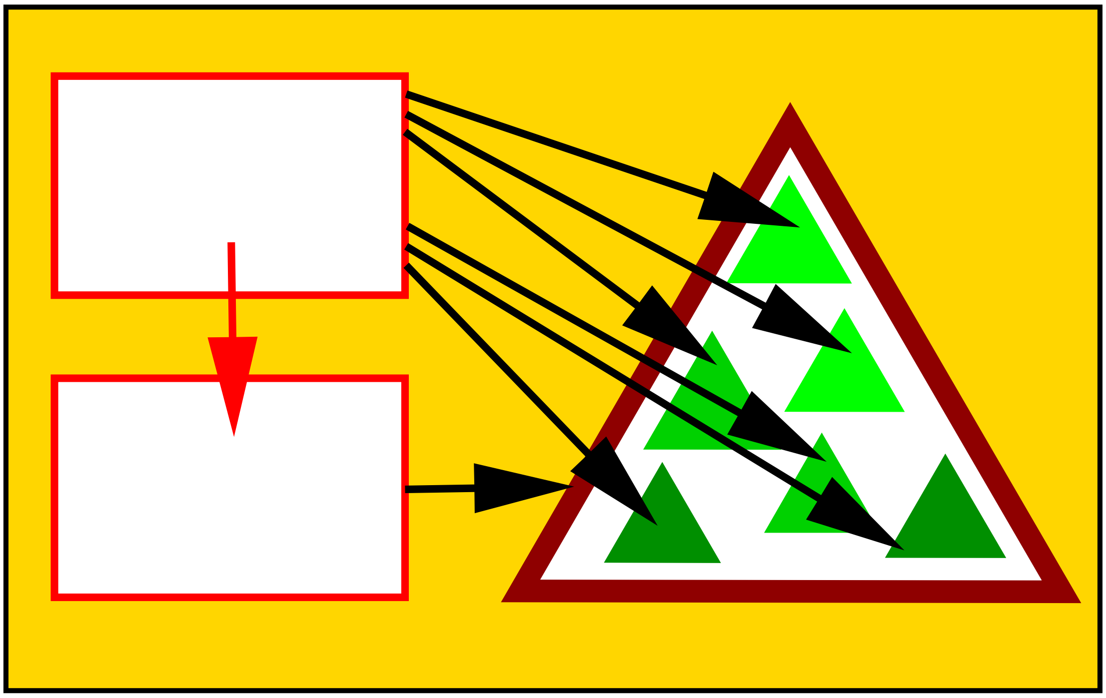
University of Amsterdam

Programming Research Group



Polymorphic Syntax Definition

Eelco Visser

Report P9701

January 1997

University of Amsterdam

Department of Computer Science

Programming Research Group

Polymorphic syntax definition

Eelco Visser

Report P9701

January 1997

E. Visser

Programming Research Group
Department of Computer Science
University of Amsterdam

Kruislaan 403
NL-1098 SJ Amsterdam
The Netherlands

tel. +31 20 525 7590
e-mail: visser@wins.uva.nl

The research was conducted with support from the Netherlands Organization for Scientific Research (NWO) under grant 612-317-420: Incremental parser generation and context-dependent disambiguation, a multi-disciplinary perspective.

Universiteit van Amsterdam, 1997

Polymorphic Syntax Definition

Eelco Visser [1]

*Programming Research Group, Department of Computer Science,
University of Amsterdam, Kruislaan 403, NL-1098 SJ Amsterdam,
The Netherlands*

Abstract. Context-free grammars are used in several algebraic specification formalisms instead of first-order signatures for the definition of the structure of algebras, because grammars provide better notation than signatures. The rigidity of these first-order structures enforces a choice between strongly typed structures with little genericity or generic operations over untyped structures. In two-level signatures level 1 defines the algebra of types used at level 0 providing the possibility to define polymorphic abstract data types. Two-level grammars are the grammatical counterpart of two-level signatures. This paper discusses the correspondence between context-free grammars and first-order signatures, the extension of this correspondence to two-level grammars and signatures, examples of the usage of two-level grammars for polymorphic syntax definition, a restriction of the class of two-level grammars for which the parsing problem is decidable, a parsing algorithm that yields a minimal and finite set of most general parse trees for this class of grammars, and a proof of its correctness.

Key words: Abstract data types, Algebraic Signatures, Decision problems, Design, Grammars, Languages, Language design, Notation, Parsing, Polymorphism, Type systems, Syntax, Two-level grammars

1 Introduction

In the algebraic approach to programming language specification, languages are considered as algebras. A sentence, program or expression in a language is an object of its algebra. The constructs for composition of expressions from smaller expressions and the operations that interpret, translate, transform or analyze expressions are the operations of the algebra. Algebraic specifications describe algebras by means of a finite structure that describes the sorts of the algebra, its operations and the relations between the operations. Any algebra that has the structure prescribed by the specification and that satisfies its relations is a model of the specification. Therefore, a specification always describes

[1] Supported by the Netherlands Organization for Scientific Research (NWO) under grant 612-317-420: Incremental parser generation and context-dependent disambiguation, a multi-disciplinary perspective.

10 January 1997

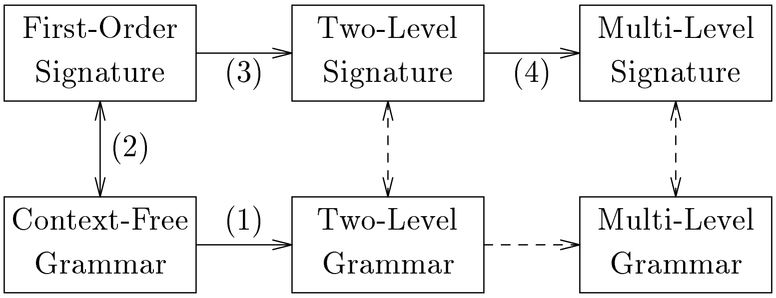
a class of algebras instead of precisely the intended algebra. There are many formalisms for algebraic specification. Depending on the expressive power of a formalism the class of algebras described by a specification can be narrowed down to the intended algebra. First-order algebraic specifications consist of a first-order signature and a set of equations over the terms generated by the signature. A first-order signature consists of a finite set of sorts and a finite number of operations over those sorts.

Grammars describe languages by means of a finite structure that describes the syntactic categories of a language and the sentences of its categories. Context-free grammars and first-order signatures generate the same class of algebras. Parse trees or abstract syntax trees can be considered as terms over a signature and the language of terms over a signature can be described by a context-free grammar (Hatcher and Rus, 1976, Goguen *et al.*, 1977). This correspondence is exploited in several algebraic specification formalisms by allowing the use of signatures with mix-fix operators (Futatsugi *et al.*, 1985, Bidoit *et al.*, 1989) or even arbitrary context-free grammars (Heering *et al.*, 1989) instead of just prefix function signatures. This provides concrete notation for functions and constructors in data type specifications and it enables definition of operations on programming languages directly in their syntactic constructs.

The rigidity of first-order signatures and context-free grammars makes it difficult to generically describe properties of an algebra. For example, an algebra with lists of integers and lists of strings can be specified with a first-order signature by declaring a sort LI (list of integers) and a sort LS (list of strings) and by defining operations like the empty list, cons, head, tail and concatenation on both sorts. However, if these list sorts have the same properties independent of the contents of the lists for some, or all, operations, this can not be expressed in a first-order specification. Similarly, if for both list sorts an operation exists that applies a function to each element of a list, this can not be expressed in a generic way in a first-order specification. This lack of genericity makes it difficult to develop libraries with specifications of common data types and generic language constructs.

A higher type algebra (Meinke, 1992b) is an algebra with an algebraic structure imposed on the set of sorts, i.e., the set of sorts is itself an algebra with operations. These sort operators are interpreted as functions from collections of carrier sets to collections of carrier sets. For instance, the sorts LI and LS above can be seen as sorts constructed from the sorts I (integer) and S (string) by the sort operator L that constructs the sort of sequences of integers and strings, respectively. In such algebras more generic statements about (classes of) objects and operations of the algebra can be made. For example, one can say that, for an arbitrary sort x, the tail function is a function from Lx to Lx that yields the argument sequence without its first element, where we abstract from the fact that x is equal to I or S. One could say that higher type algebras provide a higher resolution in the sort space of an algebra. Algebraic specifications in higher types (Poigné, 1986, Möller, 1987, Meinke, 1992a, Hearn and Meinke, 1994, Visser, 1996) describe higher type algebras by means of two (or more) levels of signatures. Each level specifies the sort operations for the next level, i.e., the terms over the signature at level $i + 1$ are the sort expressions of the signature at level i. Sort expressions with variables are polymorphic sorts that describe all sorts obtained by substituting sorts for the variables. Polymorphic sorts are used to specify polymorphic functions that uniformly apply to many sorts.

In this paper we discuss polymorphic syntax definition by means of context-free and two-level grammars. We argue that the grammatical counterpart of algebraic specifications with two-levels are two-level grammars. This correspondence can be extended to multi-level signatures leading to multi-level grammars. The connections between the various formalisms are summarized by the following diagram:



where we refer to the following literature: (1) (van Wijngaarden *et al.*, 1976, Pereira and Warren, 1980) (2) (Hatcher and Rus, 1976, Goguen *et al.*, 1977, Futatsugi *et al.*, 1985, Heering *et al.*, 1989) (3) (Poigné, 1986, Meinke, 1992b, 1992a, Hearn and Meinke, 1994) (4) (Hearn, 1995, Visser, 1996).

The rest of this paper is structured as follows. Section 2 contains a review of first-order signatures, context-free grammars and their correspondence and gives some examples of data type specification with context-free grammars. Section 3 defines two-level grammars and the parsing problem for two-level grammars. Section 4 illustrates how two-level grammars can be used for polymorphic syntax definition. Section 5 discusses several properties of two-level grammars including a characterization of a large class of grammars for which the parsing problem is decidable, although membership of the class is undecidable. Section 6 defines a parsing algorithm, with a correctness proof, for this class of two-level grammars that yields for each string a minimal and finite representation of the set of all parse trees for the string. Section 7 discusses related grammar formalisms and type systems and Section 8 concludes the paper.

2 Signatures and Grammars

In this section we review many-sorted algebras, context-free grammars, the correspondence between first-order signatures and context-free grammars and the use of context-free grammars in the algebraic specification of languages and data types.

2.1 Many-Sorted Algebra

Many-sorted algebras or Σ-algebras were introduced by Higgins (1963) as a generalization of the theory of abstract algebra. Here we give the basic constructs needed in this paper. For a further introduction to the theory of universal algebra see for instance Meinke and Tucker (1992), who also give several example applications. A note on notation: We will frequently use the notion of a family, which is a collection of sets indexed by some, finite or infinite, index set. If F is a family indexed by I, we denote by $F(i)$ the set at index $i \in I$ and write $F = (F(i) \mid i \in I)$. If we want to indicate that x is an element of some $F(i)$, we loosely write $x \in F$ identifying F with $\bigcup_{i \in I} F(i)$.

Definition 2.1 (Signature) A *many-sorted signature* Σ is a pair $\langle S, F\rangle$ where $S = \mathrm{S}(\Sigma) \subseteq \mathbf{S}$ is a set of sort names and $F = \mathrm{F}(\Sigma) \subseteq \mathbf{O} \times \mathrm{S}(\Sigma)^+$ a set of function declarations (with $\mathbf{S}$ and $\mathbf{O}$ some sets of sort names and operation names, respectively). We write $f : \tau_1 \times \cdots \times \tau_n \to \tau_0$ if $\langle f, \tau_1 \ldots \tau_n \tau_0\rangle \in \mathrm{F}(\Sigma)$. $\Sigma \cup \mathrm{V}$ is the extension of a signature Σ with a $\mathrm{S}(\Sigma)$-indexed family of sets of variables V. We write $x : \tau$ if $x \in \mathrm{V}(\tau)$. The class of all signatures is denoted by SIG.

Definition 2.2 (Σ-Algebra) A Σ-*algebra* $\mathcal{A}$ is an $\mathrm{S}(\Sigma)$-indexed family of carrier sets $A(\tau)$ and an assignment of each $f : \tau_1 \cdots \times \tau_n \to \tau_0$ in $\mathrm{F}(\Sigma)$ to an $\mathcal{A}$ function $f_{\mathcal{A}} : A(\tau_1) \times \cdots \times A(\tau_n) \to A(\tau_0)$ such that $f_{\mathcal{A}}(a_1, \ldots, a_n) \in A(\tau_0)$ if $a_i \in A(\tau_i)$ $(1 \leq i \leq n)$. $\mathrm{Alg}(\Sigma)$ denotes the collection of all Σ-algebras.

An equational many-sorted algebraic specification consists of a signature and a set of equations that define the relations between objects of the algebras described by the specification. Note that the theory of universal algebra does not limit algebras to have finitely many operations or sorts, but that an algebraic specification must be a finite structure. The following example illustrates the definitions above. We use the keywords `sorts`, `functions` and `variables` to indicate the declaration of $\mathrm{S}(\Sigma)$, $\mathrm{F}(\Sigma)$ and V, respectively. Furthermore, we make use of modular specifications consisting of modules that can import other modules, where a module with imports denotes the pointwise union of the imported and importing specification.

Example 2.3 The following is an example of a first-order algebraic specification of the algebra of natural numbers.

```
module naturals
  sorts nat;
  functions
    zero : nat;
    succ : nat -> nat;
    add  : nat # nat -> nat;
  variables
    I, J : nat;
  equations
    add(zero, I)    = I;
    add(succ(I), J) = succ(add(I, J))
```

Definition 2.4 (Terms) The $\mathrm{S}(\Sigma)$-indexed family $\mathrm{Tree}(\Sigma)$ of *well-formed terms (or trees) over signature* Σ is defined by the inference rules below such that $t \in \mathrm{Tree}(\Sigma)(\tau)$ iff $\Sigma \vdash t : \tau$.

$$\frac{x \in \mathrm{V}(\tau)}{\Sigma \cup \mathrm{V} \vdash x : \tau} \qquad \text{(Var1)}$$

$$\frac{f : \tau_1 \times \ldots \times \tau_n \to \tau \in \mathrm{F}(\Sigma),\ \Sigma \vdash t_i : \tau_i\ (1 \leq i \leq n)}{\Sigma \vdash f(t_1, \ldots, t_n) : \tau} \qquad \text{(App1)}$$

Definition 2.5 (Homomorphism) A Σ-*homomorphism* $h : \mathcal{A} \to \mathcal{B}$ is an $\mathrm{S}(\Sigma)$-indexed family of functions h_τ such that for any $f : \tau_1 \times \ldots \times \tau_n \to \tau \in \mathrm{F}(\Sigma)$, $h_\tau(f_\mathcal{A}(a_1, \ldots, a_n)) = f_B(h_{\tau_1}(a_{\tau_1}), \ldots, h_{\tau_n}(a_n))$. A Σ-algebra $\mathcal{A}$ is *initial* in $\mathrm{Alg}(\Sigma)$ if for any $\mathcal{B} \in \mathrm{Alg}(\Sigma)$ there is a unique homomorphism from $\mathcal{A}$ to $\mathcal{B}$.

Because there is a unique homomorphism $h_\mathcal{A} : \mathrm{Tree}(\Sigma) \to \mathcal{A}$ for any $\mathcal{A} \in \mathrm{Alg}(\Sigma)$, i.e., $h_\mathcal{A}(f(t_1, \ldots, t_n)) = f_\mathcal{A}(h_\mathcal{A}(t_1), \ldots, h_\mathcal{A}(t_n))$, we have

Proposition 2.6 $\mathrm{Tree}(\Sigma)$ *is an inital algebra in* $\mathrm{Alg}(\Sigma)$.

Definition 2.7 (Substitution) A *substitution* $\sigma : \mathrm{V} \to \mathrm{Tree}(\Sigma \cup \mathrm{V})$ is a $\mathrm{S}(\Sigma)$-indexed function mapping variables to terms. The function $\bar{\sigma} : \mathrm{Tree}(\Sigma \cup \mathrm{V}) \to \mathrm{Tree}(\Sigma \cup \mathrm{V})$ is the homomorphic extension of a substitution σ that replaces all variables in a term by their σ images. A term t is an *instance* of term t' or t' is *more general* than t, written as $t' \geq t$, if there is some substitution σ such that $t = \bar{\sigma}(t')$. A term t is *strictly more general* than t', $t > t'$, if $t \geq t'$ and not $t' \geq t$. In case $t' \geq t$ we also say that t *matches* t' and that σ is the match. A substitution σ is a *unifier* for two terms t and t' if $\bar{\sigma}(t) = \bar{\sigma}(t')$. A unifier σ is a *most general unifier* for t and t' if for each unifier σ' we have that $\bar{\sigma}(t) \geq \bar{\sigma}'(t)$. A substitution σ is a *renaming* of t if $\bar{\sigma}(t) \geq t$, i.e., if $range(\sigma | vars(t)) \subseteq \mathrm{V}$, with $vars(t)$ the set of variables in t. Two terms t and t' are equal up to renaming of variables ($t \doteq t'$) if there is a renaming σ such that $\sigma(t) = t'$.

The $\mathrm{S}(\Sigma)$-indexed family of equations of an algebraic specification with signature Σ is a subfamily of the family $\mathrm{Eq}(\Sigma \cup \mathrm{V})$ such that $\mathrm{Eq}(\Sigma \cup \mathrm{V})(\tau) \subseteq \mathrm{Tree}(\Sigma \cup \mathrm{V})(\tau)^2$. A Σ-algebra $\mathcal{A}$ satisfies an equation $t_1 = t_2$, if for any substitution σ, $h \circ \bar{\sigma}(t_1) = h \circ \bar{\sigma}(t_2)$ in $\mathcal{A}$, where h is the unique homomorphism $h : \mathrm{Tree}(\Sigma) \rightarrow \mathcal{A}$.

2.2 Context-Free Grammars

Context-free grammars can be used to define languages, i.e., sets of strings and analyses of strings in the form of parse trees. The structure of parse trees corresponds to the structure of terms over a signature as we shall see in the next subsection. However, grammars provide more flexible notation for terms.

Definition 2.8 (Grammar) A *context-free grammar* $\mathcal{G}$ is a triple $\langle S, L, P\rangle$ with $S = \mathrm{Ss}(\mathcal{G})$ a set of *sort symbols* or *nonterminals*, $L = \mathrm{Sl}(\mathcal{G})$ a set of *literals* or *terminals*, with $\mathrm{Ss}(\mathcal{G}) \cap \mathrm{Sl}(\mathcal{G}) = \emptyset$ and $\mathrm{S}(\mathcal{G}) = \mathrm{Ss}(\mathcal{G}) \cup \mathrm{Sl}(\mathcal{G})$ the set of *symbols* of $\mathcal{G}$, and $P = \mathrm{P}(\mathcal{G}) \subseteq \mathrm{S}(\mathcal{G})^* \times \mathrm{Ss}(\mathcal{G})$ a set of *productions*. We write $\alpha \rightarrow \tau$ for a production $\langle \alpha, \tau\rangle \in \mathrm{P}(\mathcal{G})$. $\mathcal{G} \cup \mathrm{V}$ is the extension of a grammar with variables. We write $x \rightarrow \tau$ if $x \in \mathrm{V}(\tau)$. The class of all context-free grammars is denoted by CFG.

Observe that productions are reversed in order to make them look like function declarations in a signature—conventionally a production $\alpha \rightarrow \tau$ is written as $\tau \rightarrow \alpha$ or $\tau ::= \alpha$. Also note that in the conventional definition of context-free grammars a single symbol has the role of start symbol from which all sentences of the grammar are generated. In the definition above all sort symbols are start symbols. Rus and Jones (1995) make a distinction between context-free grammars that have a single start symbol or *axiom*, *algebraic grammars* that have all nonterminals as start symbol, and *multi-axiom grammars* with a subset of the nonterminals as start symbol. In that terminology our grammars might more appropriately be called algebraic grammars. However, in our definition of language generated by a grammar (below), we distinguish the sets generated by each nonterminal, whereas in the definition of Rus and Jones (1995) the language of a grammar is the union of all strings generated by all axioms, weakening the expressive power of the formalism. With Goguen *et al.* (1977) we stick with the familiar 'context-free grammar'.

As concrete syntax for grammars in examples we adopt the style of the syntax definition formalism SDF (Heering *et al.*, 1989). The keywords `sorts`, `syntax` and `variables` indicate the sets of sort symbols, context-free productions and variables declarations, respectively. Strings of characters between double quotes represent the literals of the grammar and identifiers are used as sort symbols. The sort symbols are explicitly declared in the sorts section, whereas literals are implicitly declared by their usage in productions. Grammars can be divided in modules and modules can import other modules. A module with imports denotes the pointwise union of the imported and importing grammar.

Example 2.9 The following specification uses a context-free grammar as signature in the specification of succesor naturals. This specification is similar to the specification in Example 2.3, but in the equations we can use the more natural infix notation familiar from mathematics.

```
module naturals-cfg
  sorts nat;
  syntax
    "0"          -> nat;
    "s" nat      -> nat;
    nat "+" nat -> nat {left};
    "(" nat ")" -> nat {bracket};
  variables
    "I" -> nat; "J" -> nat;
  equations
    0    + I = I;
    s(I) + J = s(I + J)
```

The attributes attached to the productions are meant for disambiguation. The attribute `left` indicate the left associativity of the addition function and the attribute `bracket` indicates that parentheses around a natural number behave as the identity function. Disambiguation will be further discussed below.

Definition 2.10 (Parse Trees) The $\mathrm{S}(\mathcal{G})$-indexed family $\mathrm{Tree}(\mathcal{G})$ of *parse trees over grammar* $\mathcal{G}$ is defined by the inference rules below such that $t \in \mathrm{Tree}(\mathcal{G})(\tau)$ iff $\mathcal{G} \vdash t : \tau$.

$$\frac{L \in \mathrm{Sl}(\mathcal{G})}{\mathcal{G} \vdash L : L} \tag{Lit2}$$

$$\frac{x \in \mathrm{V}(\tau)}{\mathcal{G} \cup \mathrm{V} \vdash var(x, \tau) : \tau} \tag{Var2}$$

$$\frac{\tau_1 \ldots \tau_n \to \tau \in \mathrm{P}(\mathcal{G}),\ \mathcal{G} \vdash t_i : \tau_i\ (1 \le i \le n)}{\mathcal{G} \vdash app(\tau_1 \ldots \tau_n \to \tau, [t_1, \ldots, t_n]) : \tau} \tag{App2}$$

Example 2.11 As an example of this inference relation consider the following parse tree for the sentence `0 + I` over the grammar of Example 2.9.

$$app(\texttt{nat "+" nat} \to \texttt{nat}, [app(\texttt{"0"} \to \texttt{nat}, [\texttt{"0"}]), \texttt{"+"}, [var(\texttt{"I"}, \texttt{nat})]])$$

Rule (App2) defines the construction of *application* tree nodes for productions of a grammar. Observe that the complete production is used as label in such application nodes.

Because the structure of parse trees is different from terms over a signature, we redefine the notion of substitution.

Definition 2.12 (Substitution) A substitution $\sigma : (\mathrm{V} \times \mathrm{S}(\mathcal{G})) \to \mathrm{Tree}(\mathcal{G} \cup \mathrm{V})$ is a $\mathrm{S}(\mathcal{G})$-indexed family of functions mapping variables to trees. The extension $\bar{\sigma}$ of σ to trees is defined as

$$\begin{aligned}
\bar{\sigma}(L) &= L \\
\bar{\sigma}(\tau)(var(x,\tau)) &= \sigma(var(x,\tau)) \\
\bar{\sigma}(\tau)(app(\tau_1 \ldots \tau_n \to \tau, [t_1, \ldots, t_n])) &= \\
& app(\tau_1 \ldots \tau_n \to \tau, [\bar{\sigma}(\tau_1)(t_1), \ldots, \bar{\sigma}(\tau_n)(t_n)])
\end{aligned}$$

All other notions defined in Definition 2.7 are defined in the same way for parse trees.

Definition 2.13 (Language) The *language* $\mathrm{L}(\mathcal{G})$ generated by a context-free grammar $\mathcal{G}$ is the $\mathrm{S}(\mathcal{G})$-indexed family of strings such that $\mathrm{L}(\mathcal{G})(\tau) = yield(\mathrm{Tree}(\mathcal{G})(\tau))$, where the function $yield : \mathrm{Tree}(\mathcal{G} \cup \mathrm{V}) \to (\mathrm{S}(\mathcal{G}) \cup \mathrm{V})^*$ is defined by

$$\begin{aligned}
yield(L) &= L \\
yield(var(x,\tau)) &= x \\
yield(app(\tau_1 \ldots \tau_n \to \tau, [t_1, \ldots, t_n])) &= yield(t_1) \ldots yield(t_n)
\end{aligned}$$

and applied to a set of trees denotes the pointwise extension to sets.

Definition 2.14 (Parsing) A parser is a function $\Pi(\mathcal{G}) : \mathrm{S}(\mathcal{G})^* \to \mathcal{P}(\mathrm{Tree}(\mathcal{G}))$ that maps a string of symbols to a sub-family of $\mathrm{Tree}(\mathcal{G})$ such that

$$\Pi(\mathcal{G})(w)(\tau) = \{t \in \mathrm{Tree}(\mathcal{G})(\tau) \mid yield(t) = w\}$$

A recognizer is a predicate $\in \mathrm{L}(\mathcal{G})$ that decides whether a string is in the language generated by $\mathcal{G}$ or more specifically a predicate $\in \mathrm{L}(\mathcal{G})(\tau)$ that decides whether a string is in the language generated by sort symbol τ.

2.3 Correspondence of Signatures and Grammars

There is a correspondence between the trees generated by first-order signatures and context-free grammars such that grammars can be used to describe the structure of algebras (Hatcher and Rus, 1976, Goguen *et al.*, 1977, Heering *et al.*, 1989).

Proposition 2.15 *There are mappings* $grm : \mathrm{SIG} \to \mathrm{CFG}$ *and* $sig : \mathrm{CFG} \to \mathrm{SIG}$ *such that* $\mathrm{Tree}(grm(\Sigma)) \cong \mathrm{Tree}(\Sigma)$ *and* $\mathrm{Tree}(sig(\mathcal{G})) \cong \mathrm{Tree}(\mathcal{G})$.

Proof. Define grm such that for a signature Σ a grammar is constructed that expresses the syntax of terms over a signature Σ by taking as nonterminals the sorts of Σ and as literals the operator symbols of Σ, parentheses and commas.

$$\begin{aligned}
\mathrm{Ss}(grm(\Sigma)) &= \mathrm{S}(\Sigma) \\
\mathrm{Sl}(grm(\Sigma)) &= \{\texttt{"f"} \mid f : \tau_1 \times \cdots \times \tau_n \to \tau_0 \in \mathrm{F}(\Sigma)\} \cup \{\texttt{"("}, \texttt{")"}, \texttt{","}\} \\
\mathrm{P}(grm(\Sigma)) &= \{\texttt{"f""("}\tau_1\texttt{","} \ldots \texttt{","}\tau_n\texttt{")"} \to \tau_0 \\
&\qquad\qquad \mid f : \tau_1 \times \cdots \times \tau_n \to \tau_0 \in \mathrm{F}(\Sigma)\}
\end{aligned}$$

Now we can translate terms over Σ to parse trees over $grm(\Sigma)$ by means of the function $i_{grm} : \mathrm{Tree}(\Sigma) \to \mathrm{Tree}(grm(\Sigma))$ as follows:

$$\begin{aligned}
i_{grm}(\tau)(f(t_1, \ldots, t_n)) = app(&\texttt{"f""("}\tau_1\texttt{","} \ldots \texttt{","}\tau_n\texttt{")"} \to \tau_0, \\
&[\texttt{"f""("}i_{grm}(\tau_1)(t_1)\texttt{","} \ldots \texttt{","}i_{grm}(\tau_n)(t_n)\texttt{")"}])
\end{aligned}$$

for each $f : \tau_1 \times \cdots \times \tau_n \to \tau \in \mathrm{P}(\Sigma)$. Define sig such that a grammar is translated to a signature in which the productions of the grammar have the role of function names.

$$\begin{aligned}
\mathrm{S}(sig(\mathcal{G})) &= \mathrm{S}(\mathcal{G}) \\
\mathrm{F}(sig(\mathcal{G})) &= \{\texttt{"}\alpha \to \tau\texttt{"} : \tau_1 \times \cdots \times \tau_n \to \tau \mid \alpha \to \tau \in \mathrm{P}(\mathcal{G}), \alpha = \tau_1 \ldots \tau_n\} \\
&\quad \cup\ \{\texttt{"L"} :\to L \mid L \in \mathrm{Sl}(\mathcal{G})\}
\end{aligned}$$

Now we can translate parse trees to terms by means of the function i_{sig} : $\mathrm{Tree}(\mathcal{G}) \to \mathrm{Tree}(sig(\mathcal{G}))$ as follows:

$$\begin{aligned}
i_{sig}(L)(L) &= L() \\
i_{sig}(\tau)(var(x, \tau)) &= x \\
i_{sig}(\tau)(app(\tau_1 \ldots \tau_n \to \tau, [t_1, \ldots, t_n])) &= \\
&\texttt{"}\tau_1 \ldots \tau_n \to \tau\texttt{"}(i_{sig}(\tau_1)(t_1), \ldots, i_{sig}(\tau_n)(t_n))
\end{aligned}$$

It is clear that i_{grm} and i_{sig} are isomorphisms. [2] □

The following proposition tells us that we can use context-free grammars as many-sorted algebraic signatures, where productions play the role both of function symbol and type declaration. We can thus speak of the class of algebras $\mathrm{Alg}(\mathcal{G})$ generated by a context-free grammar $\mathcal{G}$, where the grammar symbols are interpreted as carrier sets and productions as algebraic operations. It is clear that the family $\mathrm{Tree}(\mathcal{G})$ of parse trees over $\mathcal{G}$ is an initial algebra in $\mathrm{Alg}(\mathcal{G})$. The language $\mathrm{L}(\mathcal{G})$ is also an element of $\mathrm{Alg}(\mathcal{G})$, with *yield* as the unique homomorphism from $\mathrm{Tree}(\mathcal{G}) \to \mathrm{L}(\mathcal{G})$. However, $\mathrm{L}(\mathcal{G})$ is not necessarily initial in $\mathrm{Alg}(\mathcal{G})$. A context-free grammar is ambiguous if there is some string $w \in \mathrm{L}(\mathcal{G})$ for which more than one parse tree exists.

Proposition 2.16 $\mathrm{L}(\mathcal{G})$ *is initial in* $\mathrm{Alg}(\mathcal{G})$ *iff* $\mathcal{G}$ *is unambiguous.*

[2] Note that grm and sig are not isomorphisms from SIG to CFG and vice versa: $\Sigma \neq sig(grm(\Sigma))$ and $\mathcal{G} \neq grm(sig(\mathcal{G}))$.

For if $\mathcal{G}$ is ambiguous, *yield* is not injective, hence not an isomorphism. This entails that algebraic properties do not apply to the strings used to denote trees. For example, in a grammar of arithmetic expressions with the production e "$-$" $e \to e$, the composition of the strings x, $-$ and $y - z$ does not correspond with the composition of their trees, i.e., $x - (y - z)$, but with $(x - y) - z$, which usually has a different semantic interpretation. We could require the use of unambiguous grammars. However, it is undecidable whether a context-free grammar is ambiguous. There are decidable subclasses of CFG, e.g., the LR(k) grammars, that are unambiguous, but these classes are much more restrictive than the class of unambiguous grammars and, moreover, not closed under union of grammars, which is a handicap when developing modular specifications. Furthermore, to disambiguate a grammar it is often necessary to introduce new sort symbols and to restrict the possibility to compose expressions.

Klint and Visser (1994) propose a method for disambiguation of context-free grammars by means of *disambiguation filters*. A filter $\mathcal{F}(\mathcal{G})$ selects a subset from the parse trees for a string, i.e., $\mathcal{F}(\mathcal{G})(\Pi(\mathcal{G})(w)) \subseteq \Pi(\mathcal{G})(w)$. A filter is completely disambiguating if for each string w, $|\mathcal{F}(\mathcal{G})(\Pi(\mathcal{G})(w))| \leq 1$.

Proposition 2.17 *If a filter* $\mathcal{F}(\mathcal{G})$ *is completely disambiguating, then* $\mathcal{F}(\mathcal{G}) \circ \Pi(\mathcal{G})$ *is an injection* $\mathrm{L}(\mathcal{G}) \to \mathrm{Tree}(\mathcal{G})$*.*

However, the trees that are not selected by the filter become unreachable with this method, i.e., $\mathcal{F}(\mathcal{G}) \circ \Pi(\mathcal{G})$ is not surjective. A solution to this problem is to try to add bracket productions, which are interpreted as identity functions, to the grammar such that all trees become reachable.

Proposition 2.18 *If* $\mathrm{Tree}(\mathcal{G}) \cong \mathrm{Tree}(\mathcal{G} \cup \mathcal{G}_{br})/ =_{br}$ *and* $\mathcal{F}(\mathcal{G} \cup \mathcal{G}_{br})$ *is completely disambiguating, then* $\mathrm{L}(\mathcal{G}) \cong \mathrm{Tree}(\Sigma)$*.*

For a further discussion of this topic we refer to Klint and Visser (1994). In the sequel we will assume that we are dealing with such grammars that we can use strings to denote trees. In examples we use a simple method for disambiguation by priority and associativity declarations. For instance, in the grammar of naturals above we used the production attribute `left` to declare the addition operator as left associative. Furthermore, the `bracket` attribute declares the production `"(" nat ")" -> nat {bracket}` as the identity function on natural numbers and makes all trees in Tree(`nat`) reachable by means of strings.

2.4 Data Type Specification

By means of grammars as signatures we have a flexible framework for syntax definition in the algebraic specification of data types, for example, the typical stack constructors might be defined as `"[]" -> stack; "push" int "on" stack -> stack` In algebraic specification of programming languages, context-free grammars can be used for instance to specify the syntax of a programming language as in `var ":=" exp -> stat` and the syntax of operations on programs such as typecheckers `decl "|-" exp -> bool; decl "|-" stat -> bool` that characterize the well-typed expressions and statements, interpreters `"eval" "[[" stat "]]" "(" env ")" -> env` that interpret statements as functions from environments to environments and compilers `"trans" "[[" stat "]]" -> smc` that translate statements to stack machine code.

The disadvantage of first-order signatures and context-free grammars is the rigid monomorphic typing scheme. For instance, we can not express that for each sort τ, the sort τ^* of sequences of τs can be constructed and that for each function $f : \tau_1 \rightarrow \tau_2 \in \mathrm{F}(\Sigma)$ the function $f^* : \tau_1^* \rightarrow \tau_2^*$ extends f to sequences such that $f^*(a_1 \ldots a_n) = f(a_1) \ldots f(a_n)$. The consequence is that for each special case of a generic construct such as sequences and for each instance of a generic function such as $\_^*$, a separate definition has to be given.

One solution to overcome this rigidity is to loosen the typing requirements. In Visser (1993) terms of typed combinatory logic are encoded as simple untyped applicative terms. In van den Brand *et al.* (tion) a similar structure is defined for the representation of parse trees and other structure data by means of a generic format for term representation. We study a combination of the ideas from those papers. The following grammar of generic terms (aterms) is defined by van den Brand *et al.* (tion) to represent parse trees and abstract syntax trees over arbitrary grammars. A term is a function symbol (afun), an application of a function symbol to a list of arguments $F(T_2, \ldots, T_n)$, or a list of terms $[T_1, \ldots, T_n]$. A function symbol is a literal or an identifier. Identifier function symbols have to defined explicitly. The module `literals` that is imported in module `aterms` defines the syntax of literals, i.e., strings of characters between double quotes.

```
module aterms
  imports literals
  sorts aterms atermlist afun aterm
  syntax
    aterm                -> aterms;
    aterm "," aterms     -> aterms;
    "[" "]"              -> atermlist;
    "[" aterms "]"       -> atermlist;
    literal              -> afun;
    afun                 -> aterm;
    afun "(" aterms ")"  -> aterm;
    atermlist            -> aterm;
  variables
```

```
"T" -> aterm; "Ts" -> aterms; "Tl" -> aterms;
```

With this term structure it is possible to define higher-order functions. For instance, the following module defines the function `*` that applies a function F to each element of a list of terms and the function `:` that adds an element to the front of a list. Functions that are passed as arguments to higher-order functions are also represented as terms. The function `@` defines the application of such symbolically represented functions to their arguments.

```
module listops
imports aterms;
  syntax
    aterm ":" atermlist -> atermlist {right};
    aterm "*" atermlist -> aterm {right};
    "map"               -> afun;
    aterm "@" aterm     -> aterm;
    "(" aterm ")"       -> aterm {bracket};
  variables
    "Fun" -> aterm;
  equations
    T : []        = [T];
    T : [Ts]      = [T, Ts];
    Fun * []      = [];
    Fun * [T]     = [Fun @ T];
    Fun * [T, Ts] = (Fun @ T) : (F * [Ts]);
    map(Fun) @ T  = Fun * T;
```

Such a definition works well as long as sensible terms are considered. However, `([] * map)`, the empty list mapped over the function `map`, is also a syntactically correct term, but does not have a clear interpretation. We would rather forbid this term on the basis of some typing rule without losing the genericity of the term structure.

An application of the generic term structure of aterms, is the representation of parse trees. We add the following function symbols

```
module atrees
  imports aterms
  syntax
    "var" -> afun; "app" -> afun; "prod" -> afun; "lit" -> afun;
```

The following proposition shows how this language can be used to represent parse trees over arbitrary grammars. Observe that we use the concrete syntax of aterms to represent elements of Tree(aterms).

Proposition 2.19 *For any CFG $\mathcal{G}$, there is an injection* $\ulcorner\_\urcorner : \mathrm{Tree}(\mathcal{G}) \to \mathrm{Tree}(\texttt{aterms})$ *such that* $\mathrm{Tree}(\mathcal{G})$ *is isomorphic with its* $\ulcorner\_\urcorner$ *image in* aterms, *i.e.,* $\mathrm{Tree}(\mathcal{G}) \cong \ulcorner\mathrm{Tree}(\mathcal{G})\urcorner$.

Proof. Given some CFG $\mathcal{G}$ first define $\ulcorner\_\urcorner : \mathrm{S}(\mathcal{G}) \to \mathrm{Tree}(\mathrm{aterms})$ as

$$\begin{aligned} \ulcorner L\urcorner &= \texttt{lit}(\texttt{"}L\texttt{"}) \\ \ulcorner \tau\urcorner &= \texttt{"}\tau\texttt{"} \text{ if } \tau \in \mathrm{Ss}(\mathcal{G}) \end{aligned}$$

then define $\ulcorner\_\urcorner : \mathrm{Tree}(\mathcal{G}) \to \mathrm{Tree}(\mathrm{aterms})$ as

$$\begin{aligned} \ulcorner L\urcorner &= \texttt{"}L\texttt{"} \\ \ulcorner var(x,\tau)\urcorner &= \texttt{var}(\texttt{"}x\texttt{"},\ulcorner\tau\urcorner) \\ \ulcorner app(\tau_1 \ldots \tau_n \to \tau_0, [t_1, \ldots, t_n])\urcorner &= \\ &\texttt{app}(\texttt{prod}([\ulcorner\tau_1\urcorner, \ldots, \ulcorner\tau_n\urcorner], \ulcorner\tau_0\urcorner),\ [\ulcorner t_1\urcorner, \ldots, \ulcorner t_n\urcorner]) \end{aligned}$$

Now we have $\mathrm{Tree}(\mathcal{G}) \cong \ulcorner\mathrm{Tree}(\mathcal{G})\urcorner$. □

As a result, any sentence in a context-free language can be represented as a string in the fixed language of aterms. For example, the parse tree for the string `s 0` according to the grammar for natural numbers is translated as follows:

$$\begin{aligned} &\ulcorner app(\texttt{"s"}\ \mathrm{nat} \to \mathrm{nat}, [\texttt{"s"}\ app(\texttt{"0"} \to \mathrm{nat}, [\texttt{"0"}])])\urcorner = \\ &\qquad \texttt{app(prod([lit("s"), "nat"], "nat"),} \\ &\qquad\qquad \texttt{["s", app(prod([lit("0")], "nat"), ["0"])])} \end{aligned}$$

The resulting string does not only have a fixed syntax, it is also self descriptive. The grammar $\mathcal{G}$ can be derived from the aterm that encodes a parse tree. With this encoding we can define very generic, language independent operations on parse trees like substitution, unification and searching of subtrees. Again, the disadvantage of this scheme is that there are (many) aterms that are not encodings of parse trees, e.g., `"abc"("def")` is a syntactically correct aterm but is not an element of $\ulcorner\mathrm{Tree}(\mathcal{G})\urcorner$ for any $\mathcal{G}$. Therefore, specifications and programs that manipulate aterms encoding parse trees have to type check the terms they receive and have to preserve well-formedness of the terms they process and construct.

3 Two-Level Grammars

Context-free grammars provide either a strongly typed but rigid syntactic structure or a generic but untyped structure. Two-level grammars provide a method for polymorphic syntax definition that supports definition of generic structures with type constraints. Two-level grammars have been defined in several variants after the original formulation for the definition of the syntax of Algol68 in van Wijngaarden *et al.* (1976). Here we introduce a definition of two-level grammars that is straightforwardly formulated as two levels of context-free grammars, where level 1 defines the syntax of the nonterminals of level 0. The productions at level 0 of a two-level grammar are *production schemata* that uniformly describe sets of context-free productions in the same way that polymorphic functions in a framework like ML (Milner, 1978) describe collections of functions. Given the extension of context-free grammars to two-level grammars, it is straightforward to generalize two-level grammars to multi-level grammars, in the same way as multi-level specifications are defined in Visser (1996). In this paper we will restrict our attention to two-level grammars.

Definition 3.1 (Two-Level Grammar) A *two-level grammar* Γ is a pair $\langle \mathcal{G}_1, \mathcal{G}_0 \rangle$ of context-free grammars such that the sort symbols of $\mathcal{G}_0$ are terms, possibly with variables, over $\mathcal{G}_1$, i.e., $\mathrm{Ss}(\mathcal{G}_0) \subseteq \mathrm{Tree}(\mathcal{G}_1 \cup \mathrm{V}_1)$.

The following definition gives the meaning of finite two-level grammars in terms of, possibly infinite, context-free grammars.

Definition 3.2 A two-level grammar Γ corresponds to a, possibly infinite, context-free grammar $[\![\Gamma]\!]$ that is derived from Γ by taking all substitutions of symbols $\mathrm{S}([\![\Gamma]\!]) = \{\bar{\sigma}(\tau) \mid \tau \in \mathrm{S}(\mathcal{G}_0),\ \sigma : \mathrm{V}_1 \to \mathrm{Tree}(\mathcal{G}_1 \cup \mathrm{V}_1)\}$ and productions $\mathrm{P}([\![\Gamma]\!]) = \{\bar{\sigma}(\tau_1) \to \bar{\sigma}(\tau_2) \mid \tau_1 \to \tau_2 \in \mathrm{P}(\mathcal{G}_0),\ \sigma : \mathrm{V}_1 \to \mathrm{Tree}(\mathcal{G}_1 \cup \mathrm{V}_1)\}\rangle$.

Through the translation of a two-level grammar Γ to a CFG $[\![\Gamma]\!]$ we immediately have the definitions of the term algebra $\mathrm{Tree}([\![\Gamma]\!])$ and the language $\mathrm{L}([\![\Gamma]\!])$. Another characterization of the trees generated by a two-level grammar is given by means of inference rules in the following definition.

Definition 3.3 The $\mathrm{S}(\mathcal{G}_0)$-indexed family $\mathrm{Tree}(\Gamma)$ of *parse trees over two-level grammar* Γ is defined by the inference rules below such that $t \in \mathrm{Tree}(\Gamma)(\tau)$ iff $\Gamma \vdash t : \tau$.

$$\frac{L \in \mathrm{Sl}(\mathcal{G}_0)}{\Gamma \vdash L : L} \tag{Lit3}$$

$$\frac{x \in \mathrm{V}(\tau'), \tau' \geq \tau}{\Gamma \cup \mathrm{V} \vdash var(x, \tau) : \tau} \tag{Var3}$$

$$\frac{p \in \mathrm{P}(\mathcal{G}_0),\ p \geq \tau_1 \ldots \tau_n \to \tau,\ \Gamma \vdash t_i : \tau_i\ (1 \leq i \leq n)}{\Gamma \vdash app(\tau_1 \ldots \tau_n \to \tau, [t_1, \ldots, t_n]) : \tau} \tag{App3}$$

Recall from Definition 2.7 that the relation $p \geq p'$ holds if production p' is an instance of p, i.e., p is more general than p'.

We observe that the two ways of defining the terms generated by a two-level grammar are equivalent.

Proposition 3.4 $\Gamma \vdash t : \tau$ *iff* $[\![\Gamma]\!] \vdash t : \tau$

Proof. ($\Rightarrow$) by induction on t: (i) if $t = L$ then $[\![\Gamma]\!] \vdash L : L$ by (Lit2) (ii) if $t = var(x, \tau)$ then $[\![\Gamma]\!] \vdash var(x, \tau) : \tau$ by (Var2) (iii) if $t = app(p', [t_1, \ldots, t_n])$, by induction hypothesis $[\![\Gamma]\!] \vdash t_i : \tau_i$, and by (App3) there is some $p \in \mathrm{P}(\mathcal{G}_0)$ such that $p \geq p'$, i.e., there is some σ such that $\bar{\sigma}(p) = p'$, but then $p' \in p([\![\Gamma]\!])$, by (App2) $[\![\Gamma]\!] \vdash app(p', [t_1, \ldots, t_n])$. ($\Leftarrow$) similarly. $\square$

Corollary 3.5 $\mathrm{Tree}([\![\Gamma]\!]) = \mathrm{Tree}(\Gamma)$ *and* $\mathrm{L}(\Gamma) = \mathrm{L}([\![\Gamma]\!])$

Definition 3.6 (Substitution) A two-level substitution φ is a pair $\langle \sigma_1, \sigma_0 \rangle$ of a type substitution $\sigma_1 : (\mathrm{V}_1 \times \mathrm{S}(\mathcal{G}_1)) \rightarrow \mathrm{Tree}(\mathcal{G}_1 \cup \mathrm{V}_1)$ and an object substitution $\sigma_0 : (\mathrm{V}_0 \times \mathrm{S}(\mathcal{G}_0)) \rightarrow \mathrm{Tree}(\mathcal{G}_0 \cup \mathrm{V}_0)$. The extension $\bar{\varphi}$ of φ to level 0 trees is defined as

$$\bar{\varphi}(L) = L$$

$$\bar{\varphi}(\tau)(var(x, \tau)) = \sigma_0(var(x, \bar{\sigma}_1(\tau)))$$

$$\begin{aligned}\bar{\varphi}(\tau)(app(\tau_1 \ldots \tau_n \rightarrow \tau, [t_1, \ldots, t_n])) = \\ app(\bar{\sigma}_1(\tau_1) \ldots \bar{\sigma}_1(\tau_n) \rightarrow \bar{\sigma}_1(\tau), [\bar{\varphi}(\tau_1)(t_1), \ldots, \bar{\varphi}(\tau_n)(t_n)])\end{aligned}$$

All other notions defined in Definition 2.7 are defined in the same way for parse trees. A two-level substitution with $\sigma_0 = id$ equal to the identity function is also denoted by $\bar{\bar{\sigma}}$, i.e., a function that substitutes type variables throughout a term.

Definition 3.7 (Parsing) Given a two-level grammar Γ and a string w the parsing problem is to find the set of parse trees $\Pi(\Gamma)(w)$ such that

$$\Pi(\Gamma)(w)(\tau) = \{t \mid \Gamma \vdash t : \tau \wedge yield(t) = w\}$$

Discussion 3.8 According to the definition above, *trees* over level 1 are used as sort symbols in level 1. However, if we write such grammars, we want to use strings instead of trees, i.e., $S(\mathcal{G}_0) \subseteq \mathrm{L}(\mathcal{G}_1) \cup \mathbf{S}$ instead of $\mathrm{S}(\mathcal{G}_0) \subseteq \mathrm{Tree}(\mathcal{G}_1 \cup \mathrm{V}_1) \cup \mathbf{S}$. This entails that the syntax of two level grammars is not fixed, the syntax of the symbols of level 0 is determined by level 1. To parse a two-level grammar we first have to parse level 1 with a parser for a context-free grammar formalism in order to construct a parser for level 0. Note that we use the same, SDF style, notation for productions and modules at both levels.

4 Examples

In this section we discuss several examples of two-level grammars. The syntax of grammars is the adaptation of the syntax of the multi-level specifications of Visser (1996) to grammars, i.e., function declarations become productions. It is not our intention to explain every detail of the notation used, but we do want to illustrate the general utility of two-level grammars for specification of data types.

4.1 Naturals

Module `nat` defines the syntax of natural number expressions. Level 1 introduces the sort `type` and the `type` constant `nat`. The expression `nat` can then be used as sort at level 0. Consider for example the production `"s" nat -> nat` of level 0. The expression `nat` in this production, is the constant `"nat" -> type` defined at level 1.

```
module nat
level 1
  sorts type;
  syntax
    "nat" -> type;
level 0
  syntax
    "0"          -> nat;
    "s" nat      -> nat;
    nat "+" nat -> nat {left};
    "(" nat ")" -> nat {bracket};
  variables
    "I" -> nat; "J" -> nat;
  equations
    0    + I = I;
    s(I) + J = s(I + J);
```

4.2 Booleans and Polymorphic Conditional

The grammar in module `nat` defines monomorphic syntax for natural numbers. Each production has one instance, i.e., the production itself. The following module defines the data type of Booleans. At level 1 the `type` constant `bool` is introduced, which is used as sort at level 0. In addition to the ordinary Boolean connectives, the module defines a polymorphic conditional for any `type`. The `type` variable `A` in the if-then-else-fi production can be instantiated with any `type` expression. The production actually denotes the set of all instantiations of this production. Furthermore, the module defines a polymorphic bracket function.

```
module bool
imports nat;
level 1
  syntax
```

```
    "bool" -> type;
  variables
    "A" -> type; "B" -> type; "C" -> type;
level 0
  syntax
    "true"  -> bool; bool "/\" bool -> bool; "not" bool -> bool;
    "false" -> bool; bool "\/" bool -> bool;
    "if" bool "then" A "else" A "fi" -> A;
    "(" A ")" -> A {bracket};
  variables
    "B" -> bool; "X" -> A; "X'" -> A;
  equations
    true  /\ B = B;     true  \/ B = true; not true  = false;
    false /\ B = false; false \/ B = B;    not false = true;
    if true  then X else X' fi = X;
    if false then X else X' fi = X';
```

4.3 Polymorphic Lists

Most grammar formalisms provide a built-in notion of lists. The next example shows how such notation can be introduced with two-level grammars. Module `list` introduces type operators at level 1 denoting the type of polymorphic lists. The operators {_ _}+ and {_ _}* denote the type of non-empty and possibly-empty lists with separators, respectively. The operators `"_+"` and `"_*"` denote the type of non-empty and possibly-empty lists without separators, respectively. The latter two operators are defined in terms of the former two by means of the equations that define lists without separators as lists with empty separators, where `empty` is a `sep`.

At level 0 polymorphic constructor functions for these types are defined. A non-empty list of `A`s separated by `Sep`s is either an `A` or two lists concatenated by a `Sep`. The first equation expresses that `Sep` concatenation associates to the right. An {`A Sep`}* list is either empty or a non-empty list of `A`s. {`A Sep`}*-lists can be concatenated by means of the operator `[_ Sep _]`. Note that `"^"` is used as a variable to denote separators.

```
module list;
imports bool;
level 1
  sorts regtype, sep;
  syntax
    "{" type sep "}" "*" -> regtype;  type "*" -> regtype;
    "{" type sep "}" "+" -> regtype;  type "+" -> regtype;
    "[" regtype "]"      -> type;     "empty"  -> sep;
```

```
  variables
    "Sep" -> sep; "R" -> regtype;
  equations
    A* = {A empty}*; A+ = {A empty}+;
level 0
  syntax
    A                              -> {A Sep}+;
    {A Sep}+ Sep {A Sep}+          -> {A Sep}+ {right};
                                   -> {A Sep}*;
    {A Sep}+                       -> {A Sep}*;
    "[" {A Sep}* Sep {A Sep}* "]" -> {A Sep}* {right};
                                   -> empty;
    "(" R ")"                      -> R {bracket};
  variables
    "L" -> {A Sep}*; "Lp" -> {A Sep}+; "^" -> Sep;
  equations
    (Lp1 ^ Lp2) ^ Lp3 = Lp1 ^ (Lp2 ^ Lp3);
    [    ^ L  ]       = L;
    [L   ^    ]       = L;
    [Lp1 ^ Lp2]       = Lp1 ^ Lp2;
```

Observe again how expressions over the syntax defined at level 1 are used as sorts at level 0. For instance, in the production `{A Sep}+ -> {A Sep}*`, the syntax of the expression `{A Sep}+` is defined by the production `"{" type sep "}" "+" -> regtype` and by the variables `"A" -> type` and `"Sep" -> sep`.

We have introduced a new sort `regtype` at level 1 as the sort of list type constructors in order to avoid an infinite chain caused by the injection of arbitrary types in the corresponding list type. If we would have declared `"{" type type "}" "+" -> type`, the production `A -> {A Sep}+` would give rise to the productions `{A Sep}+ -> {{A Sep}+ Sep}+`, `{{A Sep}+ Sep}+ -> {{{A Sep}+ Sep}+ Sep}+`, etc., causing each expression to have infinitely many non-unifiable types. By introducing the new sort `regtype`, lists are not automatically embedded in types, i.e., `A` does not unify with `{A Sep}+` because their sorts are different.

The usage of list types is illustrated in the following grammar of a fragment of an imperative language. A statement is either an assignment, a while-do loop or a list of statements separated by semicolons.

```
module while
imports list, exp;
level 1
  syntax
    "var"  -> type; "exp" -> type;
    "stat" -> type; "‘;’" -> sep;
```

```
level 0
  syntax
    var ":=" exp              -> stat;
    "while" exp "do" stat     -> stat;
    "begin" {stat ';'}* "end" -> stat;
    ";"                       -> ';';
```

The expression {stat ';'}* is defined by the polymorphic productions in module `list`, which have the following instantiations.

```
  stat                         -> {stat ';'}+;
  {stat ';'}+ ';' {stat ';'}+  -> {stat ';'}+;
                               -> {stat ';'}*;
  {stat ';'}+                  -> {stat ';'}*;
```

4.4 Polymorphic Operations

Now that we have a polymorphic definition of list construction we can also define polymorphic functions over lists. For instance, the length function that computes the number of elements of a list can be polymorphically defined by the following specification:

```
module length
imports list;
level 0
  syntax
    "length" "(" {A Sep}* ")" -> nat;
  equations
    length()          = 0;
    length(X)         = s(0);
    length(Lp1 ^ Lp2) = length(Lp1) + length(Lp2);
```

4.5 Higher-Order Functions

Another example of a type constructor is the arrow `=>` of function types. A term of sort `A => B`, i.e., a function from `A` to `B`, can be applied to a term of sort `A` yielding a `B`.

```
module arrow
imports list;
level 1
  syntax
    type "=>" type -> type {right};
level 0
  syntax
    (A => B) "(" A ")"    -> B;
```

```
    (A => B) "*" {A Sep}* -> {B Sep}*;
  variables
    "F" -> (A => B);
```

The higher-order function * (map) takes as arguments a function from `A` to `B` and a list of `A`s and applies the function to each element of the list.

```
  equations
    F * ()          = ();
    F * (X)         = F(X);
    F * (Lp1 ^ Lp2) = (F * Lp1) ^ (F * Lp2);
```

If we want to pass functions such as length and map themselves as arguments to some higher-order function we need to define the combinators (curried versions) associated with the functions as follows:

```
  syntax
    "if"      -> (bool => A => A => A);
    "flength" -> ([{A Sep}*] => nat);
    "(*)"     -> (A => B) => ([{A Sep}*] => [{B Sep}*]);

    (A   => [R'])  "(" A ")" -> R';
    ([R] => B)     "(" R ")" -> B;
    ([R] => [R'])  "(" R ")" -> R';
  equations
    if(B)(X)(Y) = if B then X else Y fi;
    flength(L)  = length(L);
    (*)(F)(L)   = F * L;
```

Observe the usage of the operator `[_]` that injects `regtypes` into types in order to reuse the functionality for `type` expressions. We added extra application operators to apply functions like `flength` to lists.

These examples illustrate how two-level grammars provide user-definable syntax for sort symbols and generic definition of polymorphic mix-fix functions and constructors over data types. More advanced examples of two- and multi-level specifications (with prefix function signatures instead of grammars) can be found in Hearn and Meinke (1994), Hearn (1995), Visser (1996).

5 Properties

We have seen how two-level grammars can be used for polymorphic syntax definition in algebraic specification. To actually use two-level grammars in an executable specification formalism, it is necessary that we can parse strings over the language specified by a grammar. Unfortunately, the parsing problem for two-level grammars is in general undecidable as shown by the following theorem.

Theorem 5.1 (Sintzoff, 1967) *For every semi-Thue system T we can construct a Van Wijngaarden Grammar W such that the set $S(T)$ generated by T is the set $S(W)$ generated by W.*

Corollary 5.2 (Sintzoff, 1967) *Every recursively enumerable set is generated by a Van Wijngaarden grammar.*

Corollary 5.3 (Sintzoff, 1967) *The problem of determining, of a given string, whether or not it is generated by a given Van Wijngaarden grammar, is recursively unsolvable.*

Although the version of two-level grammars defined in this paper is somewhat weaker because it uses trees instead of strings as nonterminals at level 0, these constructs can be translated to our two-level grammars. From these theorems it follows that we cannot construct terminating parsers for arbitrary two-level grammars in a general way. However, for the purpose of polymorphic syntax definition we are interested only in restricted forms of the formalism.

One view on two-level grammars is that they are used to abbreviate frequently occurring patterns in context-free grammars, but that in the end we want only a finite context-free grammar from a ground subgrammar and the appropriate instantiations of generic productions. For instance, the grammar of the programming language in module `while` gives rise to the instantiation of the list construction functions for `{stat ';'}*` and to the instantiation of the list and map functions for lists of statements. This is the effect that is reached when reuse of functions is obtained by means of parameterized modules for which only finitely many instantiations are requested. Although it is clear by looking at a grammar, which instantiations of productions are needed for the implementation of a certain subgrammar, we have not yet found a syntactic characterization of productions such that such subgrammar operations are possible. A promising approach might be the extension of the layering operations of Hatcher and Rus (1976), Rus and Jones (1995) to two-level grammars.

In context-free grammars empty (ϵ) and chain productions are the cause of infinite ambiguities. In two-level grammars they are the cause of the undecidability of the formalism. In the rest of this section we investigate the restriction of the usage of such productions in order to achieve a subclass of the two-level grammars with a decidable parsing problem that still allows the kind of grammars as shown in Section 4.

Definition 5.4 (ϵ-elimination) The conventional method for eliminating ϵ-productions from context-free grammars applied to two-level grammars works by adding productions to the grammar according to the rule

$$\frac{\alpha A \beta \to B \in \mathcal{G}_0, \epsilon \to A' \in \mathcal{G}_0, \sigma(A') = \sigma(A)}{\sigma(\alpha\beta) \to \sigma(B) \in \mathcal{G}_0}$$

where σ is a most general unifier of A and A'. After no more productions can be added, all ϵ-productions are removed. Define $ee(\Gamma)$ to be the result of removing ϵ-productions from two-level grammar Γ by the above procedure.

Note that ϵ-elimination preserves both the language and the trees generated by the grammar (if ϵ-trees are identified).

Proposition 5.5 $\mathrm{L}(ee(\Gamma)) = \mathrm{L}(\Gamma)$ *and* $\mathrm{Tree}(\Gamma) \cong \mathrm{Tree}(ee(\Gamma))$

Deussen (1975) shows that this method can turn finite two-level grammars into infinite ones. Consider the following grammar that gives type `<a`^{n}`>` to each sentence $\mathtt{a}^m$ with $m \leq n$.

```
level 1
  sorts type, list;
  syntax
    "<" list ">" -> type
    "a"          -> list
    list "a"     -> list
  variables
    "L" -> list
level 0
  syntax
    <L> <a> -> <L a>;
    "a"     -> <a>;
            -> <a>;
```

If we try to eliminate the last production by substituting it in the first production we get the productions

```
   <L> -> <L a>;    %% <a> can be empty
       -> <a a>;    %% <L> unifies with the rhs of -> <a>
       -> <a a a>;  %% <L> unifies with the rhs of -> <a a>
```

and all other productions of the form `-> <a`^{n}`>` for $n > 0$. However, for many applications ϵ-productions can be eliminated. For instance, the production `-> A*` in the list grammar in Section 4 can be eliminated by means of the procedure outlined above, resulting in a finite two-level grammar defining the same language.

In a similar fashion chain productions can be eliminated from grammars.

Definition 5.6 (Chain Elimination) To eliminate chain production from a two-level grammar Γ, first take the transitive closure of all chains in $\mathcal{G}_0$:

$$\frac{\tau_1 \to \tau_2 \in \mathcal{G}_0, \tau_3 \to \tau_4 \in \mathcal{G}_0, \sigma(\tau_2) = \sigma(\tau_3)}{\sigma(\tau_1) \to \sigma(\tau_4) \in \mathcal{G}_0}$$

then use chain productions as substitutions

$$\frac{\tau_1 \to \tau_2 \in \mathcal{G}_0, \alpha\tau_3\beta \to \tau_4 \in \mathcal{G}_0, \sigma(\tau_2) = \sigma(\tau_3)}{\sigma(\alpha\tau_1\beta \to \tau_4) \in \mathcal{G}_0,}$$

and finally remove all chain productions from $\mathcal{G}_0$ resulting in $ce(\Gamma)$.

This procedure preserves the meaning of the grammar.

Proposition 5.7 $\mathrm{L}(\Gamma) = \mathrm{L}(ce(\Gamma))$ *and* $\mathrm{Tree}(\Gamma) \cong \mathrm{Tree}(ce(\Gamma))$.

Also chain elimination does not terminate for all grammars. Take for instance the grammar for lists in the previous section. If we redefine the syntax of the list operators as

```
"{" type sep "}" "*" -> type;  "{" type sep "}" "+" -> type
```

then we have that `A` unifies with `{A Sep}+` resulting in infinitely many productions

```
A                  -> {A Sep}+;
{A Sep}+           -> {{A Sep}+ Sep}+;
{{A Sep}+ Sep}+    -> {{{A Sep}+ Sep}+ Sep}+;
...
```

So we see that ϵ-elimination and chain elimination will not terminate for arbitrary grammars. However, for the grammars for which it succeeds we have the following corollary from Theorem 6.8 that we will prove in the next section.

Corollary 5.8 *If Γ is a finite two-level grammar without ϵ- and chain productions, then the question $w \in \mathrm{L}(\Gamma)$ is decidable.*

The intuition behind this result is that with ϵ- and chain-free grammars at most n reductions can be done for a string of length n. Based on the same idea, the next definition defines a characterization of a larger class of two-level grammars for which the parsing problem is decidable.

Definition 5.9 (Finite Chain Property) A two-level grammar Γ has the *finite chain property* if it is (1) ϵ-free, (2) its chain productions are non-cyclic and have a finite transitive closure and (3) it does not contain redundant productions, where a production $p \in \mathrm{P}(\mathcal{G}_0)$ is redundant if there is some $p' \in \mathrm{P}(\mathcal{G}_0)$ such that $p \neq p'$ and $p \geq p'$.

In the next section we will define a parsing algorithm for two-level grammars and prove that it is a decision procedure for membership of languages defined by finite chain two-level grammars.

On the positive side we have a subclass of the two-level grammars with a decidable parsing problem. On the negative side, membership of the class itself is undecidable.

Proposition 5.10 *It is undecidable whether a two-level grammar satisfies the finite chain property.*

However, decidability of the finite chain property is not essential for using two-level grammars for language specification. The situation can be compared to ambiguity of context-free grammars: Although it is undecidable whether a context-free grammar is ambiguous, it is a good formalism for defining unambiguous languages. A large class of grammars is evidently non-ambiguous and for others ambiguities will turn up when working with the grammars.

The examples presented in Section 4 satisfy the finite chain property, except for the empty production `->` `{A Sep}*` for lists. As remarked above this production is not a problematic ϵ-production because it can be eliminated from the grammar. In general we can follow the following procedure for determining whether a grammar has the finite chain property: (1) Try to eliminate ϵ-productions by the method of definition 5.4. (2) Try to eliminate chain rules by means of the method in definition 5.6. (3) If this terminates we know that the grammar has the finite chain property and that we can parse with it (see next section). (4) If either step (1) or step (2) takes too long, this is a hint that it does not terminate. In such cases we can inspect the list of ϵ-productions or chain productions added by the elimination procedures. These traces will give a clue about the productions that cause the nontermination, because these will lead to a repetition of similar productions, as we saw in the example above. This information can be used to redesign the grammar such that it satisfies the finite chain property.

6 Parsing

In this section we define a parsing algorithm for finite chain two-level grammars. The parsing algorithm below is a parallel bottom-up parsing algorithm that computes all parse trees for a sentence. This procedure is similar to the Hindley-Milner type assignment procedure used in functional languages, that assigns to each expression a single principal type (Damas and Milner, 1982). The difference is that in two-level grammars strings can have more than one principal type due to ambiguities. It will turn out that for finite chain two-level grammars there are only finitely many principal types for a string. We first define a function that gives the type of a parse tree.

Definition 6.1 The type of a parse tree is defined as:

$$\begin{aligned}\mathrm{type}(L) &= L\\ \mathrm{type}(var(x,\tau)) &= \tau\\ \mathrm{type}(app(\tau_1 \dots \tau_n \to \tau, [t_1, \dots, t_n])) &= \tau\end{aligned}$$

Next we define the data structure of parse configurations that is used in parsing.

Definition 6.2 A parse configuration $(\boldsymbol{t} \bullet a_1 \dots a_n)_\Phi$ is an element of the set $\mathrm{Tree}(\Gamma \cup \mathrm{V})^* \times \mathrm{Sl}(\Gamma)^* \times Set(\mathrm{V}_1)$, i.e., a triple consisting of a list of trees $\boldsymbol{t} = t_1 \dots t_m$ (the stack), a list of literals $a_1 \dots a_n$ (the remaining input) and a set of sort variables Φ (the sort variables over level 1 that are used in $\boldsymbol{t}$).

Algorithm 6.3 Define the function $parse(\Gamma) : \mathrm{S}(\Gamma)^* \to Set(\mathrm{Tree}(\Gamma))$ as

$$parse(\Gamma)(w) = \{t \mid (\epsilon \bullet w) \Rightarrow^*_\Gamma (t \bullet \epsilon)\}$$

where $\Rightarrow^*_\Gamma$ is the transitive closure of the one-step parse relation $\Rightarrow_\Gamma$ on parse configurations, which is defined by the rules

$$(\boldsymbol{t} \bullet a_1\ a_2 \dots a_n)_\Phi \Rightarrow_\Gamma (\boldsymbol{t}\ a_1 \bullet a_2 \dots a_n)_\Phi \qquad \text{(Shift)}$$

$$\frac{x \in \mathrm{V}(\tau'), \tau = \bar{\rho}(\tau')}{(\boldsymbol{t}\ x \bullet \boldsymbol{a})_\Phi \Rightarrow_\Gamma (\boldsymbol{t}\ var(x,\tau) \bullet \boldsymbol{a})_{\Phi \cup vars(\tau)}} \qquad \text{(Var)}$$

$$\frac{p \in \mathrm{P}(\mathcal{G}_0), \bar{\rho}(p) = \alpha \to \tau, |\alpha| = m, mgu(\alpha; \mathrm{type}(t_1, \dots, t_m)) = \sigma}{(\boldsymbol{t}\ t_1 \dots t_m \bullet \boldsymbol{a})_\Phi \Rightarrow_\Gamma (\bar{\bar{\sigma}}(\boldsymbol{t}\ app(p, [t_1, \dots, t_m])) \bullet \boldsymbol{a})_{\Phi \cup vars(\bar{\sigma}(p))}} \qquad \text{(Red)}$$

where $\rho : \mathrm{V}_1 \to \mathrm{V}_1$ is a renaming of sort variables occurring in Φ such that $\rho(\Phi) \cap \Phi = \emptyset$. We identify configurations that are the same up to renaming of sort variables.

We now prove that the algorithm is a correct implementation of $\Pi(\Gamma)$ for finite chain two-level grammars. We first show that the trees for the parser are correct parse trees.

Lemma 6.4 (Sound) $\forall t \in \mathrm{Tree}(\Gamma) : t \in parse(\Gamma)(w) \Rightarrow yield(t) = w$

Proof. We first prove that if $(\boldsymbol{t_1} \bullet \boldsymbol{a_1}) \Rightarrow_\Gamma (\boldsymbol{t_2} \bullet \boldsymbol{a_2})$, then $yield(\boldsymbol{t_1})\boldsymbol{a_1} = yield(\boldsymbol{t_2})\boldsymbol{a_2}$. For (Shift) and (Var) the property clearly holds. In (Red) we see $yield(\boldsymbol{t}\ t_1 \dots t_m)\boldsymbol{a} = yield(\bar{\bar{\sigma}}(\boldsymbol{t}\ app(p, [t_1 \dots t_m])))\boldsymbol{a}$ by definition of *yield* and by the fact that type substitutions do not affect the *yield* of a tree. But then also for $(\boldsymbol{t_1} \bullet \boldsymbol{a_1}) \Rightarrow^*_\Gamma (\boldsymbol{t_2} \bullet \boldsymbol{a_2})$ we have $yield(\boldsymbol{t_1})\boldsymbol{a_1} = yield(\boldsymbol{t_2})\boldsymbol{a_2}$. In particular, if $(\epsilon \bullet w) \Rightarrow^*_\Gamma (t \bullet \epsilon)$ we have that $w = yield(\epsilon)w = yield(t)\epsilon$. □

Next we show that the parser is *complete*, in the sense that any parse tree for the sentence can be derived by instantiating one of the parse trees produced by the parser.

Lemma 6.5 (Complete) $\forall t \in \mathrm{Tree}(\Gamma) : \mathit{yield}(t) = w \Rightarrow \exists t' \geq t : t' \in \mathit{parse}(\Gamma)(w)$

Proof. By induction on t: (1) if $t = L$ then $\mathit{parse}(\Gamma)(L) \ni L \geq L$.
(2) if $t = \mathit{var}(x, \tau)$, there is some $\tau' \geq \tau$ such that $x \in \mathrm{V}(\tau')$, but then $\mathit{parse}(\Gamma)(x) \ni \mathit{var}(x, \tau') \geq \mathit{var}(x, \tau)$.
(3) If $t = \mathit{app}(p, [t_1, \ldots, t_n])$ with $p = \tau_1 \ldots \tau_n \to \tau$ (a) By definition of *yield* we have $\mathit{yield}(t) = \mathit{yield}(t_1) \ldots \mathit{yield}(t_n) = w_1 \ldots w_n$ with $\mathit{yield}(t_i) = w_i$ for $1 \leq i \leq n$. (b) By induction we have $t'_i \geq t_i$ for $1 \leq i \leq n$—and thus $\tau'_i \geq \tau_i$ with $\tau'_i = \mathrm{type}(t'_i)$—such that $t'_i \in \mathit{parse}(\Gamma)(w_i)$. (c) By (App) there is some $\tau''_1 \ldots \tau''_n \to \tau'' = p'' \in \mathrm{P}(\mathcal{G}_0)$ such that $p'' \geq p$ (variables of p'' and p disjunct). By (b) and (c) there is a substitution σ_0 such that $\sigma_0(\tau''_1 \ldots \tau''_n) = \tau_1 \ldots \tau_n = \sigma_0(\tau'_1 \ldots \tau'_n)$. Then there is also a most general unifier, say σ. Now take $t' = \bar{\bar{\sigma}}(\mathit{app}(p'', [t'_1 \ldots t'_n])) \in \mathrm{Tree}(\Gamma)$. It is clear that $t' \geq t$ and that $(\epsilon \bullet w_1 \ldots w_n) \Rightarrow^*_\Gamma (t'_1 \ldots t'_n \bullet \epsilon) \Rightarrow_\Gamma (t' \bullet \epsilon)$ □

Next we show that the set of parse trees produced by the algorithm is *minimal* in the sense that it generates only the most general parse trees for a string.

Lemma 6.6 (Minimal) $\forall t, t' \in \mathit{parse}(\Gamma)(w) : t \geq t' \vee t' \geq t \Rightarrow t \doteq t'$

Proof. Assume that $t, t' \in \mathit{parse}(\Gamma)(w)$ and that $t > t'$. Because both trees are in the set there must be sequences of configurations for their derivation. Because $t > t'$, the trees have the same structure, i.e., the configuration sequences have the same number of reductions and shifts. But also because $t > t'$, there must be some point at which the sequences diverge, i.e.,

$$(\epsilon \bullet w) \Rightarrow^*_\Gamma (\boldsymbol{t_1 t_2} \bullet \boldsymbol{a}) \Rightarrow_\Gamma (\bar{\bar{\sigma}}_1(\boldsymbol{t_1}\ \mathit{app}(p_1, \boldsymbol{t_2})) \bullet \boldsymbol{a}) \Rightarrow^*_\Gamma (t \bullet \epsilon)$$

and

$$(\epsilon \bullet w) \Rightarrow^*_\Gamma (\boldsymbol{t_1 t_2} \bullet \boldsymbol{a}) \Rightarrow_\Gamma (\bar{\bar{\sigma}}_2(\boldsymbol{t_1}\ \mathit{app}(p_2, \boldsymbol{t_2})) \bullet \boldsymbol{a}) \Rightarrow^*_\Gamma (t' \bullet \epsilon)$$

for t and t', respectively. Because $t > t'$ we must have $p_1 > p_2$. Now we have either (1) $p_1 = p_2$ and $\sigma_1 > \sigma_2$, but then σ_2 is not a most general unifier and hence $t' \notin \mathit{parse}(\Gamma)(w)$ or (2) if $p_1 \neq p_2$, then Γ does not satisfy the finite chain property because it has the redundant production p_2. □

Finally we prove that *parse* yields a finite set of parse trees, entailing that *parse* is effectively computable.

Lemma 6.7 (Finite) $|\mathit{parse}(\Gamma)(w)| \in N$

Proof. (1) For each configuration and each production there is at most one reduction step (Red) because there is at most one most general unifier for α and $\tau_1 \ldots \tau_m$. For each configuration there is at most one (Shift) step and one (Var) step. Therefore, the graph of the relation $\Rightarrow_\Gamma$ is finitely branching.
(2) The length of configurations does not increase (no ϵ-productions). For any configuration $(\boldsymbol{t} \bullet \boldsymbol{a})$, $|\boldsymbol{a}|$ (Shift) steps can be done. A (Red) step with a production $\alpha \to \tau$ such that $|\alpha| > 1$ decreases the length of a configuration, therefore at most $|\boldsymbol{t}|/2$ such reductions can be performed for a configuration $(\boldsymbol{t} \bullet \boldsymbol{a})$. By finite chain property only finitely many chain reductions can be done, i.e., for each configuration $(\boldsymbol{t}\ t \bullet \boldsymbol{a})$ there is maximal value n such that $(\boldsymbol{t}\ t \bullet \boldsymbol{a}) \Rightarrow_\Gamma^n (\boldsymbol{t}\ t' \bullet \boldsymbol{a})$. Therefore, the graph of the relation $\Rightarrow_\Gamma$ has no infinite paths.
(3) From any configuration $(\boldsymbol{t} \bullet \boldsymbol{a})$ only finitely many configurations are reachable. In particular, for any string w only finitely many configurations of the form $(t \bullet \epsilon)$ are reachable from $(\epsilon \bullet w)$. $\square$

Finally, we see that Algorithm 6.3 is a correct implementation of a parser for finite chain two-level grammars.

Theorem 6.8 (Correct) *For any finite chain two-level grammar* Γ *and any string* $w \in \mathrm{Sl}(\Gamma)^*$, $parse(\Gamma)(w)$ *is a minimal and finite set of parse trees, unique up to renaming of sort variables, that generates* $\Pi(\Gamma)(w)$.

Proof. By Lemma 6.4 and Lemma 6.5 all and exactly the trees in $\Pi(\Gamma)(w)$ can be derived from $parse(\Gamma)(w)$. By Lemma 6.7 $parse(\Gamma)(w)$ is finite and by Lemma 6.6 it is minimal. $\square$

As a result the recognition problem for finite chain two-level grammars is decidable.

Corollary 6.9 (Decidable) *For a finite chain two-level grammar* Γ *it is decidable whether* $w \in \mathrm{L}(\Gamma)$ *and* $w \in \mathrm{L}(\Gamma)(\tau)$.

The relation $\Rightarrow_\Gamma$ defines a tree shaped search space. Only the types of trees in the configuration matter for the rest of the process. We would like to identify configurations $(\boldsymbol{t_1} \bullet \boldsymbol{a})$ and $(\boldsymbol{t_2} \bullet \boldsymbol{a})$ for which $|\boldsymbol{t_1}| = |\boldsymbol{t_2}|$ and $\mathrm{type}(\boldsymbol{t_1}) = \mathrm{type}(\boldsymbol{t_2})$. This would lead to a generalization of the graph structured stack and the parse forests of Tomita (1985) to parsing for two-level grammars.

7 Related Formalisms

In the same way that context-free grammars correspond to first-order signatures, two-level grammars correspond to two-level signatures. The type system of the functional programming language ML (Milner, 1978) can be considered as two-level signatures in which the expressions over level 1 are single-sorted expressions of sort `type`. This system was used to introduce parametric polymorphic functions. Two-level signatures are discussed in Poigné (1986), Möller (1987), Meinke (1992a). After a two-level signature is expanded, a, possibly infinite, one-level signature results that can again be used as the specification of the sort space of a level 0 signature. In this manner the extension of signatures to two-level signatures can be generalized to signatures with three and more levels. Hearn and Meinke (1994) introduce the three-level algebraic specification formalism Atlas, which is generalized by Hearn (1995) to a multi-level specification formalism. The complete and formal specification of the related multi-level specification formalism MLS is presented in Visser (1996). MLS supports overloading of function symbols, which entails that a term can have infinitely many types, but only finitely many most general or principal types. This property is not respected by general two-level grammars as discussed in the previous section.

On the grammatical side, many variants of two-level grammars have been proposed in the literature for various purposes. Van Wijngaarden grammars (VWG) (van Wijngaarden *et al.*, 1976) were developed to express the syntax and semantics of Algol68. In VWGs strings, instead of trees, over level 1 are used as nonterminals (hypernotions) at level 0. This leads to the problem of grammatical unification—whether two sentential forms over a context-free grammar are unifiable by means of a substitution of nonterminals with strings—which Maluszynski (1984) shows to be undecidable. The transparent two-level grammars of Maluszynski (1984) are a restriction of VWGs such that grammatical unification comes down to term unification. Another restriction of VWGs are the Extended Affix Grammars (EAG) (Watt, 1977) that restrict the order in which the variables in nonterminals at level 0 can be instantiated.

The observation that two-level grammars are Turing equivalent sparked another development: two-level grammars as logic or functional programming languages. The (context-free) metagrammar (level 1) is used to define the syntax of language and semantic domains. The hypergrammar (level 0) is used to define the operations on the data. See for example Maluszynski (1984).

Definite Clause Grammars (DCG) introduced by Pereira and Warren (1980) are grammars embedded in Prolog programs. They are equivalent to two-level grammars with a fixed level 1 equivalent to the following grammar

```
module dcg
level 1
  sorts fun, term;
  syntax
    [a-z][A-Za-z0-9]*        -> fun;
    fun                      -> term;
    fun "(" {term ","}* ")" -> term;
  variables
```

```
[A-Z][A-Za-z0-9]* -> term;
```

that defines an untyped domain of terms that can be used as grammar symbols in level 0. These terms are then typically used at level 0 in productions such as the following from a tiny natural language grammar:

```
np(N) vp(N)          -> s;
det(N) n(N) rel(N) -> np(N);
```

Parsing of DCGs—parsing as deduction (Pereira and Warren, 1983)—uses Prolog's built-in resolution strategy to answer queries like $w \in \mathrm{L}(\mathcal{G})(\mathtt{s})$. With the normal evaluation strategy of Prolog this comes down to top-down backtrack parsing. Problems with this strategy are that it cannot cope with left-recursion and that already computed answers are not reused. The tabulation strategy described in Warren (1992) partially overcomes these problems. One of the problems of the latter approach is that unification in Prolog is not many-sorted, disabling solutions like that with `regtype` in Section 4.

8 Conclusions

Algebraic specification with first-order signatures or context-free grammars enforce a choice between strongly typed structures with little genericity or generic operations over untyped structures. Polymorphism combines genericity with typedness, making it possible to develop libraries of specifications. In this paper we have discussed how the integration of algebraic specification with user-definable syntax and polymorphism can be materialized. The extension with polymorphism of algebraic specification formalisms that use context-free grammars as signatures, e.g., OBJ or ASF+SDF, leads necessarily to formalisms with two-level grammars as signatures. Likewise, the extension with user-definable syntax of formalisms that have polymorphic signatures, including polymorphic functional and logic programming languages, leads to two-level grammars.

In two-level grammars level 1 defines the syntax of sort symbols used at level 0. Sort terms with variables are interpreted as sort schemata that can have many instantiations. Productions at level 0 with such sorts are production schemata, i.e., declarations of polymorphic functions with mix-fix syntax. Thus two-level grammars combine polymorphism with user-definable syntax, as we illustrated by means of a number of examples of polymorphic syntax definition in data type and programming language specifications.

Although the parsing problem for context-free grammars and the type-assignment problem for two-level signatures are decidable, the parsing problem for the integration of both formalisms is undecidable if no restrictions are considered. We defined an intuitive restriction of the class of two-level grammars that results in a class of two-level grammars for which the parsing problem is decidable and for which we defined a parsing algorithm that yields a minimal and finite set of most general parse trees for each string.

Acknowledgments I thank Mark van den Brand, Arie van Deursen, Annius Groenink, Jan Heering, Jasper Kamperman, Paul Klint and Janos Sarbo and several referees for discussions and/or comments on previous versions of this paper.

References

Bidoit, M., Gaudel, M.-C., and Mauboussin, A. (1989). How to make algebraic specifications more understandable: An experiment with the PLUSS specification language. *Science of Computer Programming*, **12**, 1–38.

van den Brand, M., Klint, P., Olivier, P., and Visser, E. (in preparation). ATerms: Representing structured data for exchange between heterogeneous tools.

Damas, L. and Milner, R. (1982). Principal type-schemes for functional programs. In *Conference Record of the Ninth Annual ACM Symposium on Principles of Programming Languages*, pages 207–212. ACM.

Deussen, P. (1975). A decidability criterion for van Wijngaarden grammars. *Acta Informatica*, **5**, 353–375.

Futatsugi, K., Goguen, J., Jouannaud, J.-P., and Meseguer, J. (1985). Principles of OBJ2. In B. Reid, editor, *Conference Record of the Twelfth Annual ACM Symposium on Principles of Programming Languages*, pages 52–66. ACM.

Goguen, J. A., Thatcher, J. W., Wagner, E. G., and Wright, J. B. (1977). Initial algebra semantics and continuous algebras. *Journal of the ACM*, **24**(1), 68–95.

Hatcher, W. S. and Rus, T. (1976). Context-free algebras. *Journal of Cybernetics*, **6**, 65–76.

Hearn, B. M. (1995). *The Design and Implementation of Typed Languages for Algebraic Specification*. Ph.D. thesis, University of Wales, Swansea.

Hearn, B. M. and Meinke, K. (1994). ATLAS: A typed language for algebraic specification. In J. Heering, K. Meinke, B. Möller, and T. Nipkow, editors, *Proc. First Int. Workshop on Higher-Order Algebra, Logic and Term Rewriting - HOA '93*, volume 816 of *lecture Notes in Computer Science*, pages 146–168, Berlin. Springer-Verlag.

Heering, J., Hendriks, P. R. H., Klint, P., and Rekers, J. (1989). The syntax definition formalism SDF – reference manual. *SIGPLAN Notices*, **24**(11), 43–75. Most recent version available at ftp://ftp.cwi.nl/pub/gipe/reports/SDFManual.ps.Z.

Higgins, P. J. (1963). Algebras with a scheme of operators. *Mathematische Nachrichten*, **27**, 115–132.

Klint, P. and Visser, E. (1994). Using filters for the disambiguation of context-free grammars. In G. Pighizzini and P. San Pietro, editors, *Proc. ASMICS Workshop on Parsing Theory*, pages 1–20, Milano, Italy. Tech. Rep. 126–1994, Dipartimento di Scienze dell'Informazione, Università di Milano.

Maluszynski, J. (1984). Towards a programming language based on the notion of two-level grammar. *Theoretical Computer Science*, **28**, 13–43.

Meinke, K. (1992a). Equational specification of abstract types and combinators. In E. Boerger, G. Jaeger, H. K. Buening, and M. M. Richter, editors, *Computer Science Logic - CSL'91*, volume 626 of *Lecture Notes in Computer Science*, pages 257–271, Berlin. Springer-Verlag.

Meinke, K. (1992b). Universal algebra in higher types. *Theoretical Computer Science*, **100**, 385–417.

Meinke, K. and Tucker, J. (1992). Universal algebra. In S. Abramsky, D. Gabbay, and T. Maibaum, editors, *Handbook of Logic for Computer Science. Volume I: Mathematical Structures*, pages 189–141. Oxford University Press, Oxford.

Milner, R. (1978). A theory of type polymorphism in programming. *Journal of Computer and System Sciences*, **17**(3), 348–375.

Möller, B. (1987). Algebraic specification with higher-order operators. In L. Meertens, editor, *Program Specification and Transformation*, pages 367–398. Elsevier Science Publishers B.V. (North-Holland).

Pereira, F. C. N. and Warren, D. H. D. (1980). Definite Clause Grammars for language analysis—a survey of the formalism and a comparison with augmented transition networks. *Artificial Intelligence*, **13**, 231–278.

Pereira, F. C. N. and Warren, D. H. D. (1983). Parsing as deduction. In *Proceedings of the 21st Annual Meeting of the Association for Computational Linguistics*, Massachusetts Institute of Technology, Cambridge, Massachusetts.

Poigné, A. (1986). On specifications, theories, and models with higher types. *Information and Control*, **68**, 1–46.

Rus, T. and Jones, J. S. (1995). Multi-layered pipeline parsing from multi-axiom grammars. In A. Nijholt, G. Scollo, and R. Steetskamp, editors, *Algebraic Methods in Language Processing AMiLP'95*, volume 10 of *Twente Workshops in Language Technology*, pages 65–81, Enschede, The Netherlands. Twente University of Technology.

Sintzoff, M. (1967). Existence of a van Wijngaarden syntax for every recursively enumerable set. *Annales de la Société Scientifique de Bruxelles*, **81**(II), 115–118.

Tomita, M. (1985). *Efficient Parsing for Natural Languages. A Fast Algorithm for Practical Systems*. Kluwer Academic Publishers.

Visser, E. (1993). *Combinatory Algebraic Specification & Compilation of List Matching*. Master's thesis, Department of Computer Science, University of Amsterdam, Amsterdam.

Visser, E. (1996). Multi-level specifications. In A. van Deursen, J. Heering, and P. Klint, editors, *Language Prototyping. An Algebraic Specification Ap-*

proach, volume 5 of *AMAST Series in Computing*, pages 105–196. World Scientific, Singapore.

Warren, D. S. (1992). Memoing for Logic Programs. *Communications of the ACM*, **35**(3), 94–111.

Watt, D. A. (1977). The parsing problem for affix grammars. *Acta Informatica*, **8**(1), 1–20.

van Wijngaarden, A., Mailloux, B. J., Peck, J. E. L., Koster, C. H. A., Sintzoff, M., Lindsey, C. H., Meertens, L. G. L. T., and Fisker, R. G., editors (1976). *Revised Report on the Algorithmic Language Algol 68*. Springer-Verlag, Berlin Heidelberg New York.

Technical Reports of the Programming Research Group

Note: These reports can be obtained using the technical reports overview on our WWW site (URL `http://www.wins.uva.nl/research/prog/reports/`) or using anonymous ftp to `ftp.wins.uva.nl`, directory `pub/programming-research/reports/`.

[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.*

[P9609] T.B. Dinesh and S.M. Üsküdarlı. *The VAS formalism in VASE.*

[P9608] J.A. Hillebrand. *A small language for the specification of Grid Protocols.*

[P9607] J.J. Brunekreef. *A transformation tool for pure Prolog programs: the algebraic specification.*

[P9606] E. Visser. *Solving type equations in multi-level specifications (preliminary version).*

[P9605] P.R. D'Argenio and C. Verhoef. *A general conservative extension theorem in process algebras with inequalities.*

[P9602b] J.A. Bergstra and M.P.A. Sellink. *Sequential data algebra primitives (revised version of P9602).*

[P9604] E. Visser. *Multi-level specifications.*

[P9603] M.G.J. van den Brand, P. Klint, and C. Verhoef. *Reverse engineering and system renovation: an annotated bibliography.*

[P9602] J.A. Bergstra and M.P.A. Sellink. *Sequential data algebra primitives.*

[P9601] P.A. Olivier. *Embedded system simulation: testdriving the ToolBus.*

[P9512] J.J. Brunekreef. *TransLog, an interactive tool for transformation of logic programs.*

[P9511] J.A. Bergstra, J.A. Hillebrand, and A. Ponse. *Grid protocols based on synchronous communication: specification and correctness.*

[P9510] P.H. Rodenburg. *Termination and confluence in infinitary term*

rewriting.

[P9509] J.A. Bergstra and Gh. Stefanescu. *Network algebra with demonic relation operators.*

[P9508] J.A. Bergstra, C.A. Middelburg, and Gh. Stefanescu. *Network algebra for synchronous and asynchronous dataflow.*

[P9507] E. Visser. *A case study in optimizing parsing schemata by disambiguation filters.*

[P9506] M.G.J. van den Brand and E. Visser. *Generation of formatters for context-free languages.*

[P9505] J.M.T. Romijn. *Automatic analysis of term rewriting systems: proving properties of term rewriting systems derived from* ASF+SDF *specifications.*

[P9504] M.G.J. van den Brand, A. van Deursen, T.B. Dinesh, J.F.Th. Kamperman, and E. Visser (editors). ASF+SDF *'95: a workshop on Generating Tools from Algebraic Specifications, May 11&12, 1995, CWI Amsterdam.*

[P9503] J.A. Bergstra and A. Ponse. *Frame-based process logics.*

[P9208c] J.C.M. Baeten and J.A. Bergstra. *Discrete time process algebra (revised version of P9208b).*

[P9502] J.A. Bergstra and P. Klint. *The discrete time ToolBus.*

[P9501] J.A. Hillebrand and H.P. Korver. *A well-formedness checker for* μ*CRL.*

[P9426] P. Klint and E. Visser. *Using filters for the disambiguation of context-free grammars.*

[P9425] B. Diertens and A. Ponse. *New features in PSF II: iteration and nesting.*

[P9424] M.A. Bezem and A. Ponse. *Two finite specifications of a queue.*

[P9423] J.J. van Wamel. *Process algebra with language matching.*

[P9422] R.N. Bol, L.H. Oei J.W.C. Koorn, and S.F.M. van Vlijmen. *Syntax and static semantics of the interlocking design and application language.*

[P9421] J.A. Bergstra and A. Ponse. *Frame algebra with synchronous communication.*