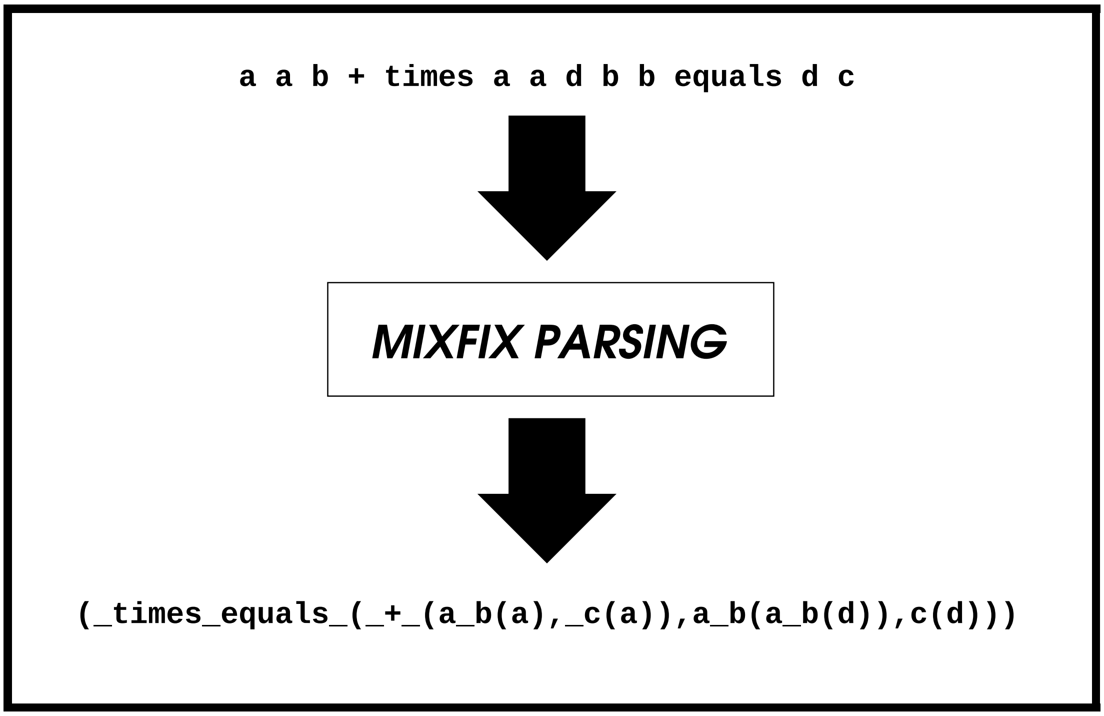
University of Amsterdam

Programming Research Group



Introduction & Basic Tooling for CASL using ASF+SDF

Bjarke Wedemeijer

Report P9809 october 1998

University of Amsterdam

Department of Computer Science

Programming Research Group

Introduction & Basic Tooling for CASL using ASF+SDF

Bjarke Wedemeijer

Report P9809 october 1998

B. Wedemeijer

Programming Research Group
Department of Computer Science
University of Amsterdam

Kruislaan 403
NL-1098 SJ Amsterdam
The Netherlands

tel. +31 20 525 7463
e-mail: wedemeij@wins.uva.nl

Introduction & Basic Tooling for CASL using ASF+SDF

Bjarke Wedemeijer Universiteit van Amsterdam
Afstudeerdocent: prof. dr. P. Klint
Begeleiders: dr. M.G.J. van den Brand prof.dr. P.D.Mosses

October 27, 1998

Contents

Chapter 1

Introduction

A few years ago an initiative was started to develop a general algebraic specification language. The designers based their work on existing algebraic languages. The creation of this new language should also be supported by existing tools for algebraic specification formalisms. The new language should not be seen as a deus ex machina, but as a generalization of existing algebraic techniques. At the time of writing this thesis the language was still very tentative. By using existing algebraic specification tools like ASF+SDF to define the new language, the designers are capable of comparing the new language with existing languages and furthermore, whether the new language has unwanted properties[1].

This masters thesis is based on work done at the University of Aarhus in cooperation with Peter Mosses. Only a small part of the work done is presented in this thesis. A lot of time is spent on the development of the CASL-grammar since it was quite tentative. We used the ASF+SDF META-ENVIRONMENT as tool to examine and develop the CASL-grammar. ASF+SDF generated quite a lot of feedback, with respect to ambiguities, which influenced the original design of the concrete syntax of CASL.

1.1 Problem Description

There is not much tool support for CASL at the moment. When designing a new language the tool support is always very restricted. It makes no sense to develop complete parsers before the syntax is fully stable. On the other hand the development of a parser at an early stage may reveal all kinds of ambiguities and shortcomings in the (concrete) syntax of the new developed language. However, using tools like YACC when the syntax is not yet stable may be very tedious because of the effort invested to resolve conflicts in the parse table. Therefore we used ASF+SDF to do language prototyping for a part of CASL. We restrict ourselves to the examination of the syntax of a sublanguage of CASL: Basic Items.

[1] e.g. unwanted parses, ambiguities, etc.

By implementing a sublanguage of CASL in ASF+SDF we are able to test, verify and parse CASL specifications belonging to that sublanguage. An implementation of this sublanguage in ASF+SDF is still easier said than done. The grammar of CASL is divided in two parts:

- **Concrete Syntax**: the concrete syntax is used when writing concrete specifications, it introduces keywords, constructions, etc. The concrete syntax of CASL is completely defined, but unfortunately this grammar is quite ambiguous and not yet stable.
- **Abstract Syntax**: an abstract syntax is only meant for automatic processing, it has no syntactic sugar and is not ambiguous. The semantics of CASL are defined on this syntax.

Since all CASL specifications are written in an ambiguous concrete syntax, whereas the semantics are defined in terms of a non-ambiguous abstract syntax, a conversion from concrete to abstract syntax is necessary for the following reason. Tools developed by different groups interact with each other at the level of abstract syntax. Tools like parsers, typecheckers, and rewrite engines, can be "connected" by means of the abstract syntax.

Of course, a dozen of other arguments can justify the conversion from concrete to abstract syntax. But this reason is of significant importance when developing tools and therefore it motivates our main question of this thesis: is it possible to convert the concrete CASL syntax, restricted to the sublanguage Basic Items, to abstract CASL syntax using ASF+SDF?

1.2 Organization

A general introduction of the CASL language is given in Chapter 2 together with a historical overview of the Common Framework Initiative (CoFI) project. Furthermore the development of CASL is discusses by presenting the basic design issues and finally a critical examination of the industrial goals of CASL is given. We are examining the syntax of a sublanguage, therefore we are giving an introduction in Chapter 3 to its syntax.

The main question of our thesis is addressed in Chapter 4 and 5. The specification of the CASL grammar can be done in relatively few rules. Chapter 4 discusses also the conversion of concrete to abstract CASL syntax which is not so easy as the definition of the concrete CASL grammar in ASF+SDF. A restricted number of rules establish a huge part of the conversion but since concrete CASL allows full mixfix syntax the rest of the conversion can not be done so easily. Chapter 5 presents a solution for the conversion of mixfix syntax with one exception: if the general mixfix syntax contains ambiguities they are not discovered. Chapter 6 gives a possible solution to handle these ambiguities. This results in a positive answer for the main question. Since the main question has quite a complex answer, other (sub) problems occur, e.g., can the conversion from concrete CASL be done in an adequate way? At the end of Chapters 5 and

6 this adequacy is discussed. These subproblems are stated in the beginning of the chapters and their answers are part of the main answer of this thesis. At the end a concluding chapter discusses the results, possible improvements, and the reuse of the mixfix parser for other tool builders.

1.3 Hints before reading

One should be aware that before reading this masters thesis some basic knowledge about ASF+SDF [14, 15, 16], general algebraic specification literature [3, 6, 7], and general notions about CASL and CoFI [9, 10, 19] is assumed.

1.4 Acknowledgments

A lot of people supported me during the preparation of this thesis and now I want to thank them all. First of all, I would like to thank Peter Mosses and Mark van den Brand. Without them, this thesis was never finished. Peter[2] supported me answering all my silly questions with great patience and diplomacy. He arranged the best working atmosphere one can image. After asking Peter I could ask Mark[3] everything I wanted and he supported me to the end. Even if I made stupid mistakes as introducing my own ambiguities he pinpointed me the spot and made some humoristic remarks. He was just extremely helpful.

While working with ASF+SDF Chris Verhoef[4] supported me by sending me now and then humoristic email's and he was seriously interested in CASL. I definitely want to thank my parents for their unconditional support, without them this thesis was never written. Lene Giel, Peter Kuiters and my sister supported me continuous. None of the above would have happened without my wife Sandie.

[2]http://www.brics.dk/~pdm
[3]http://adam.wins.uva.nl/~markvdb
[4]http://adam.wins.uva.nl/~x

Chapter 2

CASL and CoFI, A Political Achievement

2.1 In Abstracto

A number of researchers from the algebraic community[1] initiated a collaborative effort to design a common framework for the development and specification of software. The main idea of this effort is that there is a need for such a common framework because the absence of it hinders the industrial standardization, distribution, and application of research results in the field of algebraic specifications. Such a uniform framework might create a positive attitude form the industry towards algebraic specifications. However, not only commercial thoughts are at stake here, a widespread acceptance of a common framework is urgently needed within the research community as well. Due to a perfect Babel of tongues a lot of energy is used on interpreting each others materials, examples, etc.

The goal of the CoFI-project (Common Framework Initiative) is that the framework is mainly based on a selection of properties that have been investigated in other algebraic specification formalisms. The framework unifies these specification languages at different levels. This family of languages has one expressive and central language called CASL (Common Algebraic Specification Language) which covers all sublanguages. Some fields where CASL could be used are:

- for specifying conventional software (design, requirements, etc.).
- in connection with prototyping and verification tools.
- oriented towards certain programming paradigms (OOP, reactive).
- for mapping from existing languages to members of the CASL-family.

[1]e.g., P.D. Mosses, D. Sanella, M. Bidoit, and others.

2.2 History and Background

Quite a number of algebraic specification formalisms have been developed in the past decades. Some are extremely ambitious, huge formalisms to cover complete software development. Others are quite modest and small. Some of these formalisms are popular but none has an overall support within the algebraic specification community. The majority of these formalisms were created at universities or by individual researchers. That is why these formalisms have a lot of rural characteristics. Two exceptions are LARCH [12] and OBJ [11]. Due to those rural characteristics the convictions behind these formalisms are a bit narrow minded which do not stimulate general acceptance. The conclusion is that there does not exist a completely general, common, and widespread formalism. Standardization of algebraic methods could improve their popularity not only within the research community but also within industry. However, there are quite a number of counter examples of languages, such as COBOL, which are heavily used in industry and not really standardized. Not only these commercial and academical thoughts lie behind CoFI. For standard examples, education, and case studies it would be quite useful to use one common framework. The algebraic community, related commercial organizations and educational organizations must agree on such common framework. However, at the moment only academics are discussing and developing this common framework. Commercial organizations are not yet involved. It looks like the common framework is only for academical use.

2.3 CoFI

The idea of a common framework started in June 1994 by members of COMPASS and IFIP Workgroup 1.3. COMPASS was an ESPRIT (European) Basic Research Workgroup regarding algebraic specifications. IFIP Workgroup 1.3 (Foundations of System Specification) was founded in 1992, the members came not only from Europe but from all over the world. In Oslo, autumn 1995, the main ideas and aims were clarified: CoFI was born. During that combined meeting of COMPASS and IFIP in Oslo the decision was made to start the Common Framework Initiative. A number of task groups were formed, such as the tool task group and language design task group. The IFIP Workgroup 1.3 is currently the driving force behind CoFI, they initiate proposals and keep the CoFI flame burning.

2.3.1 Overall Aims of CoFI

The overall aims of CoFI were formulated in [9]. A summary:

- A common framework for algebraic specification and software development is to be designed, developed, and spread around the world.
- The production of the common framework is to be a collaborative effort.

- In the short term (around 1997) the common framework is to be accepted as an appropriate basis.
- Specifications in the common framework should have an uniform, user-friendly syntax and straightforward semantics.
- The common framework must be able to replace many existing algebraic specifications formalisms.
- The common framework is to be supported by reference manuals, users guides, libraries of specifications, tools, and educational materials.
- In the longer term the common framework is to be made attractive for use in industrial contexts.
- The common framework is to be available free of charge, both to academic institutions and to industrial companies.

More unofficial aims are also named in [9]. These aims concern documentation, verification, formal development of algebraic specifications, reuse, logical consequences (theorem proving, prototyping, rewriting etc.). All these aims can be seen as a shopping list for the implementation of the framework.

2.4 The Framework and Languages

The goals of this general framework is to include all concepts of previous specification languages. So, all extensions and restrictions of each algebraic specification language must be covered. This raises immediately the following question: is this possible? The main idea of CoFI is not to provide one language but a family of languages closely related to each other. A number of restrictions and extensions of other formalisms are exploited by introducing a main language CASL. However, certain restrictions or extensions can not be covered by this main language and therefore sublanguages of CASL are introduced to deal with them. The concepts of CASL have been chosen in such a way that there is a certain balance between simpler and more advanced languages. It was decided that, the basic concept of CASL, is the specification of functional requirements and design of conventional software packages such as abstract data types.

2.4.1 Restrictions and Extension of the Main Language

The restrictions of the main language correspond to languages used with existing tools for rapid prototyping, term rewriting, verification, and so on. These restrictions may concern syntactic/semantic restrictions. Although these restricted languages might be different internally, they allow basically single or many-sorted equational specifications. Only a specific class of languages can be processed by each available tool. These tools impose restrictions on, for example, sorts, overloading, parsing only prefix operations, etc.

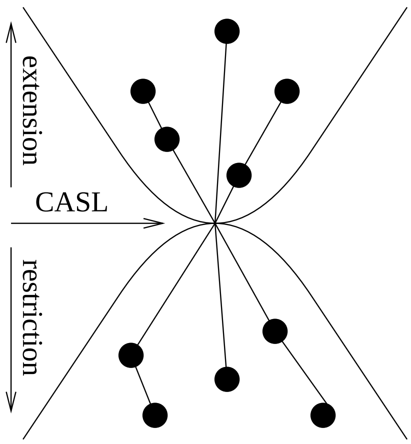


Figure 2.1: Restriction and Extension

Extension of the main language can be seen as extension on basic programming paradigms, for example higher-order functions, OOP, and reactive. These extensions are not part of the main language. The main language is a balance between the mentioned restrictions and these extensions. There extensions are not in the main language (see Figure 2.1) Of course, combining extensions with the main language can give severe technical difficulties. Unfortunately the CoFI community does not give a solution at the moment for these difficulties. This is still being investigated.

2.5 Other Features of the Framework

Because a framework is not only a language, other things have to be developed as well. [9] states the following:

> The common framework is also to provide an associated development methodology, training materials, tool support, libraries, a reference manual, formal semantics and conversion from existing formalisms.

Learning from history this aim is maybe a bit too ambitious. A lot of formalisms just did not have the resources to develop all these things. The CoFI community keeps the thought that investing resources into the framework will solve these problems. But where do these resources come from? If the amount of resources is below a certain level the framework may terminate before it is fully developed. A solution to some of the problems is to exchange and/or reuse existing tools and formalisms in combination with the common framework[2].

One of the best things of the framework is the possibility to create a library of useful specifications in a single language. Libraries have always been a problem

[2]This is already done by involving for example ASF+SDF into the initiative.

because a library of one language could not be reused by another language. This problem will be solved in the future when CoFI is fully developed.

2.6 A Definite Proposal

On the basis of all the above mentioned investigations and discussions the final proposal for the design of CASL, the main language of CoFI, has been given to IFIP WG 1.3 in June 1997. A lot of work is still to be done. Severe problems, as mentioned in the previous sections, are present. The main problem, for CoFI and its main language CASL, is to survive. Not only a resource problem exists, also the acceptance of CASL as the main specification language is a problem. But maybe if the acceptance increases users of other frameworks, such as ASF+SDF, might switch to CASL instead.

2.7 Critics

A lot of critical notes have to be made. Some important ones are:

- The main language CASL should be kept as simple as possible. If the language becomes to complex the link to industry as well as education will be extremely difficult. This contradicts with the tendency in many projects where complexity grows because the developers want the best and most complete language.

- People are rebelling against uniformity. Therefore an universal algebraic specification language will not be accepted easily. Also some political and commercial points could be mentioned here. Research groups are not willing to give up their own work because a lot of money is invested and they have to give up their own creative work.

- At the moment only academics are involved in the initiative but industry must also have a voice in the initiative. If this is not the case the common framework will be another academical framework with a small amount of people working on it. The discussion about for example concrete/abstract syntax is only held at an academical level. It is obvious: if the framework is to be accepted outside the academical world this might be a problem.

Chapter 3

Language Introduction

In this chapter we are examining the syntax of the sublanguage Basic Items of CASL. Since CASL is a new language, a more elaborate introduction to the CASL syntax is given. In this chapter the following steps are taken:

- We introduce briefly the theory behind CASL.
- A very simple example is shown, intuitively one can guess what the example specifies.
- Every construction of the sublanguage Basic Items is described on a syntax level.

After this chapter one is able to recognize and specify simple CASL specifications. This is needed for understanding the rest of the thesis.

3.1 Theory

We start with a brief theoretical approach towards algebraic specification formalisms. For more information about the theoretical background of algebraic specifications, see [3, 6, 7]. Algebraic specification formalisms may be formalized in terms of so-called institutions and proof-systems. A basic algebraic specification formalism inhabits:

- a class **Sig** of signatures Σ. Every element of this class defines symbols which meaning is going to be specified. There exist morphisms between signatures.
- a class $\mathbf{Mod}(\Sigma)$ of models. For each signature Σ there exists a homomorphism between the models of that signature Σ.
- a set $\mathbf{Sen}(\Sigma)$ of axioms (or sentences) for each signature Σ.
- a satisfaction relation $\models$ between axioms and models over the same signature.

- a proof system to derive axioms/sentences from the set of axioms given.

A specification consists of a combination of a signature Σ and a set of sentences from $\mathbf{Sen}(\Sigma)$. The semantics of this specification are exactly the class of models in $\mathbf{Mod}(\Sigma)$ that satisfy all specified axioms. If there exists no model, in which all sentences are satisfied, the specification is called *inconsistent*; otherwise it is called *consistent*.

3.1.1 Signatures

A many-sorted signature is a tuple $\Sigma = (S, \Omega, P)$. The elements stand for sorts, operations[1], and predicates. The signature is a kind of syntax for the algebra. The functions Ω fall, within CASL, apart in two sets. There are total and partial functions. Each function consists of a tuple:

$$n : s_1 \times \cdots \times s_k \rightarrow s$$

where $s_1 \times \cdots \times s_k \in \mathrm{S}$ and $k \geq 0$, where n is the function name and k its arity. The sorts $s_1 \times \cdots \times s_k$ are called the argument sorts of the function, s a target sort[2]. The special case where $k = 0$, the function $n :\rightarrow s$, is called a constant of sort s. Function names may be overloaded. Set P contains predicate symbols. They are often represented as boolean functions which is in fact not totally correct. Because in boolean arithmetic true has the same status as false whereas a predicate is only defined and it can not be undefined.

3.1.2 Partial versus Total Function

A function is partial if there is no way of deriving every element of the target sort using the sorts of the specification. Informally: a partial function is undefined on some elements of the existing sorts. There may be no *sensible* result for some input. The properties defining the function do not guarantee or compute a result. Therefore, some input can be refused. A total function guarantees that for every input there is some sensible result. So, a total function is defined for *every* input.

3.1.3 Models

Looking into a many-sorted signature $\Sigma = (S, \Omega, P)$ a many-sorted algebra $M \in \mathbf{Mod}(\Sigma)$ is a many-sorted first-order structure consisting of (ω denotes $s_1 \times \cdots \times s_k$)

- a non-empty carrier set s^M for each sort $s \in S$.
- a partial function f^M from ω^M to s^M for each function symbol $f \in \Omega$.

[1]Operation is sometimes confused with function. We assume they are the same. In fact, during the development of CASL the operation was basically called function, even in the concrete syntax.

[2]The target sort is also called the coarity of the function.

- a predicate $p^M \subseteq \omega^M$ for each predicate symbol $p \in P$.

A homomorphism h from M_α to M_β, M_α and $M_\beta \in \mathbf{Mod}(S, \Omega, P)$ is a function $h_s : s^{M_\alpha} \to s^{M_\beta}$ for each $s \in S$. These homomorphisms preserve the values of functions, the truth values of predicates, and the definedness of the functions.

3.1.4 Sentences

The terms are many-sorted on a signature $\Sigma = (S, \Omega, P)$. These terms are built from:

- variables from X, the set of sorted variables.
- function symbols in Ω.

For our many-sorted signature the sentences are ordinary first-order logic formulæ. They are built from quantification over sorted variables and logical connectives. Sentences are also built from qualified predicate symbols $p \in P$, assertions about the definedness of qualified terms, and existential/strong equations. The sentences may include sort-generation constraints as well. A sort-generation constraint introduces a new sort based on functions from Ω defining that new sort. These functions specify how the new sort is constructed.

3.1.5 Satisfaction

The satisfaction of a sentence within a model holds, informally said, only when all values, assigned to the occurring variables s, are in s^M. The value of a term can be undefined due to partial functions. The satisfaction of sentences is two valued. The use of a predicate symbol p towards terms holds in M if and only if the values of all terms are defined and belong to p^M. The definedness assertion with respect to a term holds if and only if the value of the term is defined. An existential equation holds when the values of the terms are both defined and equal; a strong equation holds also when the values of both terms are undefined. Thus the two forms of equations are equivalent when the values of both terms are always defined.

3.2 A First Look

There are many ways to introduce a new language. One could start with a completely formal introduction. The disadvantage of such an approach is that some people just hate formal methods. The advantage is that, for those who understand it, the language becomes crystal clear. Assuming that the world prefers examples above a formal introduction, we start with a very informal introduction of CASL based on examples. We start with a minor example and explain what is going on.

3.2.1 The Start

CASL is an algebraic specification language. Looking into the literature ([3, 6, 7, 8]) about algebraic specification languages one can easily see that there are some basic things present in almost every algebraic specification language. The intention of the designers of CASL was to keep CASL as close as possible related to those specification languages. Of course, not everything could be adopted right away, many compromises have been made. In general the following features are at least present in most algebraic languages:

- a signature Ω consisting of a sort and operation declarations.
- variables over the sorts defined in the signature. Beware: variables are formally *not* part of the signature.
- some kind of logic system with sentences (also called axioms).

More can be said about algebraic specifications. However, we take these three things and see how to involve CASL into this. CASL has indeed some kind of signature where one can define sorts and operations. Variable and axiom declarations exists within CASL as well. In algebraic specification languages, the order in which elements occur, can play a role. There exists linear-visibility in CASL(Basic Items). This means that, for example, the variable declarations may not be declared before the sorts are defined. Tools should be aware of this feature. So, to typecheck a variable declaration all the sorts mentioned in the variable declaration must be declared before the variable declaration.

An example of a CASL specification, containing all the elements mentioned above, would look like:

```
%% my first casl specification !
spec Hello_World =
  sort    String;
  var     Message: String
  op      HelloWorld : String;
          Yell__   :  String -> String
  axiom
  Message = Yell(HelloWorld)
end
```

An “equivalent” C program could be:

```
/* My first C program ! */

#include <stdio.h>
int main(void) {
printf(‘‘ Hello, World ! \n’’);
}
```

These examples may be slightly confusing. If we compile the C-program and execute it on a certain platform the text 'Hello, World!' will appear as output. If it is possible to execute[3] a CASL specification, there will be no output to the screen. CASL is an algebraic language without output features, these should be provided by the tools. One can just focus on the algebraic notions.

3.2.2 Comments

The HelloWorld-example started with the comment `%%my first casl program`. Comments start with `%%`. There are two possibilities of ending a comment:

- At the end of the line, when a newline character is seen.
- By placing `%%` on the same line[4].

It is not possible to expand the comment over newlines. If comments are spread over several lines, each of these lines should start with `%%`. Comments are allowed everywhere but not within preserved keywords. So, `sp%%oops%%ec TryOut = ...` is not correct. A good use of comments is:

```
%% start of specification
%% author: . . .

spec Tryout =
        var x,y,z   : Nat;           %% Nat is predefined
        op  __ + __ : Nat*Nat -> Nat %% addition
        axioms
           x = (x+y) + %% between formulae! %% (x+z)
end %% end of specification
```

3.2.3 Optional Semicolons

In the previous example the use of semicolon's is not consistent. In CASL there are *optional semicolons*. Every axiom, variable, operation, predicate or sort declaration may be terminated by a semicolon or not.

3.3 Sort Declarations

Sorts are the elements within the algebraic specification which one reasons about. Sorts can be seen as sets but with some special characteristics. Sorts are special because when one declares a sort some features are inherently present[5] (at least in CASL). Sort declarations in CASL look like

[3] Only a certain class of CASL can be executed.

[4] This construction is very difficult to specify in ASF+SDF, for the moment only the first option is implemented.

[5] E.g. if the sort is declared, then it is in the signature.

```
. . .
spec LetsMakeSorts =
  sort  a,b,c;
        x,y < z;
        s = { p : q . p mod 2 };
        n=m=k=l; %% optional semicolon
  . . .
end
```

The notation ... is not a feature of CASL. It is an informal notation to suppress a part of the specification. There are exactly four ways of defining sorts. After the keyword **sort/sorts** at least one of the following elements should appear:

- *ordinary*: `a,b,c` is the ordinary way of declaring sorts. Some kind of overriding is not prohibited here, it just makes no sense when it occurs. So, e.g., `sorts a,a,a` is legal but is equal to `sorts a`. If this ordinary form is used at least one sort should be defined.

- *subsorting*: `x,y < z` informally means that `x` and `y` are subsorts of `z`. At least one sort before the `<` should be defined. After the `<` only one sort is allowed.

- *predicates*: `s = {p : q . p mod 2}` defines a new sort `s`. The elements of `s` are generated from `q` using the formula `p mod 2` where `p` is a variable of sort `q`. So, `s` is in fact a subsort of `q`.

- *iso-sorts*: `n=m=k=l` defines for the sorts `n`, `m`, `k` or `l` a one to one correspondence between the elements of these sorts. Informally, there is a one to one mapping for every element even though the elements of each sort may be completely different internally.

Sorts Syntax
sort/sorts
  <sortname(s)> separated by comma's
  | *<sortname(s)> separated by comma's < <sortname>*
  | *<sort> = { <variable> : <sort> . <formula> }*
  | *<sortname(s)> separated by equal symbols*
   (minimum two sortnames)
opt-semi

The |[6] is used as in BNF, it separates the alternatives. After the keyword at least one of these four possibilities must be present, semicolons are used as separators. The semicolon as separator and the optional semicolon are confusing. The only places where optional semicolons are allowed is at the end of declarations. Sort definitions are always separated by semicolons. In the previous sort declaration example the last semicolon is thus an optional semicolon. The

[6]The | here has the same meaning as in the concrete and abstract syntax in the appendices.

notation **sort/sorts** means either **sort** or **sorts** as keyword. The keyword **sort** does not imply that only *one* sort declaration must occur, there may be more than one, and the other way around. In general, this applies for every keyword used in the syntax, the singular and plural variant are separated by /.

3.4 Variable Declarations

Variables are used within axioms and are sorted. Remember, variables do *not* contribute to the signature. A variable declaration always starts with a keyword `var` or `vars`. After the keyword comes a list of one or more variable names separated by comma's followed by a colon and the sort, for example:

```
. . .
vars    x,y,z : CoordinateSystem;
          a,b : ComplexNumbers;
. . .
```

Here `x`, `y`, `z` are variables of the sort `CoordinateSystem` and `a`, `b` of the sort `ComplexNumbers`. After the sortname more variables can be declared. The variable names within a comma separated list must be unique and overriding[7] is not allowed. The overriding principle also applies when there is more than one variable declaration in the specification. The relation between axioms and variables is that for every variable declared a universal quantification over that variable is implicitly assumed over all axioms.

Variable Syntax
var/vars
<variablename(s)> separated by comma's : <sortname>
opt-semi

Multiple variable declarations can again be defined by just separating them by semicolons as shown in the example. Variable declarations are always global. There is one exception to this rule. Local variable declarations may occur in combination with axioms. After the variable declaration(s) a number of axioms appear and the scope of the declared variables is restricted to these axioms.

Local Variable Syntax
var/vars
<variable declaration(s)> . <axiom(s)>
opt-semi

The dot between variable declaration(s) and axioms(s) is used as a separator. The specified axiom(s) contribute to the whole specification, they are *not* local.

[7] Overriding means one variable name with two sorts, so `var a:S; a:T`.

The axiom(s) are internally only separated by dots[8] The axioms do not start with a keyword.

3.5 Operation Declarations

In the first example 'HelloWorld' we introduced two operators:

```
 . . .
 op      HelloWorld : String;
         Yell__   :  String -> String
 . . .
```

The first `HelloWorld : String` defines a constant `HelloWorld` of type `String`. The second one introduces an operator `Yell` with one argument of the type `String`. The `__` is called a placeholder. There is a one-to-one linear correspondence between placeholders and sorts after the colon. The placeholder of the `Yell` operation can be replaced by a term of type `String`. The resulting type of this operation is `String`. An operation is declared by introducing an operation name with some placeholders (if required). The argument sorts are defined after the colon and are separated by an asterisk. The result sort (this can *only* be one) is defined after the arrow. Overloading of operators is allowed. CASL allows mixfix definition of operations, so placeholders can occur in any combination with the operation names, for example: (assume that the sorts `Nat` and `Bool` are defined)

```
 . . .
 op
    __ times __ equals __ : Nat*Nat*Nat -> Bool; %% mixfix
    __ __ +               : Nat*Nat -> Nat;      %% postfix
    / __ __               : Nat*Nat -> Nat;      %% prefix
    __ + __ __            : Nat*Nat*Nat          %% infix
 . . .
```

All these declarations are valid. There are some special cases:

- *prefix*: this notation is very known. A prefix operation in CASL would be `Pow __ __`. The corresponding use would then be `Pow__ __(2,3)` but could also be `Pow 2 3`. The declaration of `Pow` could also be: `Pow:Sort*Sort'->Sort''`. The application of this operation would then be `Pow(2,3)`.
- *postfix*: is almost the same as prefix only the name of the operation comes at the end not at the beginning and can only be used in, for example, the following form `1 2 +`.
- *infix*: left and right to the operation name might occur placeholders. A standard example would be `__+__`, application of this operation would be for example `a+b`.

[8] This is only here the case, general axioms can only be separated by semicolons.

- *mixfix*: now there exists no order. One can say that a mixture of postfix, prefix and infix are allowed and there can be more than one operation name in a definition (such as `__times__equals__`). The question is of course how a parser can recognize whether a mixed term is correct or not. The parsing of mixfix terms will be addressed in Chapter 5.

As seen in the formal introduction of algebraic languages, operations can be total or partial. The above mentioned operations are total. Partial operators are declared explicitly by adding a `?` just after the arrow, for example:

```
. . .
op
    __ plus __ : Nat*Nat ->  Nat; %% total
    __ div __  : Nat*Nat ->? Nat; %% partial
       1       : Nat              %% constant
. . .
```

As explained before, for some input the partial function gives no/never a *sensible* output. The `div` is chosen as a partial function because `3 div 0` is not defined. A function can be, for example, *associative* or *idempotent*. In a way it is not necessary to make it explicit that functions (e.g., operators) have keywords to indicate that they inhabit a certain property. By just stating the relating axioms in the axioms-section one can define those properties. In CASL there is an option to define those properties by means of attributes. There are four properties (examples for a $\_\_ + \_\_$ operation):

- *associative*: $(x + y) + z = x + (y + z)$ for any x, y or z.
- *commutative*: $x + y = y + x$ for any x, y or z.
- *idempotent*: $x + x = x$ for any x.
- *unit*: this means existence of an identity, $x + 0 = x$ and $0 + x = x$ for any x.

These properties can be expressed by adding one of the following attributes `assoc`, `comm`, `idem`, or `unit` to an operation. If more than one attribute is added, the attributes are separated by comma's.

Operation Syntax
op/ops
<operation name(s)> separated by comma's :
<operation type>
<operation name(s)> separated by comma's :
<operation type> , attributes
<operation name> <operation head>=<term>
opt-semi

Between multiple operation declarations semicolons are used as separators. The first two alternatives of operation declarations are shown in the example. The <operation type> of these two alternatives is defined as follows:

Operation Type

*<sortnames> separated by * → <sortname>*
*<sortnames> separated by * →? <sortname>*
<sortname>
? <sortname>

The third alternative of operation declaration is used to define procedure-like functions. This type of operation definition is very familiar in languages like Pascal and C. The operation head contains possible arguments. These arguments can be used locally in the term. For example, assume we have an infix operation `__ + __`, a possible operation definition would be: `op plus(a : s) : Nat = a + a`. Operation heads can introduce new variables. To state that the operation declaration is not defined for some of the arguments there is a possibility to define partiality and totality of the operation. Partiality is expressed with a **?** after the colon in the arguments, in case of totality there is no question mark.

Operation Head

(<argument declaration(s)>) : <sortname>
: <sortname>
(<argument declaration(s)>) :? <sortname>
:? <sortname>

If there is more than one argument declaration a semicolon is used as separator. Argument declarations are defined as:

Argument Declaration

<variable(s)> : <sort>

Variables (if more than one) are separated by comma's. So, an example with multiple argument and variable declarations could be:

```
. . .
op
 payment(expired         : Date;
         bonus,
         employee_number : Nat ) =

           getsalary(employee_number) + bonus
          when
           get_current_date < expired
          else
           getsalary(employee_number)
. . .
```

This operation states that if the `date` is not expired a bonus will be given to an employee.

3.5.1 Associative

The conjunction `__and__` is associative from a mathematical point of view. In CASL we write this operations as follows:

```
op __AND__ : Bool*Bool -> Bool, assoc
```

Thanks to the `assoc` attribute this definition of `__and__` means the following. It does not matter whether parentheses are used or not. For example,

```
false AND true AND false
```

has two parses:

```
(false AND true) AND false
false AND (true AND false)
```

but by associativity these two parses are equal. This, and the following features, are not automatically present in CASL because there are functions which are not associative.

3.5.2 Commutative

Sticking to our previous example, the conjunction is also commutative. In CASL the operation now looks like:

```
op __AND__ : Bool*Bool -> Bool, assoc, comm
```

Every attribute can be specified by the user self. In this example the code to produce a commutative conjunction would be (`X` and `Y` are variables)

```
X AND Y = Y AND X
```

due to the `comm` attribute at the operation declaration this equation is automatically present.

3.5.3 Idempotent

Our conjunction is also idempotent. Now our function declaration looks like:

```
op __AND__ : Bool*Bool -> Bool, assoc, comm, idem
```

`idem` is equivalent to the equation

```
X AND X = X
```

Again, by stating the `idem` in the operations declaration also this equation is automatically present.

3.5.4 Unit

Our conjunction has `true` as (right and left) identity. In CASL it would look like:

```
op __AND__ : BOOL*BOOL -> assoc, comm, idem, unit true
```

In fact, the attribute `unit true` is equivalent to the following equations

```
X AND true = X;
true AND X = X
```

Since `comm` is also given, mathematically seen only one of the equations is enough. After `unit` a term must be given which defines exactly the identity element.

3.6 Predicate Declarations

Besides total and partial operations another kind of operation is present in CASL: *predicates*. Predicates can be seen as operations with a one valued coarity[9]. Predicates *express* the "truth" but not the negation of the "truth" and therefore a comparison with booleans is not possible. A comparison with a partial function can be made. Assume the only element where the partial function is defined is the "truth", for all other possible elements the function is undefined. This is exactly the situation with predicates, this undefinedness can not be categorized as "not truth" or "truth". One can use predicates to express, for example in Figure 3.1, whether a number is odd or not.

The syntax for predicates starts always with **pred/preds**. A predicate has within CASL the following three forms:

- a list of predicate names with the comma as separator, a colon and a predicate type,
- a predicate name and head related to a formula, or

[9] In fact, at a moment of development of CASL the idea arose to define predicates as follows: S*...*S → truth, where truth as coarity is a kind of reserved sort.

```
spec Natsub =
sorts   pos,nat;
        pos < nat
ops     zero : nat;
        succ : nat -> pos;
pred    odd : nat

var     n:nat
axioms  not odd(zero);
        odd(succ(zero));
        odd(succ(succ(n))) <=> odd(n)
sorts   Odd = {n:nat . odd(n)}
```

Figure 3.1: Predicate example

- only a predicate name with a corresponding formula.

If predicate types are empty, then that is represented by `()`. If they are not empty, then they are represented as a list of sorts separated by `*`. Predicate heads contain, just like the head of an operation, one or more variable(s) declarations of one or more sort types.

Predicate Syntax
pred/preds
<predname(s)> separated by comma's : <predtype>
| *<predname> <predhead>* $\leftrightarrow$ *<formula>*
| *<predname>* $\leftrightarrow$ *<formula>*
opt-semi

Predicates can thus be related to a formula just like in the example below. A formula defines the actual predicate function. In the first case, where no formula is given, a number of predicates can be defined together with their parameters. This form is used when a predicate is not directly linked to a formula. A predicate is related to a formula in the second and third case. For example:

```
pred
    deny(B : BOOL) <=> not B;
op  0 : BOOL;
    1 : BOOL;
axioms
    deny(0);
```

```
spec LIST_WITH_ORDER [sort Elem
                      pred __<__ : Elem*Elem ] =
  free type List[Elem] ::= nil
                       | cons(hd:?Elem; tl:?List[Elem]);
  ops
    insert: Elem*List[Elem] -> List[Elem];
    order: List[Elem] -> List[Elem];
  axioms
    order(nil) = nil;
    order(cons(x,l)) = insert(x,order(l));
    insert(x,nil) = cons(x,nil);
    x < y => insert(x,cons(y,l)) = cons(x,cons(y,l));
    not x < y => insert(x,cons(y,l)) = cons(y,insert(x,l));
  vars x,y:Elem; l:List[Elem]
hide insert
end
```

Figure 3.2: List with order using predicates

By `deny(0)` we state that the negation of `0` is the "truth".

> **Predicate Type Syntax**
> *<sortname(s)> separated by a **
> | *( )*

A predicate head can be seen, just as in operations, as a list of argument declarations. An argument declaration consists of a list of variables separated by comma's with a target sort. If there is more than one argument declaration a semicolon is used as separator.

> **Predicate Head Syntax**
> *(<variable declarations> : <sortname>)*
> | *( )*

Figure 3.2 presents an example of a standard list specification extended with a predicate <. Due to this predicate it is possible to stress that insertion into the list must occur in a specific order.

3.7 Data-type Declaration

The creation of your own data-types is a good way to produce better programs in the sense that they are better readable and they can express things in a more abstract way. Data-types are also present in CASL. A data-type declaration always defines a new sort. There are several ways to specify data-types. In

```
spec BASICS =
free types

    PumpNumber ::= Pump1 | Pump2 | Pump3 | Pump4;

    PumpState ::= Open | Closed ;

    PumpControllerState ::= Flow | NoFlow;

    PhysicalUnit ::= Pump(PumpNumber)
                  | PumpController(PumpNumber)
                  | OutputOfSteam
                  | WaterLevel;

    Modes ::= Initialization | Normal | Degraded | Rescue
          | EmergencyStop
end
```

Figure 3.3: Data-type example (original F. Voisin as appendix in [19])

CASL it is done by referring to either operations (total or partial) or by declaring other (already defined) sorts. See, for example, Figure 3.3.

A data-type declaration is defined in the following way. They always start with the new sort name followed by `::=` and a list of alternatives to define the new sort. There are four possibilities for these alternatives:

- operation name with a list of components between brackets.
- the same as before but with a "?" defining partial functions within components.
- only an operation name.
- `sort/sorts` with a list of sorts indicating that the new sort is an union of the sorts in the comma separated list.

Components are always a list of operator name(s) separated by comma's followed by a colon and sort ("?"just after the colon means also here partial) or only a single sort.

Data-type Syntax
type/types
<*sort*> *::=* <*alternatives*> *separated by* |
opt-semi

Multiple data-type declarations are separated by semicolons.

Alternatives
<operation name> (<components>)
| *<operation name> (<components>)?*
| *<operation name>*
| **sorts/sort** *<sortnames> separated by comma's*

Data-types can thus be defined by a number of sorts or by means of operations[10]. These operations are used to implement some sort constraints which construct the new sort. By using the keyword `free` and `generated` before `type` one can create generated data-types and free data-types.

3.8 Axioms

Basically, we have defined all syntax features of CASL. The next that will be defined are axioms. Axioms are used to express and reason about the variables, operations and predicates. The axiom-section starts with a keyword: `axiom/axioms`.

An axiom is well-formed when it is constructed from atomic formulæ. All the variables occurring in that axiom, are declared either in a quantification or in the specification by means of a variable declaration. If an axiom is well-formed then it determines a sentence of the specification[11]. A formula consists of a combination of quantifiers, logical connectives, atomic formulæ, application of predicates, operations, and sorts.

Axioms
axiom/axioms
| *<formulæ> separated by semicolons*
opt-id

3.8.1 Quantifiers

Quantifiers
<quantifier> list of <variable declarations> .
list of <formulæ>

The list of variables and formulæ are separated by semicolons. A quantification with more than one variable, abbreviates a nesting of quantifications with the same quantifier. The scope of a variable declaration in this quantification is

[10] These operations are called constructors and selector functions.
[11] Remember that *all* variables are universally quantified over every axiom.

the list of formulæ. If a variable declared in the quantification is also declared outside the quantification then the inner variable overrules the outermost one.

3.8.2 Logical Connectives

Logical Connectives

$< formula > \wedge < formula > \wedge ... \wedge < formula >$
$\mid\ < formula > \vee < formula > \vee ... \vee < formula >$
$\mid\ < formula > \rightarrow < formula >$
$\mid\ < formula >$ *if* $< formula >$
$\mid\ < formula > \Leftrightarrow < formula >$
$\mid$ *not* $< formula >$

The logical connectives defines conjunction, disjunction, implication, equivalence, and negation. The conjunction and disjunction are only well-formed when they consist of two or more formulæ.

3.8.3 Atomic Formulæ and Applications

An atomic formula is well-formed if it is well-sorted and expands to an unique formula. This means that only one interpretation is possible for every atomic formula. It can occur that a unique formula does not exist because some predicates or functions are overloaded. In such a case more than one expansion is possible.

(Atomic) Formulæ

true | **false**
| *<predname> with <arguments> between parentheses*
| *<terms>*
| **def** *<terms>*
| *<terms>* **=e=** *<terms>*
| *<terms>* = *<terms>*
| *<term>* **in** *<sort>*
| *(<formula>)*

The atomic formulæ `true` and `false` are always well-sorted they represent basic sentences by themselves. Predicates exist of a single predicate name or a predicate name with arguments. The term behind the keyword **def** is always defined. The equational (=) and existential (=**e**=) equation must have a term on both sides. The **in** keyword defines that a term is of a specific sort. There can always be parentheses around a formula. A formula can be a list of terms defining an application in prefix, postfix, infix, or mixfix notation. A little example of the above mentioned features:

```
pred p : ();
op   a __ : s -> s;
     b    : s;
axioms
   p;          %% predicate axiom
   def a b     %% a b is defined
   a b = true  %% strong equation
   a a b       %% terms in mixfix notation
```

3.9 The Signature

Now that all basic notations have been informally defined a total overview can be given. A CASL specification consists of variable and axiom sections together with a signature consisting of sections defining:

- *sorts*: defines one or more sorts in several ways.
- *operations*: one or more partial and total operations.
- *predicates*: one or more predicates
- *data-type*: defines a data-type as new sort.

Of course, not all notions are fully explained but the basic issues of the CASL-syntax are given. This is needed to understand the next chapters where a mapping from concrete to abstract syntax is defined.

3.10 A Bigger Example

A very informal introduction of the language CASL was given. Some syntax and a bit of the semantics have been defined. A complete formal introduction of CASL can be found in [21]. A bigger example is given to illustrate everything we have seen so far. A quick tour will be made through the example in Figure 3.4.

The example starts with a declaration of sorts `pos` and `nat` where `pos < nat` implies that `pos` is a subsort of `nat`(all sorts are assumed to be predefined). The definition of the operations `zero`, `succ`, and `predec` is defined. `predec` is more interesting because this operation is overloaded. It is defined as a total as well as partial function. The total case manipulates positive numbers, because they have always a predecessor given the fact that `zero` is defined as `nat`. The definition of predecessor on naturals is partial because in our specification there is no predecessor of zero defined.

Predicate `odd` and a variable `n` of sort `nat` are defined. The actual axioms are now defined. The first axiom declares that `zero` is not `odd`, the second declares that the `succ` of `zero` is `odd`. The last one declares that the `odd` of `succ(succ(n))` equals the `odd` of `n`. In natural language these three lines

```
spec Natsub =

sorts   pos,nat;
        pos < nat
ops     zero : nat;
        succ : nat -> pos;
        predec : pos -> nat;
        predec : nat ->? nat;
pred    odd : nat

var     n:nat
axioms  not odd(zero);
        odd(succ(zero));
        odd(succ(succ(n))) <=> odd(n);
        predec(succ(n)) = n
sorts   Odd = {n:nat . odd(n)}
```

Figure 3.4: General example

actual specify: zero is not odd, succ(zero) is odd, and for every other natural number substract two and perform the odd of the result. The definition of the predecessor is straightforward. The predecessor of $n + 1$ is n. The last definition generates a new sort `Odd` by taking `n` and applying `odd(n)`. The predicate `odd(n)` generates all the odd numbers of `nat`.

3.11 Annotations

A special mechanism is introduced in CASL to influence the behaviour of tools: annotations. Annotations are written in the form `%!....%!` or `%[...]`. The annotation may contain extra information which cannot be specified in another way. The tools may use, ignore, or add annotations. In our first CASL-specification (see Section 3.2.1) we could write just behind the `Yell` operation an "output" annotation. This enables some tool to interpret the `Yell` operator as a output function. If the tool is processing the axioms where this operation is used, the output can then be constructed and displayed. The only currently provided annotations are for tagging constructs with labels and for changing the display. An example of label annotation is (`Bool` is predefined):

```
vars m,n : Bool;
axioms
%[double_negation] not(not(m)) = m
%[DeMorgans]       not(m \/ n) = not(m) /\ not(n)
```

The label construct is always standing before a construction. Display annotations are used to change the output. One can specify, e.g., explicit LaTeX(`Nat` is predefined),

```
vars m,n : Nat;
op   __ pow __   : Nat*Nat -> Nat;
     __ min __   : Nat*Nat -> Nat;
     __ times __ : Nat*Nat -> Nat;
axioms
   m pow n = m * m pow ( n - 1 )
   %!<latex> $m^n = m * m ^ ( n - 1 )$%!
```

At the moment only these two annotation forms are available. In the future more forms will be developed. For example, the mixfix notation in CASL might not be parsable for tools. An annotation for mixfix notation might define whether a list of terms is in mixfix notation or not. We might introduce a new annotation form:

```
op succ __   : Nat -> Nat;
   __ + __ : Nat*Nat -> Nat;
   1         : Nat;
var n        : Nat;
axioms
   succ succ n = 1 + succ n %!mixfix%!
```

3.12 Extensions of CASL

It would be a disaster if one has to write the naturals or booleans every time one wants to use CASL. Maybe one wants to extend the basic naturals with prime, powers, etc. To avoid writing the same code over and over again a lot of features are present in CASL. These features are not stable in the sense that the syntax is not yet completely defined. Only a very small overview will be given here.

3.12.1 Structured and Generic Specifications

Structured and generic specifications consist of e.g. extensions, unions, hiding and revealing, etc. The main idea is to structure, reuse, reorganize, and extend specifications. A complete overview of all these features can be found in [19, 25]. For example, to reuse the specifications `Nat` and `Bool` in a `New` specification one could simply specify:

```
spec New = Nat and Bool then
   %% rest of the specification
```

3.12.2 Architectural Specifications and Libraries

An extension of CASL is architectural specifications. This extension enables the structuring of CASL specifications into modules. In fact it is combining several specifications into one unit. For example, one can define an unit `arithmetic` by combining the CASL specifications `naturals`, `booleans`, `reals`, etc. The libraries extension extends CASL so that the modules (units) can be used over a distributed network. Both architectural and libraries are very tentative for the moment.

3.13 Final Remarks

This chapter introduced most of the syntactic issues of the so called Basic Specifications of CASL this is the starting point for defining the concrete and abstract syntax of CASL in SDF which will be presented in the next chapter. From this point only the Basic Specifications are considered, the extensions of CASL are left out.

Chapter 4

Concrete → Abstract

In the previous chapter we have given an introduction to the concrete syntax of CASL. The main question of this thesis is whether it is possible to convert the concrete CASL syntax, restricted to the sublanguage Basic Items, to abstract CASL syntax using ASF+SDF? The two main reasons why such a conversion is necessary are:

- To develop or reuse existing tools, to support and process CASL.
- Tools developed by different groups interact with each other at the level of abstract syntax. Tools like parsers, typecheckers, and rewrite engines, can be "connected" by means of the abstract syntax.

The previous introduction to the concrete syntax of CASL is useful since we are converting this concrete syntax to abstract syntax. In this chapter two things are defined to specify this conversion:

1. Since we are using ASF+SDF as development environment we are forced to specify the CASL grammar in SDF. This will be done by presenting general rules for the translation of the concrete and abstract CASL grammar presented in Appendices A and B to SDF.
2. The actual conversion can then be done by using ASF+SDF since abstract and concrete grammar are specified in SDF. General rules will be presented to establish the conversion.

So, we have to convert the concrete and abstract CASL grammar (G) into another one: an SDF[1] grammar (H). Because both grammars use different grammar constructs a one to one translation is not possible.

Assume we have a correct CASL specification $w \in L(G)^2$. To specify G in H the following property must be preserved:

$$\forall w : w \in L(G) \Leftrightarrow w \in L(H) \tag{4.1}$$

[1] Syntax Definition Formalism.

[2] L means Language, $L(G)$ is the language L generated by grammar G.

To create such H, every production rule of G must in some way be translated into production rules in H. If it is possible to specify exactly every production rule form G in H then 4.1 is preserved. However, some grammar constructs used in G may not be present in H. It is very well possible that one production rule $p \in G$ is presented by $q_1 \ldots q_n \in H$ $n \geq 1$. Assume that p recognizes a part of the language $L(G)$ (notation $L(p)$), then the following must hold:

$$L(p) \Leftrightarrow L(q_1 \ldots q_n) \tag{4.2}$$

The expressiveness of grammar constructs of H may be bigger than G and therefore it is possible to express $p_1 \ldots p_n \in G$ in $q \in H$ Again, there must hold:

$$L(p_1 \ldots p_n) \Leftrightarrow L(q) \tag{4.3}$$

In other words, the language generated by G must be the same language that H generates.

$$L(G) \Leftrightarrow L(H) \tag{4.4}$$

The transformation is not correct when this equation fails.

4.1 Grammar Rules

The original CASL grammar (G) defines production rules in the following BNF[3]-like way. N is a nonterminal in uppercase-words with hyphens; t a terminal not in uppercase and allows all other characters. The ::= is used as meta-notation like in BNF:

$$N ::= N_1' t_1 \ldots N_n' t_m \quad m, n \geq 0$$

If m or n equals 0 there is, respectively, no t nor N present. If m and n equal 0 this represents the empty case $N ::=$ which is not present in our grammar G. The grammar has only one nonterminal on the left and thus no productions of the form $N_1 \ldots N_k ::= \ldots$[4] exists, we can directly map each production rule in SDF **context-free syntax** via the conversion:

$$N ::= N_1' t_1 \ldots N_n' t_m \;\Rightarrow\; N_1' \text{"} t_1 \text{"} \ldots N_n' \text{"} t_m \text{"} \rightarrow N \tag{4.5}$$

Just like in BNF is | a choice. In G it is possible to write $N ::= t \mid t' \mid \ldots$ which is a short hand for $N ::= t, N ::= t', \ldots$. The first notation can not be expressed in SDF, but the second can be expressed in SDF (z is a possible mixture of one or more terminals and nonterminals and must be transformed using the above mentioned equations before applying)

$$N ::= z \mid z' \mid \ldots \;\Rightarrow\; z \rightarrow N, z' \rightarrow N, \ldots \tag{4.6}$$

[3]Backus-Naur Form.

[4]In fact $N_1 \ldots N_k ::= \ldots$ is context-sensitive while G and H are context-free.

4.2 Special Grammar Constructs

In G there are some special constructs. Every possible construct in G must have an equivalent in H.

- A string of characters surrounded by quotation marks "..." always stands for the enclosed characters themselves.
- $N\ t \ldots t\ N$ describes one or more repetitions of N separated by t
- $N \ldots$ is simply one or more repetitions of N
- For example **var/vars** or **pred/preds** indicates that the singular or plural form may be used randomly. **vars** does not imply more than one and **var** does not imply one. Furthermore there is a special case `;/` which expresses an optional semicolon.

If strings are surrounded by quotation marks there are two possibilities:

- The string "$c_1 \ldots c_n$" between quotation marks should be seen as terminals. In H that would be directly "$c_1 \ldots c_n$"

$$"c_1 \ldots c_n" \Rightarrow "c_1 \ldots c_n" \tag{4.7}$$

- In G there is only one character which forms a separator for a list. SDF allows a more complex literal as list separator. The list-transformation from G to G' is therefore no problem and quite straightforward. If we have $N"."\ldots"."N$ then we directly transform the "." into a separator (see transformation of lists with separators below)

In G there are two list-constructs which can be directly transformed into SDF:

$$N\ t \ldots t\ N \Rightarrow \{N\ "t"\ \}+ \tag{4.8}$$

$$N \cdots \Rightarrow N+ \tag{4.9}$$

Examples: $N"."\ldots"."N \Rightarrow \{N\ "."\ \}+$ and $N; \ldots ; N \Rightarrow \{N\ ";"\ \}+$ The other special constructs are translated naively (in all cases must z be transformed using the above mentioned rules)

- v/vs will be defined as a special nonterminal in H so that every occurrence of v/vs will be substituted by that nonterminal.

$$N ::= v/vs\ z \Rightarrow V\ z \rightarrow N \tag{4.10}$$

  where V is defined as "v" $\rightarrow V$, "vs" $\rightarrow V$.

- The optional semicolon ;/ can be defined in almost the same way as v/vs. We create a new nonterminal O with the rules: $\rightarrow O$, ";" $\rightarrow O$.

$$N ::= z;/ \Rightarrow zO \rightarrow N \tag{4.11}$$

  where O is thus $\rightarrow O$, ";" $\rightarrow O$.

Every nonterminal on the right of $\rightarrow$ must be defined in the **sorts** of the SDF specification (R are rules, β expression on the left-hand side of a SDF rule, S are sorts):

$$\begin{array}{lcl} \textbf{sorts} & & \textbf{sorts} \\ \quad S^* & & \quad S^* N \\ \textbf{context}-\textbf{free syntax} & \Rightarrow & \textbf{context}-\textbf{free syntax} \\ \quad R_1^*\ \beta\ \rightarrow\ N\ R_2^* & & \quad R_1^*\ \beta\ \rightarrow\ N R_2^* \end{array} \tag{4.12}$$

$$R_1^*\ \beta \rightarrow N\ R_2^*\ \beta \rightarrow N\ R_3^* \Rightarrow R_1^*\ \beta \rightarrow N\ R_2^*\ R_3^* \tag{4.13}$$

Rules that are already present in the grammar H should not be redefined, this causes ambiguities.

$$\textbf{sorts}\ S_1^*\ S\ S_2^*\ S\ S_3^* \Rightarrow S_1^*\ S\ S_2^*\ S_3^* \tag{4.14}$$

4.3 Example of Translating to SDF

Taking the following small selection of the CASL grammar G,

```
(1) BASIC-SPEC   ::= BASIC-ITEMS... | { }
(2) BASIC-ITEMS  ::= var/vars VAR-DECL;...;VAR-DECL
                     . FORMULA"."..."."FORMULA ;/
```

by using the equations presented in Section 4.2 this can easily be transformed to SDF. Since the choice operator `|` can not be expressed directly in SDF, we use rule (4.6) on `BASIC=SPEC ::= BASIC-ITEMS...  |  :`

```
(1) BASIC-SPEC   ::= BASIC-ITEMS...
(2) BASIC-SPEC   ::= { }
(3) BASIC-ITEMS  ::= var/vars VAR-DECL;...;VAR-DECL
                     . FORMULA"."..."."FORMULA ;/
```

Using (4.9) on (1); (4.5) with $n = 0$, $m = 2$ on (2) ; (4.10),(4.8), (4.8), (4.11) all on (3), and (4.12), (4.13), (4.14) produces the SDF specification in Figure 4.1. By applying the equations defined in Section 4.2 the total grammar G, will produce an ASF+SDF grammar H which preserves (1.1).

4.4 Defining Priorities

A grammar is ambiguous if it is possible to parse a sentence in two or more ways using different production rules. By using priorities in SDF these ambiguities can be solved. Priorities define that a production rule has a higher precedence than another one. The grammar G has definitely some ambiguities of the above mentioned kind. To keep the target grammar H so compact and readable as possible it is advisable to keep the priority sections as small as possible. To

```
sorts
   BASIC-ITEMS BASIC-SPEC KEYWORD-VAR
   VAR-DECL FORMULA OPT-SEMI
context-free syntax

BASIC-ITEMS+                              -> BASIC-SPEC
"{" "}"                                   -> BASIC-SPEC
KEYWORD-VAR {VAR-DECL ";"}+
     "." {FORMULA "."}+ OPT-SEMI          -> BASIC-ITEMS
"var"                                     -> KEYWORD-VAR
"vars"                                    -> KEYWORD-VAR
";"                                       -> OPT-SEMI
                                          -> OPT-SEMI
```

Figure 4.1: Translated (CASL) grammar in SDF

denote that production rule X has a higher precedence then Y we write $X > Y$ or $Y < X$.

There is another way of creating priorities in SDF by using priority levels in an Algol68-like way (see [29]). Assume that we have a specification based on boolean logic[5]:

```
(1) "true"                             -> FORMULA
(2) "false"                            -> FORMULA
(3) FORMULA "if" FORMULA               -> FORMULA
(4) FORMULA "/\\" {FORMULA "/\\"}+  -> FORMULA
```

It is obvious that (1) and (2) have a lower priority than (3) and (4); (4) has a lower priority than (3). Grouping every production rule with the same priority gives us (3) > (4) > (2,1). By giving each group of equal priority a new nonterminal we are able to separate the groups. If $a_1 \ldots a_n \ldots z_1 \ldots z_n$ are production rules with priority $(a_1 \ldots a_n) > \cdots > (z_1 \ldots z_n)$ we give every group a new nonterminal as result sort say $A \ldots Z$ All production rules had the same result sort RS. Transformation will be:

$$a_1 \ldots a_n \to RS, \ldots, z_1 \ldots z_n \to RS \Rightarrow a_1 \ldots a_n \to A, \ldots, z_1 \ldots z_n \to Z \tag{4.15}$$

It is obvious that $A > \cdots > Z$ and that apparently all production rules had the same result sort. There is a way to express Z in Y ... in A by just stating the relations. The highest priority set (in our case A) must be expressed in the original result sort.

$$Z \to Y, Y \to X, \ldots, B \to A, A \to RS$$

[5]This specification is taken since it is a subset of CASL.

If we use this technique with some new, well-chosen nonterminals we get:

```
(1) FORMULA-1                          -> FORMULA
(2) FORMULA-2                          -> FORMULA-1
(3) ATOM                               -> FORMULA-2
(4) "true"                             -> ATOM
(5) "false"                            -> ATOM
(6) FORMULA "if" FORMULA               -> FORMULA-1
(7) FORMULA "/\\" {FORMULA "/\\"}+     -> FORMULA-2
```

The techniques presented here are only used when there are three or more production rules of the above mentioned structure. If there are less then three production rules the expansion of the grammar is bigger than stating the priorities in the priority section. These priorities will be defined in the priority section. The solution here is cheaper as well. If we used a priority rule, a part of the parse tree is built but never used.

4.5 Conversion from Concrete to Abstract

The specification of the original CASL grammars in SDF was done in the previous sections. The conversion of concrete to abstract CASL syntax can therefore be specified in ASF+SDF. Such a conversion function requires the specification of the syntax of the conversion function (SDF) and its semantics (ASF).

So, the conversion from concrete to abstract syntax is mainly a translation between two grammars: how can one be expressed in terms of the other one? If one takes two grammars such a translation might not be possible at all. The difference is that CASL is defined in terms of abstract syntax and that the concrete syntax of the language is derived from the abstract one[6]. So, there *is* a logical connection between the concrete and abstract grammar. Demonstrating how to translate in ASF+SDF a subset (TINY-CASL) of the concrete and abstract syntax of Basic Items will be considered. The following questions should be answered:

- Is it possible to make a conversion function *c2a* in ASF+SDF from concrete to abstract syntax?
- If such a conversion function is possible, what information of the parsed program P do we lose? If none, is it possible to write a bijective function:

$$\forall P : P = c2a(c2a^{-1}(P)) \tag{4.16}$$

- What is the easiest way to express such a conversion function in ASF+SDF?

Before actually investigating the possibility of creating such a conversion function a few remarks must be made:

[6] In January 1998, Peter Mosses and the author, derived the concrete syntax from the abstract syntax.

- The implementation of our abstract and concrete TINY-CASL syntax can be done using the techniques presented in the previous sections.
- In order to write the conversion function, it is necessary to introduce variables for every nonterminal in the concrete (and abstract) syntax. These variables are generated according to the following guidelines (N is a nonterminal)
  - There exist a variable for every ordinary nonterminal. The name of the variable is the name of the nonterminal by converting *uppercase* letters, to *lower-case*.

    $$N \Rightarrow n \rightarrow N \tag{4.17}$$

  - Every list variable without separators is generated as in (4.17) but with the concatenation of, respectively, *star* or *plus*.

    $$N+ \Rightarrow nplus \rightarrow N + \tag{4.18}$$

    $$N* \Rightarrow nstar \rightarrow N * \tag{4.19}$$

  - Every list variable with separator is generated by using (4.18) and (4.19). Only if there is more than one list construction with the same nonterminal but with different separators, the textual representation of the separator will be inserted in the variable name. This assures uniqueness for every variable.

    $$\{\ N\ ","\ \}+ \Rightarrow ncommaplus \rightarrow \{\ N\ ","\ \}+ \tag{4.20}$$

    $$\{\ N\ "."\ \}* \Rightarrow nbulletstar \rightarrow \{\ N\ "."\ \}* \tag{4.21}$$

  - The nonterminals in the ASF+SDF specification of TINY-CASL have the following form:
    * If we deal with a *concrete* nonterminal the nonterminal name starts with a capital, the rest is in lower-case. A capital is also allowed after a hyphen. The rest is always in lower-case.
    * An *abstract* nonterminal is *always* in capitals.
- We assume that the lexical syntax of CASL is present. The grammar of the lexical syntax can be found in [20] Chapter 4. The translation of this grammar into an appropriate ASF+SDF-module can be done by using the techniques presented in Chapter 4.

Assume we have an ASF+SDF representation of concrete (C) and abstract (A) TINY-CASL. We want to build a function $c2a$ so that the following holds:

$$c2a(C) = A \tag{4.22}$$

```
BASIC-SPEC ::= basic-spec BASIC-ITEMS+
BASIC-ITEMS::= var-items VAR-DECL+
             | SIG-ITEMS
SIG-ITEMS  ::= sort-items SORT-ITEM+
             | op-items OP-ITEM+
SORT-ITEM  ::= sort-decl SORT+
OP-ITEM    ::= op-decl OP-NAME+ OP-TYPE OP-ATTR*
OP-TYPE    ::= total-op-type   SORTS SORT
OP-ATTR    ::= associative | commutative
SORTS      ::= sorts SORT*
VAR-DECL   ::= var-decl VAR+ SORT
OP-NAME    ::= ID
SORT       ::= ID
VAR        ::= SIMPLE-ID
```

Figure 4.2: Abstract Tiny-Casl

```
Basic-Spec ::= Basic-Items...
Basic-Items::= var/vars Var-Decl;...
             | Sig-Items
Sig-Items  ::= sort/sorts Sort-Item;...
             | op/ops Op-Item;...
Sort-Item  ::= Sort,...,Sort
Op-Item    ::= Op-Name,...,Op-Name : Op-Type
Op-Item    ::= Op-Name,...,Op-Name : Op-Type
               , Op-Attr,...,Op-Attr
Op-Type    ::= Sorts -> Sort | Sort
Op-Attr    ::= assoc | comm
Sorts      ::= Sort*...*Sort
Var-Decl   ::= Var,...,Var : Sort
Op-Name    ::= Id
Sort       ::= Id
Var        ::= Simple-Id
```

Figure 4.3: Concrete Tiny-Casl

```
context-free syntax

  "basic-spec" BASIC-ITEMS+             -> BASIC-SPEC
  SIG-ITEMS                             -> BASIC-ITEMS
  "var-items" VAR-DECL+                 -> BASIC-ITEMS
  "sort-items" SORT-ITEM+               -> SIG-ITEMS
  "op-items" OP-ITEM+                   -> SIG-ITEMS
  "sort-decl" SORT+                     -> SORT-ITEM
  "op-decl" OP-NAME+ OP-TYPE OP-ATTR*   -> OP-ITEM
  "total-op-type" SORTS SORT            -> OP-TYPE

  associative            -> OP-ATTR
  commutative            -> OP-ATTR
  "sorts" SORT*          -> SORTS
  "var-decl" VAR+ SORT   -> VAR-DECL

  Id              -> SORT
  Id              -> OP-NAME
  Simple-Id       -> VAR
```

Figure 4.4: Abstract Tiny-Casl in ASF+SDF

```
context-free syntax
  Basic-Items+                      -> Basic-Spec
  Sig-Items                         -> Basic-Items
  Var-S {Var-Decl ";"}+             -> Basic-Items
  Sort-S {Sort-Item ";"}+           -> Sig-Items
  Op-S {Op-Item ";"}+               -> Sig-Items
  { Sort "," }+                     -> Sort-Item

  { Op-Name "," }+ ":" Op-Type "," {Op-Attr ","}+-> Op-Item
  { Op-Name "," }+ ":" Op-Type                   -> Op-Item

  Sorts "->" Sort        -> Op-Type
  Sort                   -> Op-Type
  {Sort "*"}+            -> Sorts
  "assoc"                -> Op-Attr
  "comm"                 -> Op-Attr

  {Var ","}+  ":" Sort   -> Var-Decl
  Id              -> Sort
  Id              -> Op-Name
  Simple-Id       -> Var
```

Figure 4.5: Concrete Tiny-Casl in ASF+SDF

4.6 A First Approach: Tiny-Casl

The grammars in Figures 4.2 and 4.3 use the same grammar constructs as presented in the Appendices A and B. These two grammars can be directly converted into ASF+SDF using the techniques introduced in the previous sections. The results are shown in Figure 4.4 and 4.5. The **sorts** section is left out because it is trivial. Comparing the two ASF+SDF-grammars in Figures 4.4 and 4.5:

- Both grammars are tightly linked to each other, there is *almost* an one to one correspondence.
- Linking each grammar rule of the concrete syntax to the abstract syntax can not be done automatically, this requires some human intervention.

4.6.1 A First Mapping Function

We need a function that is capable of dealing with every nonterminal or list nonterminal when processing the concrete grammar. If we have such a function *c2a* (4.22) then every nonterminal or list nonterminal could occur as input parameter. The output is a bit more complicated. It is necessary to define precisely which abstract output (list) nonterminal corresponds to a concrete one. Assume that the abstract and concrete syntax of Tiny-Casl are defined in ASF+SDF. A straightforward definition of the *c2a* is possible. N is a nonterminal[7], t is a terminal, c and a means concrete and abstract.

The concrete ASF+SDF function looks like:

$$N_1^{'c} t_1^c \dots N_m^{'c} t_n^c \rightarrow N^c \tag{4.23}$$

The abstract ASF+SDF function looks like:

$$N_1^{'a} t_1^a \dots N_x^{'a} t_y^a \rightarrow N^a \tag{4.24}$$

If m or n is zero then no t or N is present. If m, x and n, y are zero no N and t are present. It is almost obvious that our *c2a* conversion function could be generated by:

$$c2a(N^c) \rightarrow N^a \tag{4.25}$$

However, the decision which N^c corresponds to N^a can *not* be made automatically.

So far we have defined the syntax structure of the translation function. In order to define the actual translation we add the semantics to these functions by means of (ASF) equations. To add the semantics to (4.25) we use:

[7] N represents either a normal nonterminal or one of the following list-nonterminals: $N+$, $N*$, {N "t" }* or {N "t" }+.

```
context-free syntax
  c2a(Basic-Spec)   -> BASIC-SPEC
  c2a(Basic-Items+) -> BASIC-ITEMS+

variables
  basicitem+        -> Basic-Items+

equations

[1] c2a(basicitemsplus) = basic-spec c2a(basicitemsplus)
```

Figure 4.6: A first conversion function

- (4.17), (4.18), (4.19), (4.20) and (4.21) to define all the variables needed.
- the assumption that all *c2a* functions are present on every N^c using (4.25) in combination with (4.23) and (4.24)

This will produce an ASF equation where N_i^{vc} represents variables for the non-terminal N_i^c.

$$c2a(N_1^{'vc}\ t_1^c \ldots N_m^{'vc}\ t_n^c) = c2a(N_1^{'vc})\ t_1^a \ldots c2a(N_m^{'vc})\ t_y^a \tag{4.26}$$

4.6.2 Conversion Example

Assume we have the following abstract context-free syntax rule of Tiny-Casl:

```
"basic-spec" BASIC-ITEMS+ -> BASIC-SPEC
```

and its corresponding concrete context-free syntax rule:

```
Basic-Items+                -> Basic-Spec
```

The ASF+SDF functions, variables, and equations are defined by using (4.25) and (4.26). We have a very quick way of defining a conversion function if this approach succeeds. However, there are two big objections:

- The function *c2a* is heavily overloaded. For every N^c there is a function declaration of *c2a*. Such amount of overloading of one function quickly leads to ambiguities. Already in our little example in Figure 4.6 ambiguities arise. Imaging we have a list of two `Basic-Items`. The abstract syntax of these two `Basic-Items` should contain only one terminal `basic-spec` but since one `Basic-Items` is also a `Basic-Spec` the two `Basic-Items` can be iterated over by the function `c2a(Basic-Spec) -> BASIC-SPEC` which yields then exactly two times "`basic-spec`". The main problem is that in almost all cases the abstract syntax uses a terminal whereas the corresponding concrete syntax does not. It is possible to avoid these ambiguities. This can be done by introducing new function names for every nonterminal that has to be converted.

- ASF+SDF has the property that result sorts can not be lists. To allow a function to have a list as result sort, one should introduce a new nonterminal which contains the needed list. So, our example in Figure (4.6) is not a correct ASF+SDF specification and therefore it can not be parsed at all.

4.6.3 Removing Ambiguities

The following adjustments are made to our little example in (4.6):

- The name of the conversion function uses the nonterminal to be converted in such a way that every *c2a* is unique. We have to expand (4.25) and (4.26) in a straightforward way. The name of every conversion function *c2a* is appended by a hyphen and the nonterminal correspond to the input parameter. List nonterminals use as extra extension the words `list`, `star` or `plus`. If there exists lists with the same nonterminal but with different separators, the name of the separator will make each function unique. The extension of the *c2a* function in this way is denoted $c2a - L$.

- A list can only be introduced as a result sort by introducing a new abstract nonterminal representing that list. That new nonterminal can then be used as result sort. To use lists on the right-hand side of an equation a new variable has to be introduced for that list nonterminal in combination with a conditional equation structure. Of course, the new abstract nonterminal must occur in the **sorts** definition of the abstract syntax module. The new nonterminal represents a list and is therefore build on the list nonterminal with the addition of `LIST`, `STAR`, or `PLUS`. `LIST` is used to give somewhat more freedom in specifying the function name and gives thus a bit syntactic freedom. As we are dealing with abstract nonterminals no lists with separators are present. The expanded nonterminals and the normal nonterminals will be denoted as K.

- The new introduced variables ranging over the abstract syntax are built from at least one or more capital characters. This property establishes the distinction between concrete and abstract variables. Concrete variables are completely in lower-case while abstract variables have at least one capital character. The abstract variables are denoted as K^v.

These observations lead to extension of (4.25) and (4.26). For every new introduced K an extra grammar rule must be appended to the abstract syntax. In this case is $N\alpha$ the corresponding list notation of list nonterminal N^a[8]. So

[8] As mentioned before N is an ordinary nonterminal or a list nonterminal. $N\alpha$ is only a list nonterminal.

```
context-free syntax
  c2a-Spec(Basic-Spec)            -> BASIC-SPEC
  BASIC-ITEMS*                    -> BASIC-ITEMS-LIST
  c2a-Basic-Items(Basic-Spec)     -> BASIC-ITEMS-LIST

variables
  bspec           -> Basic-Spec
  BITEMlist       -> BASIC-ITEMS*

equations

[basic-spec-1]
        c2a-Basic-Items(bspec) = BITEMlist
        ======================================
        c2a-Spec(bspec) = basic-spec BITEMlist
```

Figure 4.7: Conditional approach

$N^a\alpha$ is either N^a+ or N^a*.

$$N^a\alpha \rightarrow K \tag{4.27}$$

Beware, the extension of concrete nonterminals L only effect the name of the conversion functions while (4.27) appends extra functionality to the abstract syntax. Future list variables ranging over the abstract syntax are *not* of type K but of type $N^a\alpha$.

Adjusted ASF+SDF function and equation

$$c2a - L(N^c) \rightarrow K \tag{4.28}$$

$$\frac{\begin{array}{c} c2a - L_1^{'}(N_1^{'vc}) = K_1^{'v}, \\ \vdots \qquad\qquad \vdots \\ c2a - L_m^{'}(N_m^{'vc}) = K_m^{'v} \end{array}}{c2a - L(N_1^{'vc}t_1^c \ldots N_m^{'vc}t_n^c) = K_1^{'v}t_1^a \ldots K_m^{'v}t_y^a} \tag{4.29}$$

4.7 Conditional Approach

What are the effects of the observations made in the previous section on our example? We disambiguate all conversion functions by renaming them. The equations introduce new variables ranging over the abstract syntax because ASF+SDF can not directly produce a list as result sort. Our example is presented in Figure 4.7. The effect on TINY-CASL is that

- For almost every equation this conditional form should be applied (see 4.29). The exceptions are when the conversion function is only producing abstract terminals.
- The number of variables explode in the sense that at least for every $N^a\alpha$ a variable should be introduced. In the future we could use these variables ranging over the abstract syntax while specifying a conversion from abstract syntax to, for example, an intermediate CASL representation.
- Besides the *c2a* function for regular nonterminals a new kind of *c2a* functions should be applied. These functions handle lists. So for every nonterminal used within a list, a separate *c2a* function should be written ($c2a - L$). They have for example the following form:

  ```
  c2a-Basic-Item-List(Basic-Item*) -> BASIC-ITEM-LIST
  ```
- The user can not directly see what is going on. There is no overview due to the huge amount of conditional equations.

4.7.1 Applicability of Conditional Equations

Almost all equations will be conditional equations (80-90%). We can reduce some conditional equations to ordinary ones when no list variables are used on the right-hand-side. As this reduces so little the amount of conditional equations and it increases complexity of our conversion function, this step will not be taken. Assume we have a way of specifying our equations in a non-conditional way (*linear approach*). If this is possible we have to change our abstract grammar in the sense that no lists are directly on the right-hand-side of the equations and we are still able to iterate over those lists.

approach	*nb. equations*	*plus*	*minus*
conditional	≈ 8,5 pag.	abstract variables used	missing overview
linear	≈ 3 pag	good overview	grammar changes

The number of pages are counted in the following way. Every concrete syntax has an average of two nonterminals on the right. Every nonterminal is handled in the conditional way on one line (see 4.29). Adding three lines for:

- the original equation,
- equation line (=================),
- empty line

This gives 5 lines for each concrete grammar rule. We have 84 concrete syntax rules[9]. For conditional equations this will be $5 \times 84 = 420$ lines in total and,

[9] Only the concrete syntax rules of Basic Items are counted.

with an average of 50 lines per page, 8,5 pages. In the linear way we have one line for the equation and one for an empty line. This leads to $2 \times 84 = 168$ lines in total, so about 3 pages.

The choice to be made is between a good overview but a non-orthodox concept by changing the target grammar *or* a bad overview but an orthodox concept. If both concepts are working correctly in practice the only advantage of the conditional approach is the possible disadvantage of changing the grammar in the linear case and the reuse of abstract variables.

4.8 Changing the Grammar in Favor of ASF+SDF

By abandoning the conditional approach we are in fact back at rules (4.25) and (4.26). The only difference is that:

- An expansion of abstract list nonterminals should be made. This is just K.
- The abstract grammar should be adapted. This is almost done by (4.27) and (4.28) . The only problem is that we are not able to iterate over lists.

The abstract syntax is changed in the following way. First every list nonterminal is rewritten (K). New functions are added to the abstract syntax as in (4.27). Future abstract list variables will still use $N^a\alpha$ as type. In the previous section we made the selection of conditional equations because we were not able to iterate over lists. Apparently that is the core of the problem. It is not possible to iterate over the abstract list nonterminals generated as in (4.27). There is only an injection into K and this does not imply an injection from K back to the original list representation. We can expand (4.27) with an extra grammar rule that implies lists iteration.

$$\begin{aligned} &N^a\alpha \rightarrow K \\ &N^aK \rightarrow K \end{aligned} \tag{4.30}$$

In abstract CASL syntax no lists with separators are present. So, by giving one example the rest follows automatically. We had an original context-free abstract syntax rule:

```
"basic-spec" BASIC-ITEMS+      -> BASIC-SPEC
```

The nonterminal `BASIC-ITEMS+` with the list operation (α was $+$ or $*$) is renamed to `BASIC-ITEMS-LIST`. The definition of `BASIC-ITEMS-LIST` is defined exactly in (4.30). This results in a changed abstract grammar:

```
"basic-spec" BASIC-ITEMS-LIST       -> BASIC-SPEC
BASIC-ITEM+                         -> BASIC-ITEM-LIST
BASIC-ITEM BASIC-ITEM-LIST          -> BASIC-ITEM-LIST
```

```
context-free syntax
 c2a-Spec(Basic-Spec)                   -> BASIC-SPEC
 c2a-Basic-Items-List(Basic-Spec)       -> BASIC-ITEMS-LIST
 c2a-Basic-Items(Basic-Items)           -> BASIC-ITEMS
 c2a-Var-Decl-List({Var-Decl ";"}+)     -> VAR-DECL-LIST
 c2a-Var-Decl(Var-Decl)                 -> VAR-DECL
 c2a-Var-List({Var ","}+)               -> VAR-LIST
 c2a-Var(Var)                           -> VAR
 c2a-Sort(Sort)                         -> SORT

variables
 bspec          -> Basic-Spec
 bitem          -> Basic-Items
 bitems         -> Basic-Items+
 varkeyword     -> Var-S
 vardecls       -> {Var-Decl ";"}+
 vardecl        -> Var-Decl
 varlist        -> {Var ","}+
 sortname       -> Sort
 varr           -> Var
```

Figure 4.8: TINY-CASL SDF conversion function

Of course, the new nonterminal `BASIC-ITEM-LIST` is assumed to added to the **sorts** section.

It is possible that the two rules in (4.30) can cause ambiguities. To avoid this, the two rules will be appended to the priority section of the abstract syntax. The second one has higher priority than the first one. These priorities are in fact very expensive[10]. The extension of the nonterminals leads to expansion of (4.26):

$$c2a - L(N_1^{vc} t_1^c \dots N_m^{vc} t_n^c) = c2a - L'(N_1^{vc}) t_1^a \dots c2a - L''(N_m^{vc}) t_y^a \tag{4.31}$$

4.9 TINY-CASL Applied

Extending the abstract SDF definition of TINY-CASL in Figure 4.4 is straightforward with (4.30). To write a conversion function for TINY-CASL one should keep in mind that some abstract list nonterminals are changed and that (4.28) is applied. In Figure 4.8 contains the SDF definition of the conversion functions. The new equation in (4.31) generate the equations of Figure 4.10. An example TINY-CASL specification, say `c2a-Spec(var a,b : R;a,b : R)` results in Figure 4.9. The approach for all Basic Items is:

[10] Expensive in the sense that two parse trees are built, while only one is used.

```
basic-spec var-items var-decl
                        a b
                        R
                     var-decl
                        a b
                        R
```

Figure 4.9: Result of `c2a-Spec(var a,b : R;a,b : R)`

```
equations
[1] c2a-Spec(bspec) =
            basic-spec c2a-Basic-Items-List(bspec)
[2] c2a-Basic-Items-List(bitem) =
            c2a-Basic-Items(bitem)
[3] c2a-Basic-Items-List(bitem bitems) =
            c2a-Basic-Items(bitem) c2a-Basic-Items-List(bitems)
[4] c2a-Basic-Items(varkeyword vardecls) =
            var-items c2a-Var-Decl-List(vardecls)
[5] c2a-Var-Decl-List(vardecl) =
            c2a-Var-Decl(vardecl)
[6] c2a-Var-Decl-List(vardecl ; vardecls)=
            c2a-Var-Decl(vardecl) c2a-Var-Decl-List(vardecls)
[7] c2a-Var-Decl(varlist : sortname)=
            var-decl c2a-Var-List(varlist) c2a-Sort(sortname)
[8] c2a-Var-List(varr) = c2a-Var(varr)
[9] c2a-Var-List(varr , varlist) =
            c2a-Var(varr) c2a-Var-List(varlist)
[10] c2a-Var(varr) = varr
[11] c2a-Sort(sortname) = sortname
```

Figure 4.10: TINY-CASL ASF conversion function

1. (4.28), this defines all the SDF definitions of the conversion function.

2. (4.31), this defined all the semantics of the conversion function.

3. the assumption that the abstract syntax is expanded with (4.30) and appropriate variables and priorities. In all cases the assumption is made that we deal with K in the abstract syntax. All new introduced nonterminals in (4.30) are explicitly mentioned in the **sorts** sections.

4.10 Converting Basic Specifications

Now there is a way of converting from concrete to abstract TINY-CASL and the actual specification can be done concerning Basic Specifications of CASL. So, time is come to check whether the observations have some impact. Some side effects are discovered by using the current approach with the Basic Items:

- Every keyword in Basic Items should, according to the previous chapter, have a variable with the keyword name in lower-case. Sometimes these keywords in lower-case clash with actual CASL keywords. Therefore we define that *every* keyword variable starts with the text "keyword".[11]

- The conversion of `TERM`s is not straightforward. There is not enough information to decide how to convert the concrete `TERM` to abstract syntax. There is need of information about the defined variables (local or not) and operations. To acquire this information a pre-parse of the specification should be made where all variables and operations are collected. CASL terms can be in mixfix notation; the parse depends on the definition of operations and variables. So, to make a conversion of concrete terms to abstract syntax a special parser should be written for those terms. Modulo `TERM`s the current approach works well. These `TERM`s will be parsed using a mixfix parser presented in Chapter 6.

- Since optional semicolons are not of semantic relevance, they do not occur in the abstract syntax. Therefore, it is impossible to create a bijective *c2a*-function on a syntactic level: we just do not know where to put the optional semicolons. It is also possible to add unlimited parenthesizes in terms, they will also not be present in the abstract syntax.

- During the development of the *c2a*-function an implementation of *c2a* for TINY-CASL is made for the conditional approach and the linear approach. No difference in delays or huge memory consumptions are measured. For the grammars used, both approaches work fine in ASF+SDF. For overview reasons is chosen for a linear approach.

[11] This is why in Figures 4.10 and 4.8 the variable `varr` is with double r. It clashes with the keyword `var`.

4.11 Conclusions

If a mixfix parser can be written the full conversion from concrete to abstract will not be a problem. During the conversion phase we do lose some information. The optional semicolons used in Basic Specifications are completely discarded by the conversion. The parsing of mixfix terms discards possible parenthesizes. This happens when the parenthesizes have no actual meaning in the term. So, (4.16) does not hold on a syntactic level. The easiest way to express a conversion function is presented along the road and can be summarized in:

- Rules (4.30), (4.28), (4.31).
- The adaptation of sorts, variables and priorities described in the previous sections.
- Generation of $c2a - L$ functions and the use of K in abstract syntax.

Chapter 5

Parsing Mixfix

In Chapter 5 we presented an almost complete conversion from concrete to abstract syntax. In this chapter we describe a first approximation to mixfix parsing that turns out to be imperfect in certain ambiguous cases. In Chapter 6 we will correct this defect.

If we are able to specify a mixfix parser, the total conversion can be made. In this chapter we examine, develop, and specify a mixfix parser. We are taking the following steps:

- An introduction to mixfix parsing (see Section 5.1) is given.
- Two requirements are needed when specifying mixfix parsing:
  - We have to change the original CASL grammar so that we can access the terms in a more easy and structured way (see Section 5.2).
  - We need a list with all the variables, constants, predicates, and operation names. Therefore, we have to specify the creation of such a list (see Section 5.3).
- The terms, on which the mixfix parser operates, consists of some complex structures. In a preparse of the list of terms these complex structures are, if possible, recognized and isolated, by adding parentheses (see Section 5.4).
- The mixfix parser inserts parentheses in such a way that their position in the list disambiguates an ambiguous list of terms. The list of terms consists of simple tokens or the previously mentioned parenthesized complex structures (see Section 5.5).
- The non-ambiguous list of terms has to be converted to abstract syntax (see Section 5.6).

Presenting the complete ASF+SDF specification with comments would not increase the insight of what is going on. Therefore, a combination of theory, the

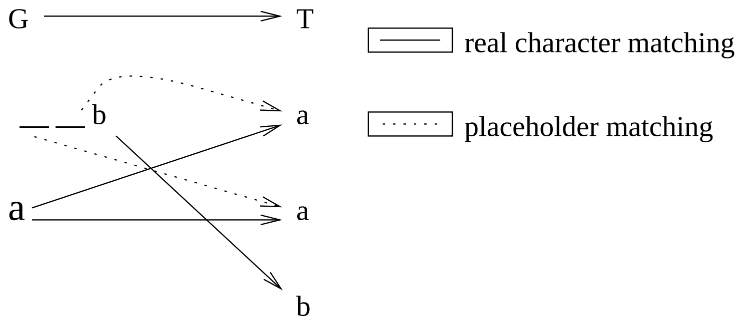


Figure 5.1: Example of mixfix mapping

concrete specification and an abstract mixfix algorithm will be used to describe the developed mixfix parser. Several questions can be formulated:

- Is it possible to create a mixfix parser and which assumptions does one have to make?
- What is the performance of such a mixfix parser?
- Are there some benefits of using ASF+SDF for mixfix parsing?

5.1 Introduction to Mixfix Parsing

The conversion from concrete to abstract syntax is possible if the ambiguous terms can be converted. It is necessary to parse these ambiguous terms with respect to the operations, variables, and constants. We must have all the variables and operations in one list (G) before actually parsing the ambiguous list of terms. This is done by processing through the concrete grammar and collecting all variables and operations into G[1].

A `Term` is in fact a list T of `Simple-Term`'s t. A mixfix-parser should investigate if there is a possible mapping from G to T. For example, for $G = (a, \_\_\ \_\_b)$ and $T = aab$ such mapping is possible (see Figure 5.1). In this case aab should be seen as $(a, a)\_\_\ \_\_b$ where b is an operation application with parameters a, a. Unfortunately the mapping from G to T is not so straightforward since a `Simple-Term` (t) is more complex than just a simple list of identifiers[2]. The concrete syntax in Appendix A gives the exact definition of t (see the corresponding ASF+SDF grammar of `Simple-Term`'s in Figure 5.2):

- *id*
- *qualified variable:* (**var** *var : sort)*

[1] Predicates will in the future also be added to G.

[2] Variables and constants inhabit the same flat structure, and in fact the identifier in the `Simple-Term`'s is meant as variable or constant.

```
%% meant as constant or variable
Id                              -> Simple-Term

%% qualified variable
"(" "var" Var ":" Sort ")"      -> Simple-Term

%% meant as qualified constant
"(" "op" Qual-Op-Name  ")"      -> Simple-Term

%% application with one term
Op-Symb ""(" Term ")"           -> Simple-Term

%% application with more than one term
Op-Symb "(" Term "," Termss ")" -> Simple-Term

%% sorted term
Simple-Term ":" Sort            -> Simple-Term

%% cast
Simple-Term "as" Sort           -> Simple-Term

%% parenthesized term
"(" Term ")"                    -> Simple-Term

%% conditional
Term "when" Formula "else" Term -> Simple-Term

%% concrete Term
Simple-Term+                    -> Term
```

Figure 5.2: SIMPLE-TERM ASF+SDF grammar

- *qualified constant:* (**op** *opname : optype)*
- *application: opsymb (term, ... , term)*
- *sorted term: t : sort*
- *cast: t as sort*
- *conditional: term* **when** *formula* **else** *term*
- *parentheses:* (T)

Some assumptions are made here to assure that a future mixfix-parser is correct. A *qualified variable*, *qualified constant*, *application* with more than one parameter, *sorted term*, and *cast* can always automatically be parsed correctly by the concrete CASL grammar in ASF+SDF, because they cause no ambiguities. We have to stress that the mixfix-parser does not work on the general concrete grammar as presented in Appendix A. This general concrete syntax causes too many unnecessary ambiguities. E.g., $a(b)$ could be a function application a with parameter b or two constants where b is a parenthesized constant. We will present an improved version of the grammar that avoids these ambiguities. This expansion can parse all correct CASL specifications but unfortunately still some not well-formed ones. Ambiguities arise between *id's*, *applications* with one parameter and *parenthesized terms*. For example, $a(b)$ is this an application with one parameter or two constants where the last one is parenthesized? For the moment it is an *id* in combination with a separate parenthesized term. The decision whether it is an application with one parenthesized argument or two identifiers where one is parenthesized, is left to the mixfix parser.

The main problem with parsing mixfix terms is that the textual information of a mixfix term is not enough to convert it from concrete to abstract syntax. Information as whether it is an operation application or constant plays an important role in the conversion. One should be aware of the number of placeholders involved. To see whether, e.g., a postfix operation matches on a list of tokens, information about the placeholders is needed. One should keep in mind that parsing of mixfix terms is not the same as type-checking, it just converts concrete ambiguous mixfix syntax to non-ambiguous abstract syntax. Sometimes predefined disambiguation rules and syntactical notions are used to disambiguate. For example, if we have a mixed term ab and operations $\_\_a, a, b$ we know, due to the shape of the operation, that the a in the mixed term can never be an operation application since there are no tokens on the left-hand side of the token a.

5.1.1 Theory and Complexity

Before investigating a possible conversion it is quite useful to look at mixfix parsing at a more theoretical level. If we forget for the moment the complexity of t and focus on a list of possible parenthesized identifiers we might gain some insight in the complexity. Assume a μ-operation consists of only a name and zero or more placeholders. If there are zero placeholders it is called a constant. Mixfix terms can be seen as a list of μ-operations (e.g. $a\_\_b\_\_c$ can be seen as $a\_\_, b\_\_, c$) where each placeholder is substituted with a subterm t.

A list of terms is well-formed if there exists a surjective mapping from all (μ) operations to the list. Assume we have such a list $r_1 \ldots r_k$ where r is a term, and the number of (μ) operations is l.

We can try to match an (μ) operation on $r_1 \ldots r_k$, there are as many possibilities as there are operations. Every placeholder also introduces new possibilities, we have to consider all available operations. It can occur that the chosen operation can not be matched after all, since one of its placeholders does not match on a sublist. Assume we have l operations with an average of k placeholders. Since every placeholder has l possibilities we have l^k as worst case. If the number of placeholders and operations is low then it can quickly be decided whether there is a positive match or not.

5.2 Specializing the Concrete Grammar

The concrete CASL grammar is too general and causes too many unnecessary ambiguities. Therefore some restrictions on the grammar are necessary to reduce the number of incorrect parses whereas the number of correct parses does not change. The CASL grammar is still quite tentative. Therefore it is hard to ensure that our mixfix parser will parse CASL mixfix terms in the future. If the CASL grammar changes in the future the algorithms and parsing techniques are reusable.

5.2.1 Reconfiguring the `Id`'s

In the original CASL grammar (Appendix A), an `Id` is a random list of placeholders and tokens. So, an `Id` is also a list of tokens and can thus represent sentences. This was never the intended meaning of an `Id`. An `Id` is correct if it is a word, or several words in combination with a random number of (at least one) placeholders. Only `Pred-Names`, `Op-Name`'s and `Simple-Term`'s can be `Id`'s in Basic Specifications. By using different grammar rules for `Id` it is possible to do the disambiguation automatically:

```
Mix-Id               -> Id
Token                -> Mix-Id
Token "__"           -> Mix-Id
"__"                 -> Mix-Id
Token "__" Mixid     -> Mix-Id
"__" Mix-Id          -> Mix-Id
```

In the original CASL grammar the identifier was originally specified as:

```
Token                -> Token-Or-Place
"__"                 -> Token-Or-Place
Token-Or-Place+      -> Id
```

We restrict the original CASL grammar since we are specifying the identifier unambiguous. Only unwanted parses are removed, correct parses are not removed by this modification[3]. A predicate and operation name can inhabit the above mentioned mixfix-structure, a `Simple-Term` can not. As mentioned before, the `id` as `Simple-Term` represents only a constant or variable. Since constants and variables do not inhabit such `Mix-Id`-structure, but only a token-structure, a new nonterminal `Var-Or-Const` should be introduced which only consists of a token.

[3]Of course, we are speaking of *intended* correct parses.

```
Token-Id                      -> Var-Or-Const
Token                         -> Token-Id
Token "[" {Id ","}+ "]" -> Token-Id %% possible extension
```

In the future, when generic and structured CASL grammars are being parsed, this definition might be extended[4].

5.2.2 Restricting Operations in `Simple-Term`'s

By changing the original grammar we achieved a reduction of the number of unwanted parses. There are still some other places in the grammar where ambiguities occur. The grammar is changed again to remove these incorrect parses as well. Given the following partial definition of `Simple-Term`'s:

```
"(" "op" Qual-Op-Name  ")"       -> Simple-Term
Op-Symb ""(" Term ")"            -> Simple-Term
Op-Symb "(" Term "," Termss ")" -> Simple-Term
"(" Term ")"                     -> Simple-Term
Simple-Term+                     -> Term
```

It is explicitly defined that the single qualified operation is a constant. Operations with one parameter must be recognized by

$$\text{Op-Symb (Term)} \rightarrow \text{Simple-Term}$$

But `(op Qual-Op-Name)` and `( Term )` are also `Simple-Terms`, this will cause ambiguities since an `(op Qual-Op-Name)` is also an `Op-Symb`. If a qualified operation is a `Simple-Term` it must be a constant. Of course, e.g., $\_\_a(b)$ is always an operation application since a placeholder is involved. By transforming:

$$\text{Op-Symb (Term)} \rightarrow \ \text{Simple-Term}$$

to

$$\text{Op-Name (Term)} \rightarrow \ \text{Simple-Term}$$

it is established that a qualified operation and parenthesized terms are always recognized separately. Whether they must be seen as separate terms or as one operation application is of later concern. This leaves us with one unwanted ambiguity: the case that `Op-Name` is a single token, e.g., $a(b)$. By stating in the priority section of ASF+SDF that

$$\text{Op-Name (Terms)} \rightarrow \text{Simple-Term}$$

has a lower priority than `(op Qual-Op-Name)` and `( Terms )`, it is assured that when `Op-Name` is a single token, it is not recognized as an application. In fact, our priority rule stresses that parenthesized terms are always recognized

[4]The extension of the tokens with a list of `ID's` between [] is still very tentative. Mixfix-notation with such extended tokens will probably be forbidden since it causes severe problems.

```
Var-Or-Const                        -> Simple-Term
"(" "var" Var ":" Sort ")"          -> Simple-Term
"(" "op" Qual-Op-Name    ")"        -> Simple-Term
Op-Name "(" Term ")"                -> Simple-Term
Op-Symb "(" Term "," Terms ")"      -> Simple-Term

Simple-Term ":" Sort                -> Simple-Term
Simple-Term "as" Sort               -> Simple-Term
"(" Term ")"                        -> Simple-Term
"when" Formula "else"               -> Simple-Term

Token-Id                            -> Var-Or-Const
```

Figure 5.3: Changed ASF+SDF `Simple-Term` grammar

separately, and they are only recognized in combination with an application name, if there is no other possibility. Whether it is an application or not is left to the mixfix-parser. As mentioned before, operations with more than one parameter are always correctly recognized since they cause no ambiguities.

5.2.3 Neglecting Conditionals

Conditionals use in some way the `Simple-Term` as recursive input. They have in a sense the lowest priority. Only the "when" and "else" keywords are mentioned in the new restricted grammar. After the mixfix parsing it will be decided which parts of a mixed term belongs to a conditional. This approach gives the following adjustment of `Simple-Terms`:

```
"when" Formula "else"      -> Simple-Term
```

During mixfix parsing it will be decided which subterms belong to a cast or sorted term. Since the conditionals are grouped from right to left, it is easier to decide after the mixfix parsing which terms belong to a conditional.

5.2.4 New `Simple-Term` Grammar

The above mentioned adjustments of the grammar results in Figure 5.3, notice the difference between Figures 5.3 and 5.2

5.3 Accessing Environment List G

G is a list containing all variables, constants, and operations. A predicate can also occur as a mixfix-operation, and should therefore also be included in G. There will be an extra check whether a term is a predicate application or not.

For the moment we just ignore the predicate application with respect to the mixfix parser but it will be discussed in Section 5.7 more elaborately.

5.3.1 Creating data-type G

G must contain all variables, operations, and predicates occurring in a CASL specification. A traversal through the specification establishes the possibility of collecting these elements and inserting them into G. We can use the same traversal technique used in the $c2a$-function. A very general data-type `App-List` is introduced to contain all elements:

```
"(" Op-Item      ")"     -> List-Item
"(" Pred-Item    ")"     -> List-Item
"(" Var          ")"     -> List-Item
"(" Alternative ")"      -> List-Item
List-Item*               -> App-List
```

The first three rules are straightforward. Operations[5] (`Op-Item`), predicates (`Pred-Item`), and variables (`Var`) are all elements of G. In the alternative section of data-types it is possible to specify operation names. These operation names can occur within mixed terms, e.g.,

```
type Nat ::= 0 | suc(pre:Nat);
op   __ + __ : Nat*Nat -> Nat;
        1    : Nat = suc(0);
vars suc,x,y:Nat;
axioms
  suc(0) = 1;
  suc(x) = x + 1;
```

In this example the operation `suc` is used within the axioms but specified in the data-type declaration. The definition of `suc` restricts the number of arguments of suc, e.g., the term `suc(0,1) = ...` is thus not well-formed. So, the alternative section must be included in G.

5.3.2 Checking G

Before the conversion from concrete to abstract is performed the list G is created and checked. G can contain inconsistent operation and predicate declarations, e.g.,

```
op __ + __ : Nat*Nat*Nat -> Nat %% Inconsistent, too many
                                %% sort names
op __ / __ : Nat                %% Missing two sort names
pred __ a  : ()                 %% Missing one sort name
```

[5]Constants are a special form of an operation.

Therefore, G is checked with respect to the number of placeholders and sort names. In case that the operation or predicate name consist of a single identifier then the declaration is always correct. In this special case the operation or predicate must always be specified in a prefix form, e.g.,

```
op   a   : Nat*Nat -> Nat;
     0,1 : Nat;
pred b   : Nat;
axioms
    a(1,1) when b(0) else a(0,0)
```

If G contains one or more inconsistent operation or predicate declarations the conversion will be aborted.

5.3.3 Sorting and Beautifying G

The previous data-type is very rough and is not suitable for direct access while parsing mixfix-terms. G contains on the one hand unused information but on the other hand it does not contain any order with respect to prefix, postfix, infix, mixfix, predicates, and variables. This order is of significant importance for the mixfix parser since the priority of the elements of G determines the disambiguation of the mixed terms.

First, we should get rid of all information that we do not need. Second, extra information should be added to G. This information classifies what kind of element we have: application, constant, variable, or predicate but also the number of parameters of a predicate or application. By creating a new data-type G' it is possible to add this extra information to the data-type:

```
"(" Id "," Status "," INT ")"  -> SimpleElement
SimpleElement*                 -> SimpleList

"application"   -> Status
"constant"      -> Status
"variable"      -> Status
"predicate"     -> Status
```

Predicate, operation, variable, and constant names are all of type `Id`, which will be the basic element in our new data-type G'. Other information, besides the names, can be discarded since it is of no use for mixfix parsing. This information consists of the result sorts, arguments, etc.

The status is determined in the following way:

1 If it is a predicate, the status will be `predicate`.

2 If it is an operation, the status will be `application` with one exception. If the operation name consists of only a token and it is one of the following forms *token* : *sort* or *token* :? *sort*, then the status is `constant`.

3 If it is a variable, the status will be `variable`.

Variables and constants have zero arguments, the number of arguments of an application or predicate is determined when transforming G to G' by counting the number of arguments.

Constants and variables consist only of one token. `Id` is also a token, therefore the insertion of variables and constants in G' is not a problem even though we have classified constants and variable as `Var-Or-Const`.

The order of the list is important. Predicates have a higher priority than applications, and therefore they should be at the beginning of the list. By creating an order into G', we automatically obtain a priority ordering. The order is (from high to low priority):

- predicates,
- applications,
- constants and variables.

Furthermore, there should be a priority order between applications in mixfix parsing. In the concrete language summary [20] it is related that the following notations have a priority from high to low:

- postfix, e.g. $\_\_x$, $\_\_\ \_\_x$, $\_\_\ \_\_\ \_\_x$
- prefix, e.g. $x\_\_$, $x\_\_\ \_\_$, $x\_\_\ \_\_\ \_\_\ \_\_$
- infix, e.g. $\_\_ + \_\_$, $\_\_\ \_\_\ x\_\_$, mixfix, e.g. $\_\_times\_\_equals\_\_$

If two operations of the same kind are defined together, the one with the biggest size will have the highest priority. If there is no difference in size the order in which they occur in the specification will determine the priority. For example, a mixfix operation $a\_\_b\_\_c$ has a higher priority than $a\_\_b$ since it has greater size. The list contains only unique elements.

The list G' is created using the priority rules just described. Due to these priority rules the mixfix parsing of operations, predicates, variables, and constants used in the mixfix-terms will follow the order in which they are meant to be applied. Assume we have the following mixfix term: ab with

$$G' = (\_\_b, application, 1)(a\_\_, application, 1)(a, constant, 0)(b, constant, 0)$$

G' has a correct priority ordering and the element with the highest priority is the first one in the list. Therefore, if we obey the priority ordering the mixfix parsing should see ab as an application of b with a parameter a. Our mixfix parser takes the first element of the list (and thus with the highest priority) and tries to match it on our mixed term. In this case, $\_\_b$ is matched on ab. Beware, the name (b in this case) is discarded since, in this case, it was for the moment recognized as an application. Therefore, the placeholder of $\_\_b$ must be matched on a. If we disobey the priority rules and for example have:

$$G' = (a\_\_, application, 1)(\_\_b, application, 1)(a, constant, 0)(b, constant, 0),$$

the result of mixfix parsing will be an application a with parameter b. $a\_\_$ is simply matched on ab and the placeholder must be matched on b. We must assure that all priority rules will be obeyed in our mixfix parser, by applying the priority rules to G'.

5.4 Massaging the `Simple-Term`'s

In the beginning of this chapter we stressed that mixfix terms do not only contain tokens but also more complex structures. In this section we will find these complex structures and isolate them by parentheses or give them some other special treatment. This results in a new list of terms T^e. Everything that is automatically recognized by ASF+SDF is surrounded by parentheses and can be matched against a placeholder. Everything that is recognized by `Var-Or-Const` (all single tokens) are not surrounded by parentheses. A special treatment of the conditionals, casts and sorted terms is presented. They will be surrounded with special markers [] and are left untouched for the moment. Their terms are seen as ordinary `Simple-Term`'s and are handled in the same way as ordinary `Simple-Term`'s

A cast `[:sort]` or sorted term `[as sort]` is well-formed if on the left-hand side of, respectively, `[:sort]` or `[as sort]` a term is found that can be matched against a placeholder. So, that term is in fact a single variable, constant, or application. This term in combination with the predefined syntax (`[:sort]` and `[as sort]`) can then be matched against a placeholder. For example, for a list $G' = (a\_\_, application, 1)(b, constant, 0)$ a list of terms $ab[as\ sort]$ is converted into $(a\_\_(b[as\ sort]))$.

A list of T^e consists of:

- `Var-Or-Const`, ergo single tokens.
- conditionals, casts and sorted terms with special markers. Keep in mind that conditionals, casts and sorted terms in the original CASL grammar are combined with terms. After and during the mixfix parsing the appropriate term will be connected with these elements.
- in all other cases, t is extended with parentheses.

Beware, since we changed the original CASL grammar an operation application with one parameter e.g. $a(b)$ is recognized as a and (b). Therefore $a(b)$ will be represented as $a((b))$ in our new list of terms T^e. Every $t^e \in T^e$ can now be seen as a token, parenthesized expression or an expression between [].[6] See Figure 5.4 for an example: every `Simple-Term` is parenthesized, except the tokens and exceptions mentioned above.

[6] t^e is thus for the moment: (qualified variable...) , (qualified constant...), (application with more than one argument), [: sort], [as sort], [when formula else], and ((T)).

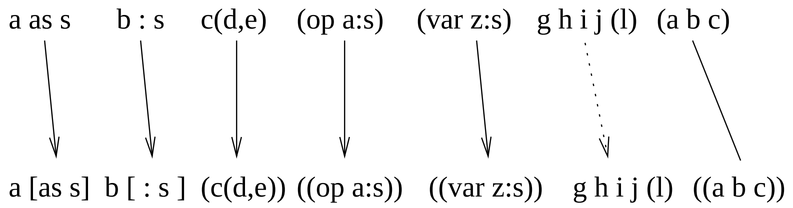


Figure 5.4: Massaging the `Simple-Terms`

5.4.1 Storing T^e

To store the massaged `Simple-Terms` we have to create a new data-type. This data-type must contain:

- The special markers `[: sort]`, `[as sort]`, and `[when formula else]`.
- An identifier to state variable, constant, predicate, and operation names.
- Parentheses to insert in the list of terms.

A new data-type `Mixfix-Element` and `Mixfix-List` is introduced:

```
"(" Simple-Term+ ")"          -> Mixfix-Element %% parentheses
Id                            -> Mixfix-Element %% identifier
"[" ":" sort  "]"             -> Mixfix-Element %% spec. marker
"[" "as" sort "]"             -> Mixfix-Element %% spec. marker
"[" "when" Formula "else" "]" -> Mixfix-Element %% spec. marker

Mixfix-Element*               -> Mixfix-List %% list of terms
```

The `Id` can inhabit a complete mixfix structure, e.g., $a\_\_b$. Our `Id` in this form consists of a single token. Therefore, for our current approach we could write directly `Token -> Mixfix-Element` but for uniformity reasons we choose `Id`[7]. Remember, that here `Id` as `Mixfix-Element` represents only a single token and not a mixture of tokens and placeholders.

5.4.2 Removal of Ambiguities

The insertion of the parentheses and the [], has the effect that almost all elements of the list of terms T^e consist of a token, parenthesized expression, or an expression between []. The previous definition of T^e implies that the list of terms is not ambiguous. Unfortunately, there are still situations where a term can be parsed as two terms. Beware, these ambiguities have nothing to do with future mixfix parsing ambiguities. The ambiguities mentioned here are ambiguities caused by the grammar not by mixfix operations. So, in fact the CASL

[7]In our mixfix parsing we always use the sort `Id`.

grammar presented in Appendix A is quite ambiguous. We have to remove the ambiguities which occur in the list of terms. There are two cases in which an ambiguity can occur:

- Assume we have a mixed term $a\_\_b(a)$. The problem is that it is not clear whether this is an operation application of $a\_\_b(a)$, or an operation application of $\_\_b(a)$ in combination with a constant a. We will look in G' to decide which one to take. If one of the two operations (in our example $\_\_b$ and $a\_\_b$) is not present in G', then it is obvious which one to take: the one that is in G'. In the case that both are present in G' as operations but the token on the left is not mentioned in G' as constant or variable, then the total name will be considered as operation name. If there exists a constant or variable with that token as name, then we have a serious ambiguity which demands a special treatment. Of course, the number of arguments between the parentheses must be the same as the number of arguments related to the operation name. Therefore the number of arguments of the operation name will be looked up in G', and must match the number of arguments between the parentheses. If we have a list of terms consisting of only an application, then the operation name is not separated, e.g.,

  $$a\_\_b(a) \rightarrow (a\_\_b(a))$$

  If our operation application should be matched with a single placeholder then the total operation name should be involved. If it should be matched with two placeholders then the single token and the rest of the operation application are matched against the two placeholders. This decision is left to the mixfix parser. Our operation application should therefore have some special treatment so that the mixfix parser can recognize its shape; the following extra grammar rule defines this special treatment:

  ```
  "[" Id "(" { Term ","}+ ")" "]"  -> Mixfix-Element
  ```

  The following examples show such ambiguous operation which should be identified with one or two placeholders.

  $$G' = \_\_c, a\_\_b, \_\_b, a, \_\_\ \_\_d$$

  $$\text{one placeholder: } a\_\_b(a)c \rightarrow (\_\_c(a\_\_b(a)))$$

  $$\text{two placeholders: } a\_\_b(a)d \rightarrow (\_\_\ \_\_d(a, \_\_b(a)))$$

- A qualified operation can be a constant or not. If it is a constant then terms of the form:

  $$(op\ \ qualopname)(term)$$

  are seen as two separate parenthesized terms. If it is not a constant then they are seen as one, e.g.:

  $$((op\ \ qualopname)(term))$$

with the restriction that the number of arguments of the qualified operation name is the same as the number of arguments between the parentheses. The following sub-term can therefore never result in a correct parse, since the number of arguments of the qualified operation is two whereas the parenthesized terms yield three:

$$(op\ a : s * s- > s)(x, y, z)$$

Beware that a *qualopname* is only a constant if it has the following form: $(op\ \ token : sort)$ or $(op\ \ token :?sort)$. In any other case it must be an application. Examples of applications are(a a variable of s':

– $(op\ \ token\_\_ : s' \to s)(a)$

– $(op\ \ \_\_token : s' \to s)(a)$

From this point is our list T^e so specialized that only tokens, parenthesized, and untouchable[8] expressions occur in it. By stating that all placeholders can be matched with tokens and parenthesized terms we have in fact a situation as in Figure 5.1.

5.5 Mixfix Parsing

In the previous sections we defined a sorted priority list G and a specialized list of terms T^e. These two elements are the input for the mixfix parser. The mixfix parser searches for a surjective mapping from G to the list of terms T^e. The description of the mixfix parser is based on an abstract algorithm.

The mixfix parser can be divided into three parts:

- A *parsing* function. This function starts the parsing of mixfix terms. It copes with special cases and its result is a non-ambiguous term which corresponds almost directly to an abstract term.
- An *apply* function. As mentioned before, we are searching for a surjective mapping from G to the list of terms. This function matches an element of G with the list of terms.
- An *applyspecial* function. This function is specially written to deal with placeholders within operation names.

These three elements are described step by step by giving an abstract algorithm.

5.5.1 Parsing Function

The parsing function is in fact the starting point of the mixfix parser. It requires a list of terms T^e (variable `lt`) and the list G (variable `lo`). All non-capital words are variables (optionally followed by an apostrophe):

[8]Everything between “[” and “]” must not be touched.

- `lt` is a list of terms.
- `lo` is a list of operations.
- `umt` is an unambiguous mixed term.
- `id, term, termss, optype` are respectively identifiers, term, a list of terms, and operation type.

```
PROC parsing(lt,lo)
BEGIN
     IF is-empty(lt)
[1]  THEN RETURN EMPTY;

     ELIF lt == (id(termss)) AND
          is-element((id, application,
                        numbers-of-terms(termss)),lo)
[2]  THEN RETURN (id(termss));

     ELIF lt == (lt')
[3]    THEN RETURN ((lt'));

     ELIF lt == lt'[: sort]
[4]    THEN RETURN (parsing(lt',lo) [:sort]);

     ELIF lt == lt'[as sort]
[5]    THEN RETURN (parsing(lt',lo) [as sort]);

     ELIF lt == [ id ( terms ) ]
          WHERE id == token __ OR token __ mixid
[6]    THEN RETURN (id(terms));

     ELSE get-first-element(lo) == id AND
          (apply(id,lt,lo)) == umt
[7]    THEN RETURN (umt);

     ELSE remove-first(lo) == lo' AND
          length-of-first(lt) > 1
[8]    THEN RETURN parsing(lt,lo');
END
```

The *parsing* function is successful in one of the following cases:

[1] The list of terms is empty, the result will automatically be empty.

[2] The list of terms is of the form $(id(terms))$ and the identifier is present as an application in G, the result is $(id(terms))$. Of course, the number of arguments of the application in G must be the same as the number of arguments specified.

[3] If the list of terms is parenthesized, the list is returned. Beware, the list of terms can in that case still be ambiguous. In the conversion from concrete to abstract the parenthesizes are removed and a recursive call to the mixfix parser will disambiguate this list of terms.

[4–5] The list of terms ends with the special markers for a cast or sorted term. The list of terms is parsed without the special markers.

[6] If `lt` consists only of the special cases:

$$[token\_\_(terms)] \texttt{ or } [token\_mixid(terms)]$$

then they are directly returned in the forms:

$$(token\_\_(terms)) \texttt{ or } (token\_mixid(terms))$$

[7] The first element of G is successfully matched with the list of terms using the *apply* function. Beware, one could combine cases [7] and [8] to:

```
IF ( ∃  el  ∈  lo ) THEN
APPLY(el, lt',lo);
```

But now it is not clear that the first element of G has a higher priority than the second. By specifying cases [7] and [8] separately we explicitly define that the elements of G are applied with respect to their priority.

[8] The first element of G will be removed and the *parsing* function is applied without that first element. Beware, if G is empty the terms can never be parsed correctly.

No error mechanism is present. If no match is possible, the mixfix parsing simply stops.

5.5.2 Apply Function

In the *parsing* function a call to the *apply* function was specified. *Apply* tries to match the operation, constant, etc. with the list of terms. Every possibility has to be examined and therefore the function inhabits many cases.

```
PROC apply(id, lt, lo)
BEGIN
     IF is-token(id) AND id == lt AND
        (is-element((id,constant,0),lo)
        OR is-element((id,variable,0),lo))
[9]  THEN RETURN (id);

     ELIF lt == (id(termss)) AND
```

```
          is-element((id, application,
                      get-numbers-of-terms(termss)),lo)
[10]   THEN RETURN (id(termss));

      ELIF (op id:op-type)(terms) == lt
[11]    THEN RETURN ((op id:op-type)(terms));

      ELIF lt == (terms)
[12]    THEN RETURN ((terms));

      ELIF  id == token __ AND
            lt == token lt'
[13]    THEN RETURN (token __ (applyspecial(__, lt', lo)));

      ELIF  id == token __  mixid AND
            lt == token lt' AND
            only-placeholders(mixid)
[14]    THEN RETURN (token __ mixid(applyspecial(__ mixid, lt', lo)));

      ELIF  postfix(id) AND
            get-postfix-token(id) = token AND
            lt == lt' token
[15]    THEN RETURN (id(applyspecial(get-placeholders(id), lt', lo)));

      ELIF infix(id) AND get-infix-token(id) = token AND
           ∃ (lt', lt'': lt == lt' token lt'') AND
           ψ == applyspecial(get-placeholders-left(id), lt', lo)) AND
           γ == applyspecial(get-placeholders-left(id), lt'', lo)))
[16]    THEN RETURN (id(ψ,γ));

      ELIF id == token __ mixid AND mixfix(id) AND
           lt == token lt' AND name-matches(id, token lt')
[17]    THEN RETURN (id(applyspecial(__mixid, lt', lo)));

      ELIF id == __ mixid AND mixfix(id) AND
           name-matches(id, lt')
[18]    THEN RETURN (id(applyspecial(__mixid, lt, lo)));

      ELIF ∃ (lt', lt'': lt == lt' [when formula else] lt''
[19]    THEN RETURN parsing(lt',lo)
                    [when formula else]
                    parsing(lt'',lo);

      ELIF lt == lt' [: sort]
[20]    THEN RETURN parsing(lt',lo) [: sort];
```

```
      ELIF lt == lt' [as sort]
[21]    THEN RETURN parsing(lt',lo) [as sort];

[22]  ELSE RETURN apply(get-first-element(remove-first(lo)),
                        lt, remove-first(lo))
END
```

The *apply* function is successful in one of the following cases:

[9] If `id` is a single token, the list of terms consist only of that token, and the token is present in G as variable and/or constant then the token is returned. The check whether the token is present as variable/constant is necessary since a single token can also be a (prefix) function application.

[10] If the list of terms consist only of $id(terms)$ and the id is present in G as operation then $(id(terms))$ is returned. Of course, the number of arguments specified in G for that id must be the same as the number of terms between parenthesizes.

[11] If the list of terms consist of $((op\ id : op - type)(terms))$ then this is directly returned. This case is already examined in the expansion of the `Simple-Term`'s.

[12] If the list of terms consist only of a parenthesized term then this is immediately returned as well. The terms between the parenthesizes can still be ambiguous. While translating non-ambiguous concrete syntax to abstract syntax the terms between the parenthesizes will be parsed using the mixfix parser.

[13] If `id` consists of a single token and a placeholder, the token must match the first element of the list of terms. The placeholder is matched on the rest by using the `applyspecial` function. If the `applyspecial` function gives a result (say γ) then $(id(\gamma))$ is returned. This case is in fact the matching of a prefix operation with one placeholder.

[14] If `id` consists of a single token with more than one placeholder the same happens as in the previous case. The only difference is that not one placeholder is matched but two or more.

[15] If `id` consists of a postfix operation we match the token with the last element of the list of terms and use the `applyspecial` to match the placeholders. The result is returned in the same form as in the two previous cases $(id(\gamma))$.

[16] If `id` consists of an infix operation we match the token with an element in the list of terms. The placeholders on the left of the token must match with the sublist of terms on the left of the element. The placeholders on the right must of course match with the sublist on the left of the element. The result is returned as in the previous case and the placeholders are

matched using *applyspecial*. If an element does not give a correct match another possibility is tried until no possibilities are left or a correct match is found[9].

[17–18] If `id` consists of a mixfix operation the name of the mixfix operation is matched with the list of terms. In all previous cases we were lucky to have the token of the operation name present. E.g., in prefix we had the form $token\_\_$ or $token\_\_mixid$ ($mixid$ consists only of placeholders). Unfortunately a mixfix operation contains more than one token in its operation name. All these tokens in the operation name must occur in the list of terms and must not clash with an operation application. E.g., assume we have operations $a\_\_b$ and a and a list of terms aab. The b in the list must be a part of the operation name $a\_\_b$. Therefore, the placeholder of the operation name must only match with the middle a. As stated before a function *applyspecial* is specially designed for this job. In that function only the placeholders are matched, the tokens of the operation name must occur in the list of terms but do not return in the result.

[19] If the list of terms consists of a conditional, the sublists on the left and right of the special markers are parsed. They are combined with the conditional `[when formula else]` in the result.

[20–21] If the list of terms consists of a cast or sorted term the sublist on the left of the special markers is parsed. This result is combined with the special markers `[as sort]` , `[:  sort]` and surrounded by parenthesizes.

[22] In the default case the procedure is recursively applied without the first element of G'. At this stage that first element could not be matched with the list of terms.

5.5.3 Applyspecial Function

We have specified all the possible matchings between the `id` as operation, constant, or variable and a list of terms. What is missing is a description of the *applyspecial* function. This function matches one or more placeholders with a list of terms. The tokens within the operation names are discarded.

```
PROC APPLYSPECIAL(id, lt, lo)
BEGIN
     IF is-token(id)
[23] THEN RETURN EMPTY;

     ELIF token __ == id AND lt == token lt'
[24]   THEN RETURN (APPLYSPECIAL(__, lt', lo));

     ELIF __ token == id AND lt == lt' token
[25]   THEN RETURN (APPLYSPECIAL(__, lt', lo));
```

[9] This is in fact the ASF+SDF list matching property.

```
     ELIF __ == id AND lt == lt' [:sort]
[26]   THEN RETURN((APPLYSPECIAL(__,lt',lo) [:sort]));

     ELIF __ == id AND lt ==  lt' [as sort])
[27]   THEN RETURN((APPLYSPECIAL(__,lt',lo) [as sort]));

     ELIF __ == id AND lt == ( lt' )
[28]   THEN RETURN((lt'));

     ELIF __ == id AND
          lt == [token φ (terms)])
          WHERE φ ==  __ OR  __ mixid
[29]   THEN RETURN(token φ (terms));

     ELIF __ __ == id AND  lt == [token φ (terms)])
          WHERE φ ==  __ OR __ mixid
[30]   THEN RETURN token,( φ(terms));

     ELIF id == __ __ mixid' AND
          lt == [token φ (terms)] lt')
          WHERE φ ==  __ OR __ mixid
[31]   THEN RETURN token,(φ(terms)),APPLYSPECIAL(mixid', lt',lo));

     ELIF id == token __ mixid AND lt == token lt'
[32]   THEN RETURN(APPLYSPECIAL(__ mixid, ;t', lo));

     ELIF id ==  __ mixid
[33]   THEN RETURN(AMBIGUITY-CHECKER(__ mixid, lt, ,lo));

[34] ELIF id == __ THEN RETURN(PARSING(lt, lo));
END
```

The result of this function is always seen as arguments of an operation application[10]. Therefore, in the case where multiple arguments are returned comma's are used to separate these multiple arguments, e.g., see cases [30,31]. The function *applyspecial* is only successful in one of the following cases:

[23] If `id` and the list of terms consists only of the same token the result will then be empty since we are only matching placeholders. The tokens are already specified before, for example:

Assume we have operations $a\_\_b\_\_c\_\_$, e,f, and g, and a list of terms $aebfcg$. At a certain moment the mixfix operation $a\_\_b\_\_c\_\_$ is going to be matched with the list of terms (see case [17]). The result is in the following form $a\_\_b\_\_c\_\_(\ldots)$. The "..." are in fact a call to the *applyspecial* function in the form $applyspecial(\_\_b\_\_c\_\_, ebfcg, G')$. But b and c are still in the list of terms but also specified in $a\_\_b\_\_c\_\_(\ldots)$. To match the placeholders with the list of terms we have to filter the tokens of the operation name ($a\_\_b\_\_c\_\_$) out of the list of terms. Therefore, tokens are matched within the list of terms but removed from the result.

[10] This function was used to match placeholders, and placeholders are inherently attached to operations.

[24] If `id` consists of `token __` then the first element of the list of terms must be a token. The rest of the list of terms is then matched with the single placeholder by a recursive call to the *applyspecial* function. The token is again removed from the result.

[25] If `id` consists of `__ token` we have the same situation as the previous one only the token must be matched with the last element of the list of terms.

[26–27] If `id` consists of a single placeholder and the list of terms ends with the special markers of the sorted term, the sublist before the special markers is parsed using again *applyspecial*. The total result is then returned.

[28] If `id` consists of a single placeholder and the list of terms contains only a parenthesized term, this parenthesized term is directly returned.

[29–31] If `lt` consists of the forms [*token__mixid*(*terms*)] or [*token__*(*terms*)] and `id` contains one placeholder this placeholder is matched with the previous forms. This result directly in (*token__mixid*(*terms*)) or (token __ (terms)). If `id` contains exactly two placeholders the result is directly *token*(*__mixid*(*terms*)) or token (__ (terms)).

[32–33] If `id` consists of `token __ mixid` then the token is matched with the first element of the list of terms. The rest (`__ mixid`) is matched with the rest by a recursive call. If `id` consists of `__ mixid` an ambiguity checker is used to match the `id` with the list of terms (see Chapter 6).

[34] The list of terms is parsed using the *parsing* function. The *parsing* function returns either a token or a parenthesized expression. Both can be placed on a single placeholder.

5.5.4 Single Placeholder Problem

In Casl there exists a very strange operation consisting of only one placeholder, e.g. $\_\_ : s \to s'$. Besides this operation, every other operation reduces in some way our list of terms to subterms. Our divide and conquer algorithm works successfully until a single placeholder comes into play. Assume we have an operation __ and a list of terms $a(bc)$ where a, b, c are constants. The total list of terms can be identified with a placeholder, but also every subterm. This results in:

$$(\_\_((\_\_(a))(\_\_((\_\_b)(\_\_c)))))$$

However every subterm of this result, together with the result, can again be identified with the single placeholder. To state this infinity it is enough to look at a single constant a as list of terms (ω indicates infinity):

$$a \to \_\_(a) \to \_\_(\_\_(a)) \to \cdots \to \_\_^{\omega}(a)$$

In the original Casl language definition it will be forbidden to have single placeholders as operation name in parsable specifications. But it is possible, to a certain degree, to apply this single placeholder operation to a list of terms. To keep the possibility of having a single placeholder as operation we make the following assumption: a single placeholder operation cannot be placed if its argument consist already of a single placeholder application. In the current specification single placeholder operations are not allowed.

5.5.5 Examples

We start with a simple example so that one easily sees how the mixfix parser works. Assume we have the following CASL specification:

```
var a    : s;
op  b __ : s -> s;
axioms
    b a = true
```

It is obvious that `b a` is the list of terms T^e and that G' consists of

$$(b\_\_, application, 1)(a, variable, 0)$$

. All numbers between "[]" are the number of all cases as presented in the abstract algorithm.

```
PARSING(b a, G')                       = [7] =>
APPLY(b __, b a,G')                    = [13]=>
(b __ ( APPLYSPECIAL( __, a, G')))     = [34]=>
(b __ ( PARSING(a, G'          )))     = [7] =>
(b __ ( APPLY(b __, a, G'      )))     = [22]=>
(G'' = (a,variable,0))
(b __ ( APPLY(a, a, G''        )))     = [9] =>
(b __ ( a ) )
```

Some more advanced examples of mixfix parsing are given now. G' is constructed from the specification and is sorted obeying the earlier presented priority rules.

$$G' = (\_\_b, application)(a\_\_b, application)(a, constant)$$

$$abb \ \overrightarrow{\_\_b_{[7+15+34]}} \ (\_\_b(ab)) \ \overrightarrow{\_\_b_{[7+15+34]}} \ (\_\_b((\_\_b(a))) \ \overrightarrow{a_{[8+8+7+9]}} \ (\_\_b((\_\_b(a)))$$

It is possible in this example to use our mixfix operation $a\_\_b$, but our list G' is a priority list and postfix $\_\_b$ has a higher priority than mixfix and therefore applied.

$$G' = (\_\_b\_\_, application)(a\_\_b, application)(c, constant)$$

$$acbcb \ \overrightarrow{a\_\_b_{[7+17+24+34]}} \ (a\_\_b(cbc))$$

$$\overrightarrow{\_\_b\_\_{}_{[7+15+2\times 34]}} \ (a\_\_b(\_\_b\_\_(cc)))$$

$$\overrightarrow{2 \times c_{[7+9]}} \quad (a\_\_b(\_\_b\_\_(cc)))$$

Here it is not possible to start with our first element of G'. If we match $\_b\_\_$ on $acbcb$ we insert parentheses correctly into $a(\_\_b\_\_(c,c))b$, since c is a constant. But subterms a and b are not parsable since we have no constant, variable, or postfix operation a or b. This might be a bit confusing. One should keep in mind that after *every* application of an operation the result must be a parenthesized term or a single constant or variable without parentheses. If, for example, we had only a constant a and a list of terms aaa then it is obvious that the three a's must be constants. But since the final result is not a single constant, variable, operation application, qualified variable, etc. the list of terms aaa can not be parsed. The reason for this is that more than one constant can never be converted to a correct abstract term since in the abstract syntax it is not allowed to have more than one constant in a term. In Figure 5.5 the exact definition of what can be defined in a term is described. A result term consists either of a single constant, variable, operation application, qualified variable, etc. with, if necessary, inserted parentheses to disambiguate. The next example is an example with a sorted term:

$$G' = (\_\_b, application)(c, constant)$$

$$cb[: sort]$$

$$\overrightarrow{[: sort]_{[4]}} \quad (cb[: sort])$$

$$\overrightarrow{\_\_b_{[7+14+34]}} \quad ((\_\_b(c))[: sort])$$

$$\overrightarrow{c_{[7+9]}} \quad ((\_\_b(c))[: sort])$$

A bit more advanced example:

$$G' = (\_\_b\_\_, application)(a\_\_b, application)(c, constant)$$

$$acbc[: sort]b$$

$$\overrightarrow{a\_\_b_{[7+15+34]}} \quad (a\_\_b(cbc[: sort]))$$

$$\overrightarrow{[: sort]_{[4]}} \quad (a\_\_b((cbc)[: sort]))$$

$$\overrightarrow{\_\_b\_\__{[7+15+2\times 34]}} \quad (a\_\_b((\_\_b\_\_(cc))[: sort]))$$

$$\overrightarrow{2 \times c_{[7+9]}} \quad (a\_\_b((\_\_b\_\_(c,c))[: sort]))$$

5.6 From Data-type to Abstract Syntax

We have reached the stage where our non-ambiguous (parenthesized) term must be converted into abstract syntax. In Figure 5.5 we already defined the five possibilities to which we want to convert to:

```
TERM            ::= SIMPLE-ID
                  | qual-var VAR SORT
                  | application OP-SYMB TERMS
                  | sorted-term TERM SORT
                  | cast TERM SORT
                  | conditional TERM FORMULA TERM
```

Figure 5.5: Abstract Term

1. A single variable or constant. Since we implemented the concrete variables and constants as `Var-Or-Const` and the abstract ones as `VAR-OR-CONST`, the conversion of variables and constants is straightforward since both are only consisting of a single token.

2. A qualified variable with the following syntax: `"qual-var" VAR SORT`. Since in the concrete syntax we just parenthesized the qualified variable with syntax `"(" "var" Var ":" Sort ")"` and both concrete and abstract variables and sorts have the same underlying structure, the conversion is straightforward as well.

3. Due to a well chosen format as mixfix result we are able to do a direct conversion from our parenthesized concrete operation application to an abstract one. We have to convert to the following abstract syntax: `"application" OP-SYMB TERMS`. In our mixfix parser we always use the following generalized notation to denote an operation application:

   $$(operationname(terms))$$

   Since our operation name has the same underlying structure as `OP-SYMB` and our terms can be parsed using the same mixfix parser, a conversion to abstract syntax is straightforward.

4. Cast and sorted terms are recognized by `[:sort]` or `[as sort]`. In our mixfix parser we assured that if a list of terms with a cast and sorted term is parsed correctly the result has the following forms: $(terms[:\ sort])$ or $(terms[as\ sort])$. Again, in abstract syntax we have the following rules: `sorted-term TERM SORT` and `cast TERM SORT`. The sort in concrete and abstract syntax has, as said before, the same structure and therefore a conversion is possible. Since our terms are parsed and non-ambiguous and is in of the five shapes shown in Figure 5.5, a conversion is straightforward as well.

5. Last but not least, our conditional form. Since we did not touch this form during the mixfix parsing, it is obvious that some work has to be done at this stage. It is for sure that if we have reached this stage, all (sub)terms are parsed correctly and consists of a structure so that the four points

before can easily be used and applied. The conditional form was in the beginning of this project left out. During the construction of the mixfix parser the grammar changed since there was a need for introducing conditionals. The syntactic form of the conditionals is `TERM when FORMULA else TERM`[11]. Our previous technique of inserting parentheses can also be adapted here. A few rules should be formulated while parsing conditionals:

- Conditionals can be identified with a placeholder, and can be seen totally apart from the mixfix parsing.
- Conditionals are grouped from the right to the left, e.g.,

  $$T_1 \ \text{ when } \ F_1 \ \text{ else } \ T_2 \ \text{ when } \ F_2 \ \text{ else } \ T_3$$

  must be seen as:

  $$(T_1 \ \text{ when } \ F_1 \ \text{ else } \ (T_2 \ \text{ when } \ F_2 \ \text{ else } \ T_3)).$$

- Every pair of terms, of every conditional, are parsed and use parentheses for disambiguation.

The insertion of the correct parentheses is straightforward. It just traverses from the right to the left through a list of terms and adds parentheses when a fully conditional form is encountered. The abstract syntax of a conditional is: `conditional TERM FORMULA TERM`. Both terms on the left and right of `when` and `else` are parsed correctly and non-ambiguous. The only thing that is not yet parsed is the `FORMULA`. Since we had in our previous chapter a function definition of converting a concrete formula to an abstract one, we just use that function to do the conversion ($c2a$-Formula). Two notes have to be stated here:

- Parsing of a formula can inhabit again some mixfix terms, therefore a copy of G' should be given as parameter to $c2a - formula$[12].
- Since there is a possibility of mixfix parsing there is also a possibility that formula can not be converted. Since formula stands between `when` and `else` such failure is only of local importance. The terms of the conditional have nothing to do with a possible failure.

By reusing the predefined functions the conversion of the conditional is also straightforward, and all mixfix parsing is performed at this stage.

[11] There is not chosen for the form `if FORMULA then TERM else TERM` since Peter Mosses and the author discovered some severe look-ahead problem with the predefined keyword `if`; `if` is also a reversed implication in CASL.

[12] In the actual implementation of our mixfix parser is G always present in every $c2a$-function since we call the mixfix parser at different stages in the conversion function.

5.7 Separating ordinary Mixfix and Predicates

Predicates can occur as mixfix terms. The mixfix parser only allows variables, constants and operations in our environment list G'. By reusing our predefined mixfix functions we are able to recognize such mixfix predicates. Assume we have two lists:

1. A list G'_p with all predicates,
2. A list G' with all variables, constants, and operations.

To recognize whether there is a possibility of having a predicate mixfix form we traverse through the list G'_p and check, for every predicate element, whether it can be matched with the list of terms. If a predicate operation name can be matched with a subterm, it should be the only one since that is, semantically seen, the only correct possibility[13].

By simply using our mixfix parser in combination with a predicate operation as possible matching element such a checker is easily developed. The differences between predicate mixfix parsing and ordinary mixfix parsing are:

- The copy of G' only contains variables, constants and operations whereas G'_p only contains predicates.
- Only *one* predicate can be applied due to semantic correctness.
- Mixfix predicates have a higher priority if there is an overlap between a predicate and an ordinary operation, so the predicate will be chosen.
- Predicate parsing results in a total abstract `FORMULA` while mixfix parsing results in `TERM`.

5.8 Examples

In Figures 5.6 and 5.7 examples are shown how these non-ambiguous parenthesized terms are converted to abstract syntax. Since both examples are on a relatively low level, a complete CASL specification will be given as example and is analyzed step by step. In Figure 5.8 we see a CASL specification with one axioms: `b = a c d a`. Our conversion function from concrete to abstract identifies this axioms as a strong equations with two lists of terms on the left-hand and right-hand side of the equal sign. The left-hand side is obviously a constant or variable. The right-hand side is quite interesting and the mixfix parser takes the following steps:

1. The list $G' = (\_\_c, application, 1)$ $(\_\_d\_\_, application, 2)$ $(a, variable, 0)$ $(b, variable, 0)$ $(d, variable, 0)$ is created.

[13] A predicate as axiom can not have a predicate application on some of its placeholders, this is semantically incorrect.

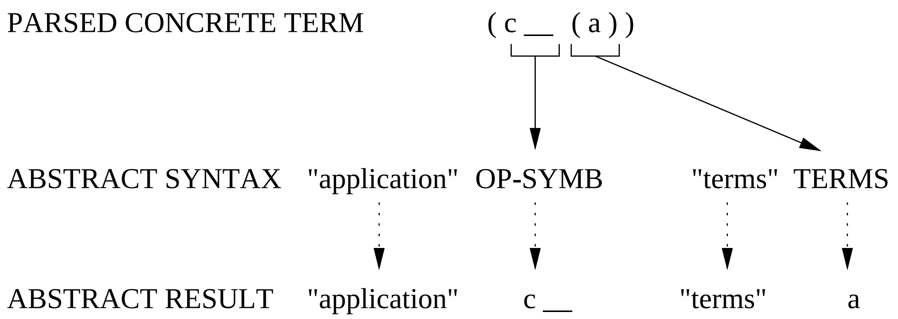


Figure 5.6: Example 1

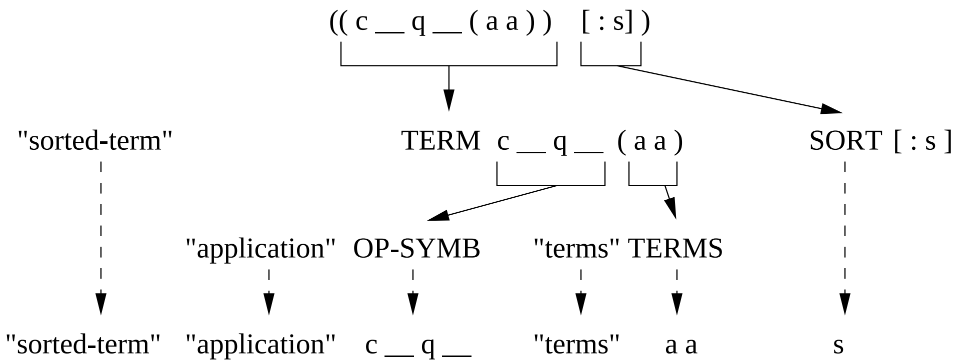


Figure 5.7: Example 2

```
sort s;
var  a,b,d   : s;
op   __ c    : s -> s;
     __ d __ : s -> s;
axioms
    b = a c d a : s
```

Figure 5.8: Example of CASL specification and mixfix parsing

2. The list of terms `a c d a :  s` is examined using the techniques used in Section 5.4. This results in a new list of terms `a c d a [:  s]`.

3. The first element of G' $\_\_c$ is matched with the list of terms. It can be matched with the list of terms but does not result in one of the five cases mentioned in Figure 5.5. But a standard case is that any operation that is matched with a list of terms consisting of a subterm `[:  s]` is correct, if the terms on the left-hand side of `[:s]` form one of the five possibilities mentioned in Figure 5.5. These terms `a c d a` are again examined by the mixfix parser and after a trial and error mechanism it results in an application of $\_\_d\_\_$, $\_\_c$, and a, into $(\_\_d\_\_(\_\_c(a), a))$. So, it is an application with two parameters. These parameters consists of an application with one parameter and a constant. This results in the following abstract syntax:

```
application __ d __ terms
    application __ c terms a
    a
```

4. The total strong equation thus results in :

```
strong-equations
b
application __ d __ terms
    application __ c terms a
    a
```

5.9 Speed and Performance

Some standard list of terms are parsed as single terms and the number of iterations is a measurement of how many reductions are performed. Of course, this is a very rough estimation and not all measurements are given in Figure 5.9 and 5.10 but they give some idea of how the size of the list of terms correlates with the number of reductions and thus with the speed.

A few additional remarks should be made:

- It makes a difference whether a list of terms inhabits a flat structure, i.e., consists of few operations, or a complex structure with many operation applications. Measurements give that around 6–7 constants are parsed in the same time as 2–3 complex operations.

- The size measurements are not quite accurate regarding conditional expressions. Measurements indicate that if the size terms of a conditional expression is greater than 2–3, e.g., `a b when c d else f g` , the time used to parse this equals the size divided by two. So, our example has about the same parsing time as with an axiom `a b c`. Of course, above 3 terms the complexity plays a more important role.

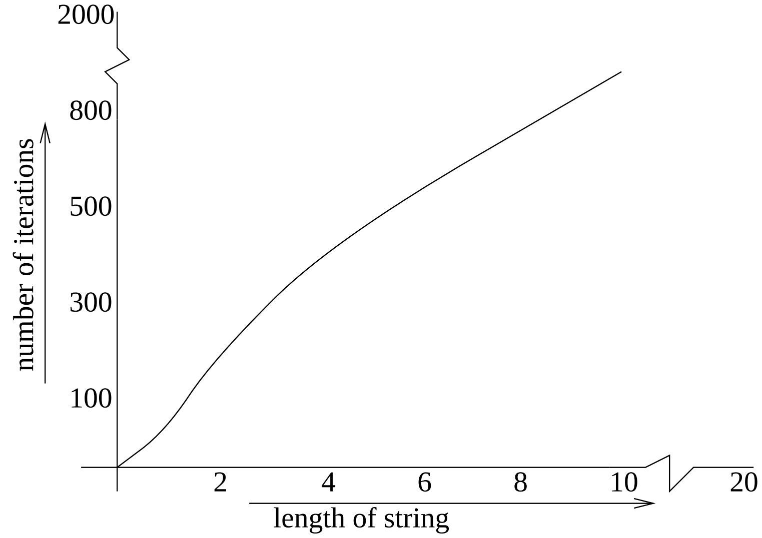


Figure 5.9: Example of mixfix mapping

String	Operations	Iterations
a	a predicate	30
c a	a constant, c __ operation	243
c a q	a constant, c __ q operation	260
c a q a	a,b constant, c __ q __ operation	385
a : s d a w a : s as s	a constant, __d__q__ operation	557
a d (b a d a w a) q a	a constant, b__, __ d __ q __ operation	947

Figure 5.10: Basic Measurements for Figure 5.9

- The number of operations, variables, constants, and predicates play also an important role. If there is not much overlap between the names of these elements, the parsing behaves as depicted in Figure 5.9. More overlap between names means more combinations that should be checked. In this case the figure will show a more vertical behavior for terms with size bigger than 3–4. Also the number of operations play a role. More operations means more overlap. In fact, in Figure 5.11, q is a reversed measurement of the complexity of the list of terms, the overlap of the operations and the number of operations. One could approximately define:

$$q = \frac{1}{complexity\ +\ overlap\ +\ number\ of\ operations}$$

  It is obvious that q' and q'' in Figure 5.11 have a greater overlap and complexity than the original function.

- When calculating the length of a list of terms, the parentheses are not taken into account, so e.g. `(a b) c` is counted as `a b c`. This follows directly from the definition that everything between parentheses is a term, and that such a term is also going through the mixfix parser.

- In the beginning of this chapter we pointed out that the speed was probably exponential. Since the size, complexity, and overlap of G' plays an important role our pre-assumption was both true and false. There exists an exponential blow-up if G' and the length of the term are sufficiently big and there is a big overlap between operations. On the other hand, if these factors stay relatively low then our functions show a linear speed correlation.

So, to draw some conclusions whether the current mixfix parser is good enough for parsing CASL specification we have to look at:

- **Complexity:** In algebraic specification it is not a fashion to write down too much complex (mixfix) operations. Probably is a mixfix operation with 5 placeholders already rare.

- **Overlap:** Measurements give that the overlap between operations has a significant effect on the number of reductions and thus the speed. It is advisable to keep the overlap as low as possible. To show how overlap can increase the speed, assume we have a $G' = \_\_b, a\_\_b, a$. To parse the list of terms $aaaaabbbb$ costs, with the current G', almost 10 times more reductions than with a $G' = a\_\_b, a$. The problem is that the application of $\_\_b$ results in an error, this is discovered at the end. Since every subterm is always parsed with a fresh G' the mistake of applying $\_\_b$ will be repeated, and this explains the blow-up in reductions.

- **Number of operations:** It is not rare to have a huge amount of operations in algebraic specification. We have seen that this can lead to severe problems with respect to above mentioned speed and complexity.

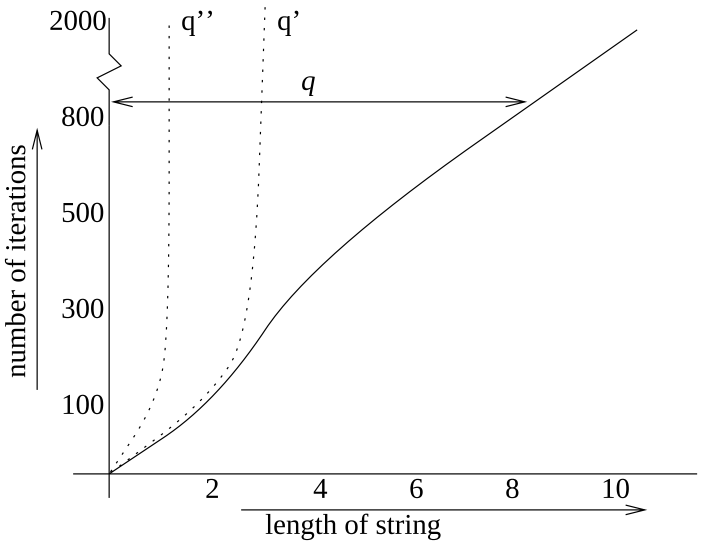


Figure 5.11: Example of mixfix mapping

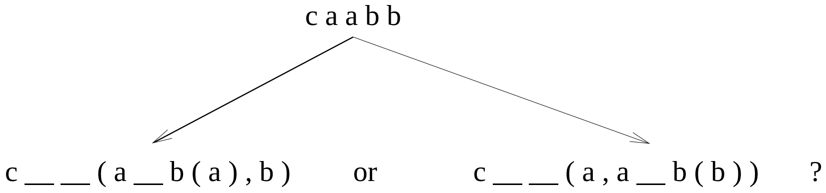


Figure 5.12: Ambiguity

If one sticks to general CASL specifications our mixfix parsing will work very well in almost linear time (small complexity, overlap, and number of operations). If one uses too much complexity, overlap and number of operations, then an exponential blow-up may occur.

5.10 Correctness

Now that we have a mixfix parser the question is of course: does it always do its job correctly? The answer is unfortunately no. Some incorrect CASL specifications are parsed correctly (see Figure 5.12). In some way the second possibility in Figure 5.12 is neglected since the first gives a positive parse. The ambiguity can be discovered when all possibilities are analyzed. In Chapter 6 an ambiguity checker will analyze all possibilities and checks whether the list of terms inhabits an ambiguity, can be parsed correctly, or can not be parsed at all.

5.11 Conclusion and Future Expansion

We can conclude that a mixfix parser can be written for the Basic Items with the following additional assumptions:

- No infinite single placeholder application is allowed since it never gives a result and is therefore not executable.
- All priorities are obeyed, with the addition that the size of two operations in the same notation, can also be used to disambiguate. If no distinction can be made between them, the order in which they occur in the specification is used to disambiguate. Of course, two operations with the same name, size, and structure, i.e., number of arguments, are only mentioned to assure uniqueness for every element.
- All operations, predicates, variables, and constants are available in the specification. We did not look at the operations in imported modules since we are only covering Basic Specifications, not Structured Specifications.
- We convert all correct specifications but unfortunately also some incorrect ones. An ambiguity checker might solve this.

The performance of the system is adequate if we keep the complexity, overlap, and number of operation low. It is very rare in algebraic specifications that the length of terms is very high. Therefore our mixfix parser for CASL is adequate. Especially a high number of operations in combination with huge overlap and complexity can lead to huge memory usage and processor time.

There are some great benefits of using ASF+SDF as implementation language for mixfix parsing. The list matching of ASF+SDF reduces the code since only the general cases should be specified, the rest will be done by recursion. If all general cases do not apply then a default equation is used. This reduces the size of the specification considerably, and has been of great help during the development of the parser.

The CASL grammar has been restricted so that less unwanted parses are "correctly" recognized by ASF+SDF. By removing some ambiguous lines, and specifying the grammar more precisely in ASF+SDF, we removed some ambiguities and made the mixfix parsing more subtle.

Our next investigation is an ambiguity checker so that only correct CASL specifications are converted to abstract syntax. In the current version of the mixfix parser all correct specifications are converted but also some incorrect ones. A future field of investigation can be the Structured Specifications and Libraries. Since the number of operations plays such a big role in the speed, a giant Library could have a huge list G'. Maybe some special search techniques could then be applied to find an appropriate operation, variable, etc.

Chapter 6

Ambiguity Removal

In the previous chapter we stated that all correct CASL `TERM`s were correctly parsed by the mixfix parser, but unfortunately some incorrect ones are correctly parsed as well. To remove an incorrect parse of CASL terms we are introducing an ambiguity checker. This ambiguity checker will try every possibility and yields, if possible, a correct parse. Otherwise, there is not a correct parse since the list of CASL terms contain ambiguities or cannot be parsed at all. One should be aware of that our understanding of a "correct" CASL specifications means only correctness on a syntactical level. While type-checking one can decide that a correctly parsed term is not correct at all since it has conflicting types. As mentioned before we are only parsing mixfix on a syntactical level. In, for example, CafeOBJ [26] and also ASF+SDF the types of the operations are used to parse the mixfix operations.

6.1 Failure of the Mixfix Parser

The parser ignores at a certain stage that there might be more than one correct parse. Experiments with only placeholders as operation names showed that in some very extreme cases the first correct parse has an operation application of lower priority then one of the following parses. The easiest thing to do is to create all possible parses, and give them some kind of priority number based on their operation applications. The one with the highest number is the correct parse. If there is more than one with the same highest number then there is an ambiguity. If there are no parses possible then the string could not be parsed at all.

The only place in the mixfix parser where other possibilities are ignored is in the `applyspecial` function ([35]). When matching `__ mixid` on a string the first positive match is taken[1]. If we had a string *abcd* and `__` can be matched on *a* and *mixid* on *bcd*, then we are ignoring all other possibilities:

[1] Positive in such a sense that it establishes a correct parse.

- __ could be matched on ab and $mixid$ could be matched on cd, or
- __ could be matched on abc and $mixid$ could be matched on d.

To check all these possibilities we have to make a function which generates all these possibilities. If we have parsed these possibilities we have to decide which one gives a correct parse. It could be that none of the possibilities is correct. The list of terms was then not parseable at all. It could be that more than one possibility is correct. The list of terms was then ambiguous.

To generate all the possibilities a simple combination An example of such a function *combine* when we start with a string *abcd*:

$$\begin{aligned}
combine(abcd) &= combine(abc,[abc,d]) \\
combine(abc,[abc,d]) &= combine(ab,[abc,d][ab,cd]) \\
combine(ab,[abc,d][ab,cd]) &= combine(a,[abc,d][ab,cd][a,bcd]) \\
combine(a,[abc,d][ab,cd][a,bcd]) &= [abc,d][ab,cd][a,bcd]
\end{aligned}$$

6.2 Parsing all Possibilities

Now that we have access to all the possibilities we can parse them, by simply using `applyspecial`. Of course, as mentioned before the only operation where ambiguities occur is `__ mixid`[2]. As with ordinary mixfix we check whether the operation names can be matched on the placeholder by using `check-name`[3].

$$[\text{cal1}]\ \frac{\begin{array}{rcl}\text{check-name}(mixid,\ mlplus') &=& \text{true}, \\ \text{applyspecial}(\_\_,\ mlplus,\ sl,\ sl) &=& applist_1, \\ \text{applyspecial}(mixid,\ mlplus',\ sl,\ sl) &=& applist_2, \\ \text{count}(applist_1\ applist_2) &=& Int, \\ \text{check-amblist}(\_\_\ mixid,\ amblist,\ sl) &=& ambtermlist\end{array}}{\begin{array}{l}\text{check-amblist}(\_\_\ mixid,\ amblist\ [mlplus,\ mlplus'],\ sl)\ = \\ ambtermlist\ [applist_1\ applist_2,\ Int]\end{array}}$$

$$[\text{cal2}]\ \frac{\text{check-amblist}(\_\_\ mixid,\ amblist,\ sl)\ =\ ambtermlist}{\text{check-amblist}(\_\_\ mixid,\ amblist\ [mlplus,\ mlplus'],\ sl)\ =\ ambtermlist}\quad \textbf{otherwise}$$

$$[\text{cal3}]\ \text{check-amblist}(\_\_\ mixid,\ ,\ sl) =$$

After having parsed all the possible terms a priority number is given to every term that was parseable. All unparseable possibilities are rejected. This number depends on the number of postfix, prefix, infix, mixfix operations in the parsed string. This number is used to determine whether we are dealing with an ambiguity. In the default case the string is rejected and we continue with the rest.

[2] If we have `token __ mixid` then that is separated in the `applyspecial` function as a single `token` and `__ mixid`.

[3] This was to remove a possible conflict between parts of a mixfix operation name and other operations.

6.3 Determining the Priority Number

The priority number is determined in the following way. Every string consists of a list of parsed terms. Every possible combination of that string is used to match `_ mixid`. Now to every application, in this string, a number is given. A postfix application yields gives four points, prefix three, infix two, mixfix one, and the rest zero. The terms of these applications will also be checked. Points given within these terms are added. The string in combination with the resulting priority number is inserted in an intermediate list. For every string, i.e., every parsed combination, this will be done. For example:

$$(\_\_a(\_\_b\_\_(c)))$$

has priority number $4(\_\_a)+2(\_\_b\_\_)+0(c) = 6$. Since our `apply`-function always returns an application in the following form $(id(terms))$ it is easy to give these points.

[c1] $\dfrac{\text{postfix}(\mathit{id}) \;=\; \text{true}}{\text{count}((\mathit{id}\ (\mathit{applist}))) \;=\; 4 + \text{count}(\mathit{applist})}$

[c2] $\dfrac{\text{prefix}(\mathit{id}) \;=\; \text{true}}{\text{count}((\mathit{id}\ (\mathit{applist}))) \;=\; 3 + \text{count}(\mathit{applist})}$

[c3] $\dfrac{\text{infix}(\mathit{id}) \;=\; \text{true}}{\text{count}((\mathit{id}\ (\mathit{applist}))) \;=\; 2 + \text{count}(\mathit{applist})}$

[c4] $\dfrac{\text{mixfix}(\mathit{id}) \;=\; \text{true}}{\text{count}((\mathit{id}\ (\mathit{applist}))) \;=\; 1 + \text{count}(\mathit{applist})}$

[c5] $\text{count}(\mathit{appelem}) \;=\; 0$
otherwise

[c6] $\text{count}(\mathit{appelem}\ \mathit{appelem}'\ \mathit{applist})$
$= \text{count}(\mathit{appelem}) + \text{count}(\mathit{appelem}'\ \mathit{applist})$
otherwise

6.4 Picking the Right One

By taking the highest priority number available in the intermediate list, we can immediately point out which parse is a correct one or whether there exists an ambiguity.

[ghp1]get-highest-priority-nr() $= 0$

[ghp2]get-highest-priority-nr([*applist*, *Int*]) = *Int*

$$\text{[ghp3]}\ \frac{\mathit{Int}_1 < \mathit{Int}_2 \ = \ \text{true}}{\begin{array}{l}\text{get-highest-priority-nr}([\mathit{applist},\ \mathit{Int}_1]\ [\mathit{applist}',\ \mathit{Int}_2]\ \mathit{ambtermlist}) \ = \\ \text{get-highest-priority-nr}([\mathit{applist}',\ \mathit{Int}_2]\ \mathit{ambtermlist})\end{array}}$$

[ghp4]get-highest-priority-nr([*applist*, Int_1] [*applist*$'$, Int_2] *ambtermlist*)
= get-highest-priority-nr([*applist*, Int_1] *ambtermlist*)
otherwise

By removing every parse with a lower priority number we get a list with only parses with the highest priority number. If the result is an empty list then there is no possible parse. If only one parse, i.e. combination, is left then that must be the right one. In every other case there is an ambiguity.

[dis1] disambiguate() =

[dis2] disambiguate([*applist*, *Int*]) = *applist*

[dis3] disambiguate([*applist*, *Int*] *ambtermlist*)
= [op error `ambiguity cant be solved / impossible parse`] *applist*
otherwise

6.5 General Approach

In general our ambiguity checker works in the following way. Create every combination of the input string (`combine`). Since our entry to the ambiguity checker was the construction (`__ mixid`) this structure should be applied on every combination (`check-amblist`). If a combination cannot be parsed with respect to (`__ mixid`) we do not insert it in the intermediate data-type structure. By giving every operation application and its terms a number we can make a difference in priority between the several combinations. By obtaining the highest priority number, and removing all combinations with lower numbers (`clear-list`) are there three possibilities left (`disambiguate`):

1. No combination are left, so there is no a correct parse at all.

2. A single combination is left, so this is *the* correct parse.

3. More than one combination is left, so there is a real ambiguity.

$$
\text{[pa-1]} \quad \frac{\begin{array}{rcl}
\text{combine}(\mathit{mlplus},) & = & \mathit{amblist}, \\
\text{check-amblist}(\_\ \mathit{mixid}, \mathit{amblist}, \mathit{sl}) & = & \mathit{ambtermlist}, \\
\text{get-highest-priority-nr}(\mathit{ambtermlist}) & = & \mathit{Int}, \\
\text{clear-list}(\mathit{ambtermlist}, \mathit{Int}) & = & \mathit{ambtermlist}', \\
\text{disambiguate}(\mathit{ambtermlist}') & = & \mathit{applist}
\end{array}}{\text{parse-ambiguity}(\_\ \mathit{mixid}, \mathit{mlplus}, \mathit{sl}) \ = \ \mathit{applist}}
$$

6.6 Examples

In the following examples the ambiguity specification is shown. By showing step by step the process, one actually sees what is going on. Only the ambiguity checker part will be shown, the rest is done by the previous mixfix specification.

6.6.1 Ambiguity Example

Assume we have a term *aabbc* and operations $a, b, a\_b, \_\ \_c$. It is obvious that the mixfix parser will apply $\_\ \_c$ on *aabbc*. In the `applyspecial` function will $\_\ \_$ be matched with *aabb* and here starts the ambiguity checker:

1. Create all possibilities: $aabb \rightarrow [a, abb][aa, bb][aab, b]$

2. Remember that we are always using the generalized form `_ mixid` to match with, here the variable `mixid` consists thus of a single placeholder. Now we are parsing every possibility with respect to $\_\ \_$

   - $[a, abb] \rightarrow a(a\_b(b))$
   - $[aa, bb] \rightarrow$ impossible to parse, cannot be matched by $\_\ \_$
   - $[aab, b] \rightarrow (a\_b(a))b$

3. Count every operation application:

   - $a(a\_b(b)) \rightarrow 0 + (1 + (0)) = 1$
   - $(a\_b(a))b \rightarrow (1 + (0)) + 0 = 1$

4. The highest priority number is one. We have two possibilities, so we have an ambiguity.

6.6.2 Parseable Example

If we had a term *aabbc* and operations $a, b, a\_, a\_b, \_b, \_\ \_c$ the previous four points, after applying $\_\ \_c$, are:

1. Create all possibilities: $aabbb \rightarrow [a, abb][aa, bb][aab, b]$

2. Parse possibilities:

- $[a, abb] \rightarrow a(\_\_b(\_\_b(a)))$
- $[aa, bb] \rightarrow (a\_\_(a))(\_\_b(b))$
- $[aab, b] \rightarrow (a\_\_(a\_\_(b)))b$

3. Count every operation application:

- $a(\_\_b(\_\_b(a))) = 0 + (4 + (4 + (0))) = 8$
- $(a\_\_(a))(\_\_b(b)) = (3 + (0)) + (4 + (0)) = 7$
- $a\_\_(a\_\_(b)))b = 3 + (3 + (0)) = 6$

4. The highest number is 8, and only one parse is thus possible $a(\_\_b(\_\_b(a)))$.

6.6.3 Another Ambiguous Example

If we had the following operations $a, b, a\_\_, a\_\_b, \_\_b, \_\_\ \_\_c$ and a term $abbc$ the four points of the first example, after applying $\_\_\ \_\_c$, are:

1. Create all possibilities: $abb \rightarrow [a, bb][ab, b]$
2. Parse possibilities:

- $[a, bb] \rightarrow a(\_\_b(b))$
- $[ab, a] \rightarrow (\_\_b(a))a$

3. Count:

- $a(\_\_b(b)) \rightarrow 0 + (4 + (0)) = 4$
- $(\_\_b(a))a \rightarrow (4 + (0)) + 0 = 4$

4. There are only two possible parses with equal number it is ambiguous.

If we have a close look there is another parse possible: $(a\_\_(b))b$. Since this is a prefix operation and the higher priority postfix operation $\_\_b$ causes an ambiguity, the total term will be ambiguous.

6.7 Ambiguity versus Parseable

In the last example we saw that higher priority operations caused an ambiguity, while a lower one could actually give a correct parse. Here the definition of priority and ambiguities contradicts each other. In the current approach we are working from high to low priority. If higher priority operations causes ambiguities, then the total term is ambiguous. But if a higher priority operation gives a correct parse, but a lower priority causes an ambiguity then the total parse will be correct and not ambiguous.

One could define that if higher priority operations causes ambiguities, the lower operations are consulted. But the rule that we use to go from a priority level to a lower one, is based on the the fact that the priority operations do not produce a *parseable* possibility. If we go to a lower priority level, since a higher one causes ambiguities, then the higher priority operation has a parseable possibility, though more than one. This contradicts with the original rule that one shifts from priority level only when higher priority operations do not create a possible parse. Obeying this rule gives that if higher priority operations give one correct parse, but lower priority operations give an ambiguity, the term *is* correctly parsed.

6.8 Effects on Speed

Measurements give that the average number of reductions is increased by a factor ten. Since the ambiguity checker is involved in the mixfix parsing whenever operations are used with size greater than two[4] it affects the execution speed enormously. In fact the ambiguity checker is the slowest part of the specification since all possibilities must be parsed. One can easily see that the length of the term now definitely is playing a role. The longer the term is, the more possibilities there are. But still the length of the term is not the worst. If we have a number of huge operations which requires one ambiguity check after the other[5] then one encounters exponential growth of speed. Fine measurements give that Figure 5.11 is still quite accurate but that the number of iterations is increased by a factor ten, the form of the figure stays the same.

6.9 Conclusions

Although our ambiguity checker increases the number of iterations with a factor ten with respect to the length of the term, the complexity and number of operations is still the worst factor[6]. We were able to specify an ambiguity checker since every possibility is investigated and a number is assigned to it. All these values, i.e., priority numbers, are used perform the disambiguation. How this is done is not arbitrary; one could chose for another approach, e.g., that if a higher operation causes ambiguities and lower ones does not, that a correct parse with the lower operation is correct. Since the shift from a higher priority level to a lower one always depends on unparsability, so an ambiguity is not enough to do that.

[4] The size of example operation $c\_\_\ \_$ is three.
[5] In a recursively notion of course.
[6] As mentioned in the previous chapter.

Chapter 7

Concluding Remarks

An introduction to CASL was given. From this introduction a BNF like CASL grammar is converted into SDF. To avoid complex priority notation in SDF an Algol–like priority method is used. All CASL semantics are defined on the non–ambiguous abstract syntax, therefore a conversion from concrete to abstract CASL is the next milestone. We developed two standard conversions: a conditional approach and a linear approach. Due to readability was chosen for a linear approach. During the conversion quite ambiguous concrete parts were discovered in the form of mixfix notation. To establish a correct conversion we were forced to analyze and convert mixfix notation in a correct way.

Mixfix notation To actually parse the ambiguous mixfix notation we first removed some ambiguous parts by specifying the CASL grammar somewhat different in SDF so that less ambiguities were created. From this point, we separated the mixfix term in two parts (`expansion`):

1. All the parts of the terms that are automatically recognized, i.e., there was no other parse possible for those parts. For example, if there did not exist an application a but only a constant, $a(a)$ could only be seen as one constant and one parenthesized constant. All the information we had on the operations is used: operation name, number of argument sorts, absence of operations, constants, or variables, etc. to recognize as much as possible.

2. All the parts of the terms that could not be automatically identified were left alone. The future mixfix parser deals with them.

Essentially parentheses are used to state whether some parts belong to each other or not. The mixfix parser uses the same technique of inserting parentheses to state which parts belong to each other or not. The mixfix parser is simply a trial and error mechanism. All the operations, constants and variables are examined whether a match on (parts) of the terms is possible. In fact the mixfix parser is a surjective function $f : A \to B$ where A is the list of operations, variables, and constants and B the term. The mixfix parser yields a parenthesized term. A simple conversion to abstract syntax is now made since the parenthesizes removed all the ambiguities, if the mixfix parser ends in a positive result[1].

The question came up whether we removed all ambiguities from the mixfix term. Unfortunately some nasty ambiguities were still not found. By creating all possible parses and a level of priority[2], we were able to remove these ambiguities successfully.

Speed and adequacy Our mixfix parser without ambiguity checker, shows a linear relation if the number of overlap and complexity was relatively low. In the case of too much overlap and complexity an exponential growth can be observed. After specifying the ambiguity checker these investigations were still valid although the number of reductions increased with a factor ten. The length of the term was going to play a bigger role then before. If we had a huge term but a small number of overlap and complexity it is easier to decide whether a combination of the term is correct or not since there is not much overlap and complexity. This relates to the fact that it is easier to detect a negative match if the overlap and complexity is low. Ergo, if the overlap and complexity is high it will be more difficult to recognize a positive match.

Near future work Since other tool builders are not so anxious to develop and specify a mixfix parser for CASL a bridge must be developed between our current mixfix parser and their parsers. This bridge is using the intermediate representation of CASL. This representation is based on ATerms [30]. Other tool builders produce abstract syntax trees in ATerms with unparsed or better incorrect parsed abstract syntax. Our bridge will gather the information about all the operations, predicates, variables, and constants from the ATerms representation. All the abstract `Terms` are reconstructed. From this point the mixfix parser can do its job and convert the `Terms` (see Figure 7.1). The author has extended Mark van den Brands work [30] by updating his specification. This led to a specification which was able to convert from abstract syntax to ATerms and vice versa.

[1] If the mixfix parser does not end in a positive result a conversion to abstract syntax is of course not possible.

[2] Our so called priority number.

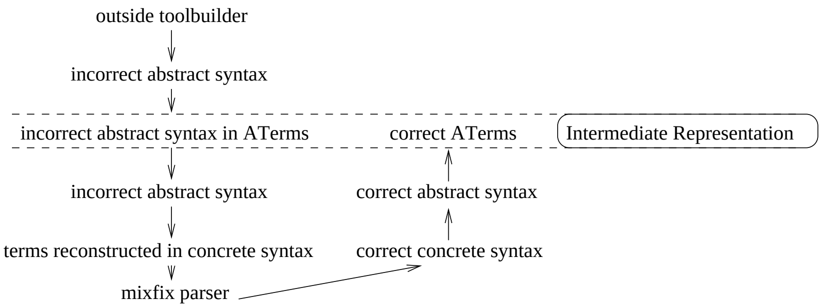


Figure 7.1: Using the current mixfix parser for other tool builders

Appendices

Syntax[3]

The grammars presented in appendices A and B are from CASL v0.99. The grammar in appendices A and B uses uppercase words for nonterminal symbols, allowing also hyphens. All other characters stand for themselves, with the following exceptions:

- ::= and | are generally used as meta-notation, as in BNF.
- A string of characters enclosed in double quotation marks { ... } always stands for the enclosed characters themselves.
- $N\ t \ldots t\ N$ indicates one or more repetitions of the symbol N separated by the symbol t (which is usually a terminal symbol, e.g. a comma or semicolon).
- $N \ldots$ is simply one or more repetitions of N.
- `var/vars` indicates that the singular and plural forms may be used interchangeably, and similarly for other keywords; `end/` indicates that the use of `end` is optional, and similarly for semicolons: `;/`.

The given grammar for input syntax is quite ambiguous. Chapter 3 in [20] provides some precedence rules for disambiguation. Note that the syntax for atomic formulae and terms takes care of the use of mixfix notation when applying predicates and operations; however, the recognition of the exact applicative structure of these constructs is left unspecified here, since it is inherently context-dependent. When mixfix applications are not used, the disambiguation ensures that normal applications are recognized.

Appendix A: Concrete Syntax

```
BASIC-SPEC      ::= BASIC-ITEMS...BASIC-ITEMS  |  { }
```

[3]This text is written by P.D.Mosses and can be found in extended form in [20]

```
BASIC-ITEMS     ::= SIG-ITEMS
                  | free type/types DATATYPE-DECL ;...; DATATYPE-DECL ;/
                  | generated type/types DATATYPE-DECL ;...; DATATYPE-DECL ;/
                  | generated { SIG-ITEMS...SIG-ITEMS } ;/
                  | var/vars VAR-DECL ;...; VAR-DECL ;/
                  | var/vars VAR-DECL ;...; VAR-DECL
                    "." FORMULA "."..."." FORMULA ;/
                  | axiom/axioms FORMULA ;...; FORMULA ;/

SIG-ITEMS       ::= sort/sorts SORT-ITEM ;...; SORT-ITEM ;/
                  | op/ops OP-ITEM ;...; OP-ITEM ;/
                  | pred/preds PRED-ITEM ;...; PRED-ITEM  ;/
                  | type/types DATATYPE-DECL ;...; DATATYPE-DECL ;/

SORT-ITEM       ::= SORT ,..., SORT
                  | SORT ,..., SORT < SORT
                  | SORT = { VAR : SORT "." FORMULA }
                  | SORT =...= SORT

OP-ITEM         ::= OP-NAME ,..., OP-NAME : OP-TYPE
                  | OP-NAME ,..., OP-NAME : OP-TYPE , OP-ATTR ,..., OP-ATTR
                  | OP-NAME OP-HEAD = TERM

OP-TYPE         ::= SOME-SORTS  -> SORT  |  SORT
                  | SOME-SORTS  ->? SORT  |  ? SORT

SOME-SORTS      ::= SORT *...* SORT

OP-ATTR         ::= assoc  |  comm  |  idem  |  unit TERM

OP-HEAD         ::= ( ARG-DECL ;...; ARG-DECL ) : SORT   |  :  SORT
                  | ( ARG-DECL ;...; ARG-DECL ) :? SORT  |  :? SORT
ARG-DECL        ::= VAR ,..., VAR : SORT

PRED-ITEM       ::= PRED-NAME ,..., PRED-NAME : PRED-TYPE
                  | PRED-NAME PRED-HEAD <=> FORMULA
                  | PRED-NAME <=> FORMULA

PRED-TYPE       ::= SOME-SORTS  |  ( )

PRED-HEAD       ::= ( ARG-DECL ;...; ARG-DECL )

DATATYPE-DECL   ::= SORT "::=" ALTERNATIVE "|"..."|" ALTERNATIVE
ALTERNATIVE     ::= OP-NAME  ( COMPONENT ;...; COMPONENT )
                  | OP-NAME  ( COMPONENT ;...; COMPONENT )?
                  | OP-NAME
```

```
                 | sort/sorts SORT ,..., SORT
COMPONENT      ::= OP-NAME ,..., OP-NAME :  SORT
                 | OP-NAME ,..., OP-NAME :? SORT
                 | SORT

VAR-DECL       ::= VAR ,..., VAR : SORT

FORMULA        ::= QUANTIFIER  VAR-DECL ;...; VAR-DECL "." FORMULA
                 | FORMULA /\ FORMULA /\.../\ FORMULA
                 | FORMULA \/ FORMULA \/...\/ FORMULA
                 | FORMULA => FORMULA
                 | FORMULA if FORMULA
                 | FORMULA <=> FORMULA
                 | not FORMULA
                 | true | false
                 | PRED-SYMB
                 | PRED-SYMB ( TERMS )
                 | TERM TERM...TERM
                 | def TERM
                 | TERM =e= TERM
                 | TERM = TERM
                 | TERM in SORT
                 | ( FORMULA )

QUANTIFIER     ::= forall  |  exists  |  exists!

PRED-SYMB      ::= PRED-NAME  |  ( pred QUAL-PRED-NAME )
QUAL-PRED-NAME ::= PRED-NAME : PRED-TYPE

TERMS          ::= TERM ,..., TERM

TERM           ::= SIMPLE-TERM...SIMPLE-TERM

SIMPLE-TERM    ::= ID
                 | ( var VAR : SORT )
                 | ( op QUAL-OP-NAME )
                 | OP-SYMB ( TERMS )
                 | SIMPLE-TERM : SORT
                 | SIMPLE-TERM as SORT
                 | TERM when FORMULA else TERM
                 | ( TERM )

OP-SYMB        ::= OP-NAME  |  ( op QUAL-OP-NAME )
QUAL-OP-NAME   ::= OP-NAME : OP-TYPE

SORT           ::= TOKEN-ID
```

```
OP-NAME        ::= ID
PRED-NAME      ::= ID
VAR            ::= SIMPLE-ID

ID             ::= TOKEN-ID | MIXFIX-ID
TOKEN-ID       ::= TOKEN
MIXFIX-ID      ::= TOKEN-OR-PLACE...TOKEN-OR-PLACE
TOKEN-OR-PLACE ::= TOKEN | PLACE
TOKEN          ::= WORDS | SIGNS | DOT-WORDS
SIMPLE-ID      ::= WORDS
```

Appendix B: Abstract Syntax

```
BASIC-SPEC      ::= basic-spec BASIC-ITEMS*

BASIC-ITEMS     ::= SIG-ITEMS
                  | free-datatype DATATYPE-DECL+
                  | sort-gen SIG-ITEMS+
                  | var-items VAR-DECL+
                  | local-var-axioms VAR-DECL+ FORMULA+
                  | axiom-items FORMULA+

SIG-ITEMS       ::= sort-items SORT-ITEM+
                  | op-items OP-ITEM+
                  | pred-items PRED-ITEM+
                  | datatype-items DATATYPE-DECL+

SORT-ITEM       ::= sort-decl SORT+
                  | subsort-decl SORT+ SORT
                  | subsort-defn SORT VAR SORT FORMULA
                  | iso-decl SORT+

OP-ITEM         ::= op-decl OP-NAME+ OP-TYPE OP-ATTR*
                  | op-defn OP-NAME  OP-HEAD TERM
OP-TYPE         ::= total-op-type   SORTS SORT
                  | partial-op-type SORTS SORT
SORTS           ::= sorts SORT*
OP-ATTR         ::= associative | commutative | idempotent
                  | unit-op-attr TERM
OP-HEAD         ::= total-op-head   ARG-DECL* SORT
                  | partial-op-head ARG-DECL* SORT
ARG-DECL        ::= arg-decl VAR+ SORT

PRED-ITEM       ::= pred-decl PRED-NAME+ PRED-TYPE
                  | pred-defn PRED-NAME  PRED-HEAD FORMULA
```

```
PRED-TYPE        ::= pred-type SORTS
PRED-HEAD        ::= pred-head ARG-DECL*

DATATYPE-DECL    ::= datatype-decl SORT ALTERNATIVE+
ALTERNATIVE      ::= total-construct   OP-NAME COMPONENTS*
                   | partial-construct OP-NAME COMPONENTS*
                   | subsorts SORT+
COMPONENTS       ::= total-select   OP-NAME+ SORT
                   | partial-select OP-NAME+ SORT
                   | SORT

VAR-DECL         ::= var-decl VAR+ SORT

FORMULA          ::= quantification QUANTIFIER VAR-DECL+ FORMULA
                   | conjunction FORMULA+
                   | disjunction FORMULA+
                   | implication FORMULA FORMULA
                   | equivalence FORMULA FORMULA
                   | negation FORMULA
                   | true | false
                   | predication PRED-SYMB TERMS
                   | definedness TERM
                   | existl-equation TERM TERM
                   | strong-equation TERM TERM
                   | membership TERM SORT
QUANTIFIER       ::= forall | exists | exists-uniquely
PRED-SYMB        ::= PRED-NAME | QUAL-PRED-NAME
QUAL-PRED-NAME   ::= qual-pred-name PRED-NAME PRED-TYPE

TERMS            ::= terms TERM*
TERM             ::= SIMPLE-ID
                   | qual-var VAR SORT
                   | application OP-SYMB TERMS
                   | sorted-term TERM SORT
                   | cast TERM SORT
                   | conditional TERM FORMULA TERM
OP-SYMB          ::= OP-NAME | QUAL-OP-NAME
QUAL-OP-NAME     ::= qual-op-name OP-NAME OP-TYPE

SORT             ::= TOKEN-ID
OP-NAME          ::= ID
PRED-NAME        ::= ID
VAR              ::= SIMPLE-ID

ID               ::= TOKEN-ID | MIXFIX-ID
TOKEN-ID         ::= TOKEN
```

```
MIXFIX-ID        ::= token-places TOKEN-OR-PLACE+
TOKEN-OR-PLACE   ::= TOKEN | PLACE
TOKEN            ::= WORDS | SIGNS | DOT-WORDS
SIMPLE-ID        ::= WORDS
```

Appendix C: Examples

The following examples are given to give better insight in the mixfix parsing. Not every tricky specification is given. Every specification goes through the mixfix parser and results in (in)correct abstract syntax trees.

PO

The partial order is described the three axioms below. For the less-then-equal sign we have chosen "<" in the ASCII-text that is being parsed.

```
sort T;

pred __ < __ : T*T

vars x,y,z : T

. x < x
. x = y if x < y /\ y < x
. x < z if z< y /\ y < z
```

This specification results after the conversion to abstract syntax in:

```
basic-spec
sort-items sort-decl T
pred-items pred-decl
             __ < __
             pred-type sorts T T
local-var-axioms
  var-decl
    x y z
    T
   predication
     __ < __
     terms x x
   implication
    strong-equation
      x
      y
    conjunction predication
```

```
                  __ < __
                  terms x y
                predication
                  __ < __
                  terms y x

   implication predication
                  __ < __
                 terms x z
               conjunction predication
                              __ < __
                              terms x y
                           predication
                              __ < __
                              terms y z
```

Unfortunately, the result of the ASF+SDF pretty printer is not so good, but the spacing must give an idea how the abstract syntax tree looks like.

MONOID

```
sort M;
ops  n,m;
     __*__ : M*M -> M, assoc, unit n;
vars x,y,z : M

. n*x=x
. x*n=x
.(x*y)*z=x*(y*z)
```

After the conversion gives:

```
basic-spec
sort-items sort-decl M
op-items op-decl
           n
           total-op-type
             sorts
             M
            op-decl
              __ * __
              total-op-type
                sorts M M
                M
              associative unit-op-attr n
local-var-axioms
  var-decl
```

```
    x y z
    M
  strong-equation
    application
      -- * --
      terms n x
    x strong-equation
        application
          -- * --
          terms x n
        x strong-equation
            application
              -- * --
              terms application
                      -- * --
                      terms x y z
            application
              -- * --
              terms x application
                        -- * --
                        terms y z
```

Ambiguity Example

To show what happens if an ambiguity is stated, we present the follow familiar specification:

```
ops a __ b : s   -> s;
    __ __ c: s*s -> s;

vars a,b : s

. a a b b c = a
```

The term section of this specification results after the conversion to abstract syntax in:

```
 strong-equation
    application
      -- -- c
      terms
      c2amix(
        [ op error ambiguity cant be solved / impossible parse ],
        ( a , variable , 0 )   ( b , variable , 0 )
        ( __ __ c , application , 2 ) ( a __ b , application , 1 ),
```

```
        ( a ) ( b ) ( a __ b : s -> s ) ( __ __ c : s*
                                                 s -> s ))
      application
        a __ b
        terms a b
    a
```

As mentioned in the conlusion error handling is not well specified. We add parenthesizes around the terms, to disambiguate and define what we actually mean. Our axiom thus look for example like:

```
. a (a b b) c = a
```

Wich results in:

```
 strong-equation
                              application
                                __ __ c
                                terms a application
                                          a __ b
                                          terms b
                              a
```

One–Two placeholder problem

As mentioned in Chapter 6 there can arise problems when an operation is matched on two or one placeholder(s). Take for example specification:

```
a __ b : s   -> s;
  __ b : s   -> s;
__ __ c: s*s -> s

vars a,b : s

. a a b c = a
```

Since operation c has two placeholders we must have an application of $\_\_\ b$:

```
 strong-equation
        application
            __ __ c
           terms a application
                       __ b terms a
        a
```

But if we removed one placeholder of c we obtain an application of $a\_\_b$:

```
strong-equation
       application
           __ c
          terms application
                    a __ b  terms a
       a
```

Bibliography

[1] Leslie Lamport, *LaTeX, A Document Preparation System*, Addison-Wesley, 2nd edition

[2] Goossens et al, *The LaTeXCompanion*, Addison-Wesley, 1994

[3] Loeckx et al, *Specification of Abstract Data Types*, Wiley-Teubner, 1996

[4] Aho, Sethi & Ullman, *Compilers: Principles, Techniques and Tools*, 1986, Bell Laboratories, Incorporated

[5] J.A. Bergstra, J.W. Klop & A. Middeldorp, *Termherschrijfsystemen*, 1989, Kluwer Programmatuurkunde

[6] J.A. Bergstra *Terminologie van Algebraïsche Specificaties*, 1987, Kluwer Programmatuurkunde

[7] W. Bouma *Algebraïsche Specificaties*, 1991, Kluwer Programmatuurkunde

[8] J.A.Bergstra & J.W. Klop *Semi-complete termherschrijfsystemen*, 1987, Kluwer Programmatuurkunde

[9] P. Mosses CoFI*: The common framework initiative for algebraic specifications*, Bulletin of the EATCS, Jun196

[10] P. Mosses CoFI*, The Common Framework Initiative*, Rationale Version 0.97

[11] Goguen et al. *Applications of Algebraic Specification using OBJ*, Cambridge Press, 1992, J.A. Goguen, D. Coleman and R.Gallimore.

[12] Guttag, James & Horning *Report on the Larch Shared Language*, Version 2.3, John V. Guttag, James J. Horning, Adres Modet, 1990, digitial systems research center.

[13] A.van Deursen *Executable Language Definitions*, PhD Thesis, Univerisity of Amsterdam, 1994

[14] M.G.J. van den Brand & P. Klint *Programmeeromgevingen I*, Overheadsheets I and II, 1995

[15] P. Klint *The* ASF+SDF *Meta-environment*, User's Guide, November 1993

[16] J. Heering, P.R.H. Hendriks, P.Klint and J. Rekers *The Syntax Definition Formalism SDF*, Reference Manual, 1993

[17] T. Mossakowski *Sublanguages of CASL*, CoFI note L-7, available at: *ftp.brics.dk/Projects/CoFI/Notes/L-7/*

[18] CoFI Semantics Task Group *The* CoFI *Algebraic Specification Language*, Tentative design: Language Summary, version 0.95, *ftp.brics.dk/Projects/CoFI/Notes/S-4/*

[19] CoFI Task Group on Language Design *The Common Algebraic Specification Language: a Summary*, version 0.99, january 1998, *ftp.brics.dk/Projects/CoFI/Documents/CASL/Summary-v0.99/*

[20] M.Bidoit, C.Choppy, B. Krieg-Brückner, P. Mosses and F. Voisin *Concrete Syntax for CASL*, version 0.99, january 1998, *ftp.brics.dk/Projects/CoFI/Documents/CASL/SyntaxIssues-v0.99/*

[21] CoFI Task Group on Semantics CASL *Semantics*, version 2.0, CASL version 0.97, oct 1997 *ftp.brics.dk/Projects/CoFI/Notes/S-6/*

[22] A. van Deursen & P. Mosses *ASD: The Action Semantic Description Tools: The Implementation*, Version 2.03, 1996

[23] P. Mosses CASL *for* ASF+SDF *Users*, september 1997, *ftp.brics.dk/Projects/CoFI/Documents/Tentative/Mosses97ASF+SDF/*

[24] M. Bidoit, D. Sannella & A Tarlecki *Architectural Specifications in* CASL, CoFI note M-4, january 1998 *ftp.brics.dk/Projects/CoFI/Notes/M-4/*

[25] T. Mossakowski *Subsorting and structured specification*, CoFI note S-3, march 1997 *ftp.brics.dk/Projects/CoFI/Notes/S-3/*

[26] T. Nakagawa, T. Sawada, K. Futatsugi *CafeOBJ User's Manual* version 1.3 *http://www.ldl.jaist.ac.jp/cafeobj/manual/cafe-manual.ps.gz*

[27] M.G.J. van den Brand, P. Klint, P. Olivier and E. Visser *Syntax trees for distributed environments* June 12, 1996

[28] A. Sellink and C. Verhoef *Development, Assessment, and Reengineering of Language Descriptions* *http://adam.wins.uva.nl/˜x/cale/*

[29] A.D. McGettrick *Algol 68 a first and second course*, Cambridge University Press, 1987

[30] M.G.J. van den Brand, P. Klint, P. Olivier *ATerms: Exchanging Data between Heterogeneous Tools for* CASL, March 23, 1998

Technical Reports of the Programming Research Group

Note: These reports can be obtained using the technical reports overview on our WWW site (`http://www.wins.uva.nl/research/prog/reports/`) or by anonymous ftp to `ftp.wins.uva.nl`, directory `pub/programming-research/reports/`.

[P9809] B. Wedemeijer. *Introduction & Basic Tooling for* CASL *using* ASF+SDF.

[P9808] J.A. Bergstra and A. Ponse. *Two Recursive Generalizations of Iteration in Process Algebra.*

[P9807] M.P.A. Sellink and C. Verhoef. *An Architecture for Software Maintenance.*

[P9806] M. B. van der Zwaag. *Some Verifications in Process Algebra with Iota.*

[P9805] M.P.A. Sellink and C. Verhoef. *Development, Assessment, and Reengineering of Language Descriptions.*

[P9804] M.P.A. Sellink and C. Verhoef. *Native Patterns.*

[P9803] J.J. Brunekreef. *Annotated Algebraic Specification of the Syntax and Semantics of the Programming Language* Alma-0.

[P9802] W. Fokkink and C. Verhoef. *Conservative Extension in Positive/Negative Conditional Term Rewriting with Applications to Software Renovation Factories.*

[P9801] P. Klint and C. Verhoef. *Evolutionary Software Engineering: A Component-based Approach.*

[P9726] J.A. Bergstra and A. Ponse. *Grid Protocol Specifications.*

[P9725] J.A. Bergstra and A. Ponse. *Process Algebra Primitives for File Transfer.*

[P9724] J.A. Bergstra and A. Ponse. *Process Algebra with Four-Valued Logic.*

[P9723] J.A. Bergstra and A. Ponse. *Kleene's Three-Valued Logic and Process Algebra.*

[P9722] J.A. Bergstra and A. Ponse. *Bochvar-McCarthy Logic and Process Algebra.*

[P9721] M.P.A. Sellink and C. Verhoef. *Reflections on the Evolution of COBOL.*

[P9720] M. van der Graaf. *A Specification of Box to HTML in ASF+SDF.*

[P9719] M. van den Brand, M.P.A. Sellink, and C. Verhoef. *Current Parsing Techniques in Software Renovation Considered Harmful.*

[P9718] M. de Jonge. *Reuse of ASF+SDF Specifications by means of Renaming.*

[P9717] A. Sellink (Ed.). *Participants Proceedings of the 2nd International Workshop on the Theory and Practice of Algebraic Specifications.*

[P9717] E. Visser. *From Context-free Grammars with Priorities to Character Class Grammars.*

[P9716] H.M. Sneed. *Dealing with the Dual Crisis — Year 2000 and Euro — What Reverse Engineering can do to Help.*

[P9715] W. Fokkink and C. Verhoef. *An SOS Message: Conservative Extension in Higher Order Positive/Negative Conditional Term Rewriting.*

[P9714] M. van den Brand, M.P.A. Sellink, and C. Verhoef. *Control Flow Normalization for COBOL/CICS Legacy Systems.*

[P9713] B. Diertens. *Simulation and Animation of Process Algebra Specifications.*

[P9712] V. Partington. *Implementation of an Imperative Programming Language with Backtracking.*

[P9711] L. Moonen. *A Generic Architecture for Data Flow Analysis to Support Reverse Engineering.*

[P9710] B. Luttik and E. Visser. *Specification ofRewriting Strategies.*

[P9709] J.A. Bergstra and M.P.A. Sellink. *An Arithmetical Module for Rationals and Reals.*

[P9708] E. Visser. *Character Classes.*

[P9707] E. Visser. *Scannerless Generalized-LR Parsing.*

[P9706] E. Visser. *A Family of Syntax Definition Formalisms.*

[P9705] M.G.J. van den Brand, M.P.A. Sellink, and C. Verhoef. *Generation of Components for Software Renovation Factories from Context-free Grammars.*

[P9704] P.A. Olivier. *Debugging Distributed Applications Using a Coordination Architecture.*

[P9703] H.P. Korver and M.P.A. Sellink. *A Formal Axiomatization for Alphabet Reasoning with Parametrized Processes.*

[P9702] M.G.J. van den Brand, M.P.A. Sellink, and C. Verhoef. *Reengineering COBOL Software Implies Specification of the Underlying Dialects.*

[P9701] E. Visser. *Polymorphic Syntax Definition.*

[P9618] M.G.J. van den Brand, P. Klint, and C. verhoef. *Re-engineering needs Generic Programming Language Technology.*

[P9617] P.I. Manuel. *ANSI Cobol III in SDF + an ASF Definition of a Y2K Tool.*

[P9616] P.H. Rodenburg. *A Complete System of Four-valued Logic.*

[P9615] S.P. Luttik and P.H. Rodenburg. *Transformations of Reduction Systems.*

[P9614] M.G.J. van den Brand, P. Klint, and C. Verhoef. *Core Technologies for System Renovation.*

[P9613] L. Moonen. *Data Flow Analysis for Reverse Engineering.*

[P9612] J.A. Hillebrand. *Transforming an ASF+SDF Specification into a ToolBus Application.*

[P9611] M.P.A. Sellink. *On the conservativity of Leibniz Equality.*

[P9610] T.B. Dinesh and S.M. Üsküdarlı. *Specifying input and output of visual languages.*