbitrary conversion functions, and this practice is, in fact, one of the strengths of mathematical notation.

Often, subset relations actually indicate the presence of embeddings. For example in the case of the integers and rationals, where the relation $\mathbb{Z} \subset$

$\mathbb{Q}$ does not denote a subset relation but the presence of an embedding, namely $n \mapsto n/1$. $\mathbb{Q}$ is usually constructed as $\mathbb{Z}\times\mathbb{Z}/\equiv$, where $\equiv$ denotes a suitable equivalence relation. Hence, $\mathbb{Z}$ cannot be a subset in the set-theoretical sense, except if it is identified with the set of its images in $\mathbb{Q}$. Sometimes the symbol "$\hookrightarrow$" is used between two sets for making this fact explicit.

Another example is that of disjoint unions. Given arbitrary sets A and B, then the two injections $in_1 : A \to A+B$ and $in_2 : B \to A+B$ of the disjoint union $A+B$ are often implicitly applied. This is especially practiced in denotational semantics, where otherwise the formulas would be full of injections.

A source of non-injective transformations arise from various forgetful functors between classes of algebraic structures. For example, every group can be treated as a monoid and every Lie group is also a topological space. Also the often-done identification of a one-sorted algebraic structure with its carrier set is due to the use of an implicit forgetful functor that extracts the underlying carrier.

All these conventions can be modeled by coercions. Without them, mathematical formulas would be cluttered up with explicit conversions and eventually become unreadable.

4.10 Requirements for a suitable meta-notation

This section discusses the features needed for a meta-notation that underlies ordinary mathematical notation. Such a fundamental formalism should be capable of describing all the facets of mathematical notation mentioned in the preceding and this chapter, namely:

- *Rich base alphabet*

  The base alphabet of digits, letters and graphical characters contains the usual mathematical characters (Section 4.1).

- *Arbitrary linear fixities*

  All possible linear fixities, as presented in 3.2, are supported.

- *No compulsory whitespaces*

In general, whitespaces are nowhere compulsory, including in the case of whitespace-less juxtaposition (4.2 and 4.3).

- *Overloading*

  Arbitrary overloading of identifiers is allowed, according to Section 4.8.

- *Coercions*

  Coercions are fully supported (4.9).

- *Support of types*

  Types are one of the essential components for disambiguating mathematical expressions during the parsing process (3.3.2). The underlying type system should be higher-order and support at least implicit polymorphism (4.7). Note that higher-orderness requires a method to access the functions underlying the various operators.

- *Suitable operator attributes*

  The available operator attributes should be powerful enough to describe and prevent ambiguities occurring in mathematical notation. They have to be context-dependent. Precedences should be given *relatively* because everyday mathematics does not know absolute priorities but utilizes a partial order on the operators.

- *Finite sequences*

  Constructs with finite sequences are supported (4.4).

- *Binding operators*

  The definition of explicit and implicit binding operators is possible. Their syntactical form is freely definable (4.5). The types of the bound variables can be left out if they can be derived with the help of type inference. This is a usual practice in mathematical notation. Implicit bound variables need a special syntactical shape as discussed in Section 4.1 because one has to recognize them in an expression unambiguously.

- *Two-dimensionality*

  Non-linear fixities are supported in a structured way.

- *Modularity and incremental extensions*

  Most mathematical texts have a section or an appendix on the notation that is used for the whole text. Hence, mathematical notation is very modular because it depends on the text currently in use. Furthermore, new operators are often added "on-the-fly" to the current context and are valid for a certain scope such as a chapter or a section. This leads to incremental extensions of the notation. Thus, the meta-notation supports the modularization of mathematical notation into usually rather small units, which in addition are extendable by new declarations.

4.11 Programming language support

We survey some programming languages for their ability to support mathematical notation as described in the preceding sections.

We stick to MODULA-2 [Wir88b] and C++ [Str91] for members of the huge family of imperative languages.[1] None of them admits the definition of new operator syntax. C++ allows a restricted form of identifier overloading but the set of non-prefix identifiers is fixed and their priorities and associativities cannot be influenced. Hence, ordinary mathematical notation cannot be expressed adequately in these languages.

Examples of functional languages are HASKELL [H+92], which can be seen as a modern successor of MIRANDA [Tur90], OPAL [Exn94], SAMPLE [GKT90], SCHEME [AIB+92], and STANDARD ML [HMT90, HMT]. HASKELL, MIRANDA, and SAMPLE use a non-strict semantics for function application, whereas the other mentioned languages are strict. SCHEME— being a Lisp dialect — is untyped, while the others are all (strongly) typed.

In SCHEME, all expressions denoting function applications are built by using lists with the operator written in a prefix style. This leads to a consistent notation without any ambiguities. On the other hand, SCHEME's notation is very different from usual mathematical notation.

STANDARD ML (SML) provides an implicit polymorphic type system including type inference, pattern-matching for function definitions, and a powerful module system based on signatures, structures, and functors (parameterized structures). The default fixity of operator application

[1] We do not distinguish between the procedural and the object-oriented paradigm.

is prefix, but infix can be declared including absolute priorities. One *has* to choose in SML between left or right associativity for an infix operator. Overloading of identifiers is not supported, except for some built-in operators.

OPAL is similar to SML but with an algebraic-specification flavor. It does not provide polymorphism (but its parameterized structures can often replace it). Arbitrary overloading of identifiers is supported. The same identifier can be used *simultaneously* in prefix, infix, and postfix notation. Implicit right associativity is universally assumed. There are no user-definable priorities or associativities for operators.

HASKELL provides pattern-matching, type inference, and polymorphism. Its type system is an extension of SML's, where polymorphic variables need not to be universal but can be constraint by type classes. Its syntactic capabilities are similar to SML: The standard fixity is prefix, but infix is possible, including absolute priorities. As opposed to SML, non-associativity is permitted. Overloading is controlled by type classes, hence arbitrary overloading is not possible. In addition, HASKELL provides sections, which are partial applications of binary operators.

SAMPLE provides prefix and infix as well as postfix operators. There are a few fixed classes of precedences for infix operators. SAMPLE's scanner takes the longest possible lexem[2]. For example, the input $x+y+z$ is in any case treated as a single identifier, independently of the context! This makes a lot of whitespaces unavoidable.

The higher-order algebraic-specification environments EXTENDED ML and SPECTRUM provide similar syntactical facilities as functional languages. Prefix is default fixity and infix can be declared. Interestingly, Spectrum uses the operator "$\_\times\_$" for denoting the cartesian product type, as opposed to "$\_*\_$" ordinarily used in programming languages.

A general conclusion can be drawn that the support for mathematical notation is rather poor in the above languages, although their paradigm is near to mathematical thinking.

Typical members of the family of computer-algebra systems are AXIOM [JS92], MAPLE [CGG+91] and MATHEMATICA [Wol91]. AXIOM possesses a typed language, while MAPLE and MATHEMATICA are both untyped. These computer-algebra systems are mainly used to implement algorithms of applied and constructive mathematics.

[2] Lexems are those substrings of the input string which are converted by the scanner into tokens.

AXIOM's higher-order type system is based on the notion of domains — types with associated operations — and categories which are the type of domains. This is similar to HASKELL's types and type classes as pointed out by Weber [Web93]. Representation-independent code can be written in categories and used by their instances. Default fixity for operators is prefix. A predefined set of infix operators can be overloaded, but their priorities and associativities cannot be influenced and are fixed (similarly to C++). AXIOM provides two different application operators, namely "_ _" vs. "_._" if expressed in mixfix syntax, which are semantically identical. The sole difference lies in the operators' associativities: The former is right associative, the latter left associative. If AXIOM's parser used the context, one of the operators would be superfluous, as discussed in Section 3.3.2.

MAPLE is based on a rather conventional Pascal-like programming language. It is not higher-order.[3] A predefined set of operators can be given their own meaning.

MATHEMATICA provides a uniform user-level language based on pattern-matching and rewrite rules. Higher-order functions can be defined. The input language is very general and user-definable [Soi95, Wol96]. MATHEMATICA permits juxtaposition but whitespaces between arguments are compulsory.

Due to the missing type system both languages cannot handle arbitrary overloading and some operators such as "_+_" or "_*_" have a built-in meaning induced by the properties of commutative algebra.

Similarly to the case of functional languages, mathematical notation is not fully supported in these systems.

All the shortcomings of systems providing mixfix syntax, as discussed in Section 3.5, also count as disadvantages for representing mathematical notation.

Some general shortcomings arising in all the systems mentioned:

- No support for whitespace-less juxtaposition.
- No arbitrary sequence and binding operators.
- Syntax of type constructors is not user-definable.

[3] The input "`(x->(y->x+y))(1)(2)`" does not evaluate to 3 as β-reduction would require.

Chapter 5

Higher-order Mixfix Syntax

Assuming a rich enough base alphabet of letters, digits and graphical characters, mixfix syntax already supports some specialties of ordinary mathematical notation, such as weird one-dimensional (linear) operator forms, juxtaposition, overloading and coercions. But to fulfill all the requirements for a meta-notation of mathematical notation we have to extend mixfix syntax by suitable features which are unknown in current mixfix formalisms. We call the resulting notation *higher-order mixfix syntax* and present details in the following sections.

We do not treat two-dimensional fixities but only the linear part of mathematical notation. This restriction seems to be more severe at first sight than it really is. Two-dimensional mathematical text has still a straightforward — but not always very readable — linear representation (for example by using TEX). In fact, the input of two-dimensional mathematical notation in computer-based systems is usually still done character by character, leading to a linear encoding. Immediately after each keystroke or at the end of input the linearized form is transformed into a two-dimensional representation [Soi95]. Hence, an x-y-pointing device such as a mouse is not required to treat two-dimensional mathematical notation, as opposed to a drawing program where a pointing device is an essential tool.

Apart from two-dimensionality, higher-order mixfix syntax fulfills all the

requirements for a meta-notation (Section 4.10).

5.1 Operator attributes

In this section, we introduce suitable operator attributes, namely relative priorities, implicit associativities and — as a novelty — *positional exclusion*, by giving a definition for the non-terminal *attrib* of Section 3.1. *attrib* is either empty or a comma-separated list of attributes (the bracket notation is influenced by OBJ3)

$$[\,attr_1, \ldots, attr_n\,]$$

where $attr_i$ is either *prio*, *assoc* or *excl*. These are defined in the next sections.

5.1.1 Relative priorities

As opposed to current mixfix formalisms, which — if at all — use absolute natural numbers for stating precedences, relative priorities are more adequate in the context of mathematical notation, as already remarked.

Furthermore, absolute priorities induce a total order, prohibiting the possibility that the precedence between two operators stays unspecified. This is done in cases where a new operator that should be independent of the current priority order is introduced, or where one wants to force the user to write parenthesis around arguments to make the association clear. For example, by leaving the priority between application and juxtaposed multiplication undefined, we force the user to write parenthesis in the expression "$\sin 2x$", which otherwise could be either interpreted as "$(\sin 2)x$" or as "$\sin(2x)$".

Hence, we must be able to state whether the currently declared operator has lower, equal or higher priority with respect to an already declared operator *id* (including its type T). Thus, *prio* takes the form

$$relop \;\; id{:}T$$

where $relop \in \{<, =, >\}$. Taking the above juxtaposition example,

$$\_\,\_ \;:\; (\alpha \to \beta) \times \alpha \to \beta \qquad\qquad [<\; \_\,\_ : \mathbb{Z} \times \mathbb{Z} \to \mathbb{Z}]$$

states a lower priority for application with respect to juxtaposed multiplication.

5.1.2 Implicit associativities

Implicit associativities are defined as usual: we can state optionally that the currently declared operator is implicit left or right associative, describing the association in interaction with an operator of the same priority. Therefore, $assoc \in \{\mathsf{left}, \mathsf{right}\}$.

For example, while declaring juxtaposed multiplication, we declare its left associativity too:

$$\_\,\_ \;:\; \mathbb{Z} \times \mathbb{Z} \to \mathbb{Z} \qquad [\,\mathsf{left}\,]$$

5.1.3 Positional exclusion

We introduce a new operator attribute for handling ambiguities not handled by the above priority and associativity schemes, as encountered in Section 3.3.6. The general idea is to forbid certain possible readings of a compounded application. This is done by *excluding* the building of applications formed by certain operators to appear at some argument position of a mixfix operator. Formally, *excl* has the form

$$\mathsf{no}\ id{:}T\ \mathsf{at}\ n$$

where n is the argument position (limited by the operator's arity). This means that at argument position n of the currently declared operator, an application built by the mixfix operator id does not occur.

For example, we want to state that juxtaposed multiplication may *not* have an application at the second argument with a unary minus as operator.

$$\_\,\_ \;:\; \mathbb{Z} \times \mathbb{Z} \to \mathbb{Z} \qquad [\,\mathsf{no}\ -\_ : \mathbb{Z} \to \mathbb{Z}\ \mathsf{at}\ 2\,]$$

As a consequence, the expression "$x - y$" cannot be read as "$x\,(-y)$", because the identifier "$-\_$" appears as operator at the second argument position of the expression. Hence, any ambiguity with the binary minus operator is avoided.

Note that the positional-exclusion attribute is strictly more powerful than the other two attributes. It can be used to model priorities and associativities but not vice versa.

5.2 Coherent use of declarations

(Normal) declarations arise in various places:

- top-level declarations for declaring global identifiers,
- in signature components,
- in binding constructs, where a new variable is introduced.

A coherent use of declarations means that a mixfix declared operator can appear everywhere where a normal declaration can occur. This leads to a uniform declaration syntax. Consequently, there are also *local* mixfix declarations, say, in the following anonymous function, whose first argument is declared as an infix operator:

$$(\_*\_ : \mathbb{Z} \times \mathbb{Z} \to \mathbb{Z}) \quad \mapsto \quad (x : \mathbb{Z} \mapsto x * x)$$

In the body of the binder, the operator $\_*\_$ can be used as a globally declared mixfix operator.

We may also pass polymorphic mixfix operators:

$$(\#\_ : \forall \alpha \,.\, \alpha^* \to \mathbb{Z}) \quad \mapsto \quad \#[\,1, 2, 3\,] \;+\; \#[\,\mathsf{true}\,] \;+\; \#[\,\text{“}abc\text{”}\,]$$

The last example cannot be typed in SML-like languages because functions with polymorphic arguments cannot be defined. Arguments have to be monomorphic because their type systems disallow the use of $\forall$-quantifiers in arbitrary type positions.

Note that we treated free-occurring type variables in a type expression as implicitly $\forall$-quantified. Hence the function

$$x : \alpha \mapsto x$$

takes a *polymorphic* constant as argument, because x possesses the polymorphic type $\forall\alpha\,.\,\alpha$; it does not denote the polymorphic identity function of type $\forall\alpha\,.\,\alpha \to \alpha$ but has the type $(\forall\alpha\,.\,\alpha) \to (\forall\alpha\,.\,\alpha)$.

5.3 Sequence operators

We want to declare operators that work on an arbitrary (finite) number of arguments. Using mixfix declarations this can easily be done if a list type is available or definable:

$$\begin{array}{rcl} \mathsf{max}\_ & : & \mathbb{Z}^* \to \mathbb{Z} \\ |\_| & : & \alpha^* \to \mathbb{Z} \end{array}$$

As mentioned in the preceding chapter, there are cases where the application of such an operator is written in an infix style using (optional) separators *between* the arguments. Hence, the above declarations cannot be used for this purpose. An example is tuple type formation, denoted by $T_1 \times \ldots \times T_n$, where the separator "$\times$" is written between the arguments. We cannot replace this by an iteratively applied binary tuple type constructor if we treat cartesian products as non-associative.

We extend the mixfix declaration mechanism by the possibility of declaring *sequence operators*. A general sequence operator on an arbitrary number of elements of type A is declared as

$$l\_s \cdots s\_r \quad : \quad A^* \to B \qquad \mathit{attrib}$$

where l, s and r are optional separators, i.e. $l, s, r \in S^* \cup \{\varepsilon\}$. The separator symbol s between the arguments is the same throughout and can be empty. The difference to ordinary mixfix declarations is the use of three centralized dots, indicating a sequence of arguments when applying the operators. The operator's form is still $l\_s \cdots s\_r$.

Given a string of the form $le_1 s\, e_2\, se_n r$, where the e_i all have type A, how large must n be, i.e. how many arguments must the sequence possess, such that the string matches the sequence operator? It cannot be 0 or 1 because in these cases there are no separators visible at all. Hence, n must be at least 2. But there is the following advantage if we require n to be at least 3. Given a binary infix operator

$$\_s\_ \;:\; A \times A \to A \qquad [\,\mathsf{no}\ \_s\_ : A \times A \to A\ \mathsf{at}\ 1,\ \mathsf{no}\ \_s\_ : A \times A \to A\ \mathsf{at}\ 2\,]$$

where the positional exclusion attributes disallow the use of the operator in a sequenced form, we do not get an ambiguity with the sequence operator

$$\_s \cdots s\_ \;:\; A^* \to A$$

because, for the latter, at least 3 arguments are required. Therefore, we can define the sequence operator's meaning by the binary one. This is usually done by some folding operation on the sequence operator's list argument. Note the similarity to MATHEMATICA's `Flat` attribute for associative operators [Wol91, p. 272].

An example of a sequence operator where the separator s is empty arises in the cycle representation of permutations on natural numbers. They are written as sequences of numbers enclosed by parenthesis which leads to the following declaration:

$$(\_ \cdots \_) \;:\; \mathbb{N}^* \to (\mathbb{N} \to \mathbb{N})$$

For example, we can now write (1 2 3 4). The meaning is the application of the operator $(\_ \cdots \_)$ to the list $[1, 2, 3, 4]$.[1]

Further examples where a sequence operator is mandatory are conjunctionally written binary relators (operators denoting a relation). Given a binary infix relator $\_R\_ : \alpha \times \alpha \to \mathbb{B}$ such as $\_ = \_$, $\_ < \_$ or $\_|\_$ (divisibility) then mathematicians often write

$$a_1 R a_2 R \ldots R a_n$$

[1] Note that permutations are an example where function composition is written juxtaposed: (1 2 3)(3 4 5).

for denoting

$$a_1 R a_2 \wedge a_2 R a_3 \wedge \ldots \wedge a_{n-1} R a_n$$

But this meaning cannot be simulated because terms of the form $a_1 R a_2 R \ldots R a_n$ are not well-typed if R is a binary operator! Using sequence operators we would declare the relator as

$$\_R \cdots R\_ \quad : \quad \alpha^* \to \mathbb{B}$$

and the above terms become legal.

In fact, Gries and Schneider introduce *two* equality operators on booleans, a conjunctional one, using a "=" symbol, and an associative one, using "≡" [GS94]. Using mixfix declarations this can be stated as:

$$\begin{array}{rcl} \_=\cdots=\_ & : & \mathbb{B}^* \to \mathbb{B} \\ \_\equiv\_ & : & \mathbb{B} \times \mathbb{B} \to \mathbb{B} \end{array}$$

Note that this sequence mechanism cannot handle sequences with *different* separators between the arguments. For example $a_1 < \ldots < a_n \leq b_1 \leq \ldots \leq b_m$ cannot be modelled.

As a last example, we declare the type constructor of polynomials over a ring and a finite set of indeterminates (symbols) as

$$\_[\_,\cdots,\_] \quad : \quad \mathsf{Ring} \times \mathsf{Symbol}^* \to \mathsf{Ring}$$

5.4 Binding operators

As noted in Section 3.5, arbitrary binding operators are not supported in current mixfix formalisms.

Suppose we want to define the summation operator $\sum_{i=n}^{m} E$ over arbitrary integers n and m. This is clearly a binding operator because it introduces a new variable i whose scope is restricted to the body E of the summation. In a higher-order system this could be reflected with the following mixfix declaration:

$$\Sigma\,\_,\_,\_ \quad : \quad \mathbb{Z}\times\mathbb{Z}\times(\mathbb{Z}\to\mathbb{Z})\to\mathbb{Z}$$

A summation $\sum_{i=1}^{9} i^2$ would then be written as "$\Sigma\,1, 9, \lambda i.i^2$", which is the application of the operator $\Sigma\,\_,\_,\_$ to the argument $\left(1,\ 9,\ \lambda i.i^2\right)$. Therefore, Σ is no longer a binding construct but a simple higher-order operator. Other binding operators can be coded analogously. Mathematicians who are used to explicit binding constructs may regard this encoding as a disadvantage.

How can we retain the usual syntax for binding operators and combine it with the semantical elegance of higher-order systems? The idea is that the λ-abstraction $\lambda x\!:\!T.E$, which forms the body of a binding construct in the higher-order declaration, can be *split* into

- its locally declared bound variable x and
- its body E.

The two parts can be accessed independently in a mixfix declaration by two new and different placeholders:

$$\lambda \underbrace{x : T}_{\_\cdot} \cdot \underbrace{E}_{\cdot\_}$$

The *left placeholder* "$\_\cdot$" stands for the declaration of the bound variable ("the left-hand side of the abstraction") and the *right placeholder* "$\cdot\_$" for the body of the abstraction ("the right-hand side"). The declaration of the bound variable can be simplified by omitting its type if this can be inferred from the body.

Binding operators can now use these two special placeholders and move the place of the introduction of the bound variable to a completely different place. Thus, the placeholders are in fact two "semi" placeholders. "$\_\cdot$" has no semantical meaning, hence there is *no* corresponding type at this position in the source of the operator! A function type is attached to "$\cdot\_$", depending on the type of the bound variable.

The above summation operator can now be declared as

$$\Sigma\ \_\cdot = \_,\_,\cdot\_ \quad : \quad \mathbb{Z}\times\mathbb{Z}\times(\mathbb{Z}\to\mathbb{Z})\to\mathbb{Z}$$

There is no semantical difference to the higher-order formalism; this is shown by the fact that the type of the operator is the same as above. An application of the form $\Sigma\, i = 1, 9, i^2$ is in fact the application of the operator $\Sigma\_. = \_, \_, .\_$ to the argument triple $\left(1,\, 9,\, \lambda i.i^2\right)$. This splitting of the abstraction into an explicit binding variable part and a body is pure syntactic sugar but allows us to write binding operators more "mathematically".

The scope of binding operators can be controlled by the ordinary operator attributes. By setting priorities we can say that $\Sigma\, i = 1, n, i + a$ should mean $\Sigma\, i = 1, n, \left(i + a\right)$ and not $\left(\Sigma\, i = 1, n, i\right) + a$. This in contrast to Révész' opinion that left- or right-open binding operators are inherently ambiguous and need an explicitly specified scope [RL91].

The mathematical binding operator for constructing anonymous functions is now syntactical sugar of the λ-abstraction, being declared as a binding operator (and defined as identity):

$$\_. \mapsto .\_ \quad : \quad (\alpha \to \beta) \to (\alpha \to \beta)$$

The let-construct found in functional programming languages could be declared as:

$$\mathsf{let}\_.{=}\_\mathsf{in}.\_ \quad : \quad \alpha \times (\alpha \to \beta) \to \beta$$

To be meaningful we have to restrict the use of these special placeholders within binding operators:

- There is *at most* one left placeholder, and
- *at least* one right placeholder.
- The type of the bound variables is compatible with all the right placeholders' sources.

A binding operator with more than one right placeholder allows us to use the introduced bound variable in more than one argument expression.

But what does a binding operator without a left placeholder mean? It is an *implicit* binder because there are no bound variables introduced explicitly. Its behavior is to bind *all* free-occurring variables in a term.

Mixfix operators consisting of a sole right placeholder are a special case. Such operators are a special form of coercion operators. For example, if Prop denotes the type of propositions [H+95, Mis94, Pau94] then

$$\_. \;:\; (\alpha \to \mathsf{Prop}) \to \mathsf{Prop}$$

can be used for denoting the invisible ∀-quantifier surrounding a top-level expression of type Prop that contains free variables. Hence, such special coercions are used exclusively at the outermost level of an expression.

5.5 Describing types by mixfix syntax

We want to derive a suitable type context that induces the language of those types which we need in mixfix declarations. But how can we declare the necessary type constructors of a higher-order polymorphic type system with mixfix declarations? The idea is to use a type of types, denoted by Type [Cro93].[2] We can now describe them by mixfix declarations with the following pseudo context:

Type	:	Type	
$\_\to\_$	:	$\mathsf{Type}\times\mathsf{Type}\to\mathsf{Type}$	[right]
$\_\times\cdots\times\_$	:	$\mathsf{Type}^*\to\mathsf{Type}$	[> $\_\to\_$: $\mathsf{Type}\times\mathsf{Type}\to\mathsf{Type}$]
$\_^*$	:	$\mathsf{Type}\to\mathsf{Type}$	[> $\_\times\cdots\times\_$: $\mathsf{Type}^*\to\mathsf{Type}$]
$\forall\_..\_$	:	$(\mathsf{Type}\to\mathsf{Type})\to\mathsf{Type}$	[< $\_\to\_$: $\mathsf{Type}\times\mathsf{Type}\to\mathsf{Type}$]
$\_.$	:	$(\mathsf{Type}\to\mathsf{Type})\to\mathsf{Type}$	

Note the last declaration, which is an example of a special binding coercion operator introduced in the preceding section. Thus, we can write type expressions without explicit ∀-quantifiers on the outermost level.

Due to its recursive structure, this context cannot be stated by ordinary mixfix declarations. It has to be predefined by the underlying type system. We can, however, access the built-in constructors by ordinary (mixfix) identifiers and make use of their attributes. Note the need for some kind of bracketing operators for the above left- and right-open

[2] Instead of the identifier Type, one also sees the symbol "∗" used as the type or kind of types [BH90, Sch94].

operators. As before, we use the polymorphic parenthesis operator for this purpose.

It is interesting to note that the language of types requires all the extra features of higher-order mixfix syntax to be described adequately.

To prevent logical anomalies by the above $\mathsf{Type} : \mathsf{Type}$ declaration — known as Girard's paradox [Coq86, How87, MR86] — Type is only a *syntactical* type of all types, similarly as in ISABELLE [Pau94, p. 133] or in [HP89]. It is the task of the underlying system to prevent or allow these effects [Car86].

If the type system allows new type constructors to be introduced, say by a datatype facility as in SML or by an explicit term algebra constructor [Mis95], we can also declare our own type constructors. For example, the sum of two types and its two injections, which can be expressed as the following term algebra induced by a suitable signature:

$$
\begin{array}{lll}
\mathsf{free} \ll & & \\
\quad \_+\_ & : \ \mathsf{Type}\times\mathsf{Type}\to\mathsf{Type} & [\ \mathsf{left}, \\
 & & \quad > \ \_\to\_ : \mathsf{Type}\times\mathsf{Type}\to\mathsf{Type}, \\
 & & \quad < \ \_\times\cdots\times\_ : \mathsf{Type}^*\to\mathsf{Type}\ ], \\
\quad \mathsf{inj}_1\_ & : \ \forall\,\alpha\,.\,\forall\,\beta\,.\ \alpha\to\alpha+\beta\,, & \\
\quad \mathsf{inj}_2\_ & : \ \forall\,\alpha\,.\,\forall\,\beta\,.\ \beta\to\alpha+\beta & \\
\gg & &
\end{array}
$$

The sum type constructor has a priority between function space and tuple type constructors, and is left associative.

5.6 Interpreting higher-order mixfix syntax in the λ-calculus λ2

In the next section, we introduce a second-order polymorphic λ-calculus in Curry-style suitable for interpreting higher-order mixfix syntax. The interpretation is given in the subsequent section by a relation defined inductively by inference rules. We will use this interpretation for stating soundness of our parsing algorithm (Section 6.3.3).

5.6.1 The definition of $\lambda 2$

$\lambda 2$ is a second-order polymorphic λ-calculus with explicit type variables in the types but not in the terms (thus a *Curry-style* $\lambda 2$-calculus [BH90]). We extend it with products, lists and some base values. Note that these extensions do not give a more powerful λ-calculus [GLT89] but are only syntactic sugar.

First, we introduce its types and terms, and then list its type system. We assume a set *Ident* of identifiers which contains at least all valid identifiers occurring in higher-order mixfix syntax and a set *TConst* of type constants. Types are then defined according to the following context-free grammar:

$$
\begin{array}{lcll}
\mathit{Type} & \Longrightarrow & \mathit{Ident} & \\
 & \mid & \mathit{TConst} & \\
 & \mid & \mathit{Type} \to \mathit{Type} & \\
 & \mid & \mathit{Type}_1 \times \cdots \times \mathit{Type}_n & (n \geq 2) \\
 & \mid & \mathit{Type}^* & \\
 & \mid & \forall\, \mathit{Ident}\ .\ \mathit{Type} &
\end{array}
$$

We have identifiers, type constants, function spaces, products, lists, and quantification over type variables. They correspond to the type constructors declared in the preceding chapter, and we take over their precedences and associativities. *TConst* contains at least Type, $\mathbb{N}$, and the unit type $()$. Equality on types is term equality modulo α-congruence, and hence decidable.

The free variables $\mathsf{FV}(T)$ of a type T are defined as usual:

$$
\begin{array}{lcll}
\mathsf{FV}\,(\alpha) & = & \{\,\alpha\,\} & \text{if } \alpha \in \mathit{Ident} \\
\mathsf{FV}\,(C) & = & \{\,C\,\} & \text{if } C \in \mathit{TConst} \\
\mathsf{FV}\,(T_1 \to T_2) & = & \mathsf{FV}(T_1) \ \cup\ \mathsf{FV}(T_2) & \\
\mathsf{FV}\,(T_1 \times \cdots \times T_n) & = & \mathsf{FV}\,(T_1) \ \cup\ \ldots \ \cup\ \mathsf{FV}(T_n) & \\
\mathsf{FV}\,(T^*) & = & \mathsf{FV}(T) & \\
\mathsf{FV}\,(\forall\,\alpha\ .\ T) & = & \mathsf{FV}\,(T) \ -\ \{\,\alpha\,\} &
\end{array}
$$

FV is extended to type contexts by $\mathsf{FV}(\Gamma) = \mathsf{FV}(T_1) \cup \ldots \cup \mathsf{FV}(T_n)$, where $\Gamma = id_1 : T_1, \ldots, id_n : T_n$.

A substitution S on types is a function in $Type \to Type$ that changes only finitely many free occurrences of variables without violating α-congruence [Bar84, Bar89]. We denote it by $[T_1/\alpha_1, \ldots, T_n/\alpha_n]$ where the α_i are the type variables to be replaced by the types T_i. Its application to a type is written in postfix notation if the above bracket notation is used.

A substitution can also be applied to type contexts by applying it componentwise to each type: $S(id_1 : T_1, \ldots, id_n : T_n) = id_1 : S(T_1), \ldots, id_n : S(T_n)$.

The language of terms (abstract syntax trees) consists of identifiers, type-decorated identifiers, constants, λ-abstractions, applications, n-tuples, and lists:

$$
\begin{array}{lcll}
\mathit{Term} & \Longrightarrow & \mathit{Ident} & \\
 & | & \mathit{Ident}_{\mathit{Type}} & \\
 & | & \mathit{Const} & \\
 & | & (\lambda\, \mathit{Ident}, \ldots, \mathit{Ident}\ .\ \mathit{Term}) & \\
 & | & (\mathit{Term}\ \ \mathit{Term}) & \\
 & | & (\ \mathit{Term}_1, \ldots, \mathit{Term}_n\ ) & (n \geq 2) \\
 & | & [\ \mathit{Term}, \ldots, \mathit{Term}\ ] &
\end{array}
$$

Const contains at least the natural numbers, i.e. $\mathbb{N} \subseteq \mathit{Const}$.

What are the differences to usual λ2-calculi?

- There are two different classes of identifiers. The undecorated ones in *Ident* are *bound variables* or *λ-variables* bound by abstraction. Their names do not matter due to α-conversion. $\mathit{Ident}_{\mathit{Type}}$ is the set of identifiers annotated with types to handle overloading of identifiers.[3] They are not λ-variables but *free variables* coming from a global type context. Note that λ-variables cannot be overloaded.
- λ-abstraction takes a sequence of bound variables to be abstracted, instead of a simple variable. A discussion of the advantages can be found in [PE88]. In fact, this is a special form of pattern-matching in heads of λ-abstractions [PJ87].

Once again, these differences are only of a syntactical nature.

[3] This can be compared with name mangling of overloaded operators in C++.

$$\frac{(id,T)\in\Sigma}{\Sigma\ \rhd\ \Gamma\ \vdash\ id_T\ :\ T} \qquad \frac{(id,T)\in\Gamma}{\Sigma\ \rhd\ \Gamma\ \vdash\ id\ :\ T} \qquad \frac{n\in\mathbb{N}}{\Sigma\ \rhd\ \Gamma\ \vdash\ n\ :\ \mathbb{N}}$$

$$\frac{\Sigma\ \rhd\ \Gamma\ \vdash\ t_1\ :\ T\to T' \qquad \Sigma\ \rhd\ \Gamma\ \vdash\ t_2\ :\ T}{\Sigma\ \rhd\ \Gamma\ \vdash\ (t_1\ t_2)\ :\ T'}\ (\to E)$$

$$\frac{\Sigma\ \rhd\ \Gamma,\, id_1 : T_1, \ldots, id_n : T_n\ \vdash\ t\ :\ T}{\Sigma\ \rhd\ \Gamma\ \vdash\ (\lambda\, id_1, \ldots, id_n\,.\, t)\ :\ T_1\times\cdots\times T_n\to T}\ (\to I)$$

$$\frac{\Sigma\ \rhd\ \Gamma\ \vdash\ t_1\ :\ T_1 \quad \ldots \quad \Sigma\ \rhd\Gamma\ \vdash\ t_n\ :\ T_n}{\Sigma\ \rhd\Gamma\ \vdash\ (t_1,\ldots,t_n)\ :\ T_1\times\ldots\times T_n}\ (\times I) \qquad (n\geq 2)$$

$$\frac{\Sigma\ \rhd\Gamma\ \vdash\ t_1\ :\ T \quad \ldots \quad \Sigma\ \rhd\Gamma\ \vdash\ t_n\ :\ T}{\Sigma\ \rhd\Gamma\ \vdash\ [t_1,\ldots,t_n]\ :\ T^*}\ ({}^*I)$$

$$\frac{\Sigma\ \rhd\ \Gamma\ \vdash\ t\ :\ \forall\,\alpha\,.\,T}{\Sigma\ \rhd\ \Gamma\ \vdash\ t\ :\ T[T'/\alpha]}\ (\forall E)$$

$$\frac{\Sigma\ \rhd\ \Gamma\ \vdash\ t\ :\ T \qquad \alpha\notin \mathsf{FV}(\Gamma) \qquad \mathsf{FV}(\Sigma)=\{\}}{\Sigma\ \rhd\ \Gamma\ \vdash\ t\ :\ \forall\,\alpha\,.\,T}\ (\forall I)$$

Figure 5.1: The typing rules of $\lambda 2$.

Given type contexts Σ and Γ, a term $t \in \mathit{Term}$, and a type $T \in \mathit{Type}$, then the typing relation

$$\Sigma\ \rhd\ \Gamma\ \vdash\ t\ :\ T$$

states that "under the type context Γ — containing the bound variables — the term t has type T relative to a context Σ of free variables". This approach with two contexts — Γ will contain the λ-variables and Σ the free variables — is similar to Crole's in [Cro93], except that he does not handle overloading at all. Note that in our setting, Γ will contain at most one occurrence of a certain identifier because λ-variables cannot be overloaded.

The defining typing rules for the relation are given in Figure 5.1. The rule $(\forall I)$ requires the assumption $\mathsf{FV}(\Sigma) = \{\}$, which is appropriate because

the types of identifiers should not contain any free type variables.

An introduction rule for the unit type and elimination rules for lists are unnecessary if the global type context Σ is augmented by suitable operator declarations:

$$\begin{array}{rcl} () & : & () \\ \mathsf{head}\_ & : & \forall\,\alpha\,.\;\alpha \times \alpha^* \to \alpha \\ \mathsf{tail}\_ & : & \forall\,\alpha\,.\;\alpha \times \alpha^* \to \alpha^* \end{array}$$

Elimination rules for products, i.e. tuple selection, are for free, due to the product structure of abstraction. For example, the first component in a pair is extracted by $(\lambda\,(x,y)\,.\;x\,)\;(t_1,t_2)$.

Perhaps one wonders that the global context Σ is not changed by any rule but only used read-only in the first rule. An additional let-construct for declaring local variables, as found in most functional languages, could be added to the language. This would suggest a rule extending the context Σ by new variable declarations. The condition that Σ may not possess free variables is satisfied by $\forall$-closing possible free variables, according to the treatment in Hindley-Milner style type inference.

5.6.2 The interpretation

Given a base alphabet C containing *Ident*, interpretation of higher-order mixfix syntax in $\lambda 2$ is given through the following relation, where Γ and Σ are contexts, t a term and T a type, as above, and additionally $s \in C^*$:

$$\Sigma \;\triangleright\; \Gamma \;\vdash\; s \;\Rightarrow\; t \;:\; T$$

This is pronounced as "under the context Γ the input string s parses to the term t with type T, relative to the global context Σ".

The relation associates strings with the $\lambda 2$ relative to the type contexts. It is defined according to the rules in the Figures 5.2 and 5.3. We reuse the "print" functions $d_{op,A,B}$ from Section 3.3.1, which are extended accordingly to sequence and binding operators. To ease the discussion we omit the treating of local mixfix declarations.

Rules (0) and (0′) correspond to the first two rules in Figure 5.1. They translate a string denoting an identifier into the corresponding identifier term. Rule (1) allows the use of the rules $(\forall E)$ and $(\forall I)$ of $\lambda 2$.

Rules (2) and (2′) denote the translation of an ordinary mixfix appli-

$$(0)\quad \frac{(id,T)\in\Sigma}{\Sigma\ \rhd\ \Gamma\ \vdash\ id\ \Rightarrow\ id_T\ :\ T} \qquad (0')\quad \frac{(id,T)\in\Gamma}{\Sigma\ \rhd\ \Gamma\ \vdash\ id\ \Rightarrow\ id\ :\ T}$$

$$(1)\quad \frac{\Sigma\ \rhd\ \Gamma\ \vdash\ s\ \Rightarrow\ t\ :\ T \qquad \Sigma\ \rhd\ \Gamma\ \vdash\ t\ :\ T'}{\Sigma\ \rhd\ \Gamma\ \vdash\ s\ \Rightarrow\ t\ :\ T'}$$

$$(2)\quad \frac{\begin{array}{c}\Sigma\ \rhd\ \Gamma\ \vdash\ id_T\ :\ A\to B\\ \Sigma\ \rhd\ \Gamma\ \vdash\ w\ \Rightarrow\ t\ :\ A\\ s\in d_{id,A_1\times\cdots\times A_n,B}\,(w_1,\ldots,w_n)\end{array}}{\Sigma\ \rhd\ \Gamma\ \vdash\ s\ \Rightarrow\ (id_T\ t)\ :\ B}$$

$$(2')\quad \frac{\begin{array}{c}\Sigma\ \rhd\ \Gamma\ \vdash\ id_T\ :\ A_1\times\cdots\times A_n\to B\\ \Sigma\ \rhd\ \Gamma\ \vdash\ w_1\ \Rightarrow\ t_1\ :\ A_1\\ \vdots\\ \Sigma\ \rhd\ \Gamma\ \vdash\ w_n\ \Rightarrow\ t_n\ :\ A_n\\ s\in d_{id,A_1\times\cdots\times A_n,B}\,(w_1,\ldots,w_n)\end{array}}{\Sigma\ \rhd\ \Gamma\ \vdash\ s\ \Rightarrow\ (id_T\,(t_1,\ldots,t_n))\ :\ B} \qquad (n\geq 2)$$

Figure 5.2: The parsing rules for higher-order mixfix syntax (contd.).

cation into a $\lambda 2$-application. This is done by searching an appropriate operator id_T and by trying recursively to find terms for each argument string. Analogously, rule (3) transforms an application of a sequence operator into a $\lambda 2$-application with a list operand.

Binding operators are treated in rule (4). This rule is a bit more complicated because we have to extract the bound variables $x_1,\ldots,x_m$ from the input string s, adding them to the type context and searching for a suitable term for the λ-body. The resulting type A' is not needed further because the whole abstraction is tested for type correctness in the last premise (alternatively, one could test whether $\alpha_1\times\cdots\times\alpha_m\to A'$ is an

$$(3)\qquad \frac{\begin{array}{c}\Sigma \rhd \Gamma \vdash id_T : A^* \to B \\ \Sigma \rhd \Gamma \vdash w_1 \Rightarrow t_1 : A \\ \vdots \\ \Sigma \rhd \Gamma \vdash w_n \Rightarrow t_n : A \\ s \in d_{id,A^*,B}\,(w_1,\ldots,w_n)\end{array}}{\Sigma \rhd \Gamma \vdash s \Rightarrow (id_T\,[t_1,\ldots,t_n]) : B}$$

$$(4)\qquad \frac{\begin{array}{c}\Sigma \rhd \Gamma \vdash id_T : A_1 \times \cdots \times A_i \times \ldots \times A_n \to B \\ \Sigma \rhd \Gamma \vdash w_1 \Rightarrow t_1 : A \\ \vdots \\ \Sigma \rhd \Gamma, x_1{:}\alpha_1, \ldots, x_m{:}\alpha_m \vdash w_i \Rightarrow t_i : A' \\ \vdots \\ \Sigma \rhd \Gamma \vdash w_n \Rightarrow t_n : A \\ s \in d_{id,A_1\times\cdots\times A_i\times\ldots\times A_n,B}\,(w_1,\ldots,w_n) \\ s = \cdots\ x_1,\ldots,x_m\ \cdots \\ \Sigma \rhd \Gamma \vdash \lambda\, x_1,\ldots,x_m.t_i : A_i\end{array}}{\Sigma \rhd \Gamma \vdash s \Rightarrow id_T\left(t_1,\ldots,t_{i-1},(\lambda\, x_1,\ldots,x_m\,.\,t_i),t_{i+1},\ldots,t_n\right) : B}$$

Figure 5.3: The parsing rules for higher-order mixfix syntax.

instance of the type A_i).

A soundness proof that only well-typed terms are generated by the rules, i.e. $(\Sigma \rhd \Gamma \vdash s \Rightarrow t : T) \to (\Sigma \rhd \Gamma \vdash t : T)$, is omitted. This can be done by structural induction.

The rules are certainly not deterministic, and due to certain coercions not terminating in general, so an algorithm cannot be derived directly from them. Because of Wells' result that type checking and type inference for second-order polymorphic λ-calculi in Curry-style are undecidable [Wel94], concrete implementations have to put some restrictions on the above rules anyway.

5.7 Other foundational formalisms

We briefly want to discuss the shortcomings of related formalisms which are also used as meta-notations to describe ordinary mathematical syntax.

Révész and Lynch use a context-free grammar to describe mathematical notation [RL91]. But, as opposed to higher-order mixfix syntax, context-free grammars cannot cope with the context sensitivities arising from higher-order polymorphic type systems and local declarations — say through binding constructs — as discussed in Section 3.4. They are more appropriate for ordinary programming languages, which have a rather fixed structure and far fewer context-sensitive features than mathematical notation.

What happens if we take a step upwards in the Chomsky hierarchy [EL92] and use context-sensitive grammars? The main problem is that the context sensitivities have to be coded into the productions, leading to huge and unreadable grammars. In fact, they are not used in practice for any purpose.

Methods such as [LCA94, Zha96] that extend context-free grammars with some form of attributes to handle context sensitivities have the following disadvantage: apart from ordinary types they also need *syntactic types* within their grammars. In fact, they do not take full advantage of higher-order abstract syntax. This approach complicates the whole description of mathematical syntax because the user is confronted with pure syntactical types. The same problem arises in Isabelle, which makes syntactical types — such as "`idt`" for the type of identifiers — visible to the user [Pau94, p. 144].

Chapter 6

Design of the Parsing System

This chapter describes the design of a parsing system that is capable of handling higher-order mixfix syntax. As mentioned in the preliminaries, the task of such a system is easily described. It transforms an input string in some fixed alphabet into a suitable abstract syntax tree. The whole transformation is done relative to some grammar, which is the representation of the context currently in use.

Usually, a scanner resides in the front of the actual parser. Its task is to remove whitespaces and comments, and to transform the lexems of the input string into tokens, which are then passed to the parser. This simplifies the grammatical description for the parser and speeds it up [ASU86].

The next section discusses the requirements for the algorithmic and the data structure part of a suitable parsing system in more detail. Current parsing methods are then investigated for their usefulness. Finally, we give designs for a bracketing dynamical scanner, suitable data structures for representing grammars induced by contexts, and for a parser.

6.1 The requirements for the parsing system

The data structure for representing contexts contains

- the type information of all occurring identifiers,
- the syntactical shape of all mixfix operators, and
- their operator attributes.

Such a data structure is a modular unit of the whole context, which allows us to manage several contexts independently. Operations on contexts should have fast counterparts in the offered set of data structure operations. The most important operations are incremental extensions by new (mixfix) declarations and the merging of two or more contexts to get a new composed context. A further point is the ability to reconstruct the original textual representation of the context from the data structure used (e.g. in order to display it in a readable form in an interactive environment).

Type inferencing is integrated into the parser, combining parsing and type inference into one process. This combination is essential for contexts containing mixfix declarations that lead to active semantic ambiguities (e.g. function application written juxtaposed). In systems such as ISABELLE, that separate the two processes [Pau94, p. 140], a lot of intermediate untyped abstract syntax trees have to be generated, only to be discarded by the later type inference pass. Analogously for operator attributes: because they can depend on types, the removal of invalid parse trees can only be performed *after* the type inference process! Thus, to be on the safe side, the parser has to produce all possible untyped parse trees — their number can be huge. The general idea of integrating type inference into parsing is simply to remove impossible abstract syntax trees at the earliest possible point in the process.

Type inference supports the possibility of leaving explicit or implicit bound variables undeclared, if their type can be derived from the actual context. This derivation is, in fact, often possible. Undeclared bound variables can be recognized by their special syntactic form, discussed in Section 4.1.

Whitespaces are nowhere compulsory. This can lead to unreadable input if too many whitespaces are omitted. But it is up to the user not to abuse

this feature, which is, after all, essential for supporting mathematical notation. This is a rather hard task for a scanner because lexems too become context-dependent! Take the following example of the type of formal power series over $\mathbb{Z}$ and an arithmetical expression, both using parenthesis:

$$\mathbb{Z}((x)) \qquad\qquad 1/((x+1)z)$$

The lexem "((" occurs in both expressions but corresponds to *one* single token in the former and to *two* tokens in the later case. Similarly for the lexem "xy" which could be one or two tokens, depending on the context. Therefore, the rule "take the longest possible match for a token" does not work! Note that most scanners use this rule, e.g. ISABELLE's scanner or the scanners generated by the lexical analyzer generator LEX [MB91].

The parsing system certainly should support features specific to higher-order mixfix syntax:

- all operator attributes, namely relative priorities, implicit associativities and positional exclusion,
- sequence operators,
- binding operators, and
- type constructors built by mixfix declarations.

Finally, parsing performance should be acceptable for small- and medium-sized input. At any rate, typical applications of such a parsing system are interactive environments where user input is at most some lines long. We cannot expect polynomial space complexity in general because one easily constructs contexts which lead to an exponential number of possible parse trees. A simple example is the context

$$x : T\,, \quad \_\_ : T \times T \to T$$

for an arbitrary type T. The number of parse trees grows exponentially with the length of the input string "$xx\cdots x$" (the numbers are the so-called Catalan numbers [Knu73]).

6.2 Existing parsing methods and their deficiencies

In the next sections, we discuss some existing parsing methods under the aspect of how they behave, taking the requirements defined in the preceding section into consideration.

A similar comparison can be found in Rekers' dissertation in the context of a meta-environment for generating interactive programming language environments [Rek92, p. 43–46].

6.2.1 Recursive-descent and shift-reduce algorithms

Recursive-descent and shift-reduce algorithms are the most widely-used techniques for parsing conventional programming languages [ASU86]. They work in linear time with respect to the input size and use tables — which have to be generated beforehand — for representing context-free grammars. The accepted grammars have to be unambiguous and the addition of new rules leads to recomputing of whole tables. Because the set of accepted grammar is not closed under union, the merging operation of grammars would be partial.

Thus, these algorithms are neither incremental nor modular and cannot parse sufficiently powerful languages.

6.2.2 Earley's algorithm

The algorithm presented by Earley accepts arbitrary context-free grammars [Ear70]. It works by attaching a set of states to each character in the input where a single state consists of

1. a production of the context-free grammar extended by a special dot symbol, which marks the already recognized part of the production, and
2. a position in the input string where this production was entered.

To start with, the state set for the first character consists of the productions with the start symbol to the left. This set is extended and transformed into the state set of the next character by the following operations applied to each state:

- If there is a non-terminal symbol to the right of the dot, all productions starting with this nonterminal are added to the current state.
- If there is a terminal to the right of the dot, the next input character is compared with this terminal and, if they match, the state is added to the next state set, moving the state's dot one position to the right.
- If the dot is at the end of the production, we have recognized a substring in the input. This state is removed and all states of previous sets with a dot to the left of the same non-terminal are moved to the current set.

This scheme corresponds to a recursive-descent parser which treats all possible parses simultaneously.

This recognizing process can be done with cubic time complexity but with a high constant. From the state sets, an abstract syntax tree can then be generated. The algorithm works directly with productions, so it is incremental and modular because no separate generation phase is needed. It does, however, make the algorithm too inefficient for interactive purposes [Rek92, p. 44], and, furthermore, priorities and associativities have to be coded directly into the grammar and cannot be given separately.

6.2.3 GLR and IPG

The generalized LR parsing algorithm (GLR) and the incremental parser generator (IPG) are extensions of shift-reduce techniques to handle the whole class of context-free grammars (GLR) and to pay more attention to incremental extensions of grammars and parse tables (IPG) [Rek92].

A GLR parser works as a normal LR parser until a shift-reduce or a reduce-reduce conflict is encountered. Then it splits up into several new parsers which work in parallel.

IPG generates LR(0) tables needed for GLR in a lazy fashion. The tables are generated while parsing the input and only those parts are generated which are needed by that specific input. Furthermore, IPG is incremental because parts not affected by grammar modifications are reused. IPG is not modular because the composition of whole parsers is not supported [Rek92, p.61].

6.2.4 Parser combinators

The method of parser combinators uses functions to encode context-free grammars in a direct way [Bur75, Pau91, Fok95]. This encoding results in executable parsers. Elementary parsers are functions which recognize terminal symbols. New parsers are constructed by parser combinators such as sequential and alternative composition, which are functions operating on parsers. Thus, higher-order functions are essential for this approach.

The main drawbacks of this approach are that left recursive productions cannot be handled due to infinite recursion, similarly to recursive-descent parsers, and that the encoding is neither incremental nor modular.

6.2.5 Bauer's mixfix algorithm

Bauer's method of parsing mixfix syntax is based on a mixture of recursive-descent and shift-reduce strategy [Bau92], and is used in the algebraic-specification system ISAR.

Depending on the precedence and implicit associativity of the currently parsed subexpression, the next input symbol is read, a reduction step is performed or the algorithm is called recursively. This scheme disallows juxtaposition because the individual arguments of an operator have to be separated by non-empty strings, similarly to operator precedence parsing [Pra73, ASU86]. Furthermore, the reuse of separators in different operators is restricted and types have no influence on parsing. Everling remedies the last two drawbacks by returning all possible parses and performing an additional type-inference pass after parsing [Eve95]. His implementation uses SML.

The main shortcomings of the method are the restriction to non-juxtaposed operators and the type-inference pass as a separate subsequent phase.

6.2.6 The OBJ3 parser

The parser of the OBJ3 system combines recursive descent parsing with backtracking, which makes the parsing expensive for larger input. Ambiguous expressions, i.e. expressions where no unique parse of lowest sort exists, are detected during parsing, but the current implementation

nevertheless fails to detect this in every case [G+92, p. 22].

6.2.7 Cigale's parsing with tries

CIGALE is specially tailored for incremental grammar construction and expression parsing [Voi86]. Voisin's clever idea was to use *tries* for representing contexts in CIGALE. Tries are standard data structures for handling finite maps (dictionaries) where the components of the argument (the key) are used to navigate the search within the underlying tree [Knu73].

In CIGALE, the whole context information is stored in a trie, except coercions, which are treated separately. Coercions induce a partial order $\leq$ on sorts (a subtype relation). As opposed to Section 3.3.1, where the subtype relation could depend on contexts, this is not the case in CIGALE because there are no polymorphic types but only sorts.

As Voisin did in his publication, we present CIGALE's parsing method as a recognizer to simplify the discussion. Furthermore, the following presentation is slightly different to Voisin's original one because we translated his imperative specification of the algorithm into a functional one. This also has a small effect on the underlying trie data structure.

In CIGALE's tries, mixfix operators and normal identifiers with a common prefix share the same path, hence one gets full factorization of paths for free, and this reduces the need for backtracking during the parsing process. The trie is built recursively by the following three kinds of nodes:

1. A terminal node contains a separator symbol and a subsequent trie, where the remaining operators with the same prefix are stored.
2. A source node contains a type required at this position in an application, and also a subsequent trie as in terminal nodes.
3. A target node contains a type, which stands for the range of the path stored above in the trie. They are the leaves of the trie.

A trie is now simply a set of nodes. This can be expressed by a fix-point equation. Given a set *Sep* of separators and a set *Sort* of sorts, the set of tries denoted by *Trie* is the least set fulfilling:

$$\mathit{Trie} \;=\; \mathsf{Set}\left(\mathit{Sep} \times \mathit{Trie} \;+\; \mathit{Sort} \times \mathit{Trie} \;+\; \mathit{Sort}\right)$$

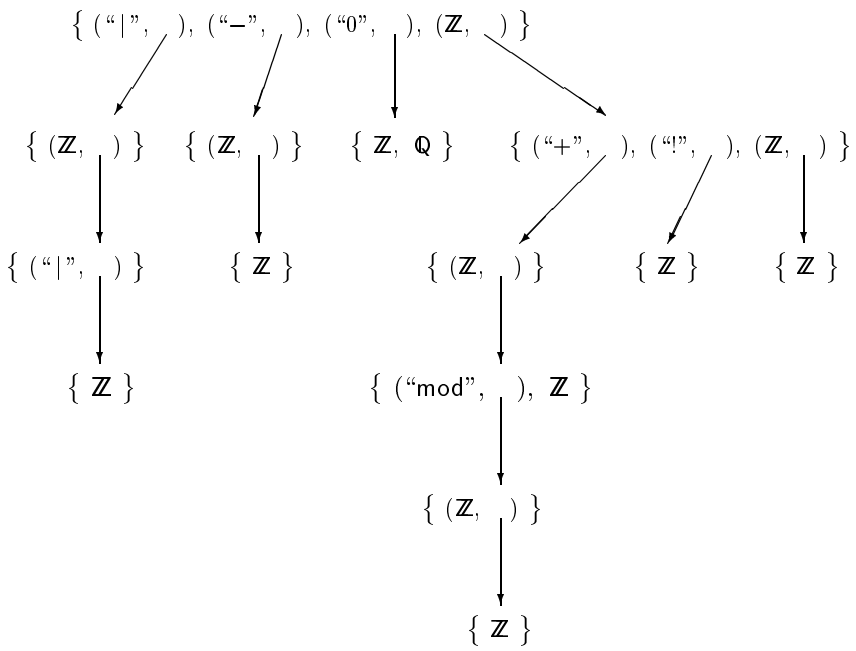

Figure 6.1: A CIGALE trie.

The set $Sep \times Trie$ denotes the terminal nodes, $Sort \times Trie$ the source nodes, and $Sort$ the target nodes in the disjoint union.

Figure 6.1 shows an example of such a trie, where we omitted the individual injections into the sums. The trie stands for the following context:

$$\begin{array}{rcl} 0 & : & \mathbb{Z}\,, \\ -\_ & : & \mathbb{Z} \to \mathbb{Z}\,, \\ \_ + \_ & : & \mathbb{Z} \times \mathbb{Z} \to \mathbb{Z}\,, \\ \_ + \_ \,\mathsf{mod}\, \_ & : & \mathbb{Z} \times \mathbb{Z} \times \mathbb{Z} \to \mathbb{Z}\,, \end{array} \qquad \begin{array}{rcl} 0 & : & \mathbb{Q}\,, \\ \_! & : & \mathbb{Z} \to \mathbb{Z}\,, \\ |\_| & : & \mathbb{Z} \to \mathbb{Z}\,, \\ \_\_ & : & \mathbb{Z} \times \mathbb{Z} \to \mathbb{Z} \end{array}$$

Note the sharing of common prefixes in the trie and the treatment of overloading in the case of the identifier "0".

Tries are incremental and modular. A newly-introduced identifier is transformed first into a target type and a list of separators and source types. Its insertion only affects the path with the same prefix. Fur-

thermore, two tries can easily be merged to form a new trie. Only the common paths between the tries need changing, the other can be reused without modification.

The parsing algorithm traverses the trie recursively, "eating" possible prefixes of the current input according to the terminal symbols in the trie. In the case of source nodes, the algorithm is called recursively. The recursion stops when a target node in the trie is reached.

Left-open operators — operators that begin with at least one placeholder — introduce the same problem as left-recursive productions in a recursive-descent parser: One gets a non-terminating recursion if left-open operators are processed naively. Hence, the algorithm has to treat left-open mixfix operators differently from left-closed ones.

As already mentioned, we present the algorithm in Figure 6.2 as a recognizer, so the final result is a boolean. The curried function parse is the main routine, which is called from outside with two arguments: the root trie representing the current context, and a sequence of separators that has to be parsed. All other functions are locally defined, hence not visible outside. The routines loop and traverse use the formal parameter *root* of parse.

The error element $\perp$ denotes an unsuccessful try to parse a prefix of the given input w. A pair $(T,\, w') \in \mathsf{Set}\,(Sort) \times Sep^*$ denotes the successful partial parse of w such that $w = w''w'$, and one of the types in T belongs to the prefix w''. We call elements in $(\mathsf{Set}\,(Sort) \times Sep^*)_\perp$ *parse states*. A parse state $(T,\, \varepsilon)$ denotes the successful parsing of the whole input w.

The functions terminals, sources, and targets extract the corresponding kinds of nodes from the passed trie, returning them as a set.

The main routine parse does a first traverse with the root trie, trying to reduce the input with a left-closed operator. The resulting parse state is then passed to loop, where left-open operators are handled.

The tail-recursive function loop — being a "while" loop in Voisin's original formulation — calls fill with the current parse state and with all the source nodes of *root*. They represent the paths of left-open operators.

The routine fill gets a set of source nodes and a parse state. It tries to find sources that are compatible with the parse state's types. It searches for an arbitrary node among the sources — expressed by the **such that** keyword — whose type is a supersort of one of the types passed in the parse state, giving preference to minimal supersorts (minimal with

$$
\begin{array}{lcl}
\mathsf{parse} & : & \mathit{Trie} \;\to\; \mathit{Sep}^* \;\to\; \mathbb{B} \\
\mathsf{parse}\ \mathit{root}\ w & = & \mathsf{loop}\ (\mathsf{traverse}\ \mathit{root}\ w) \\
 & & \\
\mathsf{terminals} & : & \mathit{Trie} \;\to\; \mathsf{Set}\,(\mathit{Sep} \times \mathit{Trie}) \\
\mathsf{terminals}\ tr & = & \{\, (s,\ tr') \mid \mathsf{inj}_1(s,\ tr') \in tr \,\} \\
 & & \\
\mathsf{sources} & : & \mathit{Trie} \;\to\; \mathsf{Set}\,(\mathit{Sort} \times \mathit{Trie}) \\
\mathsf{sources}\ tr & = & \{\, (t,\ tr') \mid \mathsf{inj}_2(t,\ tr') \in tr \,\} \\
 & & \\
\mathsf{targets} & : & \mathit{Trie} \;\to\; \mathsf{Set}\,(\mathit{Sort}) \\
\mathsf{targets}\ tr & = & \{\, t \mid \mathsf{inj}_3(t) \in tr \,\} \\
 & & \\
\mathsf{loop} & : & \big(\mathsf{Set}\,(\mathit{Sort}) \times \mathit{Sep}^*\big)_\perp \;\to\; \mathbb{B} \\
\mathsf{loop}\ \perp & = & \mathsf{false} \\
\mathsf{loop}\ (T,\ \varepsilon) & = & \mathsf{true} \\
\mathsf{loop}\ (T,\ w) & = & \mathsf{loop}\,\big(\,\mathsf{fill}\,\big(\,\mathsf{sources}\ \mathit{root},\ (T,\ w)\,\big)\,\big) \\
 & & \\
\mathsf{fill} & : & \mathsf{Set}\,(\mathit{Sort} \times \mathit{Trie}) \times \big(\mathsf{Set}\,(\mathit{Sort}) \times \mathit{Sep}^*\big)_\perp \;\to\; \big(\mathsf{Set}\,(\mathit{Sort}) \times \mathit{Sep}^*\big)_\perp \\
\mathsf{fill}\ (S,\ \perp) & = & \perp \\
\mathsf{fill}\ (\{\,\},\ (T,\ w)) & = & \perp \\
\mathsf{fill}\ (S,\ (T,\ w)) & = & \mathbf{let}\ M = \{\, ((t_1,\ tr),\ t_2) \in S \times T \mid t_2 \le t_1 \,\} \\
 & & \mathbf{in\ if}\ M = \{\,\}\ \mathbf{then}\ \perp\ \mathbf{else} \\
 & & \quad \mathbf{let}\ ((t_1,\ tr),\ t_2) \in M\ \mathbf{such\ that} \\
 & & \qquad \forall\, ((t_1',\ tr'),\ t_2') \in M \;.\; t_2 = t_2' \;\to\; t_1' \not\leq t_1 \\
 & & \quad\ \ \mathit{res} = \mathsf{traverse}\ tr\ w \\
 & & \quad \mathbf{in} \\
 & & \quad\ \ \mathbf{if}\ \mathit{res} \neq \perp\ \mathbf{then}\ \mathit{res}\ \mathbf{else}\ \mathsf{fill}\,\big(S - \{(t_1,\ tr)\},\ (T,\ w)\big) \\
 & & \\
\mathsf{traverse} & : & \mathit{Trie} \;\to\; \mathit{Sep}^* \;\to\; \big(\mathsf{Set}\,(\mathit{Sort}) \times \mathit{Sep}^*\big)_\perp \\
\mathsf{traverse}\ tr\ w & = & \mathsf{traverse}\ tr'\ w',\ \ \mathbf{if}\ \exists\ (s,\ tr') \in \mathsf{terminals}\ tr\ .\ \exists\ w' \in \mathit{Sep}^*\ . \\
 & & \qquad\qquad\qquad\qquad w = sw'\ \wedge\ \mathsf{traverse}\ tr'\ w' \neq \perp \\
\mathsf{traverse}\ tr\ w & = & \mathit{res},\qquad\qquad \mathbf{if}\ \ tr \neq \mathit{root}\ \wedge\ \mathit{res} \neq \perp \\
 & & \qquad\qquad\qquad\qquad \mathbf{where} \\
 & & \qquad\qquad\qquad\qquad\ \ \mathit{res} = \mathsf{fill}\,\big(\mathsf{sources}(tr),\ \mathsf{traverse}\ \mathit{root}\ w\big) \\
\mathsf{traverse}\ tr\ w & = & (\mathsf{targets}\ tr,\ w),\ \ \mathbf{if}\ \ (\mathsf{targets}\ tr) \neq \{\,\} \\
\mathsf{traverse}\ tr\ w & = & \perp,\qquad\qquad \mathbf{otherwise}
\end{array}
$$

Figure 6.2: CIGALE's trie walking recognizer.

respect to $\leq$). This search is carried out as long there is a successful parse of the remaining input.

The algorithm does not try to find all possible interpretations of a string but returns only the "best" one. This best one is defined by the strategy of the function traverse while processing each node set. Given an actual node set and a rest of the input sequence, the set is processed as follows:

1. *The terminal case.* A terminal node whose separator is a prefix of the input is searched. If there is no such node, then traverse continues with the next step, else it is called recursively and tries to parse the remaining sequence with the subsequent trie. If this fails then the next step is taken, else the resulting successful parse state is returned.

2. *The source case.* A recursive call of traverse with *root* as the actual trie and the remaining sequence is performed to parse an argument of the actual operator. The result is passed to fill, including the source nodes. Note that left-open operators are *not* treated.

3. *The target case.* If there are target nodes, then a parse state is built and returned.

4. *The unlucky case.* $\perp$ is returned because there is no possible parse.

The expected types in the source node case are not used to navigate within the trie, but only used to filter out unsuitable paths in fill.

CIGALE's main shortcomings are:

- Its restriction to a first-order monomorphic type system.
- Neither local mixfix declarations, nor undeclared variables, hence type inference is simple.
- The lack of priorities and associativities. If they are not imposed by the productions of the grammar, all operators are treated by the algorithm as *left-associative* and of *equal priority*.
- The algorithm's heuristics in choosing path tries results in the inability to detect ambiguous input and to recognize valid input in certain cases. Note that the method's complexity is nevertheless in general still exponential due to backtracking in fill and traverse.

Voisin discusses informally improvements to the algorithm to remedy some of these shortcomings [Voi86].

- No character-based parsing and therefore no whitespace-less juxtaposition.

6.2.8 Conclusions

The first four methods parse languages induced by restricted or arbitrary context-free grammars. In Section 3.4 we showed how to translate mixfix declarations into context-free productions, but as we said there, this can only be done by losing the possibility of representing higher-order and polymorphic types. Thus, neither of these algorithms are suited for our purposes. Similarly for the parser of the OBJ3 system and for Bauer's mixfix algorithm, which, moreover, does not use types at all for parsing.

The main advantage of CIGALE's data structure is its independence of a specific type system, especially a first-order monomorphic one. Note that this fact is not exploited by the author of CIGALE. We will do this in the next section. We are not aware of another grammar representation technique that allows the integration of type checking and the parsing process without any severe restrictions to the underlying type system.

6.3 Parsing higher-order mixfix syntax

This section presents designs for a scanner that is specially tailored for the lexical analysis of mathematical notation, an adapted trie data structure for representing the language induced by higher-order mixfix syntax, and finally a parser which translates the output of the scanner in a $\lambda 2$-term. Both the scanner and parser depend on an actual context.

We assume a base alphabet C of characters which contains as subsets the set L of letters, digits (D), graphical characters (G), and whitespace characters W, all four sets being pairwise disjoint. Besides the reserved characters $R = \{\text{“\_”}, \text{“\_.”}, \text{“.\_”}, \text{“}\cdots\text{”}\}$ required within higher-order mixfix identifiers, we have a special set of embellishment characters $E \subset G$.

Furthermore, we assume a set *Token* of tokens such that we can embed non-empty whitespace-less strings in it: $(L \cup D \cup G \cup R)^+ \hookrightarrow \mathit{Token}$. We identify in *Token* the set *Ident* of all valid higher-order mixfix identifiers.

6.3.1 Bracketing and dynamical scanning

Does a scanner make sense at all for the parsing process of mathematical notation? And if so, what are the tasks of a scanner in such a dynamical changing language environment? We show that a scanner can fulfill even more tasks than in traditional programming languages.[1]

Its traditional tasks are retained but partly adapted:

- Removal of unnecessary whitespaces.
- It detects the morphemes[2] of mathematical syntax and transforms the corresponding substrings to tokens. The parser can then work exclusively with tokens, whose main property are the faster equality and inequality tests — often in constant time — than those for substrings. Hence, it does not matter for the parser if we use long or short forms for identifiers.

  In any case, this translation process depends on the current context Σ, thus we have to maintain some scan table, derived from $\mathsf{dom}\,(\Sigma)$. Depending on $\mathsf{dom}\,(\Sigma)$, we can get more than one possible token list for a given input, as already noted in Section 6.1. Furthermore, the non-compulsory use of whitespaces between lexems complicates the scanning process.
- Recognition of identifiers not contained in the scan table. Such identifiers are locally introduced by binders or local definitions. They require incremental extensions of the scan table! We restrict the form of *undeclared* identifiers to be in the set $\mathit{UIdent} = L(L \cup D)^* E^*$. This corresponds to mathematical practice of the syntactical shape of bound variables (cf Section 4.1).
- Transforming built-in constants. Because mathematical notation has almost nothing that could count as universally built-in, we restrict ourselves to digit sequences. Hence, the scanner should recognize a sequence of digits — unseparated by whitespaces — as atomic. Digit sequences in an undeclared identifier are treated as suffix of the identifier.

[1] If we do not stick to the condition that the scanner has to be implemented as finite automaton.

[2] A morpheme is the smallest meaning carrying part of a language.

An alternative would be to take single digits as atomic, including juxtaposed operators to construct digit sequences. But this approach causes problems due to possible whitespaces between the digits. In fact, mathematical usage seems to treat digit sequences in a special manner: they never denote a juxtaposed written multiplication, as opposed to variables denoting numbers! In the case of multiplication, an infix operator is used in the case of explicit digits arguments. Similarly to the cycle representation for permutations where digits are treated as one number if not separated by whitespaces.

With the additional help of the positional exclusion attribute (Section 5.1) we may have juxtaposed multiplication for numbers as in mathematics with this special treatment of digits.

Character and string constants are the other typical examples that are treated specially. But they do not pose the same problems as digits because they are usually surrounded by a pair of bracketing symbols. We will not treat them further but restrict ourselves to digits.

The scanner thus reduces the size of the input for the parser by translating the input string into tokens. But the scanner can reduce the size even more if it takes advantage of the special syntactical form of bracketing operators (Section 3.1). Through their bracketing of intermediate arguments they already structure the input text. The scanner should detect this structure and return *structured tokens* instead of simple tokens. Accordingly, we define the set *SToken* as

$$SToken \;=\; Token \times \mathbb{B} \;+\; \mathbb{N} \;+\; Token \times \left(SToken^*\right)^*$$

Therefore, a structured token is either

- a simple token with a boolean annotation that flags if the token has the suitable form for an undeclared identifier, or
- a natural number for representing digit sequences, or
- a pair consisting of a tagging token and a list of structured tokens for each bracketed argument position. A tagging token is the unique token belonging to all bracketing operators with the same

list of separators and can be represented, for example, by putting a space character between the separators.

An example should clarify the above ideas. Given a context Σ with

$$\mathsf{dom}(\Sigma) \;=\; \{\; \mathsf{if\_then\_else\_},\; \_+\_,\; \_\_,\; (\_),\; \_|\_,\; |\_|\;\}$$

and the input string

$$\text{“}\mathit{if}\ 2|y\ \mathit{then}\ (x+y)z\ \mathit{else}\ |x|\text{”}.$$

Without structuring (bracketing) of the input, as output of an ordinary scanner we receive the following token list containing 15 elements:

$$\big[\ \text{“}\mathit{if}\text{”},\ \text{“2”},\ \text{“|”},\ \text{“}y\text{”},\ \text{“}\mathit{then}\text{”},\ \text{“(”},\ \text{“}x\text{”},\ \text{“}+\text{”},\ \text{“}y\text{”},\ \text{“)”},\ \text{“}z\text{”},\ \text{“}\mathit{else}\text{”},\ \text{“|”},\ \text{“}x\text{”},\ \text{“|”}\ \big]$$

Using the bracketing form of the operators $\mathsf{if\_then\_else\_}$, $(\_)$, and $|\_|$, we get the following output *SToken* list:

$$\big[\ \mathit{stok},\ \mathit{stok}'\ \big]$$

where

$$\begin{aligned}
\mathit{stok} \;=\; \Big(\ & \text{“}\mathit{if\ then\ else}\text{”}\,, \\
& \Big[\ \big[\,2,\ (\text{“|”},\ \mathsf{false}),\ (\text{“}y\text{”},\ \mathsf{true})\,\big], \\
& \quad \big[\,\big(\text{“(\ )”},\ \big[\,[\,(\text{“}x\text{”},\ \mathsf{true}),\ (\text{“}+\text{”},\ \mathsf{false}),\ (\text{“}y\text{”},\ \mathsf{true})\,]\,\big]\big),\ \big(\text{“}z\text{”},\ \mathsf{true}\big)\,\big]\ \Big]\ \Big) \\
\mathit{stok}' \;=\; \Big(\ & \text{“|\ |”},\ \big[\,[\,(\text{“}x\text{”},\ \mathsf{true})\,]\,\big]\ \Big)
\end{aligned}$$

We reduced the maximum size of the individual lists to four elements. Note that the following list is an equally valid output due to the multiple use of the character “|” in the context Σ:

$$\Big[\ \mathit{stok},\ \big(\text{“|”}, \mathsf{false}\big),\ \big(\text{“}x\text{”}, \mathsf{true}\big),\ \big(\text{“|”}, \mathsf{false}\big)\ \Big]$$

In general, the more structured the input is, say by explicit parenthesis, the faster the subsequent parsing process.

6.3.2 Context representation

We assume the set *Type* of types, the set *Term* of abstract syntax trees, and the set $\mathit{Ident}_{\mathit{Type}}$ of overloaded identifiers (cf Section 5.6.1).

We take CIGALE's trie data structure as a starting point for encoding the language induced by a context. We change and extend it for our purposes. Recall its recursive definition:

$$\mathit{Trie} \;=\; \mathsf{Set}\left(\mathit{Sep} \times \mathit{Trie} \;+\; \mathit{Sort} \times \mathit{Trie} \;+\; \mathit{Sort}\right)$$

We can transform it into the following fix-point equation, which better reflects the trie structure:

$$\mathit{Trie} \;=\; (\mathit{Sep} \xrightarrow{f} \mathit{Trie}) \times (\mathit{Sort} \xrightarrow{f} \mathit{Trie}) \times \mathsf{Set}(\mathit{Sort})$$

This transformation can be done because we have functional dependencies in the possible subsets of $\mathit{Sep} \times \mathit{Trie}$ and $\mathit{Sort} \times \mathit{Trie}$ due to the full factorization of the paths within the trie. Furthermore, for arbitrary sets A and B, the set $\mathsf{Set}(A+B)$ is isomorphic to $\mathsf{Set}(A) \times \mathsf{Set}(B)$, but the latter has better computational behavior because elements of sum types require some tagging. In fact, the only purpose of the functions terminals, sources, and targets in CIGALE's algorithm (Figure 6.2) was to convert from the former representation to a component of the latter.

We extend the above triple in the recursive definition of *Trie* with additional components:

$$
\begin{aligned}
\mathit{Trie} \quad = \quad & (\mathit{Type} \xrightarrow{f} \mathit{Ident}_{\mathit{Type}}) \\
& \times (\mathit{Token} \xrightarrow{f} \mathit{Trie}) \\
& \times (\mathit{Type} \xrightarrow{f} \mathit{Trie}) \\
& \times \mathit{Trie}_{\perp} \\
& \times (\mathit{Type} \xrightarrow{f} \mathit{Trie}) \\
& \times (\mathit{Type} \times \mathit{Token}_{\perp} \xrightarrow{f} \mathit{Trie}) \\
& \times (\mathit{Type} \times (\mathit{Node}^*)^* \xrightarrow{f} \mathit{Trie}) \\
\text{where} \quad \mathit{Node} \quad = \quad & \mathit{Typ} + () + \mathit{Typ} + \mathit{Typ} \times \mathit{Token}_{\perp}
\end{aligned}
$$

A trie is now a 7-tuple where the individual components store their subsequent tries either in a function or as an optional element, except the first one, which signifies an end of a path. An informal description for each component follows:

- A *target* stores the target type of a path representing a mixfix operator. As in CIGALE, it signifies the end of a possible path. The type annotated mixfix operator that corresponds to the path is assigned to each target.
- *Separators* remember the token that is needed at this place in the path.
- *Normal sources* stores the type of the corresponding argument.
- A *left source* signals the presence of a left placeholder in the path. They do not need any further information.
- Analogously, *right sources* signal the presence of a right placeholder, storing the attended functional type.
- *Sequences* are used for recognizing sequences of arguments, possibly separated by separator token (hence the optional token).
- A *structure* element represents an expected structured token. Therefore, for each gap between two separators there is a list of nodes, where a node is either of normal source, left source (denoted by the unit type), right source, or sequence kind. Conceptually, there cannot be separators or further structured elements within a node list.

Normal sources, right sources, and sequence are called *sources*.

The base case of this recursive data structure is an element of the form $(f, \bot, \bot, \bot, \bot, \bot, \bot, \bot)$ with an arbitrary function $f \in \mathit{Type} \xrightarrow{f} \mathit{Ident}_{\mathit{Type}}$ as target component. The various $\bot$-elements are certainly overloaded. They denote the corresponding empty functions or a nil value in the left-source case.

As opposed to CIGALE, we do not store normal identifiers in the trie but only in the type context, from where the trie is induced. Hence, the type context itself is also needed during parsing. As in CIGALE, coercion operators are also only stored in the type context. Additionally, a sequence operator of the form $l\_s\ \cdots\ s\_r$ is transformed internally into $l\_s\_s\_s \cdots s\_r$ (cf Figure 6.3), corresponding to the fact that sequence operators need at least 3 arguments to match (cf Section 5.3). Therefore, the minimal path length in a trie — starting counting with the root — in a complete path is 2 except in the case of closed operators, where the path length is 1.

As already mentioned, CIGALE is restricted to a first-order and monomorphic type system. What about the more powerful type system we use? Higher-order types pose no problems in the trie. But how are the polymorphic types of polymorphic operators stored, including their bound type variables? How do we split polymorphic types into the sources and the target such that we can store them within the trie?

The following fact helps. A mixfix operator is always introduced with an explicit declaration. Thus, its type does not contain free type variables because they are all — at least implicitly — bound by outer $\forall$-quantifications. Given a set of type variables $\Delta = \{\alpha_0, \alpha_1, \alpha_2, \ldots\}$, we can instantiate in the mixfix operator's type all the type variables bound by those outer $\forall$-quantifications, split it into its sources and its target and insert the mixfix operator as a monomorphic one.

With the above proceeding, we may not use the variables of Δ outside the trie. We can then assume that the occurring free type variables within the trie are all from Δ and are not used outside.

An instance of our trie data structure can be found in Figure 6.3. It is induced by the following, rather artificial, context, containing a bracketing polymorphic operator, a sequence operator, and a polymorphic binding operator (which is per definition higher-order):

$$\begin{array}{rcl} \mathsf{if\_then\_else\_} & : & \forall\,\alpha\,.\,\forall\,\beta\,.\,\mathbb{B}\times\alpha\times\alpha\to\alpha\,, \\ \_=\cdots=\_ & : & \mathbb{B}^*\to\mathbb{B}\,, \\ \forall\_\ldots\_ & : & \forall\,\alpha\,.\,(\alpha\to\mathsf{Prop})\to\mathsf{Prop} \end{array}$$

Note that the type variable $\alpha_0 \in \Delta$ occurs in two different paths, but it must nevertheless be identified and treated as different because it is implicitly $\forall$-quantified as discussed above.

We represent all possible attributes occurring in a context Σ through a filtering function:

$$\mathit{attributes} \;:\; \mathit{Ident}_{\mathit{Type}} \to \mathbb{N} \to \mathit{Ident}_{\mathit{Type}} \to \mathbb{B}$$

The truth of *attributes* id_T i $id'_{T'}$ means that the mixfix operator $id'_{T'}$ may arise in the subapplication at argument position n in an application of operator id_T. That is to say, the term

$$id_T\,(a_1,\ldots,a_{i-1},\,id'_{T'}(b_1,\ldots,b_m),\,a_{i+1},\ldots,a_n)$$

may be generated by the parser. Otherwise, the construction of such a term is forbidden to prevent ambiguities.

6.3.3 The Parser

As in the case of CIGALE, the trie data structure defined in the preceding section suggests a parsing algorithm that walks through the paths of the trie, thereby reducing identifiers and separators in the input stream and calling the algorithm recursively in the case of the various sources. But the presence of polymorphic types, (undeclared) bound variables, and visible and invisible binders complicate matters.

How has type inference in our polymorphic environment to proceed in comparison to CIGALE's simple monomorphic type derivation and to polymorphic type inference such as Milner's W algorithm?

CIGALE does not use any generalization (rule $(\forall I)$ in Figure 5.1) and instantiation (rule $(\forall E)$) steps in type inference because there are no type variables! We, on the other hand, have type variables, which can be instantiated by suitable type values. Hence, we must keep track of type variable bindings. As in algorithm W, this is done by using substitutions

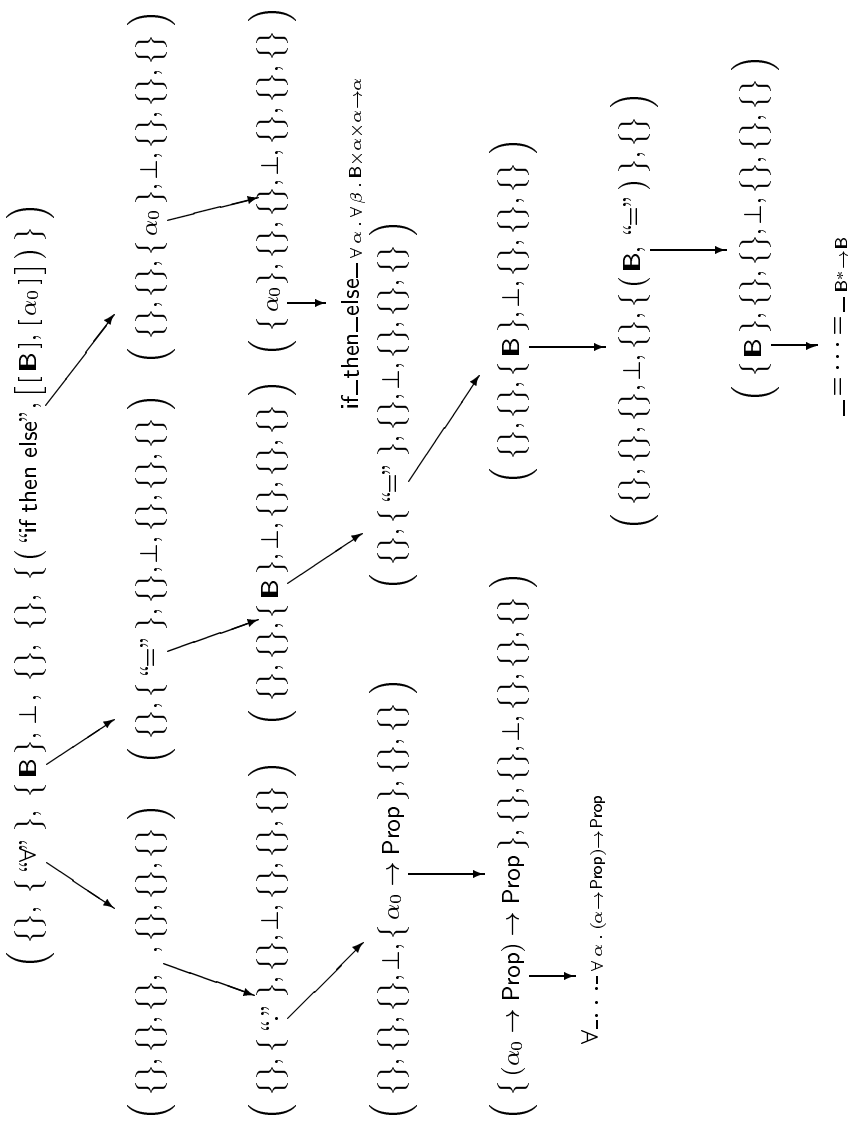

Figure 6.3: A trie containing higher-order and polymorphic mixfix operators.

that are delivered by type unification and capture the bindings. We denote the identity substitution $(x \mapsto x)$ by idsubst.

There is, however, a difference to W and similar algorithms. They all work on abstract syntax trees and thus already know operator and operands in a function application, whereas we want to *create* (well-typed) abstract syntax trees and do not have all this information! While walking on a trie path, we do not know which operator we will finally get, nor its complete type. We only have a typed abstract syntax tree for the argument and an expected type in the source component of a trie. But this information is, in fact, enough. Therefore, we assume the following function lift:

$$\mathsf{lift} \quad : \quad \mathit{Context} \rightarrow \mathit{Type} \times \mathit{Term} \times \mathit{Type} \rightarrow \mathsf{Set}\big(\mathit{Term} \times (\mathit{Type} \rightarrow \mathit{Type})\big)$$

Given a source type A and a term t with its type T, lift $\Sigma\ (A, t, T)$ results in pairs of a term t' and a substitution S such that $S(A) = S(T)$ and t' has type $S(T)$. This may be done by instantiating type variables in A and T, respectively, or by coercing the term t to $S(A)$. Therefore, the current context Σ has to be given as an argument to the routine, too. In other words, lift searches for common minimal upper bounds of A and T with respect to $\preceq_\Sigma$ (cf 3.3.1). If there are none, the empty set is returned.

Furthermore, we assume that lift respects the parsing rules of Figure 5.1. Given $(t', S) \in$ lift $\Sigma\ (A, t, T)$, we require $\Sigma \rhd \Gamma \vdash t : T \ \rightarrow \ \Sigma \rhd \Gamma \vdash t' : S(T)$.

Generalizing a type is done by the following closure operation, which we also assume. It $\forall$-quantifies the type variables occurring free in the type that are *not* free in the passed context (contexts possess the type *Context*):

$$\mathsf{close} \quad : \quad \mathit{Context} \rightarrow \mathit{Type} \rightarrow \mathit{Type}$$

We now describe the parsing algorithm. Its formal definition is shown in the Appendix A. Some of the abbreviations that occur there are defined below.

The only routines visible outside are parse and parse-as:

$$\begin{aligned} \mathsf{parse} &: \mathit{Context} \to C^* \to \mathsf{Set}(\mathit{Term} \times \mathit{Type}) \\ \mathsf{parse\text{-}as} &: \mathit{Context} \to \mathit{Type} \to C^* \to \mathsf{Set}(\mathit{Term}) \end{aligned}$$

All other functions are local to parse and have access to its local variables.

The routine parse takes a context and a string to be parsed as input. The output is a set of terms with their types. Additionally, parse-as gets a type T as a further argument, allowing the routine to search for terms of this given type only and filtering out the others. Hence, the output is a set of terms of type T. The implementation of parse-as simply calls parse and filters the resulting set. Therefore, types are not used in the parsing process to navigate within the trie, but only for type checking arguments (as in CIGALE).

The definition of parse is a bit more complicated. It builds the scan table, the root trie, and the attribute filter from the global context Σ. It then calls the scanner whose output is passed to the routine start. This returns a set of output states, where a *OState* is defined as follows:

$$\begin{aligned} \mathit{OState} = \quad & \mathit{Term} \\ & \times \mathit{Type} \\ & \times (\mathit{Ident}_{\mathit{Type}})_\perp \\ & \times \mathsf{Set}(\mathit{Ident}) \\ & \times \mathsf{Set}(\mathit{Ident}) \\ & \times \mathit{Context} \\ & \times \mathit{SToken}^* \end{aligned}$$

The individual components are:

- An abstract syntax tree representing the already parsed prefix of the input.
- The type of the term.
- The optional type annotated identifier signals that the term is an application with this identifier as the operator of the application. This is used for filtering out wrong applications to prevent ambiguities.

- A set of identifiers that contains all *newly-introduced* bound variables within the above term and which are not (yet) bound by a binder.
- Analogously for free variables. They can arise exclusively in the global context.

  A general invariant is that this and the above identifier sets are always kept disjoint by the parser. Thus, a variable appears in a term either as a free occurring bound variable — being bound later by a binder — or as a free variable coming from the global context, but not as both.
- The local type context of bound variables and their — until now — induced type.
- The rest of the input.

Parse filters away those output states that do not correspond to a complete parse of the input. Hence, all remaining input of states is empty. They are further divided into the set O_c of terms that contain no unbound variables and the set O_f of terms with free-occurring bound variables. If there are elements in O_c we are done and return O_c. Otherwise, a special coercion is searched within the global context — expressed by the awkward syntax "$T \in \Sigma.(\_)$" — which might be suitable to bind all the free occurring bound variables.

The routine start is the initiator for the trie walking algorithm walk. In addition, it checks if the first token in the input is a number or an identifier (remember that neither bound nor free variables are stored in the trie but exclusively in the context).

If it is a token but not in *UIdent*, it may be either a free variable or a *declared* bound variable. Hence its type is looked up in the corresponding context, and all possible terms are generated.

If it could be an undeclared identifier, then the terms are constructed depending on its presence in the sets of bound and free variables, respectively. If it is in neither of the two sets, both possibilities have to be generated, because we do not know beforehand if it will be treated as a bound or a free variable in the future. In this step, a new type variable is introduced in the bound variable case.

The routine walk does the effective trie traversal. It needs the actual position in the trie and the following path state, which is accumulated

during walking on a single path:

$$
\begin{aligned}
\mathit{PState} \quad = \quad & \mathit{Type} \to \mathit{Type} \\
& \times \mathit{Ident}^* \\
& \times \Big(\mathit{Term} \times (\mathit{Ident}_{\,\mathit{Type}})_\perp \; + \\
& \quad \big(\mathit{Type} \times \mathit{Term} \times \mathit{Type} \times \mathsf{Set}(\mathit{Ident}) \times \mathsf{Set}(\mathit{Ident}) \times \mathit{Context} \big) \\
& \quad \times (\mathit{Ident}_{\,\mathit{Type}})_\perp \; + \\
& \quad \big(\mathit{Term} \times (\mathit{Ident}_{\,\mathit{Type}})_\perp \big)^* \Big)^*
\end{aligned}
$$

The three components of the triple are:

- a substitution whose domain is a subset of Δ and which is updated for each passed source,
- a list of identifiers representing bound variables that are detected by passing a left placeholder, and
- a list of type-checked and type-unchecked terms, that are recognized as possible arguments of the currently parsed application. As in *OState*, the optional type annotated identifier stores the operator if the term is an application. This identifier will be used for filtering out invalid applications (depending on the attributes).

  Type-checked terms arise from passing normal sources or sequence nodes. They are type correct with respect to the source's types and need no further handling, except checking the positional exclusion attribute. Type-checked terms also arise for right placeholders — i.e., sources that may contain bound variables and will be converted to λ-terms — if we already passed a left placeholder and extracted the bound variables.

  If we have not yet been able to extract the bound variables, because their place of introduction is after their use or there are not any, we have to process right placeholders differently. We must wait until we reach the end of the path (signalled by targets). Only there can we type check the argument term! Therefore, all the needed information — namely the right source's type, the unchecked term with its type, its bound and free variables, and its local context — is packed together, stored in the argument list and processed when we reach the end of the path.

For constructing the output state, walk carries along an input state, which is accumulated while walking through all paths required by the input:

$$\begin{aligned} \mathit{IState} \quad = \quad & \mathsf{Set}(\mathit{Ident}) \\ & \times \mathsf{Set}(\mathit{Ident}) \\ & \times \mathit{Context} \\ & \times \mathit{SToken}^* \end{aligned}$$

- A set of bound variables that occur in all the above argument terms.
- The same for free variables. As with *Ostate* elements, the occurring free and bound variables are always kept disjoint.
- The local context where detected bound variables are carried along.
- The rest of the input.

Within walk, the minimal length of a path in a given set of tries is denoted by min-path. Furthermore, the routine extract-ids extracts the bound variables of the input stream in the case of a left placeholder.

The routine walk tries all possible paths that are compatible with the given input. If targets are reached in the trie, the corresponding abstract syntax trees are generated. In the case of compatible separators or structured tokens, the input stream is reduced accordingly. Structured tokens are reduced recursively by the local routine walk'. Source, right source and sequence operators collect argument terms and thus need a recursive call of the routine start. The recursion depth of these calls is limited by the minimal lengths of remaining paths in the trie and length of the remaining node list, respectively.

Note the use of continuation passing style for some of walk's local routines.

The main properties of the algorithm are (without proofs):

- It halts on every input.
- It is sound with respect to the parsing rules of Figures 5.2 and 5.3:

$$(t, T) \in \mathsf{parse}\ \Sigma\ s \quad \rightarrow \quad \Sigma \rhd \Gamma \vdash s \Rightarrow t : T$$

Note that the algorithm cannot be complete — a logical implication from right to left in the above formula — because this would contradict Well's undecidable result.

- The algorithm is at least as powerful as CIGALE in case of first-order monomorphic types because all possible paths for an input are traversed.

Chapter 7

Implementation Notes

The chosen language for converting the design of the parsing system into a runnable prototype is SML in its newest revision, called SML '97 [HMT]. We used the interactive compiler SML of New Jersey [Sml97], a public-domain product of Bell Laboratories and Princeton University, which produces rather fast executables due to modern compilation and runtime techniques.

In the next section, we explain the advantages of choosing SML as our implementation language and what novelties of SML '97 are especially relevant for us.

We assume a general knowledge of SML and of SML '97's new basis and standard libraries.

7.1 Why Standard ML '97?

SML's (non-pure) functional core allows a highly parameterized software construction in the small, by passing functions to other functions, returning possibly local functions, and by currying them (a special form of partial evaluation). Polymorphism supports this parameterized style.

Therefore, we use such higher-order and polymorphic functions excessively in our implementation. The translation of the design into an ordinary procedural or object-oriented language would have been more complex and more error-prone. Furthermore, the recursive data struc-

tures defined by fix-point equations can be expressed in a direct way with SML's data types, by transforming the equations' right-hand sides into a sum of products representation, where the products are supplied with constructors. The constructors can be used for pattern-matching, which was used heavily, too. In conventional languages, one has to use pointers for reflecting recursive data structures and selector functions instead of pattern-matching, complicating the implementation phase further.

Exceptions are the only non-pure feature of SML's core language which we used. In certain cases they allow for more concise algorithm descriptions because error values do not have to be returned as ordinary values, which would otherwise complicate the function's target.

On the other hand, SML's module system leads to a structured and component-based software construction in the large. Its powerful module system — probably the most powerful of the currently available non-experimental programming languages — supports structures, signatures and functors.

Structures are collections of identifier–value bindings, as implementation modules in MODULA-2 or environments in MIT-SCHEME [AS85] [1]. They can be compared with the notion of algebraic structures in mathematics except that the latter are usually represented by tuples. Hence, their construction and elimination is done positionally — the i-element in the tuple — whereas structures are constructed and eliminated by providing component names.

Signatures are type contexts (cf Section 2.2.2) and describe similar structures. In fact, signatures are treated in SML as the types of structures, where a structure can implement more than one signature and a signature has various structures as instances. As structures correspond to algebraic structures in mathematics, signatures can be compared with the mathematical concept of dependent sum types [MH88, Tur91]. MODULA-2 provides signatures in the form of definition modules but unfortunately, as opposed to SML, there is a strict one-to-one relation between definition and implementation modules. SML's signatures are also called specifications although SML does not support any form of properties for describing structures.

Much of the power of SML's module system lies in the presence of functors. These are parameterized structures and allow us to develop

[1] Unfortunately, environments are not included in the official definition of SCHEME [AIB+92].

abstract (generic) code relative to some class of structures. This results in a modular and parameterized style of programming, allowing also separate compilation. Functors can be compared with signature interpretations (signature morphisms) viewed as actions on models; they arise in logic and algebraic specification [Gog91, G$^+$92].

Despite some syntactical inelegance and other slight drawbacks [App92], we regard SML as one of the most powerful industrial-strength programming languages available today.

We also appreciated some of the novelties introduced in SML '97. The most important is the new basis and standard libraries of various important algorithms and data structures. They are both now standardized, leading to better portability between different SML platforms.

The complete removal of imperative type variables in SML '97 eliminated all imperative type variables occurring in some signatures of previous versions of the standard library. Furthermore, the now permitted type abbreviations in signatures enable a better control of type sharing assertions in some of our signatures.

From the base environment and the basis library of SML '97, we use the types `bool`, `char`, `int`, `string`, and the type constructors `list` and `vectors`, and, of course, the functions operating on them. Furthermore, the structure `ListPair` is used, which provides operations for pairs of lists.

From the standard library the following components are in use:

- The structure `Atom` supports a fast equality operation for strings by hashing them.
- The signature `ORD_KEY` defines linearly ordered types.
- The signature `ORD_MAP` defines structures of finite maps (dictionaries) over such an ordered type.
- Analogously, the signature `ORD_SET` defines finite sets over an ordered type.
- The functor `BinaryMapFn` takes as argument an arbitrary ordered type and produces as result a finite map, implemented as a binary tree. This can be expressed by `BinaryMapFn` : `ORD_KEY` $\rightarrow$

`ORD_MAP`. [2]

- Similarly, the functor `BinarySetFn` implements finite sets over an arbitrary ordered type by using binary trees, too, i.e. we have `BinarySetFn` : `ORD_KEY` → `ORD_SET`.

7.2 Modularization

This section discusses how the whole parsing system is modularized. Figure 7.1 shows the use relation between the individual signatures. A straight arrow between signatures S and S' means that S occurs free in S', hence, S' depends on S.

The dotted arrow between the structure `Token` and the signatures `IDENT` and `SCANNER` means that `Token` occurs free in these signatures. This breaks the signature closure rule [Pau91, p. 247]. But tokens are so fundamental for us that we take them as pervasive, i.e. globally-defined.

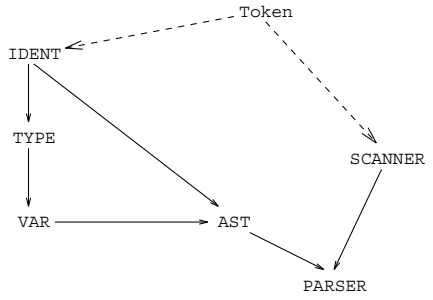

Figure 7.1: Use relation between signatures.

The individual signatures denote the following class of structures:

- `AST` specifies abstract syntax trees in the sense of *Term* (cf 5.6.1). Because certain abstract syntax trees describe types, and we want to convert such abstract syntax trees into real types, `AST` has to depend on `TYPE`.

[2] This is not a valid SML declaration because functors are not first-class citizens in SML and thus do not get an assigned type.

- `IDENT` specifies identifiers.
- `PARSER` describes what a concrete parse structure has to offer, namely context management and the actual parsing routine. Because the input to the parser is a structured token list produced by a scanner and the result of the parser is an abstract syntax tree, `PARSER` depends on `SCANNER` and `AST`.
- `SCANNER` describes abstractly a scanner that, given a string, produces a structured token list.
- `TYPE` specifies *Type*, the language of types (cf 5.6.1).
- `VAR` specifies overloaded variables as type annotated identifiers; hence it depends on `TYPE`.

Figure 7.2 shows the functorial dependencies between signatures. An arrow between the signatures S and S' means that there is a functor, mapping structures of S into structures of S'. The functors themselves are described in the next section.

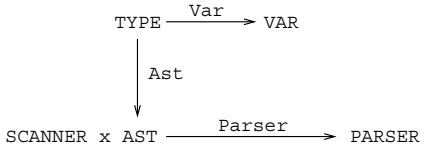

Figure 7.2: Functorial dependencies between signatures.

7.3 The modules

In this section we discuss the individual (parameterized) structures.

- Given a `Term` structure, the functor `Ast` : `TYPE` $\rightarrow$ `AST` constructs a structure for abstract syntax trees, using a recursive datatype for the terms.
- The functor `BinaryMapFn` : `ORD_KEY` $\rightarrow$ `ORD_MAP` extends `BinaryMapFn` of the standard library by some needed generic functions.

- The structure `Exceptions` exports some global exceptions.
- The structure `Ident` is an instance of `IDENT`. Implemented on top of tokens, `Ident` offers identifiers.
- The structure `Lib` defines various helper functions.
- The functor `Parser` : `SCANNER` × `AST` → `PARSER` implements the trie data structure as specified in Section 6.3.2 by using `BinaryMapFn` for representing the finite maps, with the extension that the minimal path lengths of sources are stored in the trie itself. The implementation of the parsing method in Section 6.3.3 uses `BinarySetFn` for representing the set of bound and free variables and `BinaryMapFn` for the local context. The parser uses lists instead of sets for the result. The union operations in the walk are replaced by functions that accumulate the result list (instead of simple list concatenation). Due to the use of deBruijn indices [Bar84] in types (cf below), the lifting of source type and argument term, as well as substitutions, is slightly different but more efficient.
- The structure `Scanner` is an instance of `SCANNER`. It implements the bracketing scanner.
- The structure `StringLib` exports predefined character classes and derived character operations.
- The structure `Token`, possessing signature `TOKEN`, implements tokens by using `Atom` from the standard library.
- The structure `Type`, having signature `TYPE`, implements the language *Type* of Section 5.6.1. Instead of explicit bound variables, deBruijn indices are used for the variables bound by ∀-quantification.
- The functor `Var` : `TYPE` → `VAR` implements overloaded variables (analogously to type annotated identifiers).

Chapter 8

Conclusions and Future Work

In this dissertation we presented higher-order mixfix syntax by extending ordinary mixfix syntax. Higher-order mixfix syntax is a meta-formalism to describe and extend the linear part of mathematical notation, and to deal with the various ambiguities and context sensitivities present in mathematical syntax. It includes arbitrary binding operators, finite sequence constructions, arbitrary syntax also for type constructors, and a new operator attribute for refined control of syntactical ambiguities. Furthermore, we interpreted it in a higher-order abstract syntax, using a polymorphic λ-calculus in Curry style.

On the algorithmic side, we combined parsing with Hindley–Milner style type inference for higher-order mixfix syntax into one process. The parsing process is fully parameterized by an arbitrary context. A bracketing scanner helped to reduce the input size by prestructuring the input. The resulting functional designs were translated into a prototype implementation written in the functional language Standard ML.

What main conclusions can we draw?

Mathematical notation is able to compress a lot of semantical information into few syntactical constructs. Besides the mentioned context dependence, this is done by using short identifiers — especially in the case of bound variables — built from a rich base alphabet, through the use of juxtaposition (often without any whitespaces), coercions, binding

operators, and various fixities for the operators. A meta-formalism for mathematical notation such as higher-order mixfix syntax must be capable of dealing with all these nifty details. Therefore, higher-order mixfix syntax is not restricted to use in computers but could also be used as a universal formalism for describing mathematical syntax — often in a notation section — in any kind of publication. Nevertheless, its main use is still for computer-based applications. In fact, one purpose of our work is to support mathematical notation as a programming language as was tried by Révész and Lynch [RL91]. Their approach was limited by their use of a context-free grammar for describing mathematical syntax.

We further demonstrated that the use of a context, especially the use of the types of occurring identifiers, for disambiguating mathematical expressions is essential and necessary for such a context-dependent language as mathematical notation. However, even classical disambiguating methods, such as priorities and implicit associativities, have to depend on the context. Hence, our parsing system, which accepts languages induced by higher-order mixfix syntax, is parameterized by contexts.

To partly remedy the effort from exhaustive search in the trie, we showed the usefulness of a scanner which structures the token stream given to the parser according to the special syntactical structure of bracketing operators. To prevent, or at least to reduce, the creation of abstract syntax trees that are recognized as invalid during the subsequent type-inference pass, it is important to integrate the type-inference process into the parser itself, which is the case in our algorithm.

Possible applications of higher-order mixfix syntax and its parsing system are any kind of front-ends that deal with some kind of typed mathematical input, such as computer-algebra systems, theorem-proving environments, or functional, logic and specification languages. Typical interactive input consists of a few lines in such applications.

The integration of our parsing method into interactive and two-dimensional parsing systems in the style of Soiffer's or Zhao's work [Soi95, Zha96] seems very promising for the future. This would be a great support for mathematicians who are not used to computers and who do not need to learn inadequate notations to express their thoughts.

Parsing in a parser-based interactive system means that after each keystroke the longest possible prefix of the actual input is transformed into an abstract syntax tree, which can be displayed immediately, nicely formatted. This leads to interesting modifications of the trie passing

algorithm to reuse already parsed prefixes. Furthermore, the context-sensitive attributes lead to finer control of typesetted output.

The extension of higher-order mixfix syntax with two-dimensional fixities requires surprisingly few modifications to the parser because two-dimensional input is already structured by the user, for example by using Zhao's formalization method [Zha96, p. 19–23]. The linear representation of the two-dimensional structure is similar to the structured token stream produced by our bracketing scanner, and so the same method used by the parser for translating structured tokens also applies to two-dimensional input. The required extension to the declaration syntax would be straightforward if an order on the placeholders is defined, say from left to right and then from top to bottom, such that the operator's sources can be distributed uniquely to the placeholders. For example, definite integration and the summation operator can then be declared as two-dimensional binding operators:

$$\int_{\_}^{\_} \_ \,\mathsf{d}\_ . \quad : \quad \mathbb{R} \times (\mathbb{R} \to \mathbb{R}) \times \mathbb{R} \;\to\; \mathbb{R}$$

$$\sum_{\_ \cdot = \_}^{\_} \_ \quad : \quad \mathbb{Z} \times (\mathbb{Z} \to \mathbb{Z}) \times \mathbb{Z} \;\to\; \mathbb{Z}$$

Higher-order mixfix syntax can be elegantly combined with partial evaluation[1], where the unevaluated argument positions are marked with placeholders. E.g., the successor function as partial evaluated addition is written $\_ + 1$. The extreme case of partial evaluation — where no arguments are given — fits smoothly in this framework, because it has the same syntactical appearance as writing the operator itself, correctly reflecting semantics. Compare this approach with HASKELL's sections, which serve the same purpose, but can lead to ambiguities.

Interesting aspects arise by extending the type system underlying higher-order mixfix syntax with new constructs.

Vector and matrix types often occur in mathematics but also need variables in types that denote natural numbers and not types. Say, a vector type over an arbitrary type α and an arbitrary dimension n can be written as $\forall\, \alpha. \forall\, n. \alpha^n$. Because vector types have a finer type structure than

[1] Partial evaluation is an application where not all the arguments are given, resulting semantically in an anonymous function [EL92]. Currying is a special case of partial evaluation.

lists — length information is coded in the type — sequence operators could profit by using vector types instead of list types.

Adding dependent product and dependent sum types (corresponding to signatures) to the type system enables the representation of (parameterized) mathematical structures. But the parser then needs access to the expression evaluator because types have to be β-reduced. This also requires a more powerful λ-calculus for interpreting higher-order mixfix syntax, say λP or $\lambda\omega$ [BH90, Sch94].

On the algorithmic side, one can investigate if, and how much, the parsing algorithm can profit by taking advantage of expected source types for navigating within the trie. Similarly, the set of tokens an expression of a given type can start with could be helpful (analogously to the set of first symbols of a non-terminal in a recursive descent parser [ASU86]).

Appendix A

The Parsing Algorithm

The following pages contain the functional description of the parsing algorithm.

$$\mathsf{parse} \;:\; \mathit{Context} \;\to\; C^* \;\to\; \mathsf{Set}(\mathit{Term} \times \mathit{Type})$$

$$\begin{array}{l}
\mathsf{parse}\ \Sigma\ s\ = \\
\quad \textbf{let} \\
\qquad (\mathit{scantable}, \mathit{root}, \mathit{attributes}) \;=\; \mathsf{extract}\ \Sigma \\
\qquad O \;= \bigcup \left\{ \mathsf{start}\left(\{\},\{\},[\,],\mathit{tl}\right) \;\middle|\; \mathit{tl} \in \mathsf{scan}\ \mathit{scantable}\ \mathit{ts} \right\} \\
\qquad O_c = \left\{ (t,\mathit{ty}) \;\middle|\; (t,\mathit{ty},\mathit{oid},\{\},F,\Gamma,\varepsilon) \in O \right\} \\
\qquad O_f = \left\{ (t,\mathit{ty},B,\Gamma) \;\middle|\; (t,\mathit{ty},\mathit{oid},B,F,\Gamma,\varepsilon) \in O,\, B \neq \{\} \right\} \\
\quad \textbf{in} \\
\qquad \textbf{if}\ \ O_c \neq \{\}\ \textbf{then} \\
\qquad\quad O_c \\
\qquad \textbf{else} \\
\qquad\quad \left\{ \left(\left(t'(\lambda\, x_1,\ldots,x_n\,.\,t)\right),\, S'(\alpha)\right) \;\middle|\; \begin{array}{l} \left(t,\mathit{ty},\{x_1,\ldots,x_n\},\Gamma\right) \in O_f, \\ T \in \Sigma.(_), \\ \mathit{ty}_i \in \Gamma.x_i \;\; (1 \le i \le n), \\ \alpha \text{ fresh type variable}, \\ (t',S') \in \mathsf{lift}\ \Sigma\ \left(\left(\mathsf{close}\ [\,]\ (\mathit{ty}_1 \times \cdots \times \mathit{ty}_n \to \mathit{ty})\right) \;\to\; \alpha,\, _,\, T\right) \end{array} \right\}
\end{array}$$

$$\mathsf{parse\text{-}as} \;:\; \mathit{Context} \;\to\; \mathit{Type} \;\to\; C^* \;\to\; \mathsf{Set}(\mathit{Term})$$

$$\mathsf{parse\text{-}as}\ \Sigma\ \mathit{ty}\ s \;=\; \left\{ t \;\middle|\; (t,\mathit{ty}) \in \mathsf{parse}\ \Sigma\ s \right\}$$

$$
\begin{array}{l}
\mathsf{start}\ :\ \mathit{IState}\ \rightarrow\ \mathsf{Set}(\mathit{OState}) \\
\mathsf{start}\ \big(B,F,\Gamma,\varepsilon\big)\ \ = \big\{\ \big\} \\
\mathsf{start}\ \big(B,F,\Gamma,t_1\ tl\big) = \\
\quad \mathit{conv}(t_1)\ \cup\ \mathsf{walk}\ \mathit{root}\ \big(\mathsf{idsubst},[\,],[\,]\big)\ \big(B,F,\Gamma,(t_1\ tl)\big) \\
\quad \textbf{where} \\
\qquad \begin{array}{lcll}
\mathit{conv}\ \mathsf{inj}_1\big(id,\mathsf{false}\big) & = & \big\{\ \big(id_{ty},ty,\bot,\{\},\{\},[\,],tl\big)\ \big|\ ty\in\Sigma.id\ \big\}\ \cup & \\
\mathit{conv}\ \mathsf{inj}_1\big(id,\mathsf{true}\big) & = & \big\{\ \big(id_{ty},ty,\bot,\{\},\{\},[\,],tl\big)\ \big|\ ty\in\Sigma.id\ \big\}, & \textbf{if}\ id\in F \\
\mathit{conv}\ \mathsf{inj}_1\big(id,\mathsf{true}\big) & = & \big\{\ \big(id,ty,\bot,\{\},\{\},id:ty,tl\big)\ \big|\ ty\in\Gamma.id\ \big\}, & \textbf{if}\ id\in B\wedge id\in\mathsf{dom}\ \Gamma \\
\mathit{conv}\ \mathsf{inj}_1\big(id,\mathsf{true}\big) & = & \big\{\ \big(id,\alpha,\bot,\{id\},\{\},id:\alpha,tl\big)\ \big|\ \alpha\ \text{fresh type variable}\ \big\}, & \textbf{if}\ id\in B\wedge id\notin\mathsf{dom}\ \Gamma \\
\mathit{conv}\ \mathsf{inj}_1\big(id,\mathsf{true}\big) & = & \big\{\ \big(id,\alpha,\bot,\{id\},\{\},id:\alpha,tl\big)\ \big|\ \alpha\ \text{fresh type variable}\ \big\}\ \cup & \\
 & & \big\{\ \big(id,ty,\bot,\{\},\{\},id:ty,tl\big)\ \big|\ ty\in\Gamma.id\ \big\}, & \textbf{if}\ id\notin B\wedge id\in\mathsf{dom}\ \Gamma \\
\mathit{conv}\ \mathsf{inj}_1\big(id,\mathsf{true}\big) & = & \big\{\ \big(id_{ty},ty,\bot,\{\},\{id\},[\,],tl\big)\ \big|\ ty\in\Sigma.id\ \big\}\ \cup & \\
 & & \big\{\ \big(id,\alpha,\bot,\{id\},\{\},id:\alpha,tl\big)\ \big|\ \alpha\ \text{fresh type variable}\ \big\} & \\
\mathit{conv}\ \mathsf{inj}_2(n) & = & \big\{\ \big(n,ty,\bot,\{\},\{\},[\,],tl\big)\ \big\} & \\
\mathit{conv}\ t & = & \big\{\ \big\} &
\end{array}
\end{array}
$$

$$\mathsf{walk} \;:\; \mathit{Trie} \;\to\; \mathit{PState} \;\to\; \mathit{IState} \;\to\; \mathsf{Set}(\mathit{OState})$$

$$\mathsf{walk}\ \big(tm, sepm, srcm, lsrco, rsrcm, seqm, strum\big)\ \big(S, \overbrace{[\,x_1, \ldots, x_m\,]}^{idl}, pl\big)\ \big(B, F, \Gamma, \overbrace{t_1 \cdots t_n}^{tl}\big) =$$

$$\begin{array}{l}
\quad \textbf{let}\ \mathit{targets} \;=\; \\
\qquad \bigcup_{ty' \in \mathsf{dom}(tm)} \Big\{ \big(t, ty'', id_{ty}, B', F', \Gamma', t_1\ tl\big) \;\Big|\; \big(S', B', F', \Gamma', [\,arg_1, \ldots, arg_n\,]\big) \in \mathit{filter}\ \big(1, S, B, F, \Gamma, pl\big), \\
\qquad\qquad\qquad\qquad arg \in \mathit{Term},\ n{=}1 \to arg{=}arg_1,\ n{>}1 \to arg{=}\big(arg_1, \ldots, arg_n\big), \\
\qquad\qquad\qquad\qquad id \in B' \to t = (id\ \ arg),\ id \notin B' \to t = (id_{ty}\ \ arg), \\
\qquad\qquad\qquad\qquad ty'' = \mathsf{close}\ \Gamma'\ \big(S'(ty')\big)\ \Big\} \\
\quad \textbf{where} \\
\qquad id_{ty} \;=\; tm(ty') \\
\qquad \mathit{filter}\ \big(n, S, B, F, \Gamma, [\,]\big) \;=\; \big\{\, \big(S, B, F, \Gamma, [\,]\big)\,\big\} \\
\qquad \mathit{filter}\ \big(n, S, B, F, \Gamma, \mathsf{inj}_1(t^*, oid)\, pl\big) \;= \\
\qquad\quad \Big\{ \big(S', B', F', \Gamma', t^*\ l'\big) \;\Big|\; id'_{ty'} = oid \to \mathit{attributes}\ id_{ty}\ n\ id'_{ty'}, \\
\qquad\qquad\qquad \big(S', B', F', \Gamma', l'\big) \in \mathit{filter}\ \big(n+1, S, B, F, \Gamma, pl\big)\ \Big\} \\
\qquad \mathit{filter}\ \big(n, S, B, F, \Gamma, \big(\mathsf{inj}_2(ty_1^*, t^*, ty^*, B^*, F^*, \Gamma^*), oid\big)\, pl\big) \;= \\
\qquad\quad \Big\{ \big(S', B', F', \Gamma'|\mathsf{dom}(\Gamma') - \{y_1, \ldots, y_k\}, t''\ l\big) \;\Big| \\
\qquad\qquad\qquad id''_{ty''} = oid \to \mathit{attributes}\ id_{ty}\ n\ id''_{ty''}, \\
\qquad\qquad\qquad idl = [\,] \to \{y_1, \ldots, y_k\} = B^*,\ idl \neq [\,] \to (y_1, \ldots, y_k) = (x_1, \ldots, x_k), \\
\qquad\qquad\qquad \big(B^* - \{y_1, \ldots, y_k\}\big) \cap F = \{\},\ B \cap F^* = \{\}, \\
\qquad\qquad\qquad y_i \in \mathsf{dom}(\Gamma^*) \to y_i \in B^*,\ \ ty_i \in \Gamma.y_i \quad (1 \le i \le k), \\
\qquad\qquad\qquad (t'', S'') \in \mathsf{lift}\ \big(S(ty_1^*), \big(\lambda y_1, \ldots, y_k\,.\, t^*\big), \\
\qquad\qquad\qquad\qquad\qquad \mathsf{close}\ (\Sigma, \Gamma)\big(ty_1 \times \cdots \times ty_k \to ty^*\big)\big), \\
\qquad\qquad\qquad \big(S', B', F', \Gamma', l\big) \in \mathit{filter}\ \big(n+1,\ S''|(\Delta \cap \mathsf{dom}(S'')), \\
\qquad\qquad\qquad\qquad\qquad\qquad (B^* - \{y_1, \ldots, y_k\}) \cup B,\ F^* \cup F,\ \big(S''(\Gamma), \Gamma^*\big),\ pl\big)\ \Big\} \\
\qquad \mathit{filter}\ \big(n, S, B, F, \Gamma, \mathsf{inj}_3(pl')\, pl\big) \;= \\
\qquad\quad \Big\{ \big(S', B', F', \Gamma', [\,l', arg_1, \ldots, arg_m\,]\big) \;\Big| \\
\qquad\qquad\qquad \big(S', B', F', \Gamma', l'\big) \in \mathit{filter}\ \big(n+1, S, B, F, \Gamma, pl'\big), \\
\qquad\qquad\qquad \big(S'', B'', F'', \Gamma'', [arg_1, \ldots, arg_m]\big) \in \mathit{filter}\ \big(n + |pl'| + 1, S', B', F', \Gamma', pl\big)\ \Big\}
\end{array}$$

$$
\begin{array}{l}
case\ \left(\mathsf{inj}_1\ tok\right) \quad = \ \mathsf{walk}\ sepm(tok)\ \left(S, idl, pl\right)\ \left(B, F, \Gamma, t_2 \cdots t_n\right), \qquad \textbf{if}\ tok \in \mathsf{dom}(sepm) \\
case\ \left(\mathsf{inj}_3\ (tok, tll)\right) \ = \bigcup_{(tok, nll) \in \mathsf{dom}(strum)} stru\left(\mathsf{walk}\ strum(tok, nll)\right)\ nll\ \left(S, idl, pl\right)\ \left(B, F, \Gamma, t_2 \cdots t_n\right) \\
case\ t \quad = \ \{\ \} \\
src\ cont\ ty\ n'\ \left(S, idl, pl\right)\ \left(B, F, \Gamma, t_1 \cdots t_n\right) \quad = \\
\quad \bigcup \ \{\ cont\ \left(S''|(\Delta \cap \mathsf{dom}(S'')), idl, pl\left(\mathsf{inj}_1\ (t'', oid)\right)\right)\ \left(B \cup B', F \cup F', \left(S''(\Gamma), \Gamma'\right), tl'(t_{n'+1} \cdots t_n)\right)\ | \\
\qquad \left(t', ty', oid, B', F', \Gamma', tl'\right) \in \mathsf{start}\ \left(B, F, \Gamma, t_1 \cdots t_{n'}\right), \\
\qquad \left(t'', S''\right) \in \mathsf{lift}\ \left(\Sigma, \Gamma\right)\ \left(S(ty), t', ty'\right)\ \} \\
rsrc\ cont\ ty\ n'\ \left(S, idl, pl\right)\ \left(B, F, \Gamma, t_1 \cdots t_n\right) \quad = \\
\quad \bigcup \ \{\ cont\ \left(S, idl, pl\left(\mathsf{inj}_2\ ((ty, t', ty', B', F', \Gamma'), oid)\right)\right)\ \left(B, F, \Gamma, tl'(t_{n'+1} \cdots t_n)\right)\ | \\
\qquad \left(t', ty', oid, B', F', \Gamma', tl'\right) \in \mathsf{start}\ \left(\{\}, \{\}, \Gamma, t_1 \cdots t_{n'}\right)\ \} \\
seq\ cont\ ty\ \bot\ m\ \overbrace{\left(S, idl, [p_1, \ldots, p_m]pl\right)}^{pstate}\ \overbrace{\left(B, F, \Gamma, t_1 \cdots t_n\right)}^{istate} \quad = \\
\quad cont\ \left(S, idl, \mathsf{inj}_3\left([p_1, \ldots, p_m]\right)\ pl\right)\ istate\right) \cup \\
\quad src\ \left(seq\ cont\ ty\ otok\ (m+1)\right)\ ty\ n\ pstate\ istate \\
seq\ cont\ ty\ t_1\ m\ \overbrace{\left(S, idl, [p_1, \ldots, p_m]pl\right)}^{pstate}\ \overbrace{\left(B, F, \Gamma, t_1 \cdots t_n\right)}^{istate} \quad = \\
\quad cont\ \left(S, idl, \mathsf{inj}_3\left([p_1, \ldots, p_m]\right)\ pl\right)\ istate\right) \cup \\
\quad src\ \left(seq\ cont\ ty\ otok\ (m+1)\right)\ ty\ n\ pstate\ \overbrace{\left(B, F, \Gamma, t_2 \cdots t_n\right)}^{istate} \\
seq\ cont\ ty\ otok\ m\ \left(S, idl, [p_1, \ldots, p_m]pl\right)\ \overbrace{\left(B, F, \Gamma, t_1 \cdots t_n\right)}^{istate} = \\
\quad cont\ \left(S, idl, \mathsf{inj}_3\left([p_1, \ldots, p_m]\right)\ pl\right)\ istate\right)
\end{array}
$$

$$
\begin{array}{l}
\mathit{stru}\ \mathit{cont}\ [\,]\ \mathit{pstate}\ \mathit{istate} \quad = \ \mathit{cont}\ \mathit{pstate}\ \mathit{istate} \\
\mathit{stru}\ \mathit{cont}\ \big(\mathit{nl}\ \mathit{nll}\big)\ \mathit{pstate}\ \mathit{istate} = \\
\quad \mathit{walk}'\ \big(\mathit{stru}\ \mathit{cont}\ \mathit{nll}\big)\ \mathit{nl}\ \mathit{pstate}\ \mathit{istate} \\
\quad \textbf{where}\ \mathit{walk}' \ : \ \big(\mathit{PState} \to \mathit{IState} \to \mathsf{Set}(\mathit{OState})\big) \to \mathit{Node}^* \to \big(\mathit{PState} \to \mathit{IState} \to \mathsf{Set}(\mathit{OState})\big) \\
\qquad \mathit{walk}'\ \mathit{cont}\ [\,]\ \mathit{pstate}\ \mathit{istate} \qquad\qquad\qquad\qquad = \mathit{cont}\ \mathit{pstate}\ \mathit{istate} \\
\qquad \mathit{walk}'\ \mathit{cont}\ \big(\mathit{ty}\ \mathit{nl}\big)\ \mathit{pstate}\ (B,F,\Gamma,\mathit{tl}\big) \qquad = \mathit{src}\ \big(\mathit{walk}'\ \mathit{cont}\ \mathit{nl}\big)\ \big(|\mathit{tl}|-|\mathit{nl}|+1\big)\ \mathit{pstate}\ \big(B,F,\Gamma,\mathit{tl}\big)\big) \\
\qquad \mathit{walk}'\ \mathit{cont}\ \big((\,)\ \mathit{nl}\big)\ \big(S,\mathit{idl},\mathit{pl}\big)\ \big(B,F,\Gamma,\mathit{tl}\big) = \bigcup\ \big\{\ \mathit{walk}'\ \mathit{cont}\ \mathit{nl}\ \big(S,\mathit{idl}',\mathit{pl}\big)\ \big(B,F,\Gamma,\mathit{tl}'\big)\ \big|\ \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad (\mathit{idl}',\mathit{tl}') \in \mathsf{extract\text{-}ids}(\mathit{tl})\ \big\} \\
\qquad \mathit{walk}'\ \mathit{cont}\ \big(\mathit{ty}\ \mathit{nl}\big)\ \mathit{pstate}\ \big(B,F,\Gamma,\mathit{tl}\big) \qquad = \mathit{rsrc}\ \big(\mathit{walk}'\ \mathit{cont}\ \mathit{nl}\big)\ \big(|\mathit{tl}|-|\mathit{nl}|+1\big)\ \mathit{pstate}\ \big(B,F,\Gamma,\mathit{tl}\big)\big) \\
\qquad \mathit{walk}'\ \mathit{cont}\ \big((\mathit{ty},\mathit{otok})\ \mathit{nl}\big)\ \mathit{pstate}\ \mathit{istate} \qquad = \mathit{seq}\ \big(\mathit{walk}'\ \mathit{cont}\ \mathit{nl}\big)\ \mathit{ty}\ \mathit{otok}\ 3\ \mathit{pstate}\ \mathit{istate} \\
\textbf{in} \\
\quad \textbf{if}\ \ \mathit{tl} = [\,]\ \textbf{then} \\
\qquad \mathit{targets} \\
\quad \textbf{else} \\
\qquad \mathit{targets}\ \cup\ \mathit{case}(t_1) \cup \\
\qquad \big(\textbf{if}\ \mathit{lsrco} \neq \bot\ \textbf{then} \\
\qquad\quad \bigcup\ \big\{\ \mathsf{walk}\ \mathit{lsrco}\ \big(S,\mathit{idl}',\mathit{pl}\big)\ \big(B,F,\Gamma,\mathit{tl}'\big)\ \big|\ (\mathit{idl}',\mathit{tl}') \in \mathsf{extract\text{-}ids}(\mathit{tl})\ \big\} \\
\qquad\ \textbf{else}\ \{\}\big) \cup \\
\qquad\quad \bigcup_{\mathit{ty} \in \mathsf{dom}(\mathit{srcm})} \big(\mathit{src}\ \big(\mathit{walk}\ \mathit{srcm}(\mathit{ty})\big)\ \mathit{ty}\ n'\ \big(S,\mathit{idl},\mathit{pl}\big)\ \big(B,F,\Gamma,\mathit{tl}\big)\big) \cup \\
\qquad\quad \bigcup_{\mathit{ty} \in \mathsf{dom}(\mathit{rsrcm})} \big(\mathit{rsrc}\ \big(\mathit{walk}\ \mathit{rsrcm}(\mathit{ty})\big)\ \mathit{ty}\ n'\ \big(S,\mathit{idl},\mathit{pl}\big)\ \big(B,F,\Gamma,\mathit{tl}\big)\big) \cup \\
\qquad\quad \bigcup_{(\mathit{ty},\mathit{otok}) \in \mathsf{dom}(\mathit{seqm})} \big(\mathit{seq}\ \big(\mathit{walk}\ \mathit{seqm}(\mathit{ty})\big)\ \mathit{ty}\ \mathit{otok}\ 3\ \big(S,\mathit{idl},\mathit{pl}\big)\ \big(B,F,\Gamma,\mathit{tl}\big)\big) \\
\qquad \textbf{where} \\
\qquad\quad n' \ = \ n - \mathsf{min}\big\{\ \mathsf{min\text{-}path}(\mathit{srcm}),\ \mathsf{min\text{-}path}(\mathit{rsrcm}),\ \mathsf{min\text{-}path}(\mathit{seqm})\ \big\} + 1
\end{array}
$$

Bibliography

[AIB+92] H. Abelson, N. I. Adams IV, D. Bartley, G. Brooks, R. K. Dybvig, D. P. Friedman, R. Halstead, C. Hanson, C. T. Haynes, E. Kohlbecker, D. Oxley, K. M. Pitman, G. J. Rozas, G. J. Sussman, and M. Wand. Revised^4 Report on the Algorithmic Language Scheme. Technical report, Artificial Intelligence Laboratory, Massachusetts Institute of Technology (MIT), Cambridge, Massachusetts, November 1992.

[AL91] Andrea Asperti and Giuseppe Longo. *Categories, Types, and Structures.* MIT Press, 1991.

[App92] A. Appel. A critique of Standard ML. Technical report, Princeton University, 1992.

[AS85] H. Abelson and G. J. Sussman. *Structure and Interpretation of Computer Programs.* The MIT Press, Cambridge, Mass., 1985.

[ASU86] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools.* Addison-Wesley, 1986.

[Bac88] R.C. Backhouse. An exploration of the Bird-Meertens formalism. Technical Report CS 8810, Department of Computing Science, University of Groningen, 1988.

[Bac89] R.C. Backhouse. Making formality work for us. *EATCS Bulletin*, 38:219 – 249, jun 1989.

[Bar84] Henk Barendregt. *The Lambda Calculus.* Studies in Logic 103. North Holland, 1984.

[Bar89] John Barwise. *Handbook of Mathematical Logic.* North-Holland Publishing Company, 1989.

[Bau92] Bernhard Bauer. Ein interaktives System für algebraische Implementierungsbeweise. Diplomarbeit, Universität Passau, Fakultät für Mathematik und Informatik, 1992.

[Bau93] Bernhard Bauer. An interactive system for algebraic implementation proofs: The ISAR system from the user's point of view. Technical Report 9313, Ludwig-Maximilians-Universität München, Institut für Informatik, 1993.

[BCC+87] M. Bidoit, F. Capy, C. Choppy, M.-A. Choquer, S. Kaplan, F. Schlienger, and Frédéric Voisin. Asspegique: An integrated specification environment. In *STACS 87, 4th Annual Symposium on Theoretical Aspects of Computer Science*, volume 247 of *Lecture Notes in Computer Science*. Springer-Verlag, 1987.

[BFG+91] M. Broy, Ch. Facchi, R. Grosu, R. Hettler, H. Hußmann, D. Nazareth, R. Regensburger, and K. Stølen. The requirement and design specification language SPECTRUM. Technical Report TUM-I9140, Technische Universität München, Institut für Informatik, October 1991.

[BH90] Henk Barendregt and Kees Hemerik. Types in lambda calculi and programming languages. In N. Jones, editor, *Proceedings 3rd European Symposium on Programming (ESOP '90)*, volume 432 of *Lecture Notes in Computer Science*, pages 1–35, Copenhagen, Denmark, May 1990. Springer-Verlag.

[Bur75] W.H. Burge. *Recursive Programming Techniques.* Addison-Wesley, 1975.

[C+86] R.L. Constable et al. *Implementing Mathematics with the Nuprl Proof Development System.* Prentice-Hall, 1986.

[Caj93] Florian Cajori. *A History of Mathematical Notation.* Dover Publications, 1993.

[Car86] L. Cardelli. A polymorphic λ-calculus with Type:Type. Technical Report 10, DEC Systems Research Center, 1986.

[CC94] Jacques Calmet and John A. Campbell, editors. *Integrating Symbolic Mathematical Computation and Artificial Intelligence (AISMC-2)*, volume 958 of *Lecture Notes in Computer Science*. Springer Verlag, August 1994.

[CG96] Stéphane Collart and Gaston Gonnet. Structured search in mathematical documents. *Euromath Bulletin*, 2:61–74, 1996.

[CGG+91] Bruce W. Char, Keith O. Geddes, Gaston H. Gonnet, L. Leong Benton, Michael B. Monagan, and Stephen M. Watt. *Maple V Language Reference Manual*. Springer-Verlag, New York, 1991.

[Chu40] Alonzo Church. A formulation of the simple theory of types. *Journal of Symbolic Logic*, 5:56–68, 1940.

[Coq86] Thierry Coquand. An analysis of Girard's paradox. In *Proceedings, Symposium on Logic in Computer Science*, pages 227–236, Cambridge, Massachusetts, 16–18 June 1986. IEEE Computer Society.

[Cro93] Roy L. Crole. *Categories for Types*. Cambridge University Press, 1993.

[CW85] L. Cardelli and P. Wegner. On understanding, data abstraction and polymorphism. *ACM Computing Surveys*, 17:471–522, 1985.

[Dri90] Graham C. Driscoll. Ordinary mathematical notation: Some characteristics and their implications for programming. Technical Report RC 15365, IBM Research Division, oct 1990.

[DSW94] Martin Davis, Ron Sigal, and Elaine J. Weyuker. *Computability, Complexity, and Languages*. Academic Press, second edition, 1994.

[EAG+94] E. Engeler, K. Aberer, O. Gloor, M. von Mohrenschildt, D. Otth, G. Schwärzler, and T. Weibel. *The Combinatory Programme*. Birkhäuser, 1994.

[Ear70] J. Earley. An efficient context-free parsing algorithm. *Communications of the ACM*, 13(2):94–102, 1970.

[EGL89] H. Ehrich, M. Gogolla, and U.W. Lipeck. *Algebraische Spezifikation abstrakter Datentypen.* B.G. Teubner Stuttgart, 1989.

[EL92] E. Engeler and P. Läuchli. *Berechnungstheorie für Informatiker.* B. G. Teubner, 1992.

[Eve95] Gilles Everling. Parser for mathematical notation, 1995. Diploma Thesis, ETH Zurich.

[Exn94] Jürgen Exner. The OPAL tutorial. Technical Report 94-9, TU Berlin, May 1994.

[Fok95] Jeroen Fokker. Functional parsers. In *Advanced Functional Programming*, volume 925 of *Lecture Notes in Computer Science*, pages 1–23. Springer Verlag, 1995.

[G+92] Joseph A. Goguen et al. Introducing OBJ. Technical report, SRI International, March 1992.

[GCL92] K. O. Geddes, S. R. Czapor, and G. Labahn. *Algorithms for Computer Algebra.* Kluwer Academic Publishers, 1992.

[GH93] J.V. Guttag and J.J. Horning. *Larch: Languages and Tools for Formal Specification.* Springer, 1993.

[GKT90] Michael Gloger, Stefan Kaes, and Christoph Thies. Entwicklung funktionaler Programme in der SAMPλE Programmierumgebung. Technical Report PI–R3/90, Praktische Informatik, Technische Hochschule Darmstadt, 1990.

[GLT89] Jean-Yves Girard, Yves Lafont, and Paul Taylor. *Proofs and Types*, volume 7 of *Cambridge Tracts in Theoretical Computer Science.* Cambridge University Press, 1989.

[GM92] Joseph A. Goguen and José Meseguer. Order-sorted algebra I: Equational deduction for multiple inheritance, polymorphism, and partial operations. *Theoretical Computer Science*, 105(2):217–273, November 1992.

[Gog91] Joseph A. Goguen. Types as theories. In *Topology and category theory in computer science*, pages 357–390. Oxford science publications, 1991.

[GS94] David Gries and Fred B. Schneider. *A Logical Approach To Discrete Math*. Springer-Verlag, 1994.

[Gun92] Carl A. Gunter. *Semantics of Programming Languages: Structures and Techniques*. Foundations of Computing Series. The MIT Press, 1992.

[H+92] P.R. Hudak et al. Report on the programming language Haskell, a non-strict purely functional language, Version 1.2. *ACM SIGPLAN Notices*, 1992.

[H+95] Gerard Huet et al. The Coq Proof Assistant, User's Guide, Version 5.10. Technical report, INRIA, 1995.

[Hin69] R. Hindley. The principal type scheme of an object in combinatory logic. *Transactions of the American Mathematical Society*, 146:29–60, December 1969.

[Hin86] J.R. Hindley. *Introduction to Combinators and λ-Calculus*, volume 1 of *London Mathematical Society Student Texts*. Cambridge University Press, 1986.

[HMT] Robert Harper, Robin Milner, and Mads Tofte. *The Definition of Standard ML, Revision 1997*. The MIT Press. To appear.

[HMT90] Robert Harper, Robin Milner, and Mads Tofte. *The Definition of Standard ML*. The MIT Press, 1990.

[How87] Douglas J. Howe. The computational behaviour of girard's paradox. In *Proceedings, Symposium on Logic in Computer Science*, pages 205–214, Ithaca, New York, 22–25 June 1987. The Computer Society of the IEEE.

[HP89] Robert Harper and Robert Pollack. Type checking, universe polymorphism, and typical ambiguity in the calculus of constructions. In N. Díaz and F. Orejas, editors, *TAPSOFT 89 — Proceedings of the International Joint Conference on Theory and Practice of Software Development*, volume 352 of *Lecture Notes in Computer Science*, Copenhagen, Denmark, March 1989. Springer-Verlag.

[JS92] Richard D. Jenks and Robert S. Sutor. *Axiom: The Scientific Computation System*. Springer, 1992.

[K+93] P. Klint et al. The ASF+SDF meta-environment user's guide, version 2.6. Technical report, Centrum voor Wiskunde en Informatica (CWI), March 1993.

[Knu73] D. E. Knuth. *Sorting and Searching*, volume 3 of *The Art of Computer Programming*. Addison-Wesley, 1973.

[KS] Norbert Kajler and Neil Soiffer. A survey of user interfaces for computer algebra. *Journal of Symbolic Computation*. To appear.

[Lan84] Serge Lang. *Algebra*. Addison-Wesley, 2nd edition, 1984.

[LCA94] Florian Matthes Luca Cardelli and Martin Abadi. Extensible syntax with lexical scoping. Technical report, DIGITAL Systems Research Center, 130 Lytton Avenue, Palo Alto, California 94301, feb 1994.

[MB91] Tony Mason and Doug Brown. *Lex & Yacc*. Nutshell Handbooks. O'Reilly & Associates, 1991.

[Mee86] L. Meertens. Algorithmics – towards programming as a mathematical activity. In *Proceedings of the CWI Symposium on Mathematics and Computer Science*, pages 289–334. North-Holland, 1986.

[MH88] J. Mitchell and R. Harper. The Essence of ML. In *Conference Record of ACM Symposium on Principles of Programming Languages*, pages 28–46, 1988.

[Mil78] Robin Milner. A theory of type polymorphism in programming. *Journal of Computer and System Sciences*, 17:348–375, 1978.

[Mis94] Stephan A. Missura. Theories = Signatures + Propositions Used as Types. In Calmet and Campbell [CC94].

[Mis95] Stephan A. Missura. Eliminating the shortcomings of free datatype definitions. Technical Report 242, ETH Zurich, December 1995.

[Mit90] John C. Mitchell. Polymorphic type inference and containment. In Gerard Huet, editor, *Logical Foundations of Functional Programming*, pages 153–193. Addison-Wesley, 1990.

[MR86] Albert R. Meyer and Mark B. Reinhold. 'Type' is not a type: Preliminary report. In *Conference Record of the Thirteenth Annual ACM Symposium on Principles of Programming Languages*, pages 287–295, St. Petersburg Beach, Florida, January 1986. Association for Computinge Machinery.

[MW94] Stephan A. Missura and Andreas Weber. Using Commutativity Properties for Controlling Coercions. In Calmet and Campbell [CC94].

[NPS90] Bengt Nordström, Kent Petersson, and Jan M. Smith. *Programming in Martin-Löf's Type Theory*. Oxford Science Publications, 1990.

[Pau91] Lawrence C. Paulson. *ML for the Working Programmer*. Cambridge University Press, Cambridge, 1991.

[Pau94] Lawrence C. Paulson. *Isabelle: A Generic Theorem Prover*, volume 828 of *Lecture Notes in Computer Science*. Springer Verlag, 1994.

[PE88] Frank Pfenning and Conal Elliot. Higher-order abstract syntax. In *ACM SIGPLAN '88 Conference on Programming Language Design and Implementation*, 1988.

[Pie91] B. Pierce. *Basic Category Theory for Computer Scientists*. MIT Press, 1991.

[PJ87] Simon L. Peyton Jones. *The Implementation of Functional Programming Languages*. Prentice Hall International, 1987.

[Pra73] Vaughan R. Pratt. Top down operator precedence. In *Conference Record of the ACM Symposium on Principles of Programming Languages*, pages 41–51. Association for Computing Machinery, January 1973.

[Rea89] C. Reade. *Elements of Functional Programming*. Addison-Wesley, 1989.

[Rek92] J. Rekers. *Parser Generation for Interactive Environments*. PhD thesis, University of Amsterdam, 1992.

[Rév91] György E. Révész. On translating ordinary mathematical notation. *Structured Programming*, 12:115–122, 1991.

[Rey74] John C. Reynolds. Towards a theory of type structure. In *Programming Symposium*, volume 19 of *Lecture Notes in Computer Science*, pages 408–425. Springer-Verlag, 1974.

[Rey80] John C. Reynolds. Using category theory to design implicit conversions and generic operators. In *Semantics-Directed Compiler Generation, Workshop*, volume 94 of *Lecture Notes in Computer Science*, pages 211–258. Springer-Verlag, 1980.

[RL91] György E. Révész and Kevin T. Lynch. A context-free characterization of ordinary mathematical notation encoded in TEX. Technical Report RC 16615, IBM Research Division, jul 1991.

[RS92] Lucia Rapanotti and Adolfo Socorro. Introducing FOOPS. Technical Report PRG-TR-28-92, Oxford University, 1992.

[Rud88] Walter Rudin. *Functional Analysis.* McGraw-Hill, 1988.

[San89] D. Sannella. Formal program development in Extended ML for the working programmer. Technical report, Laboratory for Foundations of Computer Science, University of Edinburgh, December 1989.

[Sch84] Peter Schreiber. *Grundlagen der Mathematik.* VEB Deutscher Verlag der Wissenschaften Berlin, second edition, 1984.

[Sch94] David A. Schmidt. *The structure of typed programming languages.* The MIT Press, 1994.

[Sml97] Standard ML of New Jersey. Bell Laboratories, URL: `ftp://ftp.research.bell-labs.com/dist/smlnj`, 1997.

[Soi95] Neil Soiffer. Mathematical typesetting in mathematica. In *Symposium on Symbolic and Algebraic Computation (ISSAC '95).* Association for Computing Machinery, 1995.

[Sto77] J. E. Stoy. *Denotational Semantics: The Scott-Strachey Approach to Programming Language Semantics.* MIT Press, 1977.

[Str91] Bjarne Stroustrup. *The C++ Programming Language.* Addison-Wesley, second edition, 1991.

[Tho91] Simon Thompson. *Type Theory and Functional Programming*. International computer science series. Addison-Wesley, 1991.

[Tri80] Hans Triebel. *Höhere Analysis*. Verlag Harri Deutsch, Thun und Frankfurt am Main, 1980.

[Tur90] David A. Turner. An overview of Miranda. In David A. Turner, editor, *Research topics in Functional Programming*. Addison Wesley, 1990.

[Tur91] Raymond Turner. *Constructive Foundations for Functional Languages*. McGraw-Hill, 1991.

[TZ82] G. Takeuti and W. M. Zaring. *Introduction to Axiomatic Set Theory*. Springer-Verlag, 1982.

[vG90] A.J.M. van Gasteren. *On the Shape of Mathematical Arguments*, volume 445 of *Lecture Notes in Computer Science*. Springer-Verlag, 1990.

[Voi86] Frédéric Voisin. Cigale: a tool for interactive grammar construction and expression parsing. *Science of Computer Programming*, 7:61–86, 1986.

[Web93] Andreas Weber. *Type Systems for Computer Algebra*. PhD thesis, Fakultät für Informatik, Universität Tübingen, July 1993.

[Wel94] J.B. Wells. Typability and type checking in the second-order λ-calculus are equivalent and undecidable. In *Proceedings, 9th Annual IEEE Symposium on Logic in Computer Science*, pages 176–185. IEEE Computer Society Press, 1994.

[Wir88a] N. Wirth. The programming language Oberon. *Software-Practice and Experience*, 18:671 – 690, 1988.

[Wir88b] Niklaus Wirth. *Programming in Modula-2*. Springer, 1988.

[Wir90] Martin Wirsing. Algebraic specification. In Jan van Leeuwen, editor, *Formal Models and Semantics*, volume B of *Handbook of Theoretical Computer Science*, chapter 13, pages 675–788. Elsevier, 1990.

[Wol91] Stephen Wolfram. *Mathematica: A System for Doing Mathematics by Computer*. Addison-Wesley, second edition, 1991.

[Wol96] Stephen Wolfram. *The Mathematica book*. Wolfram Media, 1996.

[Zha96] Yanjie Zhao. *Knowledge-Based Parsing Method of Mathematical Notation for Improving Mathematical Computation Environment*. PhD thesis, Nagoya University, 1996.

Curriculum Vitae

I was born on Feburary 20, 1968 in Grenchen (SO). From 1975 to 1981 I attended primary school in Glarus. In 1981 I entered the High School (Gymnasium) in Glarus, from which I graduated 1987 with Matura Typus C.

In 1987 I began studying computer science at ETH Zurich. I received the degree *Dipl. Informatik-Ing. ETH* in 1992. I was awarded the ETH Silver Medal for my master thesis entitled *Klassenbasierte Umgebung für algebraische Modellierungen in AlgBench*.

From 1993 to 1996 I was a research and teaching assistant at the Institute for Theoretical Computer Science of ETH Zurich in the Symbolic Computation group led by Prof. Roman Mäder. Since 1996 I have been working as a research and teaching assistant at the Logics and Computer Science group of the Mathematics Department of ETH Zurich headed by Prof. Erwin Engeler.