Natural Semantics as a Static Program Analysis Framework

SABINE GLESNER
Universität Karlsruhe
and
WOLF ZIMMERMANN
Martin-Luther-Universität Halle-Wittenberg

Natural semantics specifications have become mainstream in the formal specification of programming language semantics during the last 10 years. In this article, we set up sorted natural semantics as a specification framework which is able to express static semantic information of programming languages declaratively in a uniform way and allows one at the same time to generate corresponding analyses. Such static semantic information comprises context-sensitive properties which are checked in the semantic analysis phase of compilers as well as further static program analyses such as, for example, classical data and control flow analyses or type and effect systems. The latter require fixed-point analyses to determine their solutions. We show that, given a sorted natural semantics specification, we can generate the corresponding analysis. Therefore, we classify the solution of such an analysis by the notion of a proof tree. We show that a proof tree can be computed by solving an equivalent residuation problem. In case of the semantic analysis, this solution can be found by a basic algorithm. We show that its efficiency can be enhanced using solution strategies. We also demonstrate our prototype implementation of the basic algorithm which proves its applicability in practical situations. With the results of this article, we have established natural semantics as a framework which closes the gap between declarative and operational specification methods for static semantic properties as well as between specification frameworks for the semantic analysis. In particular, we show that natural semantics is expressive enough to define fixed-point program analyses.

Categories and Subject Descriptors: D.3.1 [**Programming Languages**]: Formal Definitions and Theory—*Syntax, semantics*; D.3.3 [**Programming Languages**]: Language Constructs and Features—*Constraints*; D.3.4 [**Programming Languages**]: Processors—*Translator writing systems and compiler generators*; F.3.1 [**Logics and Meanings of Programs**]: Specifying and Verifying and Reasoning about Programs—*Specification techniques*; F.3.2 [**Logics and Meanings of Programs**]: Semantics of Programming Languages—*Program analysis*; I.2.8 [**Artificial Intelligence**]: Problem Solving, Control Methods, and Search—*Graph and tree search strategies*

General Terms: Languages

Authors' addresses: S. Glesner, Institut für Programmstrukturen und Datenorganisation, Universität Karlsruhe, D 76128 Karlsruhe, Germany; email: glesner@ipd.info.uni-karlsruhe.de; W. Zimmermann, Institut für Informatik, Martin-Luther-Universität Halle-Wittenberg, D 06099 Halle/Saale, Germany; email: zimmer@informatik.uni-halle.de.

Additional Key Words and Phrases: Natural semantics, static program analysis, semantic analysis, fixed-point program analyses, constraint solving, compiler generators

1. INTRODUCTION

Natural semantics has become the specification method of choice for the semantics of programming languages during the last decade. It defines static semantic properties as well as the dynamic semantics of programming languages by inference rule systems. Natural semantics has grown in popularity over denotational semantics because fixed points are defined implicitly as part of the rule formalism. Moreover, natural semantics is more modular than denotational semantics because static and dynamic semantic information can be specified separately from each other without the need for a common fixed point.

In this article, we argue that natural semantics is a uniform declarative specification method for static semantic properties of programming languages. In particular, context-sensitive properties of programming languages such as well-typing or proper declarations of variables can be specified in the same manner as further static program analyses ranging from classical data and control flow analyses to type and effect systems. Usually context-sensitive properties can be computed directly during the semantic analysis phase in compilers by transporting semantic information through the abstract syntax trees of programs. In contrast, typical static program analyses need fixed-point computations to determine a solution. Hence, we establish natural semantics as a uniform framework to express the semantic analysis as well as fixed-point program analyses. This is important not only from a theoretical point of view. Many compilers do not strictly separate between the semantic analysis and succeeding program analyses. Therefore it is an important feature of a specification framework to be able to express both together. Moreover, because of its declarativity, natural semantics is especially suited to be used in the formal and, by employing automated theorem provers, mechanical verification of programming language properties. On the practical side, natural semantics specifications allow for the generation of corresponding analyses. It is the goal of our work to establish natural semantics as a formalism which meets the seemingly contrary requirements of being declarative and of being able to generate efficient analyses.

Natural semantics[1] [Kahn 1987] is a deductive method to determine program properties. Thereby, axioms and inference rules specify semantic properties with respect to language elements. For example, the type checking rule for the conditional statement would be defined as shown in Figure 1. The conclusion of the inference rule describes the program node corresponding to the whole conditional statement while the three assumptions specify the children nodes. Natural deduction [Gentzen 1935] is used to infer the properties of an entire program. Due to its logical character, natural semantics is a declarative specification method. Moreover, the program structure corresponds directly

[1]*Deductive semantics* would be a better name.

$$\frac{p \vdash \mathsf{E}_1 : \mathsf{bool} \qquad p \vdash \mathsf{E}_2 : \tau \qquad p \vdash \mathsf{E}_3 : \tau}{p \vdash \texttt{if } \mathsf{E}_1 \texttt{ then } \mathsf{E}_2 \texttt{ else } \mathsf{E}_3 : \tau}$$

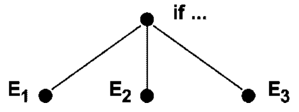

Fig. 1. Inference rule for the conditional statement.

with the structure of a proof for the static semantic properties of the program. This correspondence means that analysis implementations can be automatically generated because the proof structure is already known in advance and does not need to be computed by an extensive search. So natural semantics specifications have the potential to be declarative and, simultaneously, enable the generation of efficient parts of a compiler. This is also the reason why natural semantics specifications have become widespread in the last decade. The most prominent example for this development is the complete specification of the static and dynamic semantics of Standard-ML [Milner et al. 1990, 1997; Milner and Tofte 1991]. In contrast to the general character of natural semantics, most current tools need restricted forms as input so that declarativity and the ability to generate analyses do not exist at the same time; see Section 8 for a detailed discussion.

As a solution to bridge this gap between declarative and operational specification methods as well as between specification frameworks for the semantic analysis and further fixed-point program analyses, we present sorted natural semantics. It is a declarative specification method for static semantic properties of programming languages. In particular, it is able to define semantic properties of imperative and object-oriented programming languages. Specifications in this framework are modular. By modularity, we mean that different aspects of the language semantics can be specified independently from each other so that specifications are easily extensible and reusable. The corresponding analysis can be generated from such specifications. Therefore we represent the solution of such an analysis by the notion of a proof tree. We show that proof trees can be computed by solving an equivalent residuation problem. For the semantic analysis, we show how such constraint problems can be solved with a basic algorithm. In special cases, the efficiency of this basic algorithm can be enhanced by solution strategies. Since attribute grammars are regarded as sufficiently expressive to specify the context-sensitive properties of imperative and object-oriented programming languages, we compare our approach with them. Thereby we show that each well-defined attribute grammar can also be expressed in our specification language so that the semantic analysis can be generated. Moreover, we demonstrate that language constructs which are common in object-oriented programming languages can be expressed more easily and declaratively in sorted natural semantics than in attribute grammars. Concerning fixed-point program analyses, we discuss that the classical data and control flow analyses expressable within monotone frameworks, (cf. Nielson et al. [1999]), can also be specified in sorted natural semantics. Moreover, we show that type and effect systems (cf. also Nielson et al. [1999]), can directly be stated in sorted natural semantics. The basic algorithm for the semantic analysis corresponds directly with a concurrent constraint program.

We have implemented the basic algorithm based on this observation. A test implementation using the concurrent constraint programming language Oz shows the feasibility of the basic algorithm. The much more efficient prototype implementation using Java employed these experiences and proves that the basic algorithm can be used in practice. Throughout this article, we use two example specifications: Mini-Java is a small object-oriented programming language taking into account inheritance, subclassing, and the polymorphism of Java. Its specification demonstrates the applicability of sorted natural semantics to object-oriented programming languages. The second example language concentrates on the use of variables before their definition and is introduced to show that sorted natural semantics specifications are more declarative than attribute grammars. (Parts of this article have been published in Glesner and Zimmermann [1997, 1998] and Glesner [1998, 1999a, 1999b].)

This article is organized as follows: In Section 2, we define sorted natural semantics, thereby also using parts from the example specifications to demonstrate the method. In particular, we characterize analysis results based on the notion of proof trees. In Section 3, we introduce the basic algorithm to generate the semantic analysis. In Section 4, we describe solution strategies to enhance the basic algorithm for the semantic analysis. They exploit ideas from the theory of attribute grammars. The comparison with attribute grammars is given in Section 5. In Section 6 we show that classical data flow analyses as well as type and effect systems are expressible within sorted natural semantics. The prototype implementation is presented in Section 7. In Section 8, we discuss related work. Finally, in Section 9, we conclude and list some ideas for future work. The example specifications used throughout this article are listed in Appendices A and B, respectively.

2. SORTED NATURAL SEMANTICS

We define the logical calculus of sorted natural semantics. In particular, we demonstrate how static semantic properties of programming languages can be specified within that calculus. Then we define the notion of proofs for semantic properties. Finally we show how proofs can be found by reducing this question to a residuation problem. This residuation problem can be tackled with constraint solving techniques.

2.1 Logical Calculus

We describe semantic information as terms of a sorted logic. First we present this logic. Then we introduce axioms and inference rules which define static semantic properties of programming languages. They associate semantic information with the terminals and nonterminals of particular productions of the abstract syntax.

2.1.1 *Sorted Semantic Information.* We use abstract data types to describe static semantic information. According to Wirsing [1990], an abstract data type is defined over a signature $\Sigma = \langle S, F \rangle$ where S is a set of sorts and F is a set of function symbols. We associate with each $f \in F$ input sorts $\mathbf{T}_1, \ldots, \mathbf{T}_n \in S$ and one output sort $\mathbf{T} \in S$ denoted by $\mathbf{T}_1 \times \cdots \times \mathbf{T}_n \rightarrow \mathbf{T}$. The notion of a term t of sort $\mathbf{T}$ is defined in the standard way; for details see Wirsing [1990]. Abstract data types can be defined inductively as term algebras via constructor functions. In these cases, a sort $\mathbf{T} \in S$ is interpreted as the set of constructor terms of sort $\mathbf{T}$. Additionally, we consider finite sets over sorts. The sort of finite sets over a sort $\mathbf{T}$ is denoted by $\{\mathbf{T}\}$. Elements of $\{\mathbf{T}\}$ can be defined extensionally as finite sets because sets allow for particularly declarative specifications. Finite sets are sufficient because all information derived by static analyses of programs is finite.

To simplify the presentation, we assume that the following basic sorts **Nat**, **Bool**, **String**, and **GenInfo** do already exist. **Nat**, **Bool**, and **String** are the sorts representing the integers together with the usual arithmetic operations, the Boolean values with the constants $\underline{\mathsf{true}}$ and $\underline{\mathsf{false}}$ and the common functions on truth values, as well as the strings of finite length together with their operations and constants as, for example, string concatenation or constants to denote individual strings. **GenInfo** is a sort with the constant $\underline{\mathsf{correct}}$ whose sole function is to mark correct programs or program fragments. It is straightforward to define the basic sorts as abstract data types via appropriate constructor functions: for example, for the natural numbers we would take the constant 0 and the successor function $s : \mathbf{Nat} \rightarrow \mathbf{Nat}$. We assume the following pairwise disjoint, enumerable infinite sets of symbols: Σ, the set of all sorts, $\mathcal{C}$, the set of all constructor functions, and $\mathcal{D}$, the set of all defined functions.

Each semantic information is a pair t :: **Sort** denoting that term t is of sort **Sort**. Each sort is either a basic sort or defined by a sort equation. A specification contains a list of sort equations, each of the form $\mathbf{Sort} = \mathit{Sort\_Term}$, $\mathbf{Sort} \in \Sigma$. *Sort_Term* is a term built from basic sorts or sorts which are also defined in the list of sort equations, that is, inductive and mutually recursive definitions are also allowed. In particular, **Sort** can appear in *Sort_Term* as well. In detail, *Sort_Term* can be built as follows:

—*Term algebra*: $\mathbf{Sort} = \mathsf{F}_1(\mathbf{S}_1^{(1)}, \ldots, \mathbf{S}_{m(1)}^{(1)}) \oplus \cdots \oplus \mathsf{F}_n(\mathbf{S}_1^{(n)}, \ldots, \mathbf{S}_{m(n)}^{(n)})$ is a valid sort equation if $\mathsf{F}_i \neq \mathsf{F}_j$ for $i \neq j$, and if $\mathsf{F}_1, \ldots, \mathsf{F}_n$ are constructor functions which have not been used in any other sort equation in the specification. As an example, consider the self-explanatory definition of the set **Nat_List** of lists of natural numbers: $\mathbf{Nat\_List} = [] \oplus \underline{\mathsf{cons}}(\mathbf{Nat}, \mathbf{Nat\_List})$.

—*Cartesian product*: $\mathbf{Sort} = \mathbf{S}_1 \times \cdots \times \mathbf{S}_n$ is a valid sort definition if for all $1 \leq i \leq n$, $\mathbf{S}_i$ is defined by a sort equation and $\mathbf{S}_i$ does not depend on **Sort**. A Cartesian product defines a term algebra with constructor $\langle \cdot, \ldots, \cdot \rangle_{\mathbf{Sort}}$. We omit the subscript if it is clear from the context.

—*Renaming*: $\mathbf{Sort} = \mathbf{S}$ is a Cartesian product with $n = 1$. For example, in the specification of Mini-Java in Appendix A, we define the sort **Type** as being equal to the sort **String**, $\mathbf{Type} = \mathbf{String}$. This is useful in object-oriented programming languages because types are identified by names of classes.

—*Sets*: $\mathbf{Sort} = \{\mathbf{S}\}$ is a valid sort definition if $\mathbf{S}$ is defined by a sort equation and $\mathbf{S}$ does not depend on **Sort**. Each set sort is defined as a term algebra with the constructor functions $\emptyset_{\mathbf{Sort}}$, $\{\cdot\}_{\mathbf{Sort}}$, and $\cup_{\mathbf{Sort}}$. We omit the subscript if it is clear from the context. The constructor $\cup$ is associative, commutative, and idempotent. Consequently only finite subsets of $\mathbf{S}$ can be represented by $\mathbf{Sort}$. For example, $\mathbf{Types} = \{\mathbf{Type}\}$ defines $\mathbf{Types}$ as a sort whose elements are sets containing terms of sort $\mathbf{Type}$.

—*Lists*: $\mathbf{Sort} = [\mathbf{S}]$ is valid sort definition if $\mathbf{S}$ is defined by a sort equation and $\mathbf{S}$ does not depend on $\mathbf{Sort}$. A list sort has the constructors $[]_{\mathbf{Sort}}$ and $\mathsf{cons}_{\mathbf{Sort}} : \mathbf{S} \times \mathbf{Sort} \to \mathbf{Sort}$.

The sort equations define the signatures of the constructor functions: if $\mathbf{S} = \mathsf{F}_1(\mathbf{S}_1^{(1)}, \ldots, \mathbf{S}_{m(1)}^{(1)}) \oplus \cdots \oplus \mathsf{F}_n(\mathbf{S}_1^{(n)}, \ldots, \mathbf{S}_{m(n)}^{(n)})$ is the sort equation, then $\mathsf{F}_i : \mathbf{S}_1^{(i)} \times \cdots \times \mathbf{S}_{m(i)}^{(i)} \to \mathbf{S}$ is the signature of constructor function F_i, $1 \leq i \leq n$. Since we deal only with first-order functions, the symbol $\to$ only occurs in the function signatures and not in the arguments or results.

We allow recursion only via constructors in order to avoid inconsistencies such as, for example, $\mathbf{Sort} = \{\mathbf{Sort}\}$ or $\mathbf{Sort} = \mathbf{S} \times \mathbf{Sort}$. It is not possible to construct terms representing such sets or Cartesian products, respectively. We also do not have subsort declarations. This avoids Russell's paradox because there is no common supersort $\mathbf{Set}$ which would have $\{\mathbf{Set}\}$ as a subsort of $\mathbf{Set}$.

The sort equation $\mathbf{Sort} = \{\mathsf{R}(\mathbf{S}_1, \ldots, \mathbf{S}_n)\}$ is an abbreviation for the two definitions $\mathbf{X} = \mathsf{R}(\mathbf{S}_1, \ldots, \mathbf{S}_n)$ and $\mathbf{Sort} = \{\mathbf{X}\}$, where $\mathbf{X}$ is a new sort symbol different from all sorts in the specification. It defines the sort of all *n-ary relations* over the sorts $\mathbf{S}_1, \ldots, \mathbf{S}_n$ whereby R is the relation predicate of these relations. As an example, consider the sort equation $\mathbf{Subtyping} = \{\sqsubseteq (\mathbf{Type}, \mathbf{Type})\}$ from the Mini-Java specification in Appendix A which defines the sort of subtyping relations as a binary relation over $\mathbf{Type}$ with the relation symbol $\sqsubseteq$.

For the defined functions, the signatures are explicitly stated in the set *Sig*: $Sig = \{\mathsf{f} : \mathbf{S}_1 \times \cdots \times \mathbf{S}_n \to \mathbf{S} \mid n \in \mathbb{N}, \mathbf{S}_1, \ldots, \mathbf{S}_n, \mathbf{S} \in \Sigma, \mathsf{f} \in \mathcal{D}\}$. $\mathcal{D}_{\mathbf{Set}_{\mathbf{Sort}}} = \{\cap_{\mathbf{Sort}}, \in_{\mathbf{Sort}}, \setminus_{\mathbf{Sort}}\} \subseteq \mathcal{D}$ are the defined functions for sets of sort $\mathbf{Sort} = \{\mathbf{S}\}$. With $\mathcal{D}_{\mathbf{Set}} = \bigcup_{\{\mathbf{Sort} \mid \mathbf{Sort} \text{ is a set sort}\}} \mathcal{D}_{\mathbf{Set}_{\mathbf{Sort}}}$, we denote the set of the defined functions of all set sorts.

Terms and formulas are built as a first-order language. We assume a set $\mathcal{V}$ of sorted variables $x :: \mathbf{Sort}$. Each variable x is annotated with its sort $\mathbf{Sort}$. Terms t are also annotated with their sort: $t :: \mathbf{Sort}$. If clear from the context, we omit the sort annotation, especially for formulas. An equation is a pair of terms of the same sort, denoted by $t_1 = t_2$. An atomic formula is either an equation, or a sort membership statement of the form $t :: \mathbf{Sort}$, or a term of sort $\mathbf{Bool}$. General formulas are built as usual, (cf. Wirsing [1990]). The free and quantified variables in a formula φ and the variables contained in a term t are defined in the usual way as well, denoted by $FV(\varphi)$, $QV(\varphi)$, and $V(t)$. We use the standard interpretation for formulas; see, for example, Wirsing [1990]. $\{x_1, \ldots, x_n\}$ is an abbreviation for $\{x_1\} \cup (\cdots \cup (\{x_{n-1}\} \cup \{x_n\}))$, $\mathsf{P}(t_1, \ldots, t_n)$ is an abbreviation for $\mathsf{P}(t_1, \ldots, t_n) = \underline{\mathsf{true}}$. *Restricted quantified formulas* are formulas where only quantifications over finite sets are allowed, for example, $\forall (x :: \mathbf{S}_1) \in (M :: \{\mathbf{S}_1\}).\varphi$. This is an abbreviation for $\forall (x :: \mathbf{S}_1).(x :: \mathbf{S}_1) \in (M :: \{\mathbf{S}_1\}) \Rightarrow \varphi$. Truth values of restricted quantified formulas can easily be computed as soon as the values of all free variables representing finite sets are known ($M :: \{\mathbf{S}_1\}$). If $M = \{t_1, \ldots, t_n\}$, the above formula is equivalent to $\varphi[t_1/x] \wedge \cdots \wedge \varphi[t_n/x]$ where $\varphi[t/x]$ denotes the substitution of the free variable x by term t.

Variable substitutions are sorted in the sense that variables are replaced by terms of the same sort. Variable renamings are injective variable substitutions replacing variables by variables. Two terms t_1 and t_2 are unifiable if there exists a substitution σ such that $\sigma(t_1) = \sigma(t_2)$. σ is a most general unifier of t_1 and t_2 if, for every unifier τ, there exists a substitution π such that $\tau = \pi \circ \sigma$. Most general unifiers are computed with the Herbrand-Robinson algorithm [Robinson 1965].

A syntactic unification is not sufficient for terms of a set sort. For unification of set terms we often consider terms such $S \cup \{t_1, \ldots, t_n\}$ where $t_i \notin S$. We denote this by $S \uplus \{t_1, \ldots, t_n\}$. Therefore, we need to consider the associativity and commutativity of $\cup$ but not the idempotency. Hence, in our framework each set sort can be described by an AC1 (associative, commutative with identity element) equational theory where the empty set is the identity element with respect to $\cup$. Two semantically equivalent set terms can be syntactically different. Therefore, we need to consider AC1-unifiers which unify set terms modulo the AC1 equational theory. In general, a minimal complete set of AC1-unifiers may be doubly exponential in the size of the given AC1-problem [Domenjoud 1992]. To decide the AC1-unifiability of two terms is NP-complete [Kapur and Narendran 1992] (cf. Baader and Schulz [1998] for an overview). In Section 3, we show how AC1-unifiers can be computed efficiently in special cases which are sufficient to cover the situations arising in semantic analysis. In general, one needs to make sure that, for each specification employing set sorts, this AC1-unification problem has a reasonable solution. For now, we assume that we have AC1-unifications at our disposal.

Remark 2.1. We distinguish between properties depending on the program and general properties which are independent of the program. The latter are formalized by logical formulas, the former by means of inference rules. However, there are situations where properties stem from both, the program and general properties. For example, the subtyping relations in object-oriented programs stem from the subclass declarations in the program (program-dependent property) and from the transitivity of subtype relations (program-independent property). Such situations are modeled by relations. We therefore distinguish between predicates (completely program-independent) and relations (based also on program-dependent information).

Predicates and defined functions on the sorts are specified declaratively by equational Horn clauses. We have chosen Horn clauses because we regard them as a declarative formalism but any other equivalent will do as well. For example, the sort **Subtyping** $= \{\sqsubseteq (\mathbf{Type}, \mathbf{Type})\}$ represents inheritance relations in object-oriented programs. Such relations are transitive, expressed by the following Horn clause:

$$\begin{aligned} \mathsf{A} \sqsubseteq \mathsf{C} \in \mathsf{subtypes} \;\leftarrow\; & \mathsf{subtypes} :: \mathbf{Subtyping}, \\ & \mathsf{A} \sqsubseteq \mathsf{B} \in \mathsf{subtypes}, \\ & \mathsf{B} \sqsubseteq \mathsf{C} \in \mathsf{subtypes}. \end{aligned}$$

The head of the Horn clause has the form $\mathsf{F}(t_1, \ldots, t_n) = t$ where F is a defined function not contained in $\mathcal{D}_{\mathbf{Set}}$ except $\in$ for relations. The subgoals of the

Horn clause are either sorted variables $x :: \mathbf{Sort}$ or formulas without quantifications. Extra variables, that is, variables which occur in the body of a clause but not in the head, are allowed (cf. as an example the above transitivity rule). Furthermore, a Horn clause might have a side condition which states when this Horn clause is valid. As conditions, restricted quantified formulas are allowed whereby the free variables of the condition must be contained in the variables of the subgoals and the head of the Horn clause. Note that it is not possible to specify the behavior of constructor functions or functions on sets with Horn clauses; only defined functions and predicates as well as relations can be defined with them. Relations are special because general properties on relations are required to be defined (e.g., the transitivity of a binary relation; see above). This affects the element-function "$\in$" on relations. If there is no explicit definition for the properties of a relation, then the usual $\in$-function is assumed.

Sorts constructed from sets are particularly useful to define semantic information consisting of a collection of uniform data, for example, definition tables in the semantic analysis. Elements from such sets have typically an internal structure with certain properties. For example, they could be tuples whose first component uniquely identifies them. We specify such a search key property by defining a function $\mathsf{unique}_{\mathbf{Sort}}$. Assume that elements of Sort $\mathbf{Sort}$ are sets containing elements of sort $\mathbf{Sort}'$. Assume furthermore that elements of $\mathbf{Sort}'$ are pairs whose first component (e.g., the name of a variable) uniquely identifies them. This can be expressed with the following two Horn clauses:

$$\mathsf{unique}_{\mathbf{Sort}}(\langle x, y\rangle) = [x].$$
$$e_1 = e_2 \;\leftarrow\; \mathsf{unique}_{\mathbf{Sort}}(e_1) = \mathsf{unique}_{\mathbf{Sort}}(e_2).$$

The latter of the two Horn clauses can be specified for every sort. In general, the signature of $\mathsf{unique}_{\mathbf{Sort}}$ is given by $\mathsf{unique}_{\mathbf{Sort}} : \mathbf{Sort}' \rightarrow [\mathbf{S}]$ where $\mathbf{S}$ is the sort of the search keys. For each sort $\mathbf{S}$ of a specification, if there is no explicit definition of a unique-function, by convention it is implicitly assumed to be the identity function. The unique-functions play an important rôle in the restricted AC1-unification, as shown in Section 3.

2.1.2 *Axioms and Inference Rules.* Axioms and inference rules define the static semantic properties of nodes in the abstract syntax tree. Their assumptions and conclusions consist of judgments. *Judgments* define the properties of a program node k as a sequence of semantic information $\mathsf{t}_1, \ldots, \mathsf{t}_{l+n}$ of sorts $\mathbf{Sort}_1, \ldots, \mathbf{Sort}_{l+n}$, respectively:

$$\mathsf{t}_1 :: \mathbf{Sort}_1, \ldots, \mathsf{t}_l :: \mathbf{Sort}_l \vdash k : \mathsf{t}_{l+1} :: \mathbf{Sort}_{l+1}, \ldots, \mathsf{t}_{l+n} :: \mathbf{Sort}_{l+n}.$$

The sequence of semantic information $\mathsf{t}_1 :: \mathbf{Sort}_1, \ldots, \mathsf{t}_l :: \mathbf{Sort}_l$ is called the *context* of node k while $\mathsf{t}_{l+1} :: \mathbf{Sort}_{l+1}, \ldots, \mathsf{t}_{l+n} :: \mathbf{Sort}_{l+n}$ is called the *properties* of k. We say that the judgment is a *judgment for k*. In principle, there is no difference between the semantic information in the context and in the properties. In particular, there is no logical consequence between the semantic information in the context and in the properties. Merely in the description of programming languages, it is common to separate the semantic information of a node into that which is derived from its predecessor (the context) and into that which is derived from its children (the properties), even though this distinction is not strict here.[2] Since we use sorted semantic information, we can specify different kinds of semantic information independently from each other, making *modular specifications* possible.

Inference rules define the judgment for a node of the abstract syntax tree depending on the judgments of its successors. An inference rule consists of *assumptions* $A_1, \ldots, A_n$, a *conclusion* C, and a *side condition* φ. $A_1, \ldots, A_n, C$ are judgments.

$$\frac{A_1, \ldots, A_n}{C} \text{ if } \varphi$$

φ is a restricted quantified formula which contains as free variables only those which are already contained in the assumptions or the conclusion of the inference rule. Therefore the validity of φ is decidable and easily computable if the values of all free variables in the inference rule are known and if the free variables in the restricted quantified formulas are instantiated with finite sets. If there is a production $X_0 ::= X_1 \cdots X_n$ in the abstract syntax of the programming language, then we require that there be at least one inference rule whose conclusion is a judgment for X_0 and whose n assumptions are judgments for $X_1, \ldots, X_n$. An axiom is an inference rule without assumptions, necessary to describe the judgments for terminal nodes without successors. Substitutions σ can be applied to judgments and inference rules by replacing each contained semantic information $\mathsf{t} :: \mathbf{Sort}$ by $\sigma(\mathsf{t}) :: \mathbf{Sort}$.

Example 2.2. In the specification of Mini-Java, we specify the following inference rule for the assignment production. Thereby, we use a context which consists of five components. Names is a set that contains the names of all classes in the program, TH is a reflexive and transitive relation describing the type hierarchy, that is, the subtype relation, of the program, Intfs describes the interfaces of all methods and attributes of all classes, A is the name of the current class, and locals are the local declarations within its body. (It would have been also possible to use five separated items of semantic information in the context but for the sake of readability, we have put them together.) $stat ::= des := expr$:

$$\frac{\begin{array}{c}\langle \mathsf{Names}, \mathsf{TH}, \mathsf{Intfs}, \mathsf{A}, \mathsf{locals}\rangle :: \mathbf{Context}_3 \vdash des : \mathsf{t}_1 :: \mathbf{Type} \\ \langle \mathsf{Names}, \mathsf{TH}, \mathsf{Intfs}, \mathsf{A}, \mathsf{locals}\rangle :: \mathbf{Context}_3 \vdash expr : \mathsf{t}_2 :: \mathbf{Type}\end{array}}{\langle \mathsf{Names}, \mathsf{TH}, \mathsf{Intfs}, \mathsf{A}, \mathsf{locals}\rangle :: \mathbf{Context}_3 \vdash stat : \underline{\mathsf{correct}} :: \mathbf{GenInfo}} \text{ if } \mathsf{t}_2 \sqsubseteq \mathsf{t}_1 \in \mathsf{TH}.$$

Remark 2.3. The sort **GenInfo** distinguishes correct programs ($\underline{\mathsf{correct}}$) from incorrect ones with respect to the static semantics. For example, if it cannot be shown using the inference rules that $\vdash stat : \underline{\mathsf{correct}} :: \mathbf{GenInfo}$, then the program fragment *stat* is not correct with respect to the static semantics. This is a standard technique used in natural semantics specifications. For example,

[2]In the theory of attribute grammars, inherited attributes correspond to semantic information in the context and synthesized attributes correspond to properties. However, one of the advantages of natural semantics specifications is that these directions of attribute computations do not need to be specified.

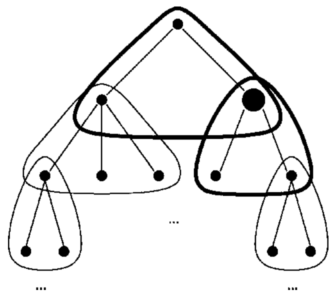

Fig. 2. Rule cover.

the static semantics of Java [Nipkow and von Oheimb 1998] uses the notation $\vdash$ *stat* : $\diamond$ for this purpose. In contrast, we use the sort **Bool** for defining auxiliary predicates (independent of the program to be analyzed) that are used in the inference rules.

A specification is *well-formed* if, in each judgment for a symbol X of the abstract syntax, the sorts of the semantic information in the context and the properties are the same. Speaking in the language of attribute grammars, this means that each symbol X has been assigned the same attributes by each judgment describing it.

2.2 Proofs in Sorted Natural Semantics

The principal task of a static analysis consists of the construction of a proof tree. A proof tree verifies that some program information is statically semantically correct. Its structure coincides with the structure of the abstract syntax tree. In this subsection, we define proof trees formally and show how they can be computed by solving an equivalent residuation problem.

A *rule cover* (Figure 2) is a mapping from the nodes of the abstract syntax tree to instances of inference rules of the specification. A rule cover[3] maps a node X_0 with successors $X_1, \ldots, X_n$ to an instance of inference rule R only if the conclusion of R is a judgment for X_0 and if R has n assumptions which are judgments for $X_1, \ldots, X_n$. Nodes without successors are mapped to axioms. We assume that the instances of the inference rules contained in the rule cover do not have common variables. A rule cover assigns one judgment to the root node and two judgments to each other node describing their static semantic information. Hence, the static semantic information of each node in the abstract syntax tree is described by two terms, except for the root node with only one term per semantic information. Note that leaves are described by two terms where one stems from an axiom.

A *proof tree* B for a given abstract syntax tree is a rule cover together with a substitution σ whereby σ satisfies the following two requirements: σ instantiates the free variables in the side conditions of the inference rules of

[3]This is an abuse of notation. Each node in an abstract syntax tree corresponds to a grammar symbol. This could be viewed as a type of an abstract syntax tree node. The notation here uses for simplicity the grammar symbols instead of the concrete nodes.

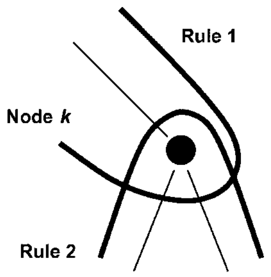

Fig. 3. Constraint-generating system.

the rule cover such that their truth values can be computed straightforwardly and such that they hold. Furthermore, for each semantic information of each node in the abstract syntax tree, it must be possible to prove the equality of its two terms using the Horn clauses. Note that this common term, also called *attribute*, may contain variables. This makes sense for example when doing separate analysis, for example, separate compilation, where only program fragments are checked. As an example, assume a programming language with procedures. Furthermore assume that such procedures shall be compiled separately. The procedure bodies may use global variables whose declarations are not known at compile time. Then we would have semantic information employing the type information of these global variables in form of logical variables which cannot be instantiated only in the context of the procedure body.

The abstract syntax tree together with all attributes is called an *attributed syntax tree*. The assignment of attributes to an abstract syntax tree B is called an *attribution of* B. A proof tree B is a most general proof tree if, for every other proof tree B', there exists a substitution π that maps the attribution of B to the attribution of B'.

To compute a proof tree, we consider all possible rule covers of the abstract syntax tree. For each such rule cover, we need to compute the static semantic information and check the side conditions of the rules. A rule cover assigns two judgments to each node in the abstract syntax tree (except the root node which has only one judgment assigned to it). Each of these two judgments defines two terms for each semantic information of each node. To compute the semantic information, we need to unify all these pairs of terms while also computing the defined functions contained in them. The combination of unification and evaluation steps where the evaluation steps are always applied to ground terms is called *residuation* Hanus [1994]. Thus, we can regard static analysis as a residuation problem. The question is how such residuation problems can be solved.

Therefore, we consider a sorted natural semantics specification as a *constraint-generating system* (Figure 3). For each rule cover, we generate automatically *constraints* whose solution is also a valid attribution of the abstract syntax tree. We introduce this process of generating constraints by an example before dealing with the general case. Let us assume an arbitrary but fixed

rule cover and a node k of the abstract syntax tree. This node k is influenced by two rules which we call rule 1 and rule 2. Assume furthermore that rule 1 and rule 2 define the following two judgments for node k, respectively: $\Gamma_1 :: \mathbf{Context} \vdash k : \mathrm{t}_1 :: \mathbf{Type}$ and $\Gamma_2 :: \mathbf{Context} \vdash k : \mathrm{t}_2 :: \mathbf{Type}$.
Then these are the constraints for k to be solved during the static analysis:

$$\mathbf{Context}_k = \Gamma_1,\ \mathbf{Context}_k = \Gamma_2,\ \mathbf{Type}_k = \mathrm{t}_1,\ \text{and } \mathbf{Type}_k = \mathrm{t}_2,$$

where $\mathbf{Context}_k$ and $\mathbf{Type}_k$ are new variables. The constraints specify that the semantic information of sort **Context** for node k must be equal to Γ_1 and Γ_2 and that the semantic information of sort **Type** for node k must be equal to t_1 and t_2.

In general, the constraints are defined as follows. We assume that the nodes in the abstract syntax tree have unique names. Using these unique names, we define sort variables for the nodes. If a judgment for a node k has the form $\Gamma_1 :: \mathbf{S}_1, \ldots, \Gamma_l :: \mathbf{S}_l \vdash k : \mathrm{t}_1 :: \mathbf{S}_{l+1}, \ldots, \mathrm{t}_n :: \mathbf{S}_{l+n}$, then its sort variables are $(\mathbf{S}_1)_k, \ldots, (\mathbf{S}_l)_k, (\mathbf{S}_{l+1})_k, \ldots, (\mathbf{S}_{l+n})_k$. Sort variables are place holders for the values of the semantic information of the corresponding node. Constraints describe requirements on these sort variables. The rules in a rule cover specify judgments for the nodes of the abstract syntax tree and, hence, terms for the values of the sort variables. If $\Gamma_1 :: \mathbf{S}_1, \ldots, \Gamma_l :: \mathbf{S}_l \vdash k : \mathrm{t}_1 :: \mathbf{S}_{l+1}, \ldots, \mathrm{t}_n :: \mathbf{S}_{l+n}$ is a judgment for a node k, then the constraints on the sort variables of k resulting from this judgment are $(\mathbf{S}_1)_k = \Gamma_1, \ldots, (\mathbf{S}_l)_k = \Gamma_l, (\mathbf{S}_{l+1})_k = \mathrm{t}_1, \ldots, (\mathbf{S}_{l+n})_k = \mathrm{t}_n$. In a rule cover, each node (besides the root node) is described by two judgments. The constraints for a node are the constraints induced by its two judgments (or by its judgment in case of the root node, respectively). The constraints of an entire program consist of the constraints of all its nodes. So to compute a proof tree, we need to solve the constraints induced by a suitable rule cover.

3. GENERATING SEMANTIC ANALYSIS

Sorted natural semantics is a general framework to specify static program analyses. As a major example, we investigate the semantic analysis in the frontends of compilers and show how it can be expressed within sorted natural semantics. The static semantics specifies and checks context-sensitive properties by computing attributes for the nodes in abstract syntax trees and by checking local consistency requirements for them. Here we concentrate on imperative and object-oriented programming languages but other programming language paradigms may be treated as well. We characterize programs of a programming language by the existence of a unique most general proof tree with respect to a sorted natural semantics specification defining the static semantics of the programming language. In this section, we show how such proof trees can be computed with a basic algorithm. First, in Section 3.1, we discuss three major requirements which we expect to hold for reasonable static semantics specifications of programming languages. We present the basic algorithm and prove its correctness in Section 3.2. Finally, in Section 3.3, we show how the basic algorithm works by analyzing small programs written in the example language

DEMO from Appendix B. As a larger demonstration, the specification of the static semantics of Mini-Java is given and explained in Appendix A.

3.1 Requirements of the Semantic Analysis of Programming Languages

The principal task of the semantic analysis is the transport of information from one program point to another. For example, the information from a variable declaration needs to be known at program points where this variable is used in order to do some correctness checks. The attribution of programs is assembled from such transported information or information inferred from it. There may be programs whose syntax is consistent with the abstract syntax of the programming language but that do not satisfy the static semantic conditions. By definition, these programs do not belong to the programming language and may have more than one most general proof tree (representing ambiguous information) or no proof tree at all (representing inconsistent information). Hence, we can state our first requirement to be satisfied by sorted natural semantics specifications of successful semantic analyses:

First requirement: Specifications of sorted natural semantics for semantic analysis are written such that, for each program of the defined programming language, there exists a unique most general proof tree upto renaming and AC1-equivalence.

Thus, if during semantic analysis, it is detected that there is more than one most general proof tree, then the semantic analysis is not successful and the analyzed program is rejected.

During semantic analysis, it is decidable whether an attribution is valid, that is, whether the side conditions of the inference rules are fulfilled. In general, the side conditions may contain variables, turning the test of validity into a nondecidable task. Nevertheless, the side conditions specifying the correctness checks must be decided after the computation of all attributes. In particular, for each program point, it must be decided whether the side conditions of its associated inference rule are evaluated to true or false. The second requirement is a sufficient condition for this property:

Second requirement: The free variables in the side conditions of the inference rules are completely instantiated in a most general proof tree.

Note that this requirement still allows for variables in the attributes of a most general proof tree. Only variables in the side conditions are excluded. Again, the second requirement is only necessary for successful semantic analyses. If it is detected during semantic analysis that, after the instantiation of the inference rules in a most general proof tree, there are side conditions with free variables, then the analyzed program is rejected.

The third assumption concerns the data structures, that is, the sorts and their functions and predicates. It is our goal to have declarative specifications, which means also declarative descriptions of the data structures. Therefore we define them with sort equations and Horn clauses. Since for semantic analysis the attribution must be unique, the defined function values must evaluate to unique constructor terms, if their arguments do not contain free variables. This leads us to the third requirement which suffices for the uniqueness of the evaluation of defined functions:

Third requirement: The Horn clauses define a ground-confluent conditional term-rewrite system. The most general proof tree can be computed without evaluating terms containing free variables.

This requirement states that defined functions must be unambigously defined. Nevertheless, as a consequence, the defined functions can be implemented differently. From a practical point of view, this is desirable and follows the tradition of other compiler generator tools [Eli n.d.; Kastens et al. 1982]. Therefore, we can assume that there are correct implementations for the sorts and their functions and predicates at our disposal. Observe that the third requirement still allows for attributions containing variables. Only subterms of the attributes which contain defined functions need to be variable-free.

It is undecidable if a given sorted natural semantics specification conforms to the above three requirements. Nevertheless when specifying the static semantics of a programming language, one might be able to prove them for a specific specification. In the following subsection, we answer the question how the most general proof tree can be computed for sorted natural semantics specifications satisfying the above requirements.

3.2 Basic Algorithm

The basic algorithm considers all possible rule covers and checks which of them can be completed to a unique proof tree. For now, we assume that all rule covers are tested separately and discuss improvements of this strategy at the end of this subsection. For each rule cover, the basic algorithm generates the corresponding constraints as defined in Section 2.2. We can think of the constraints as pairs of equations ($S = t_1$, $S = t_2$) where S is a sort variable and t_1 and t_2 are the terms constraining the value of S. (Remember that each node except for the root node is influenced by two judgments. If S is a sort variable of the root node, then without loss of generality t_1 and t_2 are syntactically identical.) To compute the proof tree and its attribution, the two terms t_1 and t_2 must be unified while simultaneously evaluating the contained defined functions. In the unification process, the sort variables are not replaced because they are merely a coding for the node and its attribute value which is represented by t_1 and t_2. If it is possible to evaluate and unify all pairs of constraints such that the side conditions of the inference rules in the rule cover are also fulfilled, then a proof tree is found.

The basic algorithm solves pairs of constraints by residuation [Hanus 1994]. Residuation nondeterministically performs unification and computation steps. In a *unification step*, a pair of terms t_1 and t_2 which are required to be equal is unified by some substitution σ. This unifier σ is also applied to all other constraints of the program. In a *computation step*, a ground subterm

$f(t_1, \ldots, t_n)$ of a term t where f is a defined function with arguments t_i is replaced by an equivalent ground constructor term. A particular run of the basic algorithm can be described by a *residuation sequence* which consists of a sequence of unification and computation steps.

Since we assume that implementations for the defined functions are available, the computation steps do not pose any problems, in contrast to the unification steps: Terms may be built with the constructor functions $\emptyset, \cup, \{\cdot\}$ for sets. Hence, in general, AC1-unifications would be required whereby a minimal complete set of AC1-unifiers may be doubly exponential in the size of the given AC1-problem. In the semantic analysis, it makes no sense to compute all these unifiers. A program is correct only if its attribution can be determined uniquely. Therefore we use a restricted version of AC1-unification by extending the Herbrand-Robinson unification algorithm [Robinson 1965].

Extending Herbrand-Robinson to restricted AC1-unification: The Herbrand-Robinson unification algorithm computes a most general unifier of two terms under which they are syntactically equal. This most general unifier is unique up to renaming of variables, that is, if σ_1 and σ_2 are both most general unifiers, then there exists a variable renaming π such that $\sigma_1 = \pi \circ \sigma_2$. The unification algorithm assumes that a term is either a variable, a constant, or a structured term. If the two input terms to be unified are not structured, then the unifier can be determined directly, if it exists. Otherwise, all corresponding subterms of the input terms have to be unified recursively. The overall unifier is the composition of the recursively computed unifiers. The order in which subterms are considered does not matter. We extend this unification algorithm in the following way: if two terms s and t being sets are to be unified, then for reasons of efficiency this is allowed in our approach only if their unifier can be determined uniquely (up to renaming of variables). We distinguish two cases:

—If $s = \emptyset$ or $t = \emptyset$, then the unique unifier can be easily determined if it exists.

—If $s = S \uplus \{s_1, \ldots, s_m\}$ and $t = T \uplus \{t_1, \ldots, t_n\}$, n not necessarily equal[4] to m, then this unification problem can be reduced to the problem of unifying $\sigma(s')$ and $\sigma(t')$ with $s' = S \uplus \{s_1, \ldots, s_{i-1}, s_{i+1}, \ldots, s_m\}$ and $t' = T \uplus \{t_1, \ldots, t_{j-1}, t_{j+1}, \ldots, t_n\}$ if $\mathsf{unique}(s_i) = \mathsf{unique}(t_j)$, $\mathsf{unique}(s_i)$ and $\mathsf{unique}(t_j)$ are variable-free, and σ is the most general unifier of s_i and t_j. The unification algorithm can proceed with the smaller problem of unifying $\sigma(s')$ and $\sigma(t')$. $\mathsf{unique}(s_i) = \mathsf{unique}(t_j)$ means that $\mathsf{unique}(s_i)$ and $\mathsf{unique}(t_j)$ are evaluated into ground terms which are equal up to AC1-equivalence. Since they are variable-free, this test is decidable. Remember that the unique-function returns the parts of an element with which it can be identified uniquely; see Section 2.

The requirement that $\mathsf{unique}(s_i)$ and $\mathsf{unique}(t_j)$ be variable-free is just for reasons of efficiency. Clearly, this restricted AC1-unification is not complete

[4]Sets S and T may have different sizes.

because it can happen that no unifier is found even though one might exist. As a simple example, consider $s = S \uplus \{s_1\}$ and $t = T \uplus \{t_1\}$ with $s_1 \neq t_1$. S and T are unifiable by instantiating S with $\{t_1\}$ and T with $\{s_1\}$. Nevertheless, the above extension of the Herbrand-Robinson algorithm will reject them because this unifier is not unique. Any other unifier which replaces S by $\{t_1\} \cup M$ and T by $\{s_1\} \cup M$ for some set M will also suffice.

LEMMA 3.1. *If σ_1 and σ_2 are two AC1 unifiers which can be computed by the restricted AC1-unification algorithm, then there is a renaming π such that σ_1 and $\pi \circ \sigma_2$ are equal up tp AC1-equivalence.*

PROOF. The correctness of the restricted AC1-unification algorithm follows directly from the correctness of the Herbrand-Robinson algorithm: a unifier is computed only if it can be determined uniquely up to variable renaming. In particular, sets are only unified if the correspondence between their elements is unique. Thereby the ordering of the elements in a set does not matter. Besides these modifications, the unifier is computed according to the Herbrand-Robinson algorithm. Hence the unifier is unique up to renaming. □

THEOREM 3.2. *Each residuation sequence used in a run of the basic algorithm yields the same result modulo variable renaming.*

PROOF. The basic algorithm computes unique (up to variable renaming and AC1-equivalence) unifiers. Furthermore, the order in which unification and computation steps are performed does not matter. Computation steps affect only variable-free terms which are not modified by the unification steps. Even though performing certain computation steps may be necessary in order to enable some subsequent unification steps, the final solution is nevertheless the same up to AC1-variants. Hence, each residuation sequence leads to the same result up to renaming of variables and AC1-equivalence. □

The *size* of a term t, denoted by $|t|$, is recursively defined by $|c| = 1$ for a constant or variable c and $|f(t_1, \ldots, t_k)| = |t_1| + \cdots + |t_k|$. The *size of a constraint* $\mathbf{S} = t$ is $|t|+1$ and the *size of a constraint set* is the sum of the size of its elements.

THEOREM 3.3. *Given the constraints of a single rule cover, the time complexity to solve them with the basic algorithm is $\mathcal{O}(n^2 \log n)$ where n is the size of the constraint set, provided that every defined function call $f(t_1, \ldots, t_k)$ can be evaluated in time $\mathcal{O}(|f(t_1, \ldots, t_k)|)$.*

PROOF. The basic algorithm performs standard unification steps interleaved with the evaluation of defined functions and with the restricted AC1-unification. The standard unification can be done in linear time using sharing of common subexpressions [Paterson and Wegman 1978]. Whenever this standard unification stops, it is necessary to either evaluate defined functions or to unify sets. Hence, the overall time complexity is the sum of the time complexity for the standard unification, of the time complexity for the evaluation of the defined functions, and of the time complexity to find all pairs of elements of sets to be unified (those whose unique-parts are identical). In the rest of this proof, we show that all defined functions can be evaluated in time $\mathcal{O}(n^2)$ and that the pairs of set elements to be unified can be found in time $\mathcal{O}(n^2 \log n)$. From these results it follows that the time complexity of the basic algorithm is $\mathcal{O}(n^2 \log n)$.

To evaluate subterms starting with defined function symbols, we proceed as follows: we keep two lists with references to subterms that start with a defined function symbol. In the first list, we keep all variable-free subterms starting with a defined function symbol while the second list contains all subterms starting with a defined function symbol which still contain variables. Moreover, we define a reference from each variable to the subterms in the second list in which this variable is contained. Whenever a variable is substituted, either with a variable-free term or with a term containing other variables, the two lists are updated: in the first case, the substituted, variable-free subterm is transferred from the second into the first list. In the second case, the references from variables to the terms in the second list are updated appropriately. Both kinds of updates can be done in time $\mathcal{O}(n)$. The overall time complexity to evaluate the subterms starting with a defined function symbol (as soon as they are in the first list) is therefore $\mathcal{O}(n^2)$.

The time complexity to find all pairs of elements of sets to be unified is $\mathcal{O}(n^2 \log n)$: the worst case appears whenever two sets $\{s_1, \ldots, s_n\}$ and $\{t_1, \ldots, t_n\}$, both of size $\Theta(n)$, are to be unified. Then we need to find a pair (s_i, t_j) such that $\mathsf{unique}(s_i)$ and $\mathsf{unique}(t_j)$ are identical. Therefore, we sort the elements of both sets lexicographically, each set separately. Then we take the first element of the first set and try to find, by binary search, a corresponding element in the second set. If this search is successful, then we are done. Otherwise, we proceed with the second (third, ...) element of the first set until we find a corresponding element in the second set. The cost for a single binary search is $\mathcal{O}(\log n)$; hence for the overall search it is $\mathcal{O}(n \log n)$. In the worst case, we need to repeat such a search n times, giving us a time complexity of $\mathcal{O}(n^2 \log n)$. □

Up to now, we have assumed that one rule cover is checked after the other. An easy calculation shows that if there are alternative inference rules in a specification, then the number of possible rule covers is exponential in the size of the abstract syntax tree. It is too costly to consider these many rule covers separately. Therefore we carry out the following efficiency enhancement: First, we consider only the constraints which are the same for all rule covers, that is, the constraints stemming from inference rules which do not have alternatives. We evaluate and unify them as far as possible. Then we sort out dynamically as many of the alternative inference rules and, in turn, rule covers as possible by either showing that their side conditions are not fulfilled or by proving that the unification fails. This is in fact a clear improvement—experiments in our prototype implementation indicate that, in most semantic analyses, only linearly many (instead of exponentially many) rule covers need to be checked (cf. Section 7).

3.3 Example Semantic Analyses

In this subsection, we demonstrate the application of the basic algorithm to programs of the language DEMO defined in Appendix B. DEMO is a very simple

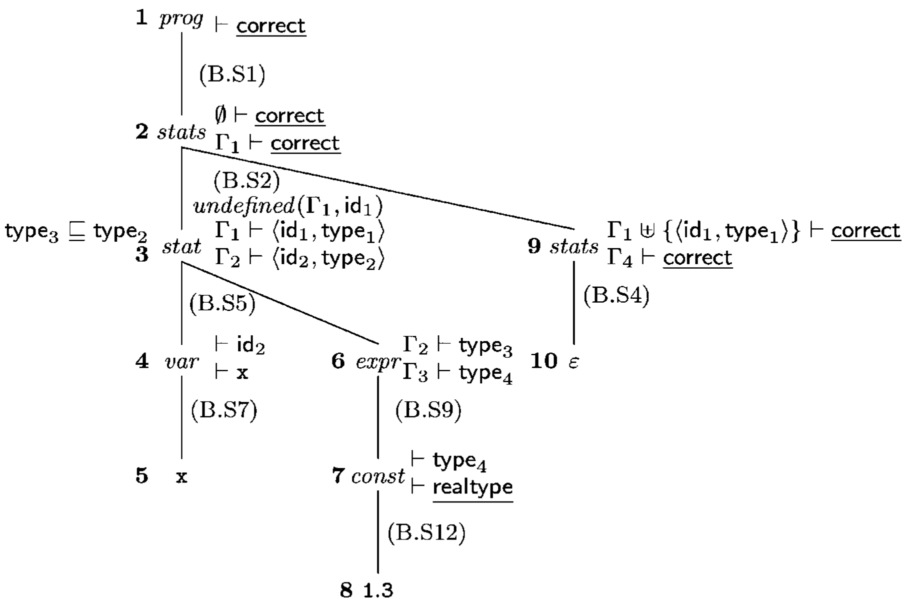

Fig. 4. Rule cover for the program in Example 3.4.

imperative language whose programs consist of a list of assignments and variable declarations. Declarations of variables do not need to occur before their use. However, every variable used in a DEMO-program must be declared. It is possible to declare a variable twice. In this case, the declaration, that is, the type of the variable, must be identical. DEMO has two types, integers and reals. Integers are coercible to reals but not vice versa. Example 3.4 shows that the attribution is ambiguous if a variable is used but not declared. This is erroneously fixed in Example 3.5. This example demonstrates that there exists no proof if a program is ill-typed. Example 3.6 shows the proof if the erroneous type declaration is fixed. Here, the variable is declared after its use.

Example 3.4. Consider the DEMO-program `x:=1.3`. It contains the undeclared variable x. Figure 4 shows the abstract syntax tree according to the syntax of DEMO (cf. Figure 31). The names of the nodes correspond to the nonterminals. A node has up to five annotations. The annotation below a node is the applied inference rule. At the upper left corner, we annotate side conditions stemming from an inference rule. Alternatively, this may also be denoted below the name of the inference rule. At the lower left corner of a node we annotate its depth-first order number. The upper right corner and lower right corner of a node are annotated with the judgments stemming from the applied inference rules at its parent and the node, respectively. For better readability, we omit the sorts. Thus, the annotations represent a complete rule cover constructed by the basic algorithm. The following constraints are derived from

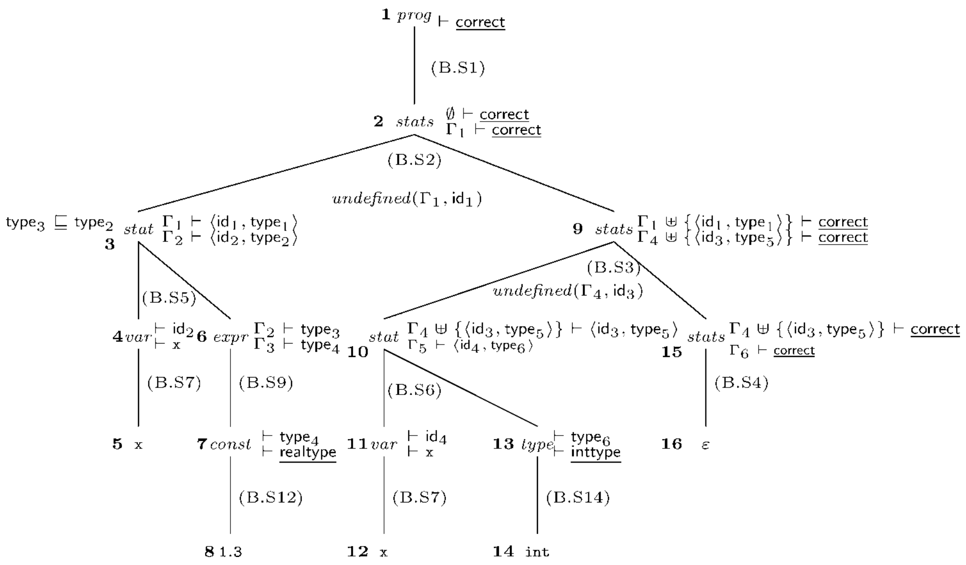

Fig. 5. A Rule cover for the program in Example 3.5.

Figure 4:

$$
\begin{array}{rlrlrl}
\mathbf{Context}_2 &= \emptyset, & \mathbf{Id}_4 &= \mathsf{id}_2, & \mathbf{Type}_6 &= \mathsf{type}_4,\\
\mathbf{Context}_2 &= \Gamma_1, & \mathbf{Id}_4 &= \mathtt{x}, & \mathbf{Type}_7 &= \mathsf{type}_4,\\
\mathbf{Context}_3 &= \Gamma_1, & \mathbf{Context}_6 &= \Gamma_2, & \mathbf{Type}_7 &= \underline{\mathsf{realtype}},\\
\mathbf{Context}_3 &= \Gamma_2, & \mathbf{Context}_6 &= \Gamma_3, & \mathbf{Context}_9 &= \Gamma_1 \uplus \{\langle \mathsf{id}_1, \mathsf{type}_1\rangle\},\\
\mathbf{Decl}_3 &= \langle \mathsf{id}_1, \mathsf{type}_1\rangle, & \mathbf{Type}_6 &= \mathsf{type}_3, & \mathbf{Context}_9 &= \Gamma_4.\\
\mathbf{Decl}_3 &= \langle \mathsf{id}_2, \mathsf{type}_2\rangle, & & & &
\end{array}
$$

A solution algorithm will stop with the following solutions:

$$
\begin{array}{rlrlrl}
\mathbf{Context}_2 &= \emptyset, & \mathbf{Id}_4 &= \mathtt{x}, & \mathbf{Type}_7 &= \underline{\mathsf{realtype}},\\
\mathbf{Context}_3 &= \emptyset, & \mathbf{Context}_6 &= \emptyset, & \mathbf{Context}_9 &= \{\langle \mathtt{x}, \mathsf{type}_1\rangle\}.\\
\mathbf{Decl}_3 &= \langle \mathtt{x}, \mathsf{type}_1\rangle, & \mathbf{Type}_6 &= \underline{\mathsf{realtype}}, & &
\end{array}
$$

The side condition $\mathsf{type}_3 \sqsubseteq \mathsf{type}_2$ becomes $\underline{\mathsf{realtype}} \sqsubseteq \mathsf{type}_1$. Hence, requirement 2 is violated, implying that the static semantics, that is, the context-sensitive properties of the program, are not correct. Due to the second requirement, every successful semantic analysis would not end with a side condition containing free variables. Hence, the context-sensitive properties of the program are not correct.

Example 3.5. The DEMO-program `x := 1.3; x:int` is not correctly typed. Figure 5 shows an annotated abstract syntax tree of this program computed by the basic algorithm.

The basic algorithm derives from the rule cover in Figure 5 the same constraints for nodes 1–8 as in Example 3.4 and additionally the following constraints:

$$\begin{array}{lll}
\mathbf{Context}_9 = \Gamma_1 \uplus \{\langle \mathsf{id}_1, \mathsf{type}_1\rangle\}, & \mathbf{Decl}_{10} = \langle \mathsf{id}_3, \mathsf{type}_5\rangle, & \mathbf{Type}_{13} = \mathsf{type}_6, \\
\mathbf{Context}_9 = \Gamma_4 \uplus \{\langle \mathsf{id}_3, \mathsf{type}_5\rangle\}, & \mathbf{Decl}_{10} = \langle \mathsf{id}_4, \mathsf{type}_6\rangle, & \mathbf{Type}_{13} = \underline{\mathsf{inttype}}, \\
\mathbf{Context}_{10} = \Gamma_4 \uplus \{\langle \mathsf{id}_3, \mathsf{type}_5\rangle\}, & \mathbf{Id}_{11} = \mathsf{id}_4, & \mathbf{Context}_{15} = \Gamma_4 \uplus \{\langle \mathsf{id}_3, \mathsf{type}_5\rangle\}, \\
\mathbf{Context}_{10} = \Gamma_5, & \mathbf{Id}_{11} = \mathtt{x}, & \mathbf{Context}_{15} = \Gamma_6.
\end{array}$$

This constraint system has the solution

$$\begin{array}{lll}
\mathbf{Context}_1 = \emptyset, & \mathbf{Type}_6 = \underline{\mathsf{realtype}}, & \mathbf{Decl}_{10} = \langle \mathtt{x}, \underline{\mathsf{inttype}}\rangle, \\
\mathbf{Context}_2 = \emptyset, & \mathbf{Type}_7 = \underline{\mathsf{realtype}}, & \mathbf{Id}_{11} = \mathtt{x}, \\
\mathbf{Decl}_3 = \langle \mathtt{x}, \underline{\mathsf{inttype}}\rangle, & \mathbf{Context}_9 = \{\langle \mathtt{x}, \underline{\mathsf{inttype}}\rangle\}, & \mathbf{Type}_{13} = \underline{\mathsf{inttype}}, \\
\mathbf{Id}_4 = \mathtt{x}, & \mathbf{Context}_{10} = \{\langle \mathtt{x}, \underline{\mathsf{inttype}}\rangle\}, & \mathbf{Context}_{15} = \{\langle \mathtt{x}, \underline{\mathsf{inttype}}\rangle\}. \\
\mathbf{Context}_6 = \emptyset, & &
\end{array}$$

Using this solution, the side condition $\mathsf{type}_3 \sqsubseteq \mathsf{type}_2$becomes $\underline{\mathsf{realtype}} \sqsubseteq \underline{\mathsf{inttype}}$. Thus, this side condition fails. Therefore the rule cover cannot be completed to a proof.

There are two alternatives for the rule cover: at node 2, inference rule (B.S3) can be applied instead of (B.S2). At node 9, inference rule (B.S2) can be applied instead of (B.S3).

Consider first the former case. The constraint $\mathbf{Context}_2 = \Gamma_1$ is replaced by $\mathbf{Context}_2 = \Gamma_1 \uplus \{\langle \mathsf{id}_1, \mathsf{type}_1\rangle\}$. Hence, the basic algorithm tries to unify $\emptyset$ and $\Gamma_1 \uplus \{\langle \mathsf{id}_1, \mathsf{type}_1\rangle\}$. This unification fails. Therefore, the rule cover cannot be completed to a proof.

In the latter case, the constraints $\mathbf{Context}_9 = \Gamma_4 \uplus \{\langle \mathsf{id}_3, \mathsf{type}_5\rangle\}$ and $\mathbf{Context}_{10} = \Gamma_4 \uplus \{\langle \mathsf{id}_3, \mathsf{type}_5\rangle\}$ are replaced by constraints $\mathbf{Context}_9 = \Gamma_4$ and $\mathbf{Context}_{10} = \Gamma_4$. The basic algorithm unifies Γ_4 and $\{\langle \mathtt{x}, \mathsf{type}_1\rangle\}$. Furthermore, it unifies $\langle \mathtt{x}, \mathsf{inttype}\rangle$ and $\langle \mathsf{id}_3, \mathsf{type}_5\rangle$. If the resulting substitutions are applied to the side condition of inference rule (B.S2), we have to evaluate $\mathit{undefined}(\{\langle \mathtt{x}, \mathsf{type}_1\rangle\}, \mathtt{x})$ which fails. Hence, the rule cover cannot be completed to a proof tree either. Furthermore, the solution for the resulting constraint system contains the free variable type_1.

Example 3.6. The DEMO-program `x:=1; x:real` declares the variable `x` after its use. Figure 6 shows the rule cover for this program. The constraints derived from this program are the same as those derived from the program in Example 3.5 except that the constraints $\mathbf{Type}_7 = \underline{\mathsf{realtype}}$ and $\mathbf{Type}_{13} = \underline{\mathsf{inttype}}$ are replaced by the constraints $\mathbf{Type}_7 = \underline{\mathsf{inttype}}$ and $\mathbf{Type}_{13} = \underline{\mathsf{realtype}}$, respectively. The constraint system has the solution:

$$\begin{array}{lll}
\mathbf{Context}_1 = \emptyset, & \mathbf{Type}_6 = \underline{\mathsf{inttype}}, & \mathbf{Decl}_{10} = \langle \mathtt{x}, \underline{\mathsf{realtype}}\rangle, \\
\mathbf{Context}_2 = \emptyset, & \mathbf{Type}_7 = \underline{\mathsf{inttype}}, & \mathbf{Id}_{11} = \mathtt{x}, \\
\mathbf{Decl}_3 = \langle \mathtt{x}, \underline{\mathsf{realtype}}\rangle, & \mathbf{Context}_9 = \{\langle \mathtt{x}, \underline{\mathsf{realtype}}\rangle\}, & \mathbf{type}_{13} = \underline{\mathsf{realtype}}, \\
\mathbf{Id}_4 = \mathtt{x}, & \mathbf{Context}_{10} = \{\langle \mathtt{x}, \underline{\mathsf{realtype}}\rangle\}, & \mathbf{Context}_{15} = \{\langle \mathtt{x}, \underline{\mathsf{realtype}}\rangle\}. \\
\mathbf{Context}_6 = \emptyset, & &
\end{array}$$

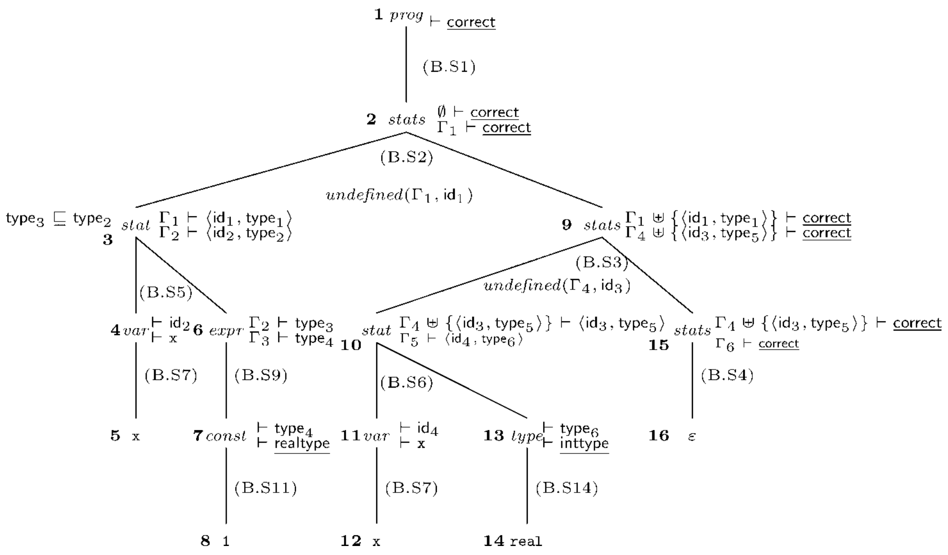
1 prog ⊢ correct
(B.S1)
2 stats ∅ ⊢ correct Γ1 ⊢ correct
(B.S2)
undefined(Γ1, id1)
type3 ⊑ type2 3 stat Γ1 ⊢ ⟨id1, type1⟩ Γ2 ⊢ ⟨id2, type2⟩
9 stats Γ1 ⊎ {⟨id1, type1⟩} ⊢ correct Γ4 ⊎ {⟨id3, type5⟩} ⊢ correct
(B.S5)
(B.S3)
undefined(Γ4, id3)
4 var ⊢ id2 ⊢ x
6 expr Γ2 ⊢ type3 Γ3 ⊢ type4
10 stat Γ4 ⊎ {⟨id3, type5⟩} ⊢ ⟨id3, type5⟩ Γ5 ⊢ ⟨id4, type6⟩
15 stats Γ4 ⊎ {⟨id3, type5⟩} ⊢ correct Γ6 ⊢ correct
(B.S7)
(B.S9)
(B.S6)
(B.S4)
5 x
7 const ⊢ type4 ⊢ realtype
11 var ⊢ id4 ⊢ x
13 type ⊢ type6 ⊢ inttype
16 ε
(B.S11)
(B.S7)
(B.S14)
8 1
12 x
14 real

Fig. 6. Rule cover for the program in Example 3.6.

Then, the following side conditions have to be evaluated:

—*undefined*(∅, x) stemming from node 2,
—inttype ⊑ realtype stemming from node 3,
—*undefined*(∅, x) stemming from node 9.

All these side conditions evaluate to <u>true</u>. Hence, the rule cover in Figure 6 can be completed to a proof tree.

The same arguments as in Example 3.5 show that there is no other proof tree. Hence, the basic algorithm succeeds.

4. SOLUTION STRATEGIES

If the basic algorithm succeeds to compute a solution, then this solution is unambiguous, that is, there is no other solution provided the sorted natural semantics specification satisfies the three requirements of Section 3.1. However, since it is undecidable whether these three requirements are satisfied, it cannot be guaranteed that the basic algorithm finds such a solution for all legal programs (with respect to the static semantics). This section discusses solution strategies which are independent from the concrete program to be analyzed. A solution strategy induces one possible residuation sequence to construct a proof tree, that is, it will not find a solution in cases where the basic algorithm will not find any but, instead, might be able to improve the efficiency of the semantic analysis. Thereby, our goals are twofold: first, we want to be able to analyze a sorted natural semantics specification in advance whether a solution strategy will succeed. Second, we want to perform the semantic analysis more efficiently than with the approach in Section 3.

In contrast to the previous sections we consider here only semantic analyses for complete programs. This implies that the attributions of complete programs do not contain free variables. This holds because otherwise the attribution would be ambiguous since each ground-term can be substituted for a free variable. In contrast, in case of separate compilation, the attribution could contain free variables that are substituted when linking the units together. Section 4.1 shows that, if there is a solution to the constraints of the semantic analysis, then each solution strategy computes the same (unambiguous) solution. Section 4.2 presents the particular solution strategy LNS(1). It guarantees that every rule cover can be completed to a proof tree by one left-to-right traversal through the corresponding abstract syntax tree. Finally, Section 4.3 sketches some generalizations.

4.1 Invariance of the Solution Strategy

Each inference rule defines a dependency graph connecting constraints with their variables. The basic idea of a solution strategy is to assign directions to the edges of this dependency graph such that, whenever its inference rule is applied, the constraints can be solved in a topological order. A solution strategy is such a topological order. We first introduce the notion of dependency graphs, then we define the notion of (legal) direction assignments. We finally show that any solution strategy based on a legal direction assignment will compute the same solution.

Definition 4.1. Let $\mathfrak{S}$ be a sorted natural semantics specification, and R be an inference rule for production $X_0 ::= X_1 \cdots X_m$ of the form

$$\frac{\begin{array}{c} t_1^1 :: \mathbf{S}_1^1, \ldots, t_{k_1}^1 :: \mathbf{S}_{k_1}^1 \vdash X_1 : t_{k_1+1}^1 :: \mathbf{S}_{k_1+1}^1, \ldots, t_{n_1}^1 :: \mathbf{S}_{n_1}^1 \\ \vdots \\ t_1^m :: \mathbf{S}_1^m, \ldots, t_{k_m}^m :: \mathbf{S}_{k_m}^m \vdash X_m : t_{k_m+1}^m :: \mathbf{S}_{k_m+1}^m, \ldots, t_{n_m}^m :: \mathbf{S}_{n_m}^m \end{array}}{t_1^0 :: \mathbf{S}_1^0, \ldots, t_{k_0}^0 :: \mathbf{S}_{k_0}^0 \vdash X_0 : t_{k_0+1}^0 :: \mathbf{S}_{k_0+1}^0, \ldots, t_{n_0}^0 :: \mathbf{S}_{n_0}^0} \quad \text{if } \varphi,$$

where $\mathbf{S}_i^j$ are sort symbols and t_i^j are terms. The *dependency graph of R* is a bipartite graph $DP(R) = (V, C, E)$ with the two node sets V and C and the set E of edges where V is the set of all sort variables and logical variables of R, $C = \{\mathbf{S}_i^j = t_i^j : 0 \leq j \leq m, 1 \leq i \leq n_j\}$ is the set of all constraints implied by R, and $E = \{\{\mathbf{S} = t, \mathsf{x}\} : \mathbf{S} = t \in C \wedge \mathsf{x} \in V(t) \cup \{\mathbf{S}\}\}$ is the set of edges such that there is an edge between a variable and a constraint if and only if this variable is contained in the constraint. If t contains a subterm of the form $t' = f(t_1, \ldots, t_k)$ where f is a defined function, then the edges $\{\mathbf{S} = t, \mathsf{x}\}$ for $\mathsf{x} \in V(t')$ are directed from the variable to the constraint. All other edges are undirected. Let p be a program and δ_p be a rule cover for p. The *dependency graph $DP(\delta_p)$* is the union of all dependency graphs of the inference rules in the rule cover where the sort variables from judgments of a node occur only once.

As an example, Figure 7 shows the dependency graph for the rule (B.S10)

$$\frac{\Gamma :: \mathbf{Context} \vdash expr_1 : \mathsf{type}_1 :: \mathbf{Type} \qquad \Gamma :: \mathbf{Context} \vdash expr_2 : \mathsf{type}_2 :: \mathbf{Type}}{\Gamma :: \mathbf{Context} \vdash expr_0 : \mathsf{type}_1 \sqcup \mathsf{type}_2 :: \mathbf{Type}}$$

of the example language DEMO.

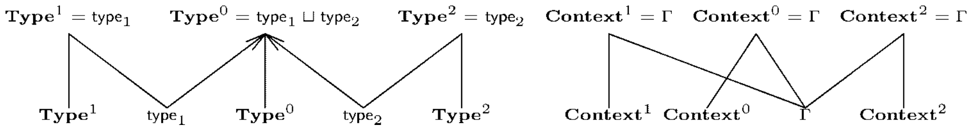

Fig. 7. A dependency graph for inference rule (B.S10).

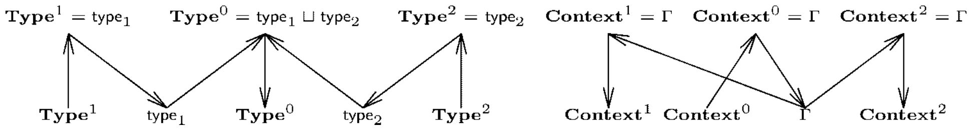

Fig. 8. The graph of Figure 7 after a direction assignment.

Remark 4.2. A dependency graph defines a data flow driven computation. Values of the sort and logical variables can flow along the edges, thereby paying attention to their directions. In this sense, the directed edges formalize the requirement that a defined function can be evaluated only if all its arguments are known. Undirected edges formalize that the flow can be in either direction in order to solve the constraint.

When defining a solution strategy, directions are assigned to the undirected edges. These directions are assigned not independently for each inference rule but need to consider the rule covers of all potential input programs. If the resulting graph is acyclic such that the constraints can be solved when their predecessors are known, then a topological order induces a solution strategy to solve the constraints.

Definition 4.3. A constraint $\mathbf{S} = t$ is *directly solvable* iff there is at most one ground-term substitution σ for $V(t) \cup \{\mathbf{S}\}$ such that $\sigma(\mathbf{S}) = \sigma(t)$. In particular, if t contains a subterm $f(t_1, \ldots, t_n)$ with a defined function f, then the arguments $t_1, \ldots, t_n$ must be ground-terms. $\mathbf{S} = t$ is *solvable with respect to a set of variables* V iff, for all substitutions σ substituting the variables $v \in V$ by ground-terms, the constraint $\sigma(\mathbf{S}) = \sigma(t)$ is directly solvable.

As an example, consider the constraints in Figure 7. None of them is directly solvable. The constraint $\mathbf{Type}^1 = \mathsf{type}_1$ is solvable with respect to $\{\mathbf{Type}_1\}$, with respect to $\{\mathsf{type}_1\}$, and with respect to $\{\mathbf{Type}_1, \mathsf{type}_1\}$. The constraint $\mathbf{Type}^0 = \mathsf{type}_1 \sqcup \mathsf{type}_2$ is solvable with respect to a set of variables X only, if $\mathsf{type}_1, \mathsf{type}_2 \in X$ because $\sqcup$ is a defined function. For example, the constraint $\mathbf{Context}_0 = \Gamma$ is solvable with respect to $\{\mathbf{Context}_0\}$, and the constraint $\mathbf{Context}_1 = \Gamma$ and $\mathbf{Context}_2 = \Gamma$ are solvable with respect to $\{\Gamma\}$.

Let $G = (V, E)$ be a graph with directed and undirected edges. A *direction assignment* is a bijective mapping $\alpha : E \to E'$ such that $\alpha(e) = e$ for each directed edge and $\alpha(\{u, v\}) = (u, v)$ or $\alpha(\{u, v\}) = (v, u)$ for each undirected edge. Hence, a direction assignment implicitly defines a directed graph $\alpha(G) = (V, E')$.

Figure 8 shows a graph $\alpha(G)$ stemming from a direction assignment to the dependency graph of Figure 7.

Definition 4.4. Let $\mathfrak{S}$ be a sorted natural semantics specification, p be a program, and δ_p be a rule cover for p. A direction assignment α for $DP(\delta_p)$ is *legal* iff $\alpha(DP(\delta_p))$ is acyclic, each variable has at least one predecessor in $\alpha(DP(\delta_p))$, and every constraint $\mathbf{S} = t$ is solvable with respect to its predecessors in $\alpha(DP(\delta_p))$. A *solution strategy with respect to* α is a topological order of $\alpha(DP(\delta_p))$.

Each solution strategy computes the same result:

THEOREM 4.5. *Let $\mathfrak{S}$ be a sorted natural semantics, p be a program, and α be a legal direction assignment for $DP(\delta_p)$. Then, every solution strategy with respect to α computes the same result.*

PROOF. According to Theorem 3.2, it is sufficient to prove that every topological order of $\alpha(DP(\delta_p))$ is a valid residuation sequence. Hence, it must be shown that all computation steps replace variable-free terms. Let $v_1, \ldots, v_n$ be a topological order of $\alpha(DP(\delta_p))$. We prove by induction the following stronger claim:

> If all constraints v_j, $j < i$ have a solution, then the following two properties are satisfied: if v_i is a variable, then the value for v_i is known. If v_i is a constraint, then $\sigma(v_i)$ is directly solvable where σ is the substitution substituting the variables v_j, $j < i$, by their solution.

v_1 must be a directly solvable constraint because it has no predecessors. Suppose v_i is a variable. Then there is constraint v_k, $k < j$, which is a predecessor of v_i. This constraint is already solved and contains variable v_i. By the induction hypothesis this constraint has a ground-term solution. Hence there is a value for v_i. Suppose now v_i is a constraint. By the induction hypothesis, the values of all predecessors of v_i are known. By Definition 4.4, $\sigma(v_i)$ is directly solvable with respect to $V(v_i) \cap \{v_1, \ldots, v_{i-1}\}$. □

Theorem 4.5 is the basis for generators of efficient semantic analyses:

(1) Compute the dependency graphs for the inference rules.
(2) Compute a direction assignment α_R for each inference rule R such that, for any program and rule cover, the composed direction assignment is legal.
(3) Compute a topological order of $\alpha_R(DP(R))$ such that, for any program and rule cover, these topological orders can be composed to a solution strategy.

The generated semantic analysis solves the constraints according to the solution strategy. For the second step, it is necessary to determine whether a constraint $\mathbf{S} = t$ is solvable with respect to a set of variables. The following lemma gives a constructive sufficient criterion. Its idea is to formalize sufficient criteria for the solvability of constraints $\mathbf{S} = t$ with a sort variable $\mathbf{S}$ and term t. If t is a variable, then this criterion is rather simple. We need to know at least the value for one of the two variables $\mathbf{S}$ and t. If both values are known, then solving the constraint becomes a simple consistency check whether the values on the left-hand side and the right-hand side of the constraint are the same. If t is a term starting with a defined function symbol, then we need to know the values for all variables contained in t. This requirement is contained in the general assertion that a constraint $\mathbf{S} = t$ is solvable with respect to X if $V(t) \subseteq X$. If t is a constructor term $f(t_1, \dots, t_n)$, then we either need to know the value of $\mathbf{S}$ (then its value must be the same as the value for t) or we need to know so many variables contained in t that we can solve the smaller problems $\mathbf{S}_i = t_i$ whereby the $\mathbf{S}_i$, $1 \le i \le n$, are new sort variables. This idea is also formalized in the case that $t = \{t_1, \dots, t_n\}$ is a set term. Then the subsets of t which contain all constructor term elements as well as variables may not contain more than one element since then the substitution cannot be determined uniquely, expressed with the requirement $|(A_v \cup V(A_f)) \setminus X| \le 1$. All cases which are not covered by Lemma 4.6 are classified as not solvable even though a solution might exist. Since we are interested in defining a sufficient criterion, this is legitimate.

LEMMA 4.6. *Let $\mathbf{S} = t$ be a constraint and $X \subseteq V(t) \cup \{\mathbf{S}\}$ be a set of variables. $\mathbf{S} = t$ is solvable with respect to X if $V(t) \subseteq X$ or $V(t) \not\subseteq X$, $\mathbf{S} \in X$, and one of the following conditions is satisfied:*

(1) *t is a variable.*
(2) *$t = f(t_1, \dots, t_n)$ for a constructor f and each of the constraints $\mathbf{S}_i = t_i$, $i = 1, \dots, n$, is solvable with respect to $(X \cap V(t_i)) \cup \{\mathbf{S}_i\}$ where $\mathbf{S}_1, \dots, \mathbf{S}_n$ are new sort variables.*
(3) *$t = \{t_1, \dots, t_n\}$, each of the constraints $\mathbf{S}_i = t_i$ is solvable with respect to $(X \cap V(t_i)) \cup \{\mathbf{S}_i\}$ where $\mathbf{S}_1, \dots, \mathbf{S}_n$ are new sort variables, and for all constructor functions f, $|(A_v \cup V(A_f)) \setminus X| \le 1$ where A_v is the largest subset of t consisting of variables and A_f is the largest subset of t consisting of all terms t_i of the form $f(\cdots)$.*

PROOF. If $V(t) \subseteq X$, it follows directly from the definition that $\mathbf{S} = t$ can be solved with respect to X. Hence, suppose $V(t) \not\subseteq X$ and $\mathbf{S} \in X$. The case that t is a variable follows directly from the definition.

Consider now the case $t = f(t_1, \dots, t_n)$ for a constructor f. Let σ be any ground-term substitution on variables X. If $\sigma(\mathbf{S})$ does not have the form $f(u_1, \dots, u_n)$, then $\sigma(S) = f(\sigma(t_1), \dots, \sigma(t_n))$ has no solution. Consider now the case $\sigma(\mathbf{S}) = f(u_1, \dots, u_n)$ for some ground-terms u_i. Our goal is now to solve the constraints $u_i = \sigma(t_i)$. Consider the substitution

$$\sigma_i(x) = \begin{cases} \sigma(x), & \text{if} \quad x \neq \mathbf{S}_i, \\ u_i, & \text{if} \quad x = \mathbf{S}_i. \end{cases}$$

Since σ is a ground-term substitution and u_i are ground-terms, σ_i is also a ground-term substitution. Furthermore $\sigma(t_i) = \sigma_i(t_i)$ because t_i does not contain the variable $\mathbf{S}_i$. Since each constraint $\mathbf{S}_i = t_i$ is solvable with respect to $(X \cap V(t_i)) \cup \{\mathbf{S}_i\}$, $i = 1, \dots, n$, the constraint $\sigma_i(\mathbf{S}_i) = \sigma_i(t_i)$ (equal to $u_i = \sigma(t_i)$) is directly solvable. If one of these constraints is not solvable, then the constraint $\sigma(\mathbf{S}) = f(t_1, \dots, t_n)$ has no solution. If there are two constraints $u_i = \sigma(t_i)$ and $t_j = \sigma(t_j)$, $1 \le i < j \le n$, having incompatible solutions, that is, ground substitutions τ_i, τ_j where $\tau_i(\mathrm{x}) \neq \tau_j(\mathrm{x})$, then the constraint $\sigma(\mathbf{S}) = f(t_1, \dots, t_n)$

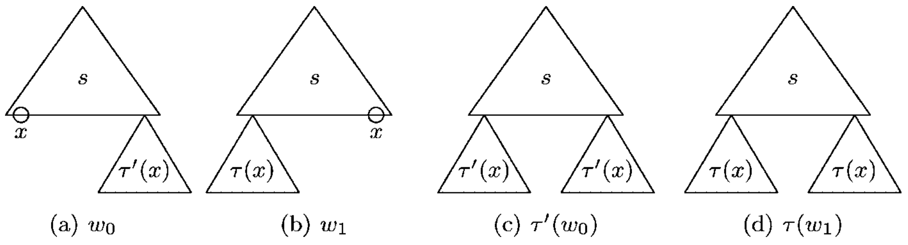

Fig. 9. Contradiction in the proof of Lemma 4.6.

has also no solution. Otherwise, it has the solution $\tau_1 \circ \cdots \circ \tau_n$ where τ_i is the solution of the constraint $u_i = \sigma(t_i)$, $i = 1, \ldots, n$. Hence, $\sigma(\mathbf{S}) = \sigma(f(t_1, \ldots, t_n))$ has at most one solution, that is, $\mathbf{S} = f(t_1, \ldots, t_n)$ is directly solvable.

Finally consider the case $t = \{t_1, \ldots, t_n\}$. We prove by contradiction that $\mathbf{S} = \{t_1, \ldots, t_n\}$ is solvable. Suppose that $\mathbf{S} = \{t_1, \ldots, t_n\}$ is not solvable. Then there is a ground-term substitution σ on variables X such that $\sigma(\mathbf{S}) = \sigma(t)$ is not directly solvable. By Definition 4.3 there are at least two different ground-term substitutions τ and τ' on $V(\sigma(t))$ such that $\sigma(\mathbf{S}) = \tau(\sigma(t))$ and $\sigma(\mathbf{S}) = \tau'(\sigma(t))$. Then, $\sigma(t) = \{u_1, \ldots, u_q\}$ for a $q \leq n$, $\sigma(\mathbf{S}) = \{s_1, \ldots, s_m\}$ for a $m \leq q$, and $u_i \in \sigma(\mathbf{S})$ for every ground-term $u_i \in \sigma(t)$ (otherwise $\sigma(\mathbf{S}) = \sigma(t)$ would have no solution). Since the constraints $\mathbf{S}_i = t_i$, $i = 1, \ldots, n$ are solvable, $\tau(\sigma(t)) = \tau'(\sigma(t))$ and $\tau \neq \tau'$, there must be a subset $W = \{w_1, \ldots, w_k\} \subseteq \sigma(t)$, $k \geq 2$, such that $\tau(W) = \tau'(W)$ and $\tau(w_i) \neq \tau'(w_i)$ for $i = 1, \ldots, n$. Without loss of generality suppose that W is minimal. Such a set W is minimal iff

$$\tau(w_1) = \tau'(w_0), \ldots, \tau(w_k) = \tau'(w_{k-1}), \tau(w_0) = \tau'(w_k). \tag{4.1}$$

Each of these terms is either a variable or starts with the same constructor symbol f. The set $V(\{w_1, \ldots, w_h\})$ contains exactly one variable x because $|(A_v \cup V(A_f)) \setminus X| \leq 1$, the terms w_i are pairwise different, and $w_i = \sigma(t_j)$ for a term $t_j \in \{t_1, \ldots, t_n\}$.

We show now that there is a contradiction for $k = 1$, that is, $\tau(w_1) = \tau'(w_0)$, $\tau(w_0) = \tau'(w_1)$, $\tau(w_0) \neq \tau'(w_0)$, and $\tau(w_1) \neq \tau'(w_1)$. The case $k > 1$ follows by a straightforward induction. Equation (4.1) implies that $\tau(\mathsf{x}) \neq \tau'(\mathsf{x})$. For simplicity we assume that each of the terms w_0, w_1 contains the variable x exactly once. The general case can be proven analogously. Two cases may occur: either w_0 is a proper subterm of w_1 (or vice versa) or they have the forms shown in Figure 9(a) and (b). Suppose w_0 is a proper subterm of w_1. Then, $\tau(w_0)$ is a proper subterm of $\tau(w_1) = \tau'(w_0)$ and $\tau'(w_0)$ is a proper subterm of $\tau'(w_1) = \tau(w_0)$. Hence $\tau(w_0) = \tau'(w_0)$. This contradicts $\tau(w_0) \neq \tau'(w_0)$. Hence w_0 is not subterm of w_1 or vice versa. According to Equation (4.1) it holds $\tau(w_0) = \tau'(w_1)$. Then, the terms w_0 and w_1 have the forms shown in Figure 9(a) and (b), respectively. This implies that the terms $\tau'(w_0)$ and $\tau(w_1)$ have the forms as shown in Figure 9(c) and (d), respectively. However, since $\tau(\mathsf{x}) \neq \tau'(\mathsf{x})$, $\tau'(w_0) \neq \tau(w_1)$ in contradiction to Equation (4.1). Hence, there cannot be two different substitutions τ and τ' which are solutions to $\sigma(\mathbf{S}) = \sigma(t)$. Thus, $\mathbf{S} = t$ is solvable with respect to X. □

Example 4.7. This example demonstrates condition (3). We show a positive and a negative example. For both examples, x and y are variables of sort **Nat**.

Consider the constraint $\mathbf{S} = \{x, \underline{succ}(x)\}$. This constraint is solvable with respect to $\{\mathbf{S}\}$: let σ be a ground substitution for $\{\mathbf{S}\}$. If $|\sigma(\mathbf{S})| \neq 2$, then there is no solution to $\sigma(\mathbf{S}) = \{x, \underline{succ}(x)\}$. If $\sigma(\mathbf{S})$ has exactly two elements, it has exactly one solution iff these elements have the form $t, \underline{succ}(t)$ for a term $t :: \mathbf{Nat}$. Thus $\sigma(\mathbf{S}) = \{x, \underline{succ}(x)\}$ is directly solvable.

It is easy to see that the constraints $\mathbf{S}_1 = x$ and $\mathbf{S}_2 =, \underline{succ}(x)$ are solvable with respect to $\mathbf{S}_1$ and $\mathbf{S}_2$, respectively. Furthermore, $A_v = V(A_{\underline{succ}}) = \{x\}$, that is, $(A_v \cup V(A_{\underline{succ}})) \setminus \{\mathbf{S}\} = \{x\}$. Hence (3) is satisfied.

Consider now the constraint $\mathbf{S} = \{x, \underline{succ}(y)\}$. This constraint is not solvable with respect to $\{\mathbf{S}\}$: if $\sigma(\mathbf{S}) = \{\underline{succ}(\underline{0}), \underline{succ}(\underline{succ}(\underline{0}))\}$, then the constraint $\sigma(\mathbf{S}) = \{x, \underline{succ}(y)\}$ has two different solutions. The first solution τ is defined by $\tau(x) = \underline{succ}(\underline{0})$ and $\tau(y) = \underline{succ}(\underline{0})$. The second solution τ' is defined by $\tau'(x) = \underline{succ}(\underline{succ}(\underline{0}))$ and $\tau'(y) = \underline{0}$.

Here, $A_v = \{x\}$ and $V(A_{\underline{succ}}) = \{y\}$. Therefore $(A_v \cup V(A_{\underline{succ}})) \setminus \{\mathbf{S}\} = \{x, y\}$. Hence (3) is violated.

The counterexample to condition (3) demonstrates the property (4.1): $\tau(\sigma(y)) = \tau'(\sigma(y)) = \underline{succ}(\underline{succ}(\underline{0}))$ and $\tau(\sigma(x) = \tau'(\sigma(y)) = \underline{succ}(\underline{0})$. The restriction $|(A_v \cup V(A_f)) \setminus X| \leq 1$ avoids these kinds of cycles. It basically states that for each constructor f there is at most one variable left for substitution if all variables in X are substituted by ground-terms.

We say, a constraint is *patently solvable* with respect to a set of variables iff the sufficient conditions of Lemma 4.6 are satisfied.

Remark 4.8. In the rest of this section, we only consider patently solvable constraints.

4.2 LNS(1)-Specifications

We now introduce a subclass of sorted natural semantics specifications that allow for the computation of a proof tree by one depth-first left-to-right traversal through the abstract syntax tree of a given program. We call these specifications LNS(1) (*Left-to-Right Natural Semantics*).

Consider an inference rule R for a production $X_0 ::= X_1 \cdots X_n$, (cf. Figure 10) and a left-to-right traversal through its nodes. The sort variables for every nonterminal $X_i, i = 0, \ldots, n$, can be partitioned into *input sorts* IS_{X_i} and *output sorts* OS_{X_i}. Before the traversal of rule R, the values for sort variables IS_{X_0} must be known. Using these values, computing values of logical variables and sort variables of rule R starts until all sort variables of IS_{X_1} have a value. This constraint solving computes alternatingly values for logical and sort variables. Then the subtree rooted at X_1 is visited. After this visit the values of the sort variables OS_{X_1} are known. Then, again, the values of logical variables and sort variables are computed alternately until the values of all sort variables in IS_{X_2} are known. This traversal repeats analogously until X_n is visited. Finally, the remaining logical variables and sort variables are computed. After R is traversed, the values of all variables of rule R must be known.

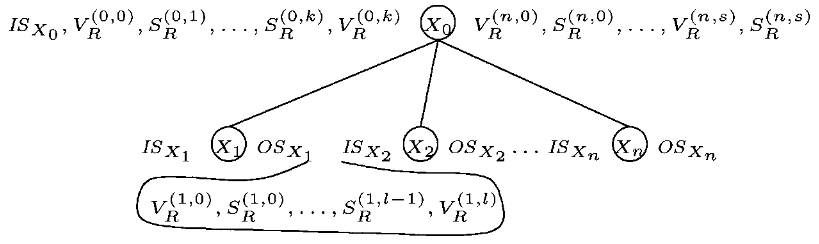
$IS_{X_0}, V_R^{(0,0)}, S_R^{(0,1)}, \dots, S_R^{(0,k)}, V_R^{(0,k)}$
X_0
$V_R^{(n,0)}, S_R^{(n,0)}, \dots, V_R^{(n,s)}, S_R^{(n,s)}$
IS_{X_1} X_1 OS_{X_1} IS_{X_2} X_2 $OS_{X_2} \dots IS_{X_n}$ X_n OS_{X_n}
$V_R^{(1,0)}, S_R^{(1,0)}, \dots, S_R^{(1,l-1)}, V_R^{(1,l)}$

Fig. 10. LNS(1)-condition.

Definition 4.9. Let $\mathfrak{S}$ be a sorted natural semantics specification, S_X be the set of sorts of grammar symbol X, $IS_X \uplus OS_X = S_X$ be a partition into a set of input and output sorts, and R be an inference rule of $\mathfrak{S}$ of production $X_0 ::= X_1 \cdots X_n$. A sequence $V_0, \dots, V_m$ of sets of logical variables and sort variables of R is *LNS(1)-computable* iff[5] $V_0 \uplus \cdots \uplus V_m = LVARS(R) \cup S_{X_0} \cup \cdots \cup S_{X_n}$, $V_0 = IS_{X_0}$, and for the sets V_k, $k \geq 1$, one of the following conditions is satisfied:

(1) $V_k \subseteq LVARS(R)$, $V_{k-1} \subseteq S_{X_0} \cup OS_{X_1} \cup \cdots \cup OS_{X_n}$, and for each $v \in V_k$ there is a constraint c adjacent to v in $DP(R)$ such that c is patently solvable with respect to $FV(c) \cap (V_0 \cup \cdots \cup V_{k-1})$.
(2) $V_k \subseteq S_{X_0}$, $V_{k-1} \subseteq LVARS(R)$, and for each $v \in V_k$ there is a constraint c adjacent to v in $DP(R)$ such that c is patently solvable with respect to $FV(c) \cap (V_0 \cup \cdots \cup V_{k-1})$.
(3) $V_k = OS_{X_i}$ for a $k \geq 1, i \geq 1$ and $V_{k-1} = IS(X_i)$.
(4) $V_k = IS_{X_i}$ for a $i \geq 1$, $V_{k-1} \subseteq LVARS(R)$, for each $v \in V_k$ there is a constraint c adjacent to v in $DP(R)$ such that c is patently solvable with respect to $FV(c) \cap (V_0 \cup \cdots \cup V_{k-1})$, $IS(X_j) \subseteq V_0 \cup \cdots \cup V_{k-1}$ for all $j = 0, \dots, i-1$, and $IS(X_j) \cap (V_0 \cup \cdots \cup V_{k-1}) = \emptyset$ for all $j = i+1, \dots, n$.

$\mathfrak{S}$ is *LNS(1)* iff for each symbol X there exists a partition $S_X = IS_X \uplus OS_X$ ($IS_Z = \emptyset$ for the start symbol Z) such that for every production each of its inference rules has a LNS(1)-computable sequence.

Intuitively, LNS(1) ensures that the variables $V_0, \dots, V_m$ can be computed in this order. The sort variables of the grammar symbols $X_1, \dots, X_n$ are computed from left to right. The conditions (1)–(4) state how the variables $v \in V_k$ can be computed, provided that the variables $w \in V_0 \cup \cdots \cup V_{k-1}$ are already computed. This process alternately computes logical variables as soon as they can be computed and sort variables. Condition (1) states the previous step V_{k-1} computed sort variables. Thus, V_k computes logical variables. In this case, there must be a constraint c that is solvable with respect to the variables $V_0 \cup \cdots \cup V_{k-1}$. Condition (2) states an analogous condition for the case that V_k computes sort variables of the grammar symbol X_0. Condition (3) states that, if the input sorts of symbol X_i are computed, then the output sorts of X_i can be computed (by recursively computing the variables of the production with left-hand side X_i in the structure tree). Condition (4) considers the remaining case that V_k computes

[5]$LVARS(R)$ is the set of logical variables of inference rule R.

Table I. Sets and Their Meanings During Traversal of Abstract Syntax Tree

$V_0 = \{\mathbf{Context}_0\}$	sort variables of $expr_0$ already computed when starting to visit the sub-tree rooted at $expr_0$
$V_1 = \{\Gamma\}$	logical variables that can now be solved by a constraint of the rule
$V_2 = \{\mathbf{Context}_1\}$	sort variables of $expr_1$ that can be solved by a constraint of the rule
$V_3 = \{\mathbf{Type}_1\}$	sort variables computed after visit of sub-tree rooted at $expr_1$
$V_4 = \{\mathsf{type}_1\}$	logical variables that can be solved after visit of sub-tree rooted at $expr_1$
$V_5 = \{\mathbf{Context}_2\}$	sort variables of $expr_2$ that can be solved by a constraint of the rule
$V_6 = \{\mathbf{Type}_2\}$	sort variables computed after visit of sub-tree rooted at $expr_2$
$V_7 = \{\mathsf{type}_2\}$	logical variables that can be solved after visit of sub-tree rooted at $expr_2$
$V_8 = \{\mathbf{Type}_0\}$	sort variables of $expr_0$ that can be solved by a constraint of R

input sort variables of symbol X_i by inference rule R. In this case, V_{k-1} must be a set of logical variables. The condition furthermore states that all sort variables of $X_1, \dots, X_{i-1}$ are computed before computing the input sorts of X_i and each of the sort variable of $X_{i+1}, \dots, X_n$ is computed after the computation of the input sorts of X_i. Thus, these conditions specify the left-to-right evaluation order.

Example 4.10. The inference rule (B.S10)

$$\frac{\Gamma :: \mathbf{Context} \vdash expr_1 : \mathsf{type}_1 :: \mathbf{Type} \qquad \Gamma :: \mathbf{Context} \vdash expr_2 : \mathsf{type}_2 :: \mathbf{Type}}{\Gamma :: \mathbf{Context} \vdash expr_0 : \mathsf{type}_1 \sqcup \mathsf{type}_2 :: \mathbf{Type}}$$

of the example language Demo is LNS(1)-computable. The grammar symbol *expr* has semantic information of sorts **Context** and **Type**. We classify theses sorts as an input sort and an output sort, respectively. Hence $IS_{expr} = \{\mathbf{Context}\}$ and $OS_{expr} = \{\mathbf{Type}\}$. In the following, $\mathbf{Context}_i$ and $\mathbf{Type}_i$ denote the sort variable **Context** and **Type** of grammar symbol $expr_i$, $i = 1, 2, 3$, respectively.

According to Definition 4.9 we have $V_0 = \{\mathbf{Context}_0\}$. Since $V_0 \subseteq S_{expr_0}$, (1) is applicable. The constraint $\mathbf{Context}_0 = \Gamma$ can be patently solved with respect to $\{\mathbf{Context}_0\}$. Thus $V_1 = \{\Gamma\}$. Now, the constraint $\mathbf{Context}_1 = \Gamma$ can be patently solved with respect to $\{\mathbf{Context}_0, \Gamma\}$. Since $V_1 \subseteq LVARS(R)$ and $IS_{expr_1} = \{\mathbf{Context}_1\}$, (4) can be applied. Therefore, $V_2 = \{\mathbf{Context}_1\}$. Now, property (3) is applicable, that is, $V_3 = OS_{expr_1} = \{\mathbf{Type}_1\}$. The constraint $\mathbf{Type}_1 = \mathsf{type}_1$ is patently solvable with respect to $\{\mathbf{Context}_1, \Gamma, \mathbf{Type}_1\}$. Thus, $V_4 = \{\mathsf{type}_1\}$. Since $V_4 \subseteq LVARS(R)$, $IS_{expr_2} = \{\mathbf{Context}_2\}$, and $\mathbf{Context}_2 = \Gamma$ is patently solvable with respect to $\{\mathbf{Context}_0, \mathbf{Context}_1, \Gamma, \mathbf{Type}_1, \mathsf{type}_1\}$, (4) is applicable. This leads to $V_5 = \{\mathbf{Context}_2\}$. Then, we construct $V_6 = OS_{expr_2} = \{\mathbf{Type}_2\}$ because only (3) is applicable. The constraint $\mathbf{Type}_2 = \mathsf{type}_2$ is patently solvable with respect to $\{\mathbf{Context}_0, \mathbf{Context}_1, \mathbf{Context}_2, \Gamma, \mathbf{Type}_1, \mathsf{type}_1, \mathbf{Type}_2\}$. Hence, $V_7 = \{\mathsf{type}_2\}$ by (1). Since $V_7 \subseteq LVARS(R)$ and the constraint $\mathbf{Type}_0 = \mathsf{type}_1 \sqcup \mathsf{type}_2$ is patently solvable with respect to $\{\mathbf{Context}_0, \mathbf{Context}_1, \mathbf{Context}_2, \Gamma, \mathbf{Type}_1, \mathsf{type}_1, \mathbf{Type}_2, \mathsf{type}_2\}$, we obtain $V_8 = \{\mathbf{Type}_0\}$. This finishes the construction because we have now encountered all variables from $LVARS(R)$. Table I summarizes these sets and their meaning with respect to the traversal of abstract syntax trees.

The specification of DEMO is not LNS(1) because there is no partition of the sorts into input sorts and outputs sorts of *stat* such that rule (B.S3) is LNS(1)-computable: it requires that **Context** is an output sort of *stat* because x and type are known only after visiting *stat*. (B.S5) implies that **Context** is also an output sort of *expr*, since otherwise the constraint $\mathbf{Context}_{expr} = \Gamma$ is not patently solvable with respect to $\emptyset$. However (B.S8) requires **Context** as an output sort of expression since otherwise the constraint $\mathbf{Context}_{expr} = \Gamma \uplus \{\langle \mathsf{x}, \mathsf{type}\rangle\}$ is not patently solvable with respect to $\{\mathbf{Context}_{expr}, \Gamma\}$.

We have intuitively stated that LNS(1)-specifications induce a particular traversal strategy. LNS(1) implies that, for all legal abstract syntax trees, the variables can be computed by a left-to-right traversal following the order induced by Definition 4.9:

THEOREM 4.11. *Let $\mathfrak{S}$ be a LNS(1)-specification. Then every rule cover can be completed to a proof by a single depth-first left-to-right traversal of an abstract syntax tree.*

PROOF. We first prove by structural induction that, for each node n of an abstract syntax tree stemming from nonterminal X, after visiting the subtree rooted at n the values of all variables are computed provided the values of the variables at n stemming from IS_X are known before the visit. If n is described by an axiom, then Definition 4.9 directly implies that the values of all variables of n can be computed. Suppose the children of n are obtained by production $X_0 ::= X_1 \cdots X_m$ and the inference rule applied is not an axiom. Let k_j, $j = 1, \dots, m$, be defined such that $V_{k_j} = IS_{X_j}$. Definition 4.9(4) implies $k_1 < k_2 < \cdots < k_m$. According to Definitions 4.9(1), 4.9(2), and 4.9(4), the values of all variables in $V_1 \cup \cdots \cup V_{k_1}$ can be computed. By the induction hypothesis, the values of all variables of the subtree rooted at the first child of n can be computed, in particular those at $V_{k_1+1} = OS_{X_1}$. By a simple induction on the child number, we can prove that, after visiting the subtree of the last child of n, all variables in $V_0 \cup \cdots \cup V_{k_m+1}$ are computed. Definitions 4.9(1) and 4.9(2) imply that, after the visit of the last child of n, the values of the remaining variables can be computed. This induction also shows that, upon each visit of a node of the abstract syntax tree, the values for its input sorts are known (and the root has no input sorts). This completes the proof. □

Example 4.12. The converse of Theorem 4.11 is not true, that is, LNS(1) is just a sufficient criterion. Figure 11 shows a sorted natural semantics specification where all abstract syntax trees can be computed by a depth-first left-to-right traversal. The sort definition for sort **S** is omitted for simplicity.

Figure 12 shows the two abstract syntax trees of this programming language. It is easy to see that, in both cases, the values of all variables can be computed by a single left-to-right traversal. However, the specification in Figure 11 is not LNS(1), because the inference rule (4.2) has no LNS(1)-computable sequence. For this inference rule, it is not possible to fix an order $V_0, \dots, V_m$ on its variables such that the variables can be computed in this order by a left-to-right traversal for *any* abstract syntax tree:

prod: $A ::= B\ C$

$$\frac{\vdash B : \langle \mathsf{x}, \mathsf{y} \rangle :: \mathbf{S} \quad \vdash C : \langle \mathsf{x}, \mathsf{y} \rangle :: \mathbf{S}}{\vdash A : \langle \mathsf{x}, \mathsf{y} \rangle :: \mathbf{S}} \tag{4.2}$$

prod $B ::= \mathsf{b}_1$

$$\vdash B : \langle \underline{\mathsf{b}}_1, \mathsf{y} \rangle :: \mathbf{S} \tag{4.3}$$

prod $B ::= \mathsf{b}_2$

$$\vdash B : \langle \mathsf{x}, \underline{\mathsf{b}}_2 \rangle :: \mathbf{S} \tag{4.4}$$

prod $C ::= D$

$$\frac{\vdash D : \langle \mathsf{x}, \mathsf{y} \rangle :: \mathbf{S}}{\vdash C : \langle \mathsf{x}, \mathsf{y} \rangle :: \mathbf{S}} \tag{4.5}$$

prod $D ::= \mathsf{d}$

$$\vdash D : \langle \underline{\mathsf{b}}_1, \underline{\mathsf{b}}_2 \rangle :: \mathbf{S} \tag{4.6}$$

Fig. 11. A sorted natural semantics specification which is not LNS(1).

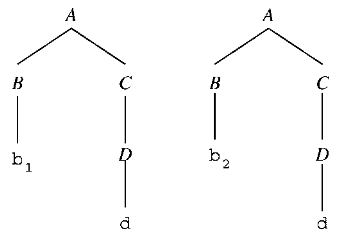

Fig. 12. The two abstract syntax trees for the language of Figure 11.

Consider first the subtree on the left of Figure 12. After the visit of the first child, the value of x from the inference rule (4.2) is known. Then, the second child is visited, and after this visit, the values of y and all other variables are known. However, for the abstract syntax tree on the right, the value of y is known after visiting the first child and the value of x is known after visiting the second child of the root. Hence, there is no LNS(1)-computable sequence for inference rule (4.2).

In the following, we present an algorithm that decides whether a sorted natural semantics specification is LNS(1), and if it is so, it returns for each inference rule an LNS(1)-computable sequence. The algorithm visits each grammar symbol. During the visit of a grammar symbol X, it visits each production with left-hand side X. The visit of a production $p : X_0 ::= X_1 \cdots X_n$ visits first every inference rule of p and then in turn each grammar symbol $X_1, \ldots, X_n$ not yet visited. During the visit of an inference rule R, the algorithm tries to find an LNS(1)-computable sequence for R. For this, it is necessary to have some assumptions on the input sorts IS_{X_0}. There are two possibilities during this construction: either a contradiction is found or it is not possible to find an LNS(1)-computable sequence for the inference rule R. In the former case, the specification is not LNS(1) and the algorithm terminates. In the latter case, the assumptions on the input sorts are revised and the visit process starts again with the revised assumptions. Initially, it is optimistically assumed that $IS_X = S_X$ for all grammar symbols X. The revision process stops iff no revision was necessary or a contradiction was found.

Figure 13 shows the algorithm for visiting an inference rule R of production $p : X_0 ::= X_1 \cdots X_n$. It greedily computes an LNS(1)-computable sequence

```
(1)  $V_0 := IS_{X_0}$, $j := 0$, $\bar{V} := V_0$; $i := 0$;
(2)  while $\bar{V} \subsetneq S_{X_0} \cup \cdots \cup S_{X_n} \cup LVARS(R) \wedge$ no contradiction to LNS(1) was found do
(3)       if $V_j \subseteq LVARS(R)$ then
(4)            $Constr := \{\mathbf{S} = t : s \in S_{X_0} \setminus \bar{V} \wedge V(t) \subseteq \bar{V}\}$;
(5)            if $Constr \neq \emptyset$ then $V_{j+1} := \{\mathbf{S} : \mathbf{S} = t \in Constr\}$
(6)            elsif $i < n$ then
(7)                 $Constr := \{\mathbf{S} = t : \mathbf{S} \in S_{X_{i+1}} \wedge V(t) \subseteq \bar{V}\}$; $V_{j+1} := \{\mathbf{S} : \mathbf{S} = t \in Constr\}$;
(8)                 if $IS_{X_{i+1}} \subseteq V_{j+1}$ then $V_{j+1} := IS_{X_{i+1}}$;
(9)                 else $IS_{X_{i+1}} := IS_{X_{i+1}} \cap V_{j+1}$; mark that the input sorts are revised; fi;
(10)                $i := i + 1$;
(11)           else there is a contradiction to LNS(1);
(12)      elsif $V_j \subseteq S_{X_0} \vee V_j \subseteq OS_{X_i}$ then
(13)           $Constr := \{ c \in Constr : c$ is patently solvable w.r.t. $FV(c) \cap \bar{V} \wedge$
                                 $c$ is not patently solvable w.r.t. $\bar{V} \setminus V_j\}$;
(14)           $V_{j+1} := \{\mathsf{x} \in LVARS(R) : \exists c \in Constr.\mathsf{x} \in FV(c) \setminus \bar{V}\}$;
(15)      elsif $V_j = IS_{X_i}$ then
(16)           $V_{j+1} := S_{X_i} \setminus IS_{X_i}$;
(17)      fi;
(18)      $\bar{V} := V \cup V_j$; $j := j + 1$;
(19) od
```

Fig. 13. Algorithm for computing an LNS(1)-computable sequence of an inference rule R for production $X_0 ::= X_1 \cdots X_n$.

$V_0, \ldots, V_m$ of R until the sequence is a partition of $LVARS(R)$ or a contradiction to LNS(1) is detected. The set $\bar{V} = V_0 \cup \cdots \cup V_j$ contains the variables already considered and X_i is the last considered symbol of p. According to Definition 4.9 only the cases specified by the conditions in lines (3), (12), and (15) need to be considered. First each iteration computes the new set *Constr* of constraints that can be solved with respect to $\bar{V}$. V_{j+1} is defined to be the maximal set of new variables whose value can be computed using the constraints of *Constr*. If $V_j \subseteq LVARS(R)$, it may happen that there are no constraints that can compute sorts of S_{X_0}. Then, according to Definition 4.9, the next set must be the input sorts of the next nonterminal X_{i+1}. If there is no such next nonterminal, there must be a contradiction to LNS(1) because no remaining constraints can be solved with respect to $\bar{V}$. If there is such a next nonterminal, then a revision is necessary if not each of its input sorts can be computed. Except the subsets of S_{X_0}, the other types of subsets may be empty. Therefore, a flag indicating the type of the set is maintained. For simplicity, we omit this detail in Figure 13.

Remark 4.13. The algorithm in Figure 13 can also be used to compute a legal direction assignment for $DP(R)$: before the computation of *Constr* all undirected edges $\{x, c\}$ are directed from logical variable or sort variable x to constraint c, if $x \in V_j$. After the computation of V_{j+1} all undirected edges $\{x, c\}$ are directed from $c \in Constr$ to $x \in V_{j+1}$.

THEOREM 4.14. *The algorithm for checking the LNS(1) property detects that a specification $\mathfrak{S}$ is LNS(1) together with the LNS(1)-computable sequences for every rule R iff $\mathfrak{S}$ is LNS(1).*

PROOF. ONLY IF: If the algorithm terminates with detection that $\mathfrak{S}$ is LNS(1), then it has computed a set of input sorts IS_X for each grammar symbol X. The definition $OS_X = S_X \setminus IS_X$ induces a partition of S_X. It remains to show that, for each inference rule R, the sequence $V_0, \ldots, V_m$ computed by the algorithm is LNS(1)-computable. Since the algorithm terminates with the detection that $\mathfrak{S}$ is LNS(1), the last visit of each inference rule neither revises a set of input sorts nor detects a contradiction to LNS(1). Hence, the loop (2)–(19) terminates with $V_0 \cup \cdots \cup V_m = S_{X_0} \cup \cdots S_{X_n} \cup V_R$, that is, after termination all sort variables and logical variables are considered. It remains to show that $V_0, \ldots, V_m$ are pairwise disjoint and that one of the conditions (1)–(4) of Definition 4.9 is satisfied. We prove this property by induction on j. Initially $V_0 = IS_{X_0}$. By the induction hypothesis, V_j satisfies one of the properties (1)–(4) of Definition 4.9. Hence, V_j satisfies one of the conditions in lines (3), (12), or (15). Hence, the set V_{j+1} is computed by one of the assignments in lines (5), (7), (14), and (16). In any case, $V_k \cap V_{j+1} = \emptyset$ for $k = 0, \ldots, j$. Furthermore, these assignments imply directly that V_{j+1} satisfies conditions (2), (4), (1), or (3) of Definition 4.9, respectively.

IF: We prove that if the algorithm terminates with output "$\mathfrak{S}$ is not LNS(1)" then $\mathfrak{S}$ is not LNS(1). This output is possible only if line (11) of Figure 13 is executed for a inference rule R. Hence $V_0 \cup \cdots \cup V_j \neq LVARS(R) \cup S_{X_0} \cup \cdots \cup S_{X_n}$, $V_j \subseteq LVARS(R)$, the last nonterminal X_n is already visited, and it is not possible to find a constraint $\mathbf{S} = t$ that can be solved with respect to $\bar{V}$. Hence, there is no LNS(1)-computable sequence for R with respect to the current partitions $S_X = IS_X \uplus OS_X$ of grammar symbols X. However, there might be other partitions. Let $\mathbf{S} = t$ be a constraint that cannot be solved with respect to $\bar{V}$. Since $S_{X_1} \cup \cdots \cup S_{X_n} \subseteq \bar{V}$, line (11) is also executed for all different partitions of $S_{X_1}, \ldots, S_{X_n}$. Furthermore, $\mathbf{S} \in OS_{X_0}$. Suppose there is a different partition of $S_{X_0} = IS'_{X_0} \uplus OS'_{X_0}$ where an LNS(1)-computable sequence for R is computed. Then, it must hold $\mathbf{S} \in IS'_{X_0}$. Therefore, there was a previous visit of R where $\mathbf{S} \in IS_{X_0}$. Since $\mathbf{S} \notin IS_{X_0}$ at the current visit, the set IS_{X_0} was revised. In particular $\mathbf{S}$ was eliminated. This can only be done if line (9) is executed for a inference rule R' of a production $Y ::= \cdots X_0 \cdots$. However, this means that $\mathbf{S}$ cannot be computed by a LNS(1)-computable sequence for R'. Hence, $\mathfrak{S}$ cannot be LNS(1). □

4.3 Other Solution Strategies

This subsection discusses informally several other solution strategies. LNS(k)-specifications guarantee that all rule covers be completed by k depth-first left-to-right traversals through the abstract syntax tree. It is a straightforward generalization of the LNS(1)-condition: for each grammar symbol X we have partitions $IS_X = IS_X^{(1)} \uplus \cdots \uplus IS_X^{(k)}$ of the input sorts and $OS_X = IS_X^{(1)} \uplus \cdots \uplus IS_X^{(k)}$ of the output sorts into k sets. With this partition, Definition 4.9 can be directly generalized to LNS(k)-computable sequences and LNS(k)-specifications. RNS(k)-specifications guarantee that all rule covers can be completed by k depth-first right-to-left traversals through the abstract syntax trees. ANS(k)-specifications guarantee that all rule covers can be completed by k depth-first traversals through the abstract syntax tree alternating between left-to-right traversals and right-to-left traversals. The idea behind these different traversal strategies is analogous to the predefined traversal strategies in attribute grammars.

It is an open problem whether there is a class of sorted natural semantic specifications for derived solution strategies analogous to ordered attribute grammars. However, if all inference rules of a sorted natural semantics specification use only defined functions (i.e., each semantic information has a term consisting only of variables and symbols for derived functions), then no logical variable is solved by unification because they must be computed before this according to the restricted AC1-unification which we employ. We will show in Section 5 that these specifications can be transformed directly into attribute grammars. Therefore this special case allows us to apply all results known from attribute grammars.

5. COMPARISON WITH ATTRIBUTE GRAMMARS

Attribute grammars are used for specifying and generating semantic analysis. This section compares the expressiveness of sorted natural semantics and attribute grammars and the efficiency of the semantic analyses generated from them. In Section 5.1 we show that each well-defined attribute grammar can be represented directly as a sorted natural semantics specification. Section 5.2 shows that the converse is not true. Section 5.3 compares LAG-grammars with LNS-specifications.

5.1 Representation of Attribute Grammars by Sorted Natural Semantics

First, we present the basic terminology for attribute grammars. Then we discuss the relation between attribute grammars and sorted natural semantics.

Definition 5.1. An *attribute grammar* is a tuple $AG = (G, A, R, C)$ where $G = (T, N, P, Z)$ is a context-free grammar with terminals T, nonterminals N, productions P, and start symbol Z describing the abstract syntax, $A = \cup_{X \in T \cup N} A(X)$ is a finite set of *attributes*, $R = \cup_{p \in P} R(p)$ is a finite set of *attribution rules*, and $C = \cup_{p \in P} C(p)$ is a finite set of *AG-conditions*.

Example 5.2. Appendix B.3 contains the attribute grammar for DEMO. $X.a$ denotes that a is an attribute of X, that is, $a \in A(X)$. Attribution rules are associated with the productions and have the form $X_i.a \leftarrow f(\cdots)$. AG-Conditions are Boolean formulas also associated with a production.

The set $AF(p)$ is the set of attributes that are computed by production p, that is, $X_i.a \in AF(p)$ iff there is an attribution rule of production p with left-hand side $X_i.a$. The attribute $X_i.a$ is *synthesized* iff X_i is the left-hand side of the production, otherwise it is *inherited*. $AS(X)$ and $AI(X)$ denote the set of synthesized and inherited attributes, respectively. An attribute grammar is *complete* iff $AI(X) \cup AS(X) = A(X)$, $AS(X) \subseteq AF(p)$ for each production with left-hand side X, and $AI(X) \subseteq AF(p)$ for each production p with a right-hand side containing X. An attribute grammar is *consistent* iff $AI(X) \cap AS(X) = \emptyset$ for all $X \in T \cup N$ and if, for each $p \in P$ and for each $X.a \in AF(p)$, there is exactly one attribution rule in $R(p)$ with left-hand side $X.a$.

Attribute grammars specify the values of the attributes by attribution rules. If the attribute values are known for the right-hand side of an attribution rule, then the attribute value of the left-hand side can be computed. Let t be an abstract syntax tree. t is *correctly attributed* iff for every node n_0 with children $n_1, \ldots, n_k$ obtained according to production $p : X_0 ::= X_1 \cdots X_k$, $X_i.a = f(X_{j_1}.a_1, \ldots, X_{j_k}.a_k)$ for every attribution rule $X_i.a \leftarrow f(X_{j_1}.a_1, \ldots, X_{j_k}.a_k) \in R(p)$ and each AG-condition $c(X_{i_1}.a'_1, \ldots, X_{i_p}.a'_p) \in C(p)$ evaluates to $\underline{\text{true}}$. Complete and consistent attribute grammars guarantee that for each abstract syntax tree there is at most one correct attribution.

For an abstract syntax tree, the attributes can be computed if the dependencies induced by the attribution rules do not define a cycle. An attribute grammar is *acyclic* iff this is true for every abstract syntax tree. An attribute grammar is *well-defined* iff it is complete, consistent, and acyclic. For well-defined grammars an attribution can be computed for each abstract syntax tree by evaluating them in a topological order of the dependencies.

The following theorem shows that each well-defined attribute grammar can be transformed automatically into an "equivalent" sorted natural semantics specification with the same attributes:

THEOREM 5.3. *For every well-defined attribute grammar $AG = (G, A, R, C)$ there is a transformation to a well-formed sorted natural semantics specification $\mathfrak{S}$ based on G with the following properties:*

(1) *$X.a \in A$ iff the judgment for nonterminal X has a semantic information of sort* $\mathbf{a}$.
(2) *The attributions obtained by AG and $\mathfrak{S}$ are equal for every abstract syntax tree.*
(3) *The basic algorithm on $\mathfrak{S}$ rejects an abstract syntax tree iff there is no correct attribution for AG.*

PROOF. Property (1) defines the sorts of $\mathfrak{S}$. Since all functions and AG-conditions used in AG are computable, they can be specified by sort definitions and Horn clauses [Andréka and Németi 1978]. Without loss of generality we assume that the attribute grammar is normalized, that is, for every production p it holds $X_j.a \notin AF(p)$ if there is a rule $X_i.a \leftarrow f(\cdots X_j.a \cdots) \in R(p)$ or an AG-condition $\phi(\cdots X_j.a \cdots) \in C(p)$. Furthermore, assume without loss of generality that there is only one AG-condition per rule. The construction of the inference rules of $\mathfrak{S}$ is based on the following transformation: for each grammar symbol X, the natural semantics specification contains judgments of the form

$$t_1 :: \mathbf{a}_1, \ldots, t_k :: \mathbf{a}_k \vdash X : t_{k+1} :: \mathbf{a}_{k+1}, \ldots t_n :: \mathbf{a}_n,$$

where $AI(X) = \{a_1, \ldots, a_k\}$ are the inherited attributes of X and $AS(X) = \{a_{k+1}, \ldots, a_n\}$ are the synthesized attributes. Our goal is to define inference rules such that for any attributed syntax tree after solving the constraints stemming from the inference rules, the value for sort $\mathbf{a}$ equals the value of the attribute a for any node of the attributed syntax tree.

For each production $p : X_0 ::= X_1 \cdots X_m$ we generate an inference rule that computes the inherited attributes of $X_1, \ldots, X_m$ and the synthesized attributes of X_0. Thus, we generate the following inference rule:

$$\frac{\begin{array}{c} t_1^1 :: \mathbf{a}_1^1, \ldots, t_{k_1}^1 :: \mathbf{a}_{k_1}^1 \vdash X_1 : \mathsf{a}_{k_1+1}^1 :: \mathbf{a}_{k_1+1}^1, \ldots, \mathsf{a}_{n_1}^1 :: \mathbf{a}_{n_1}^1 \\ \vdots \\ t_1^m :: \mathbf{a}_1^m, \ldots, t_{k_m}^m :: \mathbf{a}_{k_m}^m \vdash X_m : \mathsf{a}_{k_m+1}^m :: \mathbf{a}_{k_m+1}^m, \ldots, \mathsf{a}_{n_m}^m :: \mathbf{a}_{n_m}^m \end{array}}{\mathsf{a}_1^0 :: \mathbf{a}_1^0, \ldots, \mathsf{a}_{k_0}^0 :: \mathbf{a}_{k_0}^0 \vdash X_0 : t_{k_0+1}^0 :: \mathbf{a}_{k_0+1}^0, \ldots, t_{n_0}^0 :: \mathbf{a}_{n_0}^0} \quad \text{if } \bar{\phi}. \qquad (5.1)$$

The a_j^i are new variables for each synthesized attribute $X_i.a_j$ of $X_i, i = 1, \ldots, m$, and for each inherited attribute $X_0.a_j$ of X_0, and $\bar{\phi}$ is obtained from the conjunctions of the AG-conditions $\phi \in C(p)$ by replacing each occurrence of an attribute $X_h.a_l \notin AF(p)$ by variable a_l^h and each occurrence of an attribute $X_h.a_l \in AF(p)$ by the term t_l^h. The attributes $X_h.a_l \notin AF(p)$ are not computed within the context of production p and, hence, their values are denoted by new variables. The terms t_j^i denote the values for the attributes which are computed within the context of production p. They are obtained from the right-hand sides of the attribution rule for $X_i.a_j \in AF(p)$ by replacing each occurrence of an attribute $X_h.a_l$ by variable a_l^h, $0 \leq h \leq m$. The terms t_j^i are uniquely determined because AG is consistent.

Since there is only one inference rule per production, each abstract syntax tree has exactly one rule cover. Let t be an abstract syntax tree and n_0 be a node of t with children $n_1, \ldots, n_m$ obtained by production $p : X_0 ::= X_1 \cdots X_m$. According to the transformation (5.1) each attribute evaluation $n_i.a \leftarrow f(n_{j_1}.a_1, \ldots, n_{j_q}.a_q)$ induced by $R(p)$ corresponds one-to-one to a constraint $\mathbf{a} = f(\mathsf{a}_1, \ldots, \mathsf{a}_q)$. Since AG is well-defined, the attributes can be computed in any topological order of the dependencies. Hence, the constraints can be solved in the same order. It is a straightforward induction on this topological order to prove that the attribution of t computed by the basic algorithm with $\mathfrak{S}$ and by the attribute grammar AG is the same, that is, property (2) holds. An analogous argument shows that each side condition is evaluated with the same arguments in both cases. Hence, property (3) holds. □

Example 5.4. We apply the transformation (5.1) to the attribute grammar in Figure 35. The result is the sorted natural semantics specification shown in Figure 14. In contrast to the sorted natural semantics specification of DEMO shown in Figure 34, this specification has only one inference rule per production. Since the attribute grammar in Figure 35 has an attribute *defs* that collects all declarations of the program, the sorted natural semantics specification in Figure 14 has an additional sort **Defs** to collect all declarations of a program and to propagate them as a context through the entire abstract syntax tree. The sorted natural semantics specification of Figure 34 has only one context because constraints may be partially solved and the remaining variables substituted later.

The construction in the proof of Theorem 5.3 defines a subset of sorted natural semantic specifications, namely those corresponding to a well-defined attribute grammar. For this subset, we can generate semantic analyses with the same efficiency as if they were generated by attribute grammars. To see

Production (B.1): $prog ::= stats$

$$\frac{\mathsf{defs} :: \mathbf{Context} \vdash stats : \mathsf{defs} :: \mathbf{Defs}}{\vdash prog} \text{ if } \mathit{unambigous}(\mathsf{defs}) \tag{5.2}$$

Production (B.2): $stats_0 ::= stat; stats_1$

$$\frac{\begin{array}{c}\mathsf{context} :: \mathbf{Context} \vdash stat : \mathsf{decl} :: \mathbf{Decl} \\ \mathsf{context} :: \mathbf{Context} \vdash stats_1 : \mathsf{defs} :: \mathbf{Defs}\end{array}}{\mathsf{context} :: \mathbf{Context} \vdash stats_0 : decl \cup defs :: \mathsf{Defs}} \tag{5.3}$$

Production (B.3): $stats ::= \varepsilon$

$$\mathsf{context} :: \mathbf{Context} \vdash stats : \emptyset :: \mathbf{Defs} \tag{5.4}$$

Production (B.4): $stat ::= var := expr$

$$\frac{\begin{array}{c}\mathit{gettype}(\mathsf{context}, \mathsf{id}) :: \mathbf{Type} \vdash var : \mathsf{id} :: \mathbf{Id} \\ \mathsf{context} :: \mathbf{Context} \vdash expr : \mathsf{type} :: \mathbf{Type}\end{array}}{\mathsf{context} :: \mathbf{Context} \vdash stat : \emptyset :: \mathbf{Decl}} \text{ if } \begin{array}{l}\mathsf{type} \sqsubseteq \mathit{gettype}(\mathsf{context}, \mathsf{id}) \wedge \\ \mathit{is\_defined}(\mathsf{context}, \mathsf{id})\end{array} \tag{5.5}$$

Production (B.5): $stat ::= var : type$

$$\frac{\mathsf{type} :: \mathbf{Type} \vdash var : \mathsf{id} :: \mathbf{Id} \qquad \vdash type : \mathsf{type} :: \mathbf{Type}}{\mathsf{context} :: \mathbf{Context} \vdash stat : \{\langle \mathsf{id}, \mathsf{type} \rangle\} :: \mathbf{Decl}} \tag{5.6}$$

Production (B.6): $var ::= \mathsf{id}$

$$\mathsf{type} :: \mathbf{Type} \vdash var : v(\mathsf{id}) :: \mathbf{Id} \tag{5.7}$$

Production (B.7): $expr ::= var$

$$\frac{\mathit{gettype}(\mathsf{context}, \mathsf{id}) :: \mathbf{Type} \vdash var : \mathsf{id} :: \mathbf{Id}}{\mathsf{context} :: \mathbf{Context} \vdash expr : \mathit{gettype}(\mathsf{context}, \mathsf{id}) :: \mathbf{Type}} \text{ if } \mathit{is\_defined}(\mathsf{context}, \mathsf{id}) \tag{5.8}$$

Production (B.8): $expr ::= const$

$$\frac{\vdash const :: \mathbf{Type}}{\mathsf{context} :: \mathbf{Context} \vdash expr : \mathsf{type} :: \mathbf{Type}} \tag{5.9}$$

Production (B.9): $expr_0 ::= expr_1 + expr_2$

$$\frac{\mathsf{context} :: \mathbf{Context} \vdash expr_1 : \mathsf{type}_1 :: \mathbf{Type} \qquad \mathsf{context} :: \mathbf{Context} \vdash expr_2 : \mathsf{type}_2 :: \mathbf{Type}}{\mathsf{context} :: \mathbf{Context} \vdash expr_0 : \mathsf{type}_1 \sqcup \mathsf{type}_2 :: \mathbf{Type}} \tag{5.10}$$

Production (B.10): $const ::= \mathsf{intconst}$

$$\vdash const : \underline{\mathsf{inttype}} :: \mathbf{Type} \tag{5.11}$$

Production (B.11): $const ::= \mathsf{realconst}$

$$\vdash const : \underline{\mathsf{realtype}} :: \mathbf{Type} \tag{5.12}$$

Production (B.12): $type ::= \underline{\mathsf{int}}$

$$\vdash type : \underline{\mathsf{inttype}} :: \mathbf{Type} \tag{5.13}$$

Production (B.13): $type ::= \underline{\mathsf{real}}$

$$\vdash type : \underline{\mathsf{realtype}} :: \mathbf{Type} \tag{5.14}$$

Fig. 14. Transformation of the attribute grammar in Figure 35.

this suppose that all inference rules have the shape (5.1) and all judgments for a nonterminal X have the same sorting. Then the converse transformation can be applied. The result is an attribute grammar which can be tested for all properties (e.g., completeness, consistency, well-definedness, etc.). Thus, the generators generating semantic analyses from attribute grammars can be applied.

5.2 Representation of Sorted Natural Semantics by Attribute Grammars

The converse of Theorem 5.3 is not true: there are sorted natural semantics specifications which cannot be expressed by an equivalent attribute grammar. This implies that the specification technique of sorted natural semantics is more expressive than the specification technique of attribute grammars. For proving this claim it is sufficient to give one example of such a sorted natural semantics specification:

THEOREM 5.5. *For the sorted natural semantics specification of* DEMO *in Appendix* B.2 *there is no well-defined attribute grammar such that properties* (1)–(3) *of Theorem* 5.3 *are satisfied.*

PROOF. Suppose there were such a well-defined attribute grammar AG. According to (1), its attributes are

$$A(stats) = \{context\}, \qquad A(var) = \{string\}, \qquad A(expr) = \{context, type\},$$
$$A(stat) = \{context, decl\}, \qquad A(type) = \{type\}, \qquad A(const) = \{type\}.$$

Consider the two abstract syntax trees in Figure 15. In both cases, there must be an indirect dependency from the declaration of x to its use in the expression $\mathtt{x} + 1$.

Consider the abstract syntax tree on the left. Since *context* is the only attribute of *stats*, the node marked with $*$ implies that the production $stats_0 ::= stat; stats_1$ has an attribution rule $stats_1.context \leftarrow stats_0.context$. Otherwise there could not be any indirect dependency from the declaration of x to its use in the expression $\mathtt{x} + 1$.

Now consider the abstract syntax tree on the right. With the same arguments as above, the node marked with $+$ implies that the production $stats_0 ::= stat; stats_1$ has an attribution rule $stats_0.context \leftarrow stats_1.context$.

Combining these two attribution rules implies that AG is not well-defined, contradicting our assumption. □

Remark 5.6. There is a well-defined attribute grammar AG such that any abstract syntax tree is accepted by the sorted natural semantics specification for DEMO iff it is also accepted by AG and for all nonterminals, its number of attributes in AG being at most its number of sorts in the sorted natural semantics specification. This is an immediate consequence of the fact that one synthesized attribute per nonterminal is sufficient to describe any static program meaning [Knuth 1968, 1971]. Note that this synthesized attribute may be structured arbitrarily complex. However, if a semantic analysis is used in a compiler, the intermediate code-generation and later phases require specific attributes for

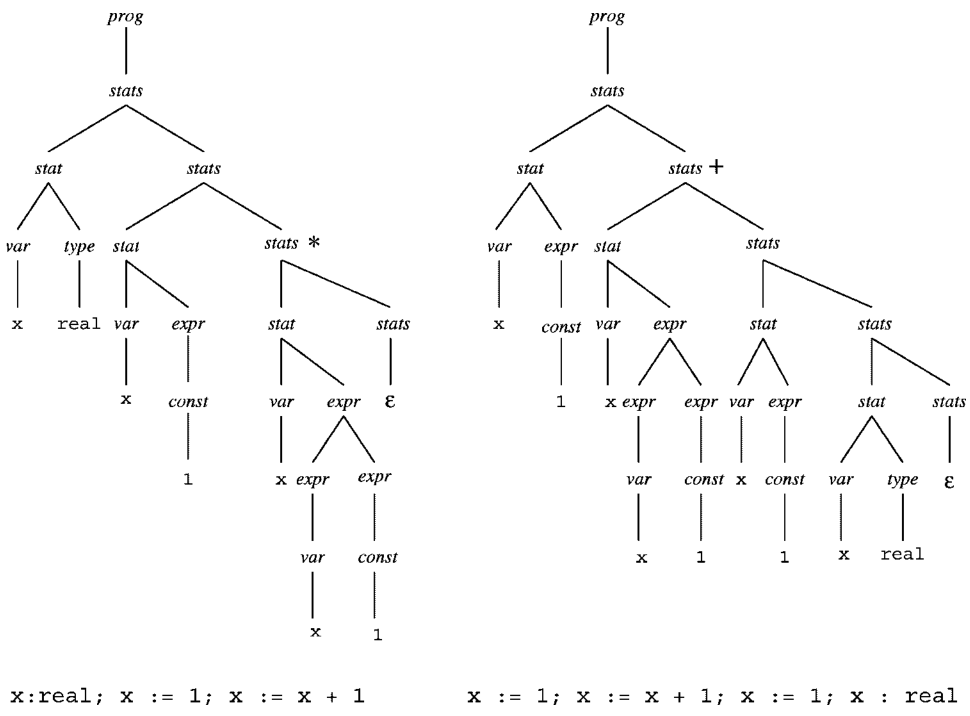

Fig. 15. Abstract syntax trees used in the proof of Theorem 5.5.

nonterminals—and this is exactly the situation reflected by Theorems 5.3 and 5.5.

5.3 LAG(1)-Grammars versus LNS(1)-Specifications

In this subsection, we show that if there is exactly one inference rule per production, then LAG(1)-attribute grammars are as expressive as LNS(1)-specifications. In particular, the transformation of attribute grammars into sorted natural semantics specifications discussed in Section 5.1 always transforms LAG(1)-attribute grammars into LNS(1)-specifications. In contrast to Theorem 5.5, LNS(1)-specifications can also be transformed into LAG(1)-grammars with the same attributes.

Definition 5.7. An attribute grammar $AG = (G, A, R, C)$ is *LAG(1)* iff, for every application of $p : X_0 ::= X_1 \cdots X_n$ in an abstract syntax tree, the attributes in $A(P) = A(X_0) \cup \cdots \cup A(X_n)$ can be computed in the following order: $AI(X_0), AI(X_1), AS(X_1), AI(X_2), \ldots, AS(X_n), AS(X_0)$.

For LAG(1)-grammars, the attributes of each abstract syntax tree can be computed by one depth-first left-to-right traversal.

THEOREM 5.8. *For every LAG(1)-grammar* $AG = (G, A, R, C)$ *there is a transformation to an LNS(1)-specification* $\mathfrak{S}$ *based on* G *with the properties* (1)–(3) *of Theorem* 5.3.

PROOF. We use the same transformation as in the proof of Theorem 5.3 and also assume that AG is normalized. It remains to show that the resulting specification $\mathfrak{S}$ is LNS(1). Each attribute $a \in A(X)$ is represented by a sort $\mathbf{a} \in S_X$ where S_X is the set of sorts of grammar symbol X. We partition the sorts S_X into a set of input sorts and a set of output sorts by defining $IS_X = \{\mathbf{a} : a \in AI(X)\}$ and $OS_X = \{\mathbf{a} : a \in AS(X)\}$. Since $A(X) = AI(X) \uplus AS(X)$, it holds $S_X = IS_X \uplus OS_X$. It remains to show how LNS(1)-computable sequences for inference rules can be defined. Consider the production $p : X_0 ::= X_1 \cdots X_m$ with the inference rule (5.1). The sequence

$$\begin{aligned}
&V_0 = IS_{X_0},\ V_1 = \{\mathrm{a}_1^0, \dots, a_{k_0}^0\},\\
&V_2 = IS_{X_1}, V_3 = OS_{X_1}, V_4 = \{\mathrm{a}_{k_1+1}^1, \dots, \mathrm{a}_{n_1}^1\},\\
&V_5 = IS_{X_2}, \dots, V_{3m-2} = \{\mathrm{a}_{k_{m-1}+1}^{m-1}, \dots, \mathrm{a}_{n_{m-1}}^{m-1}\},\\
&V_{3m-1} = IS_{X_m}, V_{3m} = OS(X_m), V_{3m+1} = \{\mathrm{a}_{k_m+1}^m, \dots, \mathrm{a}_{n_m}^m\}, V_{3m+2} = OS(X_m)
\end{aligned} \tag{5.2}$$

is an LNS(1)-computable sequence. Obviously, the sequence is a partition of the logical variables and the sort variables of inference rule (5.1). We show by induction that every V_j, $1 \leq j \leq m$ satisfies one of the properties (1)–(4) of Definition 4.9.

By construction it holds $IS_{X_0} = \{\mathbf{a}_1^0, \dots, \mathbf{a}_{k_0}^0\}$. Hence, the constraints $\mathbf{a}_i^0 = \mathrm{a}_i^0$, $i = 1, \dots, k_0$, are solvable with respect to V_0. Thus, V_1 satisfies property (1).

Case 1: $j = 3h - 1$ for a $1 \leq h \leq m$. Then $V_j = IS_{X_h}$. Suppose that there is a constraint $\mathbf{a}_i^h = t_i^h$ that is not solvable with respect to $V_0 \cup \cdots \cup V_{j-1}$. Then, t_i^h contains a variable a_s^l for a $l > h$. However, then the attribute $a_i^h \in AS(X_h)$ cannot be computed before attribute $a_s^l \in AS(X_l)$ for a $l > h$, which contradicts the LAG(1) condition. Therefore, condition (4) is satisfied.

Case 2: $j = 3h$ for a $1 \leq h \leq m$. Then $V_j = OS_{X_h}$ and condition (3) is satisfied.

Case 3: $j = 3h + 1$ for a $1 \leq h \leq m$. This case is argued analogous to V_1 using the set IS_{X_h} instead of AI_{X_0}.

Case 4: $j = 3m+2$. It holds $LVARS(R) \subseteq V_0 \cup \cdots \cup V_{3m+1}$. Hence, the constraints $\mathbf{a}_i^0 = t_i^0$, $i = k_0 + 1, \dots, n_0$ are patently solvable with respect to $V_0 \cup \cdots \cup V_{3m+1}$. Therefore (2) is satisfied. □

THEOREM 5.9. *For every LNS(1)-specification $\mathfrak{S}$ based on the abstract syntax G with one inference rule per production, there is a transformation to an LAG(1)-grammar $AG = (G, A, R, C)$ with the properties* (1)–(3) *of Theorem* 5.3.

PROOF. We define first $AG = (G, A, R, C)$ and then show that AG is LAG(1). In the proof, $X.\mathbf{S}$ denotes that $\mathbf{S}$ is the sort of a semantic information of X, and S_X denotes the set of all sorts of semantic informations of X. For every symbol X of G we define $A(X) = S_X$.

Since $\mathfrak{S}$ is LNS(1), for each symbol X of G there is a partition $S_X = IS_X \uplus OS_X$ into input sorts and output sorts such that for each inference rule there is an LNS(1)-computable sequence $V_0, \dots, V_k$. In order to construct $R(p)$ and $C(p)$ it

```
(1)  $C_{k+1}(p) := \{\varphi\}$ where $\varphi$ is the side condition of rule $\rho$;
(2)  for $j = k, \dots, 0$ do
(3)      $C_j(p) := C_{j+1}(p)$;
(3)      if $V_j \subseteq LVARS(R)$ then
(4)          for $x \in V_j$ do
(5)              $Constr_{j,x} := \{c \in Constr_j : x \in FV(c)\}$;
(6)              choose $X_i.\mathbf{S}_0 = t \in Constr_{j,x}$;
(7)              replace in $R$ and $C_j(p)$ each occurrence of $x$ by $\phi_{c,x}(X_i.\mathbf{S}_0, x_1, \dots, x_m)$
(8)              where $V(t) \cap \bar{V}_{j-1} = \{x_1, \dots, x_m\}$ ;
(9)              $C_j(p) := C_j(p) \cup \{\phi_{c,x}(\cdots) = \phi_{c',x}(\cdots) : c, c' \in Constr_{j,x} \wedge c \neq c'\}$;
(10)         od
(11)     fi
(12) od
```

Fig. 16. Elimination of logical variables from an inference rule ρ.

is necessary to know functions that compute the values of the logical variables and sort variables. Let c be a constraint that is patently solvable with respect to a set of variables $X = \{x_1, \dots, x_m\}$. Then, for each variable $x_0 \in FV(c) \setminus X$ there is a function $\phi_{c,x_0} : \mathbf{S}_1 \times \cdots \mathbf{S}_m \to \mathbf{S}_0$ where $S_i = x_i$ if x_i is a sort variable and S_i is the sort of x_i if x_i is a logical variable, $i = 0, \dots, m$. If $FV(c) = X$, then there is a function $\phi_c : \mathbf{S}_1 \times \cdots \times \mathbf{S}_m \to \mathbf{Bool}$.

Consider now an inference rule ρ of production $p : X_0 ::= X_1 \cdots X_n$. Let $V_0, \dots, V_k$ the LNS(1)-computable sequence of ρ. We use this sequence and the functions ϕ_c to construct $R(p)$ and $C(p)$. First, we eliminate the logical variables from ρ using the functions ϕ_c. During this elimination, the set $C(p)$ is computed. The second step computes the set $R(p)$. In the following discussion $\bar{V}_j = V_0 \cup \cdots \cup V_j$ and $Constr_j$ denotes the set of constraints that are patently solvable with respect to $\bar{V}_j$.

The first step traverses the LNS(1)-sequence from the back and replaces successively in the inference rule ρ the variables $V_j \subseteq V(\rho)$ by function calls and computes the AG-condition set $C(p) = C_0(p)$. Figure 16 shows the details. Each variable $x \in V_j$ is considered in turn. If there is more than one constraint in $Constr_j$ containing x, then the computed values must be equal. Therefore, these equalities are added to $C(p)$ (cf. line (9)). After one iteration of loop (4)–(10), neither ρ nor $C_j(p)$ contains x. Hence, after the transformation, the new inference rule ρ' and the AG-conditions $C(p)$ do not contain logical variables. □

LEMMA 5.10. *Let $\mathfrak{S}$ be a specification and $\mathfrak{S}'$ be the specification obtained from $\mathfrak{S}$ by elimination of the logical variables. Then, for every abstract syntax tree, a rule cover with the rules of $\mathfrak{S}$ can be completed to a proof iff a rule cover of $\mathfrak{S}'$ can be completed to a proof tree, the values for the sort variables are equal, and all side conditions are satisfied.*

PROOF. All constraints $\mathbf{S} = t$ can be evaluated from right to left by a depth-first traversal through an abstract syntax tree. An induction on the abstract syntax tree analogous to the proof of Theorem 4.11 proves that the values of the sort variables remain unchanged and that the side conditions are satisfied. □

The second step considers the inference rule ρ' of $\mathfrak{S}'$. It inductively constructs sets $R_j(p)$ of attribution rules for the sequence $V_0, \dots, V_k$. $R_0(p) = \emptyset$ and

$$R_j(p) = \begin{cases} R_{j-1}(p) \cup \{X_0.\mathbf{S} \leftarrow t : \mathbf{S} = t \in Constr_j\}, & \text{if} \quad V_j \subseteq OS_{X_0}, \\ R_{j-1}(p) \cup \{X_i.\mathbf{S} \leftarrow t : \mathbf{S} = t \in Constr_j\}, & \text{if} \quad V_j \subseteq IS_{X_i}, i \geq 1, \\ R_j(p), & \text{otherwise.} \end{cases} \tag{5.3}$$

Remember that t does not contain logical variables.

LEMMA 5.11. *Let $\mathfrak{S}'$ be defined as in Lemma* 5.10 *and $AG = (G, A, R, C)$ the attribute grammar constructed by* (5.3). *Then, for any abstract syntax tree, a rule cover can be completed to a proof tree such that all side conditions are satisfied iff the values for the sort variables define a correct attribution with respect to AG*

PROOF. It follows directly from the fact that $X_i.\mathbf{S} \leftarrow t \in R(p)$ iff $X_i.\mathbf{S} = t$ is a constraint derived from the inference rule of p. □

In this attribute grammar, $AI(X) = IS_X$ and $AS(X) = OS_X$. As above let ρ be the inference rule of p in $\mathfrak{S}$, $V_0, \dots, V_k$ be an LNS(1)-computable sequence for ρ, and $V_{j_0}, \dots, V_{j_s}$ be the subsequence consisting of sets of sort variables. Lemmas 5.10 and 5.11 imply that, for p, the attributes can be computed in order $V_{j_0}, \dots, V_{j_s}$. This sequence almost has the form in Definition 5.7. The only difference is that between $AI(X_0)$ and $AI(X_1)$, between $AS(X_i)$ and $AI(X_{i+1})$, $i = 1, \dots, n-1$, and after $AS(X_n)$, there might be sets $V_j \subseteq AS(X_0)$. However, if AG is normalized, then no attribution rule uses an attribute of $AF(p)$ on its right-hand side. Therefore all rules computing an attribute of $AS(X_0)$ can be computed after all children are visited. Therefore the normalized attribute grammar is LAG(1).

Remark 5.12. If the constraint c has the form $\mathbf{S} = f(t_1, \dots, t_n)$, then there are only two functions $\phi_{c,\mathbf{S}}$ and ϕ_c with the definition $\phi_{c,\mathbf{S}}(x_1, \dots, x_m) = f(t_1, \dots, t_n)$ and $\phi_c(x_1, \dots, x_m, \mathbf{S}) = c$. If the constraint c does not contain defined functions, then the functions ϕ_c and ϕ_{c,x_0} can be obtained by partial evaluation of the unification algorithm with respect to X. Thus, the transformation in the proof of Theorem 5.9 is automatic.

6. APPLICATION IN FIXED-POINT ANALYSES

Many static program analyses specify static semantic information of programs in such a way that it cannot be computed directly. Instead one needs to determine a solution iteratively by searching for a fixed-point with respect to the specification. Typically the solutions for such fixed-point analyses are defined by suitable sets of constraints. Thereby one needs to rely on well-developed theories to ensure three requirements. First, one needs to make sure that there is at least one fixed-point for the specification. Furthermore, one needs to establish that a fixed-point will be found by a suitable algorithm. Finally, one needs to prove that this algorithm will find the desired fixed-point, that is, the smallest or greatest depending on the analysis problem.

Such fixed-point program analyses can be described within sorted natural semantics. As a detailed example, we consider type and effect systems in Section 6.1. In Section 6.2 we discuss that classical data flow analyses fit into the framework of sorted natural semantics as well.

Taking sorted natural semantics as a common framework has the advantage that solutions of an analysis problem are described in a mathematically concise way. A solution is a proof tree in the sense of Section 2. This standardized view on static program analyses opens new possibilities concerning their uniform treatment, for example, in the mechanical verification of program properties, because the notion of a proof tree defines uniquely which interpretations a specification may have.

6.1 Type and Effect Systems

A *type and effect system* has judgments of the form $S : T \overset{\mathcal{F}}{\to} T'$ where S is a program fragment, T and T' are some types, and $\mathcal{F}$ is a set of effects giving information on the execution of S, for example, exceptions that might be raised during execution of S or new objects that are created during execution of S. The notion of type is more general than usual and includes every kind of static information, for example, reaching definitions, available expressions, etc. In this case, T denotes the information before execution of S and T' denotes that information after execution of S. Usually type and effect systems are described by means of inference rules. For more applications see, for example, Nielson et al. [1999]. Our goal is to model type and effect systems with sorted natural semantics. Consequently, we can derive automatically constraints from such a specification which can be solved by a suitable constraint solver being able to compute fixed-points.

In order to model type and effect systems with sorted natural semantics, the sorts of the types and effects have to be given, the judgments have to be converted into judgments of sorted natural semantics, and finally the inference rules of the type and effect systems have to be converted into inference rules of sorted natural semantics.

If a type and effect system has judgments of the form $S : \mathsf{T} \overset{\mathcal{F}}{\to} \mathsf{T}'$, we have to provide sort definitions $\mathbf{T}$ and $\mathbf{T}'$ for types T and T', respectively, a sort definition $\mathbf{E}$ for the effects $\mathcal{F}$, and a function definition for any function f used in the inference rules of the type and effect system. We model the above judgment of a type and effect system by the following judgment of sorted natural semantics:

$$\mathsf{T} :: \mathbf{T} \vdash S : \mathsf{T}' :: \mathbf{T}', \mathcal{F} :: \mathbf{E}.$$

The inference rules of sorted natural semantics are obtained by converting each judgment in the inference rules of the type and effect system into a judgment of the sorted natural semantics according to the above transformation.

Example 6.1. We formalize reaching definitions. The types are sets of definitions that may reach a program point. A *definition* is a pair $\langle x, l \rangle$ where x is a variable and l refers to the assignment defining this variable. Any assignment to a variable x invalidates all definitions reaching that assignment. One says that an assignment to x *kills* all definitions for x. The effects are the variables

Production $prog ::= stats$

$$\frac{stats : \emptyset \xrightarrow{\mathcal{F}} \mathsf{RD}}{prog : \emptyset \xrightarrow{\mathcal{F}} \mathsf{RD}}$$

Production $stats ::= \varepsilon$

$$stats : \mathsf{RD} \xrightarrow{\emptyset} \mathsf{RD}$$

Production $stats_0 ::= stat; stats_1$

$$\frac{\begin{array}{c} stat : \mathsf{RD}_1 \xrightarrow{\mathcal{F}_0} \mathsf{RD}_2 \\ stats_1 : \mathsf{RD}_2 \xrightarrow{\mathcal{F}_1} \mathsf{RD}_3 \end{array}}{stats_0 : \mathsf{RD}_1 \xrightarrow{\mathcal{F}_0 \cup \mathcal{F}_1} \mathsf{RD}_3}$$

Production $stat ::= var := expr$

$$\mathsf{RD} \xrightarrow{\{var.id\}} kill(\mathsf{RD}, var.id) \uplus \{\langle var.id, l\rangle\}$$

Production $stat ::=$ **if** $expr$ **then** $stats_1$ **else** $stats_2$

$$\frac{\begin{array}{c} stats_1 : \mathsf{RD}_1 \xrightarrow{\mathcal{F}_0} \mathsf{RD}_2 \\ stats_2 : \mathsf{RD}_1 \xrightarrow{\mathcal{F}_1} \mathsf{RD}_3 \end{array}}{stat : \mathsf{RD}_1 \xrightarrow{\mathcal{F}_0 \cap \mathcal{F}_1} \mathsf{RD}_2 \cup \mathsf{RD}_3}$$

Production $stat ::=$ **while** $expr$ **do** $stats$

$$\frac{stats : \mathsf{RD} \xrightarrow{\mathcal{F}} \mathsf{RD}}{stat : \mathsf{RD} \xrightarrow{\emptyset} \mathsf{RD}}$$

Fig. 17. A type and effect system for reaching definitions.

that are definitely killed by a program fragment S. Figure 17 shows a type and effect system for a simple language. This example is a slight modification of the corresponding example in Nielson et al. [1999] (they also used reaching definition as effects while we use them as types). The type and effect system assumes that each assignment has a unique label l. *var.id* denotes the identifier associated with a variable. The variables that are definitely killed in a conditional statement are only those that are killed in the then-part and in the else-part. The reaching definitions are those that may reach the end of the then-part or else-part. An assignment to variable x kills all reaching definitions for x and introduces a new definition for x. The operation $kill(R, \mathsf{x})$ deletes all definitions of x from R. The loop leads to a recursive equation on sets of definitions. Since the loop may not be executed at all, it cannot be guaranteed that there is a variable being killed by a loop.

In the first step we introduce the sorts and defined functions. We define the labels for assignments as natural numbers. The inference rules of Figure 17 require that a unique label be associated with each assignment. This association has to be modeled using sorts. One sort contains the maximal label assigned so far. The other sort contains the maximal label assigned in a statement. A definition is pair of variable name (which is a string) and a label. Reaching definitions at the beginning of a statement as well at the end of a statement are sets of definitions. The effects are sets of variables. The sort definitions in Figure 18 formalize these considerations.

$$
\begin{aligned}
\mathbf{LabIn} &= \mathbf{Nat}\\
\mathbf{LabOut} &= \mathbf{Nat}\\
\mathbf{Def} &= \mathbf{String} \times \mathbf{LabOut}\\
\mathbf{RDIn} &= \{\mathbf{Def}\}\\
\mathbf{RDOut} &= \{\mathbf{Def}\}\\
\mathbf{Eff} &= \{\mathbf{String}\}
\end{aligned}
$$

Fig. 18. Sorts used for reaching definitions.

We have to define the operation $kill : \mathbf{RDIn} \times \mathbf{String} \rightarrow \mathbf{RDOut}$:

$$kill(\emptyset, x) = \emptyset,$$
$$kill(R \uplus \{\langle x, l\rangle\}, x) = kill(R, x),$$
$$kill(R \uplus \{\langle y, l\rangle\}, x) = kill(R, x) \uplus \{\langle y, l\rangle\} \leftarrow x \neq y.$$

The other operations are classical functions on sets.

Finally, we have to transform the inference rules. A judgment in sorted natural semantics has the form

$$\mathsf{l} :: \mathbf{LabIn}, \mathsf{RD} :: \mathbf{RDIn} \vdash stats : \mathsf{l}' :: \mathbf{LabOut}, \mathsf{RD}' :: \mathbf{RDOut}, \mathcal{F} :: \mathbf{Eff},$$

where l is the maximal label used before *stats*, the RDs are the reaching definitions before *stats*, l' is the maximal label used after *stats*, the RDs' are the reaching definitions after *stats*, and $\mathcal{F}$ are the variables that are definitely killed in *stats*. Figure 19 shows the inference rules for reaching definitions in sorted natural semantics that are obtained from the type and effect system in Figure 17.

6.2 Data Flow Analysis

Data flow analyses are the classical form of static program analyses and implemented in many compilers. They regard programs in form of control flow graphs. Nodes of a control flow graph are the basic blocks which are connected by edges according to the control flow of the program. A typical data flow analysis specifies the control flow graphs of programs inductively over the program structure. Characteristic ingredients are definitions for the control flow of the program, for its basic blocks, for its initial and final blocks, and for whatever is important for the respective analysis. As an example, consider the available expressions analysis. It determines, for each program point, which expressions must have already been computed, and not later modified, on all paths to the program point. To define this analysis formally, one would need to define the set of expressions which are contained in a program and, based on it, the set of expressions which are available at a certain program point. Details of such data flow analyses can be found in textbooks on static program analysis [Nielson et al. 1999; Muchnick 1997]. The solution of data flow analyses can usually be determined iteratively. Thereby one needs to make sure that a unique minimal (or maximal, respectively), solution exists and can be found by suitable algorithms.

Production *prog* ::= *stats*

$$\frac{\underline{0} :: \mathbf{LabIn}, \emptyset :: \mathbf{RDIn} \vdash \mathit{stats} : \mathsf{l} :: \mathbf{LabOut}, \mathsf{RD} :: \mathbf{RDOut}, \mathcal{F} :: \mathbf{Eff}}{\underline{0} :: \mathbf{LabIn}, \emptyset :: \mathbf{RDIn} \vdash \mathit{prog} : \mathsf{l} :: \mathbf{LabOut}, \mathsf{RD} :: \mathbf{RDOut}, \mathcal{F} :: \mathbf{Eff}}$$

Production *stats* ::= ε

$$\mathsf{l} :: \mathbf{LabIn}, \mathsf{RD} :: \mathbf{RDIn} \vdash \mathit{stats} : \mathsf{l} :: \mathbf{LabOut}, \mathsf{RD} :: \mathbf{RDOut}, \emptyset :: \mathbf{Eff}$$

Production *stats*$_0$ ::= *stat*; *stats*$_1$

$$\frac{\begin{array}{c}\mathsf{l}_1 :: \mathbf{LabIn}, \mathsf{RD}_1 :: \mathbf{RDIn} \vdash \mathit{stat} : \mathsf{l}_2 :: \mathbf{LabOut}, \mathsf{RD}_2 :: \mathbf{RDOut}, \mathcal{F}_0 :: \mathbf{Eff} \\ \mathsf{l}_2 :: \mathbf{LabIn}, \mathsf{RD}_2 :: \mathbf{RDIn} \vdash \mathit{stats} : \mathsf{l}_3 :: \mathbf{LabOut}, \mathsf{RD}_3 :: \mathbf{RDOut}, \mathcal{F}_1 :: \mathbf{Eff}\end{array}}{\mathsf{l}_1 :: \mathbf{LabIn}, \mathsf{RD}_1 :: \mathbf{RDIn} \vdash \mathit{stats} : \mathsf{l}_3 :: \mathbf{LabOut}, \mathsf{RD}_3 :: \mathbf{RDOut}, \mathcal{F}_0 \cup \mathcal{F}_1 :: \mathbf{Eff}}$$

Production *stat* ::= *var* := *expr*

$$\frac{\vdash \mathit{var} : \mathsf{id} :: \mathbf{String}}{\mathsf{l} :: \mathbf{LabIn}, \mathsf{RD} :: \mathbf{RDIn} \vdash \mathsf{l} + 1 :: \mathbf{LabOut}, \mathit{kill}(\mathsf{RD}, \mathsf{id}) \uplus \{\langle \mathsf{id}, \mathsf{l} + 1 \rangle\} :: \mathbf{RDOut}, \{\mathsf{id}\} :: \mathbf{Eff}}$$

Production *stat* ::= **if** *expr* **then** *stats*$_1$ **else** *stats*$_2$

$$\frac{\begin{array}{c}\mathsf{l}_1 :: \mathbf{LabIn}, \mathsf{RD}_1 :: \mathbf{RDIn} \vdash \mathit{stats}_1 : \mathsf{l}_2 :: \mathbf{LabOut}, \mathsf{RD}_2 :: \mathbf{RDOut}, \mathcal{F}_0 :: \mathbf{Eff} \\ \mathsf{l}_2 :: \mathbf{LabIn}, \mathsf{RD}_1 :: \mathbf{RDIn} \vdash \mathit{stats}_2 : \mathsf{l}_3 :: \mathbf{LabOut}, \mathsf{RD}_3 :: \mathbf{RDOut}, \mathcal{F}_1 :: \mathbf{Eff}\end{array}}{\mathsf{l}_1 :: \mathbf{LabIn}, \mathsf{RD}_1 :: \mathbf{RDIn} \vdash \mathit{stat} : \mathsf{l}_3 :: \mathbf{LabOut}, \mathsf{RD}_2 \cup \mathsf{RD}_3 :: \mathbf{RDOut}, \mathcal{F}_0 \cap \mathcal{F}_1 :: \mathbf{Eff}}$$

Production *stat* ::= **while** *expr* **do** *stats*

$$\frac{\mathsf{l}_1 :: \mathbf{LabIn}, \mathsf{RD} :: \mathbf{RDIn} \vdash \mathit{stats} : \mathsf{l}_2 :: \mathbf{LabOut}, \mathsf{RD} :: \mathbf{RDOut}, \mathcal{F} :: \mathbf{Eff}}{\mathsf{l}_1 :: \mathbf{LabIn}, \mathsf{RD} :: \mathbf{RDIn} \vdash \mathit{stat} : \mathsf{l}_2 :: \mathbf{LabOut}, \mathsf{RD} :: \mathbf{RDOut}, \emptyset :: \mathbf{Eff}}$$

Fig. 19. Inference rules in sorted natural semantics for reaching definitions.

Monotone frameworks as introduced in Nielson et al. [1999] summarize the common characteristics of classical data flow analyses. They can be instantiated with particular data flow analyses. In particular, one distinguishes between forward versus backward analyses and between *may* versus *must* analyses. A forward analysis takes the control flow as specified by the operational semantics while a backward analysis reverses it. A *must* analysis considers only events which will be definitely true during program execution while *may* analyses also take those events into account which may happen but do not need to become true.

Data flow analyses which fit to the monotone framework have a common characteristic: they define static semantic information for programs whereby the definitions are given along the structure of the abstract syntax trees of the programs. This implies that they can be directly stated within the framework of sorted natural semantics. Thereby it is necessary to define a sort which points to nodes in the abstract syntax tree, as it was done already in Section 6.1 with the sorts **LabIn** and **LabOut**.

7. IMPLEMENTATIONS AND RESULTS

We have implemented the basic algorithm by employing the observation that it specifies directly a concurrent constraint program. As a test to get first runtime experiences, we realized it directly in the concurrent constraint programming language Oz. The insights gained during this test helped us in accomplishing an efficient Java prototype implementation.

7.1 Concurrent Constraint Solving

The basic algorithm for the semantic analysis specifies directly a program of a concurrent constraint programming language. The basic algorithm generates equality constraints which are solved by unification interleaved with the evaluation of defined functions. Thereby it is an essential feature that also partial information can be exploited. For example, if the first component of a tuple is known, then its value can contribute to the ongoing computation while the final evaluation of the entire tuple is postponed. Furthermore, it can happen that several rule covers are possible if more than one inference rule exists for a single production. Therefore, a running computation must be split into independent ones considering the alternative constraint sets. Such constraint systems can be expressed within concurrent constraint programming languages. One such programming language is Oz [Oz n.d.].

Computations within the *Oz programming model (OPM)* take place in *computation spaces* consisting of *constraints* which are logical formulas and a common *constraint store*. The computation goes on by solving constraints. The order in which constraints are solved can be determined by *threads*. If the constraints in a given computation space cannot be solved entirely, then additional assumptions can be made and solutions can be found by a search. Therefore, a constraint C must be chosen and the original problem P is separated into the two complementary constraint problems $P_1 = P \wedge C$ and $P_2 = P \wedge \neg C$ such that P is equivalent to $P_1 \wedge P_2$. P_1 and P_2 are solved independently from each other by creating a new computation space for each of them.

We have test-implemented the basic algorithm [Geiß 1998] using Oz in order to gain insights concerning the behavior of this principle in practice. The Oz implementation takes a program as input, generates its constraints, transforms them into an Oz intern representation, and solves them. Each solution process takes place in a separate computation space. Instead of looking at all possible rule covers separately, we dynamically create new computation spaces whenever otherwise a solution cannot be determined uniquely, as discussed in Section 3.2. In doing so, we can eliminate rule covers dynamically. Even though the theoretical complexity of analyzing alternative rule covers is still exponential, this case did not show up in our experiments where the number of computation spaces is linear in the program size. In Figure 20, we visualize this number of computation spaces examined during the semantic analyses. These experimental results indicate that an efficient implementation of the basic algorithm is possible.

7.2 Prototype Implementation

Our prototype implementation in Java [Geiß 1999] follows the same principles as the idealized Oz implementation. The main difference is that the Oz implementation of the basic algorithm does not need to specify any solution order for the constraints because the underlying Oz constraint solver cares for it. In the Java implementation we need to define this solution order explicitly. Of course any random order would do but we decided to choose one fitting to the context-sensitive nature of programming languages. The Java implementation tries to

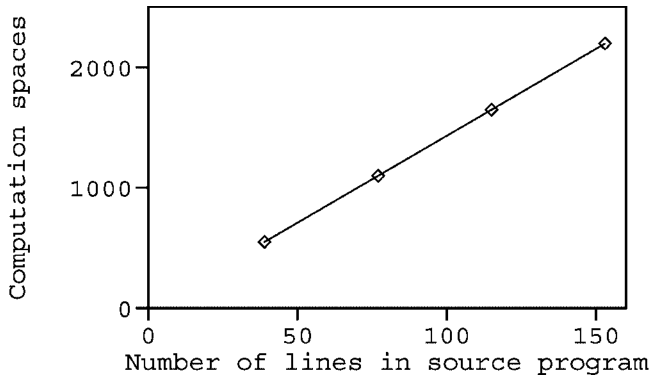

Fig. 20. Experiments: number of computation spaces.

Table II. Experimental Results for the Oz and Java Implementation

Program	Oz implementation			Java implementation	
Lines of code	Time [s]	Memory [MB]	# Comp. spaces	Time [s]	Memory [MB]
39	44.1	10	550	4.6	5
77	134.9	25	1099	5.5	5
115	289.1	52	1648	6.3	6
153	552.8	92	2197	7.0	6
191				8.3	6
229				9.2	7

solve the constraints in, possibly several, left to right depth first traversals of the abstract syntax tree. This heuristic assumes that program entities are declared before used in most of the cases. Nevertheless, it is a general solution strategy which is able to solve any constraint systems arising during semantic analysis. The abstract syntax tree is traversed as many times as necessary until nothing changes. It stops when all constraints are solved or when no more constraints can be solved. Besides this change, everything else is implemented basically as in the Oz implementation but without the overhead of the Oz system. The experimental results of this Java implementation concerning time and memory consumption when analyzing Mini-Java programs are given in Table II (when executed on an Intel Pentium with 133 MHz) and are also visualized, together with the results of the Oz test implementation, in Figures 21 and 22. We have also listed the corresponding results of the Oz test implementation to show that the efficiency of the implementation language together with the solution heuristic for the constraints have an enormous effect on the running time as well as on the memory consumption. This comparison implies directly that Oz should not be used as prototype implementation language even though its computation model fits directly because its overhead is too large.

The prototype implementation in Java demonstrates the feasibility of the semantic analysis with the basic algorithm. The experimental results show that the basic algorithm can be used in practical applications. In particular, it shows that the overall complexity is often linear in practice. The theoretical

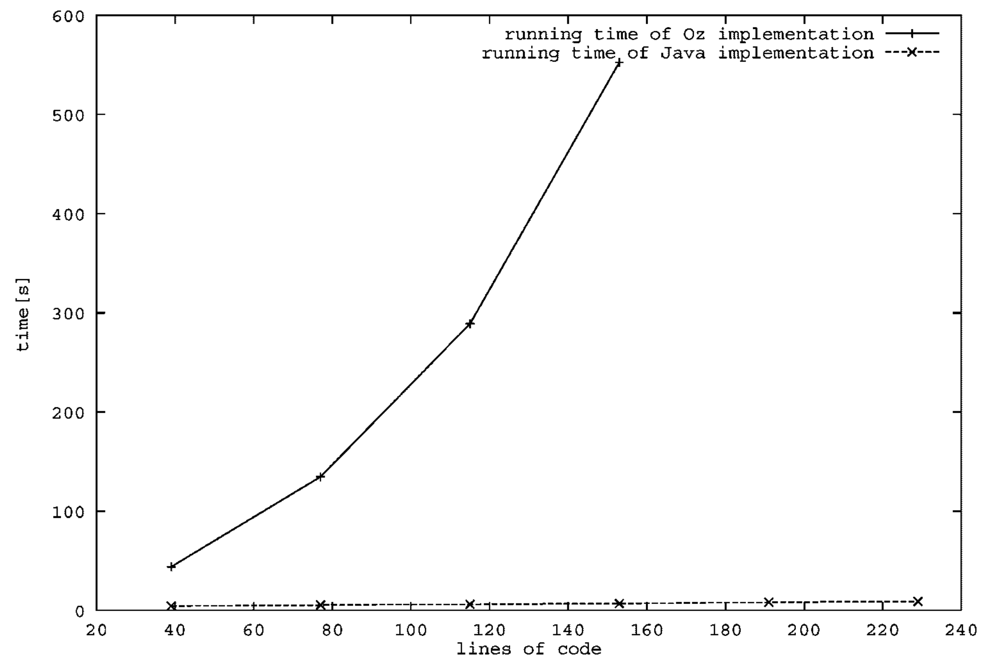

Fig. 21. Experiments: running time of the Oz and Java implementations.

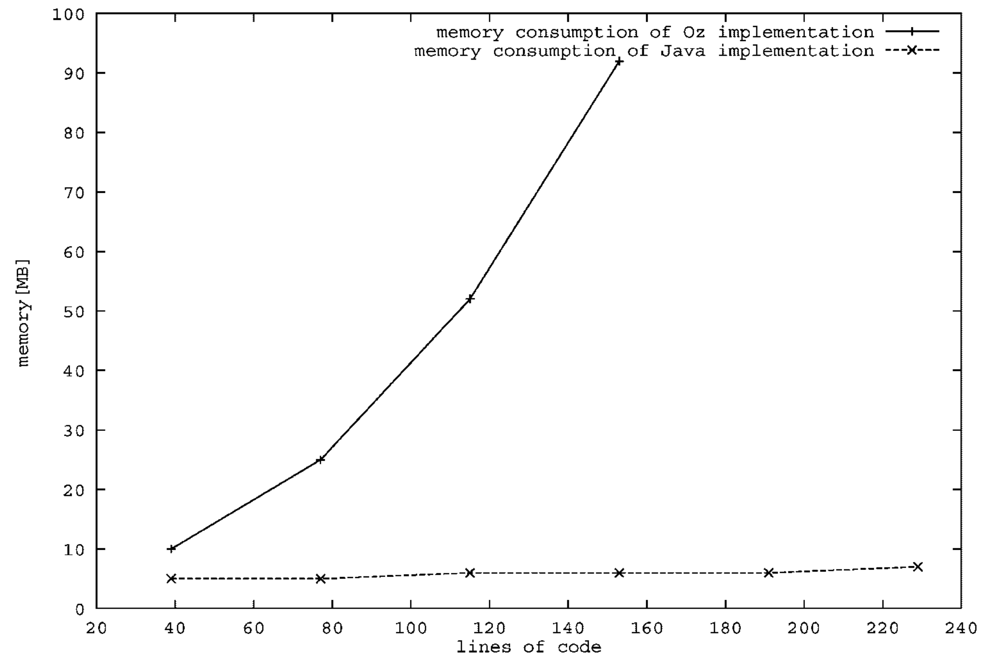

Fig. 22. Experiments: memory consumption of the Oz and Java implementations.

worst case, an exponential number of computation spaces, does not show up because the number of computation spaces grows linearly in the program size.

8. RELATED WORK

The static analysis is a phase in a compiler consisting of the semantic analysis followed by data and control flow analyses. The semantic analysis consists of name analysis (which declaration belongs to which use of a variable), type checking (determines the types of variables and expressions), and operator identification (which operation is denoted by which operator). Data and control flow analyses determine abstractions of run-time properties by fixed-point computations. In this section, we characterize related approaches for the specification of static semantic properties and the generation of the corresponding analyses.

8.1 Attribute Grammars

Attribute grammars [Knuth 1968, 1971] are used to describe the static semantics of programming languages. They associate attributes with the nodes of a program's syntax tree to express static semantic information. Functional dependencies between the attribute values within one production of the underlying context-free grammar are specified by attribution rules. The AG-conditions of the attribution rules describe consistency requirements on the attribute values which must be fulfilled by a correct attribution. An attribute grammar is well-defined if, for each correct program, the attribution is unique and computable. Since attribute grammars with a single attribute per node are sufficiently powerful, it follows directly that all well-defined attribute grammars have the same expressiveness, namely Turing-completeness. For efficiency reasons, subclasses of attribute grammars have been investigated: ordered attribute grammars [Kastens 1980] allow for the computation of the attribute values in a fixed evaluation order which is inferred from the specification. Thereby it can be checked in polynomial time whether a given attribute grammar is ordered. LAG(k)-attribute grammars require that the attributes can be evaluated in k depth-first left-to-right traversals of the abstract syntax tree [Lewis et al. 1974; Bochmann 1976]. Attribute grammars have been applied successfully in generators for the semantic analysis. The compiler generator system ELI [Eli n.d.] and the tool box Cocktail [Grosch and Emmelmann 1990] employ attribute grammars for the semantic analysis.

Altogether, attribute grammars have been proven to be a suitable method to specify the static semantics of imperative and object-oriented programming languages. Descriptions are modular because different semantic aspects of a language are defined independently from each other by different attributes. Furthermore, the semantic analysis can be generated such that the resulting analyses are efficient. Unfortunately, attribute grammars are not declarative because, when writing a specification, one must already plan in which order the attribute values need to be computed, thus leading to specifications which are more complicated than necessary and, hence, hard to use in practice. Their most severe drawback is their restriction to analyses which can be solved directly without fixed-point computations.

8.2 Natural Semantics

Sorted natural semantics is a sorted version of natural semantics [Kahn 1987]. Natural semantics takes advantage of the semantic compositionality of programming languages, which implies that the meaning of a program is always composed from the meanings of its direct subprograms. Natural semantics is not only able to specify static semantic properties but can also be used to define the operational semantics of programming languages. However, this is not within the scope of this paper.

During the last decade, natural semantics has been used for the specification of numerous programming languages. The most prominent representative is the complete description of the static and dynamic semantics of Standard-ML [Milner et al. 1990, 1997; Milner and Tofte 1991]. This specification has turned out to be very stable. Its revision shows that it has not contained any serious errors because most of the corrections affected the programming language's design but not mistakes in the specification itself. The inference rules for the type system are not completely structural because of specialization and generalization rules. These rules cannot be formalized directly in natural semantics. However, there are structural versions that can be formalized directly using natural semantics [Kfoury et al. 1994]. Further language specifications were stated at the French research institution INRIA, for example, of the dynamic semantics of Eiffel [Attali et al. 1996]. The investigations in Drossopoulou and Eisenbach [1999], Igarashi et al. [1999], Syme [1999], Nipkow and von Oheimb [1998], von Oheimb and Nipkov [1999], and von Oheimb [2001] proved the static type safety of subsets of the Java programming language based on natural semantics specifications. Other work on object-oriented calculi abstracted from a concrete programming languages and investigated typing rules for these calculi [Abadi and Cardelli 1996; Castagna 1997]. Their typing rules are also inference rules on the structure of terms of their calculi. This shows that specifications in natural semantics are convenient to formally verify properties of programming languages. In summary, natural semantics is a well-suited framework to specify imperative and object-oriented programming languages.

We note two implementations of natural semantics, the Typol [Despeyroux 1984] and the RML [Pettersson 1995, 1996] implementation. In Typol, inference rules are regarded as Prolog clauses. The semantic analysis of a program tries to find semantic information for its root node by using the Prolog search engine. Hence, the program is traversed in a single left-to-right depth-first traversal. This is analogous to the strategy of LAG(1)-attribute grammars but more powerful since Typol uses the Prolog unification algorithm. Theoretically, it is sufficient to allow a single synthesized attribute per node in the syntax trees to specify any static semantics. But in most cases, the specifications are much more declarative and readable if more flexibility and further attributes are allowed. This holds for Typol specifications as well; since Prolog and Typol are Turing-complete frameworks, every static semantics can be specified.

Nevertheless, the following example demonstrates that such specifications can become more cumbersome than necessary and desirable: think of a block-structured programming language which requires that variables are declared but allows their use before their definition. (The language DEMO in Appendix B is such a language and is discussed in detail throughout this paper.) To specify the name analysis of such a language in Typol, one needs to define two different data structures (e.g., based on lists) which collect the declarations within the block (the declaration list) as well as the variables which are used without a local declaration in the current block (the usage list). When starting to analyze a block, both lists are empty. If the use of a variable is found and if this variable has not been declared before in the block, then it is put in the usage list. If it has been declared before, nothing needs to be done. If a variable declaration is found, it is put in the declaration list. Furthermore, if this variable has been put in the usage list before, it is removed from the usage list. At the end of a local block, all its declarations are known as well as all variables which are used without a local declaration and for which a declaration must exist in the outer block. Hence, to specify the name analysis, it is necessary to define two different context attributes.

This example demonstrates a typical situation in object-oriented programming languages where all attributes and methods of a class are known in the entire class body independent of the order of definitions. In this paper, we show how specifications for such situations can be given more declaratively by using only one context attribute whereby the semantic analysis can still be generated automatically. Besides its restricted specification possibilities, Typol has also another disadvantage because its implementation is Prolog-based and inefficient. Therefore, the transformation of a subset of the Typol specifications into attribute grammars has been investigated [Attali and Franchi-Zannettacci 1988; Attali 1989] in order to evaluate them efficiently without unification. The RML specification also overcomes the inefficiency of the Typol implementation. Therefore, inference rules are regarded as procedures and the semantic information in the inference rules is strictly separated into arguments and into results. During the semantic analysis, the semantic information is computed in a left-to-right depth-first-traversal of the syntax tree. The RML implementation generates very efficient semantic analyses, as is demonstrated by many examples. But still, there are the same problems concerning flexibility and declarativity as in the Typol implementation.

Concluding, we see that natural semantics has stood the test of being a suitable specification method for the static semantics of programming languages. Since specifications are modular, natural semantics is also applicable for imperative and object-oriented languages. Nevertheless, the implementations of natural semantics have not kept abreast of this development. Though there are efficient implementations as RML, the corresponding restricted specification possibilities are not sufficiently declarative and flexible and especially not suited for object-oriented programming languages. In this paper, we present sorted natural semantics which can be used to specify object-oriented programming languages declaratively, demonstrated with the example specification of Mini-Java. When generating the semantic analysis with the basic algorithm, we do not assume a fixed traversal strategy of the syntax tree, but instead we regard the specification as a constraint system on the semantic information of a program. During the semantic analysis, these constraints are generated and solved. With this concept, we clearly exceed previous implementations of natural semantics. Moreover, we have established natural semantics as being sufficiently expressive to define fixed-point analyses.

8.3 Fixed-Point Analyses

Static program analyses are abstractions from the dynamic semantics of programs that can be determined statically. The classical approach toward static program analyses used in compilers are monotone frameworks that allow solving systems of equations over lattices by fixed-point computations. Classical compiler optimizations such as, for example, reaching definitions (definitions of variables that may reach a program point), available expressions (expressions whose value is available at a program point), copy propagation (propagates copy assignment $x := y$), constant propagation, etc., can be modeled by an equation system over lattices [Muchnick 1997; Morgan 1998]. We have shown in Section 6 on the example of reaching definitions that these analyses can be formalized using natural semantics.

Other approaches for static analysis of programs include abstract interpretation [Cousot and Cousot 1977], constraint-solving approaches [Palsberg and Schwartzbach 1994], and type and effect systems [Jouvelot and Gifford 1991]. The textbook by Nielson et al. [1999] shows a unified view on all these different approaches and discusses their commonalities from a static analysis point of view. However, these analyses are not stated in a uniform framework and their focus does not lie on the analysis of context-sensitive program properties.

8.4 Further Approaches

There are approaches describing the static semantics of programming languages based on predicate logic. In Odersky [1993] a specification defines a set of predicate logic formulas for each program. During the semantic analysis, these formulas are generated. Then it is checked whether the program is a model of these formulas, in which case it is statically semantically correct. Since this test is not decidable, the specification language must be restricted, whereupon the test becomes NP-complete. This approach is, as the author remarks, not useful in compilers but only in prototypes during the development of new programming languages. It has been implemented and tested by generating the semantic analysis for Oberon. Altogether, we characterize this specification method as declarative because predicate logic formulas are used. It is applicable for imperative and object-oriented programming languages since the descriptions are modular. The drawback lies in the inefficiency of the generated semantic analyses and we do not foresee any possibility reducing the complexity in principle. We think that this method of describing the semantics of programming languages by predicate logic is not tailored to the problem because the structure of programs, given by the abstract syntax, is not exploited.

Context relations [Snelting and Henhapl 1986] describe the static semantics of programming languages via sorted inference rules. During the semantic analysis, the syntax tree of a program is traversed bottom-up, thereby collecting for each node the relation of still possible combinations of values for its attributes. These relations are transferred to the respective predecessors in the syntax tree, thereby combining relations from different children nodes with the natural join operation. An error occurs if no elements remain as possible values. Context relations have been implemented in the PSG system (Programming System Generator) [Bahlke and Snelting 1986]. Context relations are a declarative and modular approach which allows for the generation of efficient semantic analyses. Nevertheless, they regard the name analysis as a preprocessing step which should be done during the syntactic analysis. This view does not hold for object-oriented languages: for example, if a subclass defines a method with the same name as a method of its superclass, then it is not clear a priori whether the method of the superclass is overridden by this new method or not. This depends on the inheritance relation between the parameter types of the two methods, which is not known during the syntactic analysis. Therefore context relations are not applicable for object-oriented programming languages. Moreover, context relations are not able to express fixed-point analyses because of their bottom-up nature.

9. CONCLUSIONS AND FUTURE WORK

We have presented sorted natural semantics, a declarative specification method suitable to define static program analyses by using axioms and inference rules. In particular, we have established natural semantics as being sufficiently expressive to define fixed-point analyses. We have shown that it is possible to generate static analyses from these specifications without loosing their declarativity. Due to the sortedness, the specifications are modular because the sorts allow for an independent definition of different kinds of semantic information. We have demonstrated the ability of sorted natural semantics to express two principal kinds of static analyses: semantic analysis which computes context-sensitive properties of programs and fixed-point analyses which compute abstractions of run-time properties of programs. We have defined solutions of analyses formally based on the notion of proof trees. In particular, we have shown that they can be computed by solving an equivalent set of constraints by residuation. Hence, when generating implementations, we regard the specification as a constraint-producing system and generate a set of constraints for each program under consideration. In the case of the semantic analysis, these constraints can be solved directly with the basic algorithm. In special cases, more efficient solution strategies are possible, such as for example, the LNS(1) strategy which solves the constraints in a left-to-right depth-first traversal. For more general program analyses computing abstractions of run-time properties, fixed-point algorithms are necessary. We have implemented the basic algorithm in a prototype which shows its applicability in practical situations. In contrast to other implementations of natural semantics such as Typol or RML, our method offers more degrees of freedom when writing specifications. This yields better understandable descriptions because one does not need to consider a possible evaluation strategy. Therewith we clearly exceed hitherto existing implementations of natural semantics because in the basic algorithm we can utilize partial information to compute the overall solution of the constraints. In comparison with attribute grammars as a standard specification and generation method for the semantic analysis, sorted natural semantics is clearly more expressive. Attribute grammars are not able to handle fixed-point analyses at all. This means that we do not lose any descriptive power nor the ability to generate semantic analyses compared to attribute grammars. Instead, we demonstrated that we gain power of expressiveness. Depending on the specification, a solution of the constraints can be found efficiently with a solution strategy or requires a general-solution algorithm as the basic algorithm or even a fixed-point algorithm.

The approach to semantic analysis demonstrates that special requirements stemming from the task of semantic analysis have helped us to define efficient solution strategies. Although we have not yet investigated similar considerations for fixed-point analyses, this should also be possible when they represent a monotonous framework. If the defined functions are marked as monotonous with respect to a lattice, then it should be possible to analyze the constraints and solve them using fixed-point iterations. Due to the experience in regard semantic analysis, we are optimistic that it is possible to derive from such specifications whether a backward- or forward-analysis has to be performed. Besides, the direction does not make any difference to the solution algorithm because the constraints are solved independently from the abstract syntax tree.

There are also application domains for the presented results outside the area of programming languages. In comparison with previous implementations of natural semantics, the basic algorithm offers an important advantage. It does not take advantage of the tree structure of the abstract syntax tree and could therefore also be used in application domains where the specified systems do not necessarily show tree structure. One such domain is component-based software systems where the relation between the components can be described by a graph structure. In future work, we want to apply the presented results to such problems.

APPENDIX A. SPECIFICATION OF MINI-JAVA

Mini-Java is a small object-oriented programming language featuring many Java characteristics such as concrete classes, inheritance, and polymorphism. In Mini-Java, it is not possible to overload features. Furthermore, there are no static features, no interfaces, no threads, and no privacy concepts. For simplicity, methods contain only one parameter. The imperative kernel is reduced to assignments and while loops (in Pascal-style syntax). Figure 23 shows the syntax of Mini-Java. Methods with more than one parameter and other imperative language constructs can easily be added but do not provide new insights. This appendix introduces the static semantics of Mini-Java.

Remark A.1. The distinction between the types *feature_type* and *decl_type* is necessary because the sort of the context of the judgment for productions

prog	::=	*classes*; *main*	(A.1)
$classes_0$	::=	*class*; $classes_1$	(A.2)
classes	::=	;	(A.3)
class	::=	`class` id_1 `extends` id_2; *features* `end`	(A.4)
class	::=	`class` id; *features* `end`;	(A.5)
main	::=	`class main`; *features* `end`	(A.6)
$features_0$	::=	*feature*; $features_1$	(A.7)
features	::=	;	(A.8)
feature	::=	id : *feature_type*	(A.9)
feature	::=	`method` id_1 (id_2 : $feature\_type_1$) : $feature\_type_2$; *block*	(A.10)
feature_type	::=	id	(A.11)
block	::=	`begin` *decls stats* `end`	(A.12)
$decls_0$	::=	id : *decl_type*; $decls_1$	(A.13)
decls	::=	;	(A.14)
decl_type	::=	id	(A.15)
$stats_0$	::=	*stat*; $stats_1$	(A.16)
stats	::=	;	(A.17)
stat	::=	*des* := *expr*	(A.18)
stats	::=	`while` *expr* `do` *stats* `od`	(A.19)
des_0	::=	des_1.id	(A.20)
des	::=	id	(A.21)
des_0	::=	des_1.id(*expr*)	(A.22)
des	::=	id(*expr*)	(A.23)
expr	::=	*des*	(A.24)
expr	::=	`new` id	(A.25)

Fig. 23. Syntax of Mini-Java.

Type	=	**String**
Types	=	{**Type**}
Subtyping	=	{$\sqsubseteq$ (**Type**, **Type**)}
Signature	=	**String** × **Type** × **Type**
Feature_Indicator	=	{<u>meth</u>, <u>attr</u>}
Cl_Env_Item	=	**Feature_Indicator** × **Signature**
Cl_Env	=	{**Cl_Env_Item**}
Env_Item	=	**Type** × **Cl_Env**
Env	=	{**Env_Item**}
Parameter_Indicator	=	{<u>loc</u>, <u>ret</u>, <u>inp</u>}
Local	=	**Parameter_Indicator** × **String** × **Type**
Locals	=	{**Local**}
$\mathbf{Context}_1$	=	**Types** × **Subtyping** × **Env**
$\mathbf{Context}_2$	=	$\mathbf{Context}_1$ × **Type**
$\mathbf{Context}_3$	=	$\mathbf{Context}_2$ × **Locals**
Des_Kind	=	{<u>lvalue</u>, <u>rvalue</u>}

Fig. 24. Sorts used in the specification for Mini-Java.

(A.9) and (A.10) differs from the context in the judgment for production (A.13). If there were just one nonterminal *type*, then productions (A.11) and (A.15) would be the same. Since the contexts of the judgments for *feature_type* and *decl_type* are different, and there can be at most one sorting for the judgment of a nonterminal, the specification would not be well-defined.

Figure 24 shows the sort definitions used for the specification of the static semantics of Mini-Java. There are three different levels of contexts: one on the level of classes, one within the level of classes, and one within the level of

$\mathit{overridden}$:	$\mathbf{Subtyping} \times \mathbf{Signature} \times \mathbf{Signature}$	$\rightarrow \mathbf{Bool}$
$\mathit{not\_overridden}$:	$\mathbf{Subtyping} \times \mathbf{Signature} \times \mathbf{Signature}$	$\rightarrow \mathbf{Bool}$
Thru :	$\mathbf{Subtyping} \times \mathbf{Cl\_Env} \times \mathbf{Cl\_Env}$	$\rightarrow \mathbf{Cl\_Env}$
void :		$\rightarrow \mathbf{Type}$
$\mathit{unique}_{\mathbf{Cl\_Env}}$:	$\mathbf{Cl\_Env\_Item}$	$\rightarrow \mathbf{List}$
$\mathit{unique}_{\mathbf{Env}}$:	$\mathbf{Env\_Item}$	$\rightarrow \mathbf{List}$
$\mathit{unique}_{\mathbf{Locals}}$:	$\mathbf{Local}$	$\rightarrow \mathbf{List}$

Fig. 25. Defined functions used in the specification for Mini-Java.

methods. The first context contains information on the types of the program, its subtype hierarchy, and on the features of the classes of the program. Concerning the signature of features, there are in principle two kinds, one for methods and one for attributes. Attributes have no argument type. We represent both by using semantic information of sort **Signature**. For attributes, the two items of type information are the same. When analyzing a class, the second context additionally contains information on the class currently being considered. The context for analyzing method bodies contains additionally the local variables and parameters of the method. The information on local variables requires also information on whether the variable is a local variable, an input parameter, or a return parameter. Finally, we need to distinguish whether it is possible to assign a value to designator ($\underline{\mathsf{lvalue}}$ in the positive case; $\underline{\mathsf{rvalue}}$ in the negative case). For example, if a designator refers to a method, it is impossible to assign a value to it.

The static semantics requires the defined functions shown in Figure 25. Their meaning is defined by the Horn clauses in Figure 26.

Remark A.2. Sorts are denoted in **bold face**, constructors are $\underline{\mathsf{underlined}}$, defined functions are written in *italic* style, and variables are written $\mathsf{sans\ serif}$.

The Horn clauses ((A.G1)–(A.G7) in Figure 26) define a notion of equality on the elements of class environments, environments, and sets of local variables. The basic algorithm is able to use special functions *unique* in order to define equality of elements of sets based on keys. Just for specification, the use of the functions *unique* would not be necessary. However, without them, a feasible semantic analysis would be impossible.

The constructor $\sqsubseteq$ is special in the sense that it has also Horn clauses for describing the transitivity (A.G8) of the subtype relation, that is, it implicitly refines the element-function.

The defined functions *overridden* and *not_overridden* specify the conditions when a method overrides another (cf. (A.G9) and (A.G10)). This is the case if and only if the methods have the same name and the same argument type and if the result type of the overriding method is greater or equal to the result type of the method being overridden. Finally, *Thru* filters from the set of methods of a current class those which are not overridden by the set of methods of the predecessor class (cf. (A.G11–(A.G14)). It takes three arguments. The first argument is the type hierarchy. The second argument contains the interfaces

$$unique_{\mathbf{Cl\_Env}}(\langle \underline{\mathsf{meth}}, \langle x, y, z\rangle\rangle) = [x] \tag{A.G1}$$

$$unique_{\mathbf{Cl\_Env}}(\langle \underline{\mathsf{attr}}, \langle x, y, z\rangle\rangle) = [x] \tag{A.G2}$$

$$unique_{\mathbf{Env}}(\langle x, y\rangle) = [x] \tag{A.G3}$$

$$unique_{\mathbf{Locals}}(\langle x, y, z\rangle) = [y] \tag{A.G4}$$

$$u = v \leftarrow unique_{\mathbf{Cl\_Env}}(u) = unique_{\mathbf{Cl\_Env}}(v). \tag{A.G5}$$

$$u = v \leftarrow unique_{\mathbf{Env}}(u) = unique_{\mathbf{Env}}(v). \tag{A.G6}$$

$$u = v \leftarrow unique_{\mathbf{Locals}}(u) = unique_{\mathbf{Locals}}(v). \tag{A.G7}$$

$$\begin{aligned} A \sqsubseteq C \in \mathsf{subtypes} \;\leftarrow\; & \mathsf{subtypes} :: \mathbf{Subtyping}, \\ & A \sqsubseteq B \in \mathsf{subtypes}, \\ & B \sqsubseteq C \in \mathsf{subtypes}. \end{aligned} \tag{A.G8}$$

$$\begin{aligned} & overridden(\mathsf{subtypes}, \langle \mathsf{id}, \mathsf{type}, \mathsf{res\_type}_1\rangle, \langle \mathsf{id}, \mathsf{type}, \mathsf{res\_type}_2\rangle) \;\leftarrow \\ & \quad \langle \mathsf{id}, \mathsf{type}, \mathsf{res\_type}_1\rangle :: \mathbf{Signature}, \\ & \quad \langle \mathsf{id}, \mathsf{type}, \mathsf{res\_type}_2\rangle :: \mathbf{Signature}, \\ & \quad \mathsf{subtypes} :: \mathbf{Subtyping}, \\ & \quad \mathsf{res\_type}_1 \sqsubseteq \mathsf{res\_type}_2 \in \mathsf{subtypes}. \end{aligned} \tag{A.G9}$$

$$\begin{aligned} & not\_overridden(\mathsf{subtypes}, \langle \mathsf{id}, \mathsf{type}, \mathsf{res\_type}_1\rangle, \langle \mathsf{id}, \mathsf{type}, \mathsf{res\_type}_2\rangle) \;\leftarrow \\ & \quad \langle \mathsf{id}, \mathsf{type}, \mathsf{res\_type}_1\rangle :: \mathbf{Signature}, \\ & \quad \langle \mathsf{id}, \mathsf{type}, \mathsf{res\_type}_2\rangle :: \mathbf{Signature}, \\ & \quad \mathsf{subtypes} :: \mathbf{Subtyping}, \\ & \quad \mathsf{res\_type}_2 \sqsubseteq \mathsf{res\_type}_1 \in \mathsf{subtypes}, \\ & \quad \mathsf{if}\ \mathsf{res\_type}_1 \neq \mathsf{res\_type}_2. \end{aligned} \tag{A.G10}$$

$$\begin{aligned} Thru(\mathsf{subtypes}, X, \emptyset) = X \;\leftarrow\; & \mathsf{subtypes} :: \mathbf{Subtyping}, \\ & X :: \mathbf{Cl\_Env}. \end{aligned} \tag{A.G11}$$

$$\begin{aligned} Thru(\mathsf{subtypes}, \emptyset, X) = \emptyset \;\leftarrow\; & \mathsf{subtypes} :: \mathbf{Subtyping}, \\ & X :: \mathbf{Cl\_Env}. \end{aligned} \tag{A.G12}$$

$$\begin{aligned} & Thru(\mathsf{subs}, \mathsf{Cl\_Env} \uplus \{\langle \mathsf{feat\_ind}, \mathsf{Intf}\rangle\}, X \uplus \{\langle \mathsf{feat\_ind}', \mathsf{Intf}'\rangle\}) = Thru(\mathsf{subs}, \mathsf{Cl\_Env}, X) \;\leftarrow \\ & \quad \mathsf{subs} :: \mathbf{Subtyping}, \\ & \quad \mathsf{feat\_ind} :: \mathbf{Feature\_Indicator},\ \mathsf{feat\_ind}' :: \mathbf{Feature\_Indicator}, \\ & \quad \mathsf{Intf} :: \mathbf{Signature},\ \mathsf{Intf}' :: \mathbf{Signature}, \\ & \quad \mathsf{Cl\_Env} :: \mathbf{Cl\_Env},\ X :: \mathbf{Cl\_Env}, \\ & \quad overridden(\mathsf{subs}, \mathsf{Intf}, \mathsf{Intf}'). \end{aligned} \tag{A.G13}$$

$$\begin{aligned} & Thru(\mathsf{subs}, \mathsf{Cl\_Env} \uplus \{\langle \mathsf{feat\_ind}, \mathsf{Intf}\rangle\}, X \uplus \{\langle \mathsf{feat\_ind}', \mathsf{Intf}'\rangle\}) = Thru(\mathsf{subs}, \mathsf{Cl\_Env} \uplus \{\mathsf{Intf}\}, X) \;\leftarrow \\ & \quad \mathsf{subs} :: \mathbf{Subtyping}, \\ & \quad \mathsf{feat\_ind} :: \mathbf{Feature\_Indicator},\ \mathsf{feat\_ind}' :: \mathbf{Feature\_Indicator}, \\ & \quad \mathsf{Intf} :: \mathbf{Signature},\ \mathsf{Intf}' :: \mathbf{Signature}, \\ & \quad \mathsf{Cl\_Env} :: \mathbf{Cl\_Env},\ X :: \mathbf{Cl\_Env}, \\ & \quad not\_overridden(\mathsf{subs}, \mathsf{Intf}, \mathsf{Intf}'), \\ & \quad \mathsf{if}\ \forall \mathsf{Intf}'' \in \mathsf{Cl\_Env}.not\_overridden(\mathsf{subs}, \mathsf{Intf}'', \mathsf{Intf}'). \end{aligned} \tag{A.G14}$$

Fig. 26. Horn clauses for Mini-Java.

of the current class while the third argument contains the interfaces of the predecessor class. Horn clause (A.G11) states that if the predecessor class has no methods, then nothing can be overridden and all methods of the current class will go through. If the current class has no methods, then nothing can go through, as stated in Horn clause (A.G12). Horn clause (A.G13) formalizes the situation that if there is a method in the predecessor class which overrides a method from the current class, then this method cannot go through (hence, it

Production (A.1): $prog ::= classes; main$

$$\frac{\begin{array}{c}\langle \mathsf{Names} \cup \{\underline{\mathsf{main}}, void\}, \mathsf{TH} \cup \{\underline{\mathsf{main}} \sqsubseteq \underline{\mathsf{main}}\}, \mathsf{Intfs}_1 \cup \mathsf{Intfs}_2\rangle :: \mathbf{Context}_1 \\ \vdash classes : \mathsf{Names} :: \mathbf{Types}, \mathsf{TH} :: \mathbf{Subtyping}, \mathsf{Intfs}_1 :: \mathbf{Env} \\ \langle \mathsf{Names} \cup \{\underline{\mathsf{main}}, void\}, \mathsf{TH} \cup \{\underline{\mathsf{main}} \sqsubseteq \underline{\mathsf{main}}\}, \mathsf{Intfs}_1 \cup \mathsf{Intfs}_2\rangle :: \mathbf{Context}_1 \\ \vdash main : \underline{\mathsf{main}} :: \mathbf{Type}, \{\underline{\mathsf{main}} \sqsubseteq \underline{\mathsf{main}}\} :: \mathbf{Subtyping}, \mathsf{Intfs}_2 :: \mathbf{Env}\end{array}}{\vdash prog : \underline{\mathsf{correct}} :: \mathbf{GenInfo}} \quad \text{(A.S1)}$$
$$\text{if } \forall A \sqsubseteq B \in (\mathsf{TH} \cup \{\underline{\mathsf{main}} \sqsubseteq \underline{\mathsf{main}}\}).A = B \vee B \sqsubseteq A \notin (\mathsf{TH} \cup \{\underline{\mathsf{main}} \sqsubseteq \underline{\mathsf{main}}\})$$

Production (A.2): $classes_0 ::= class; classes_1$

$$\frac{\begin{array}{c}\Gamma :: \mathbf{Context}_1 \vdash class : \mathsf{Name} :: \mathbf{Type}, \mathsf{TH}_1 :: \mathbf{Subtyping}, \mathsf{Intfs}_1 :: \mathbf{Env} \\ \Gamma :: \mathbf{Context}_1 \vdash classes_1 : \mathsf{Names} :: \mathbf{Types}, \mathsf{TH}_2 :: \mathbf{Subtyping}, \mathsf{Intfs}_2 :: \mathbf{Env}\end{array}}{\begin{array}{c}\Gamma :: \mathbf{Context}_1 \vdash classes_0 : \mathsf{Names} \cup \{\mathsf{Name}\} :: \mathbf{Types}, \mathsf{TH}_1 \cup \mathsf{TH}_2 :: \mathbf{Subtyping}, \\ \mathsf{Intfs}_1 \cup \mathsf{Intfs}_2 :: \mathbf{Env}\end{array}} \quad \text{(A.S2)}$$
$$\text{if } \forall (n :: \mathbf{Type}) \in (\mathsf{Names} :: \mathbf{Types}).\mathsf{Name} \neq n \wedge n \neq \underline{\mathsf{main}} \wedge \mathsf{Name} \neq \underline{\mathsf{main}}$$

Production (A.3): $classes ::=;$

$$\Gamma :: \mathbf{Context}_1 \vdash classes : \emptyset :: \mathbf{Types}, \emptyset :: \mathbf{Subtyping}, \emptyset :: \mathbf{Env} \quad \text{(A.S3)}$$

Production (A.4): $class ::= \texttt{class}\ \mathsf{id}_1\ \texttt{extends}\ \mathsf{id}_2; features\ \texttt{end}$

$$\frac{\begin{array}{c}\langle \mathsf{Names}, \mathsf{TH}, \mathsf{Intfs} \uplus \{\langle v(\mathsf{id}_2), \mathsf{Cl\_Intfs}_1\rangle\}, v(\mathsf{id}_1)\rangle :: \mathbf{Context}_2 \\ \vdash features : \mathsf{Cl\_Intfs}_2 :: \mathbf{Cl\_Env}\end{array}}{\begin{array}{c}\langle \mathsf{Names}, \mathsf{TH}, \mathsf{Intfs} \uplus \{\langle v(\mathsf{id}_2), \mathsf{Cl\_Intfs}_1\rangle\}\rangle :: \mathbf{Context}_1 \\ \vdash class : v(\mathsf{id}_1) :: \mathbf{Type}, \{v(\mathsf{id}_1) \sqsubseteq v(\mathsf{id}_1), v(\mathsf{id}_1) \sqsubseteq v(\mathsf{id}_2)\} :: \mathbf{Subtyping}, \\ \{\langle v(\mathsf{id}_1), Thru(\mathsf{TH}, \mathsf{Cl\_Intfs}_1, \mathsf{Cl\_Intfs}_2) \cup \mathsf{Cl\_Intfs}_2\rangle\} :: \mathbf{Env}\end{array}} \quad \text{(A.S4)}$$

Production (A.5): $class ::= \texttt{class}\ \mathsf{id}; features\ \texttt{end}$

$$\frac{\langle \mathsf{Names}, \mathsf{TH}, \mathsf{Intfs}, v(\mathsf{id})\rangle :: \mathbf{Context}_2 \vdash features : \mathsf{Cl\_Intfs} :: \mathbf{Cl\_Env}}{\begin{array}{c}\langle \mathsf{Names}, \mathsf{TH}, \mathsf{Intfs}\rangle :: \mathbf{Context}_1 \vdash class : v(\mathsf{id}) :: \mathbf{Type}, \{v(\mathsf{id}) \sqsubseteq v(\mathsf{id})\} :: \mathbf{Subtyping}, \\ \{\langle v(\mathsf{id}), \mathsf{Cl\_Intfs}\rangle\} :: \mathbf{Env}\end{array}} \quad \text{(A.S5)}$$

Production (A.6): $main ::= \texttt{class main}; features\ \texttt{end}$

$$\frac{\langle \mathsf{Names}, \mathsf{TH}, \mathsf{Intfs}, \mathsf{id}\rangle :: \mathbf{Context}_2 \vdash features : \mathsf{Cl\_Intfs} :: \mathbf{Cl\_Env}}{\begin{array}{c}\langle \mathsf{Names}, \mathsf{TH}, \mathsf{Intfs}\rangle :: \mathbf{Context}_1 \vdash main : \underline{\mathsf{main}} :: \mathbf{Type}, \{\underline{\mathsf{main}} \sqsubseteq \underline{\mathsf{main}}\} :: \mathbf{Subtyping}, \\ \{\langle \underline{\mathsf{main}}, \mathsf{Cl\_Intfs}\rangle\} :: \mathbf{Env}\end{array}} \quad \text{(A.S6)}$$
$$\text{if } \exists t_1, t_2 \in \mathsf{Names}.\langle \underline{\mathsf{meth}}, \langle \underline{\mathsf{Main}}, t_1, t_2\rangle\rangle \in \mathsf{Cl\_Intfs}$$

Fig. 27. Inference rules for productions (A.1)–(A.6).

is removed from the methods of the current class) and it cannot override any other method in the current class (hence, it is also removed from the method set of the predecessor class). Last, but not least, Horn clause (A.G14) reflects that, if a method from the predecessor class does not override any method from the current class, then it can be removed from the method set of the predecessor class. In summary, the results of the function *Thru* are those interfaces of the current class which are not overridden by the methods from the predecessor class. This defined function is needed when defining inheritance. *void* is a constant denoting a special type needed for the return type of procedures.

Finally, we discuss the inference rules for the productions. Figure 27 shows the inference rules for productions (A.1)–(A.6). The side condition of inference rule (A.S1) checks whether the type hierarchy derived from the program is acyclic. Inference rules (A.S2) and (A.S3) collect the information of the classes. The name of a class added must be unique and different from the other class names (cf. the side condition of (A.S2)). (A.S4) and (A.S5) summarize local information on classes. If a class B inherits a class A (cf. (A.S4)), then the inherited features that are not overridden also belong to B and a subtype relation is added. In any case, a type is a subtype of itself. The class $\underline{\mathsf{main}}$ contains a

Production (A.7): $\mathit{features}_0 ::= \mathit{feature}; \mathit{features}_1$

$$\frac{\begin{array}{c}\Gamma :: \mathbf{Context_2} \vdash \mathit{feature} : \mathsf{Intf} :: \mathbf{Cl\_Env\_Item} \\ \Gamma :: \mathbf{Context_2} \vdash \mathit{features}_1 : \mathsf{Cl\_Intfs} :: \mathbf{Cl\_Env}\end{array}}{\Gamma :: \mathbf{Context_2} \vdash \mathit{features}_0 : \{\mathsf{Intf}\} \cup \mathsf{Cl\_Intfs} :: \mathbf{Cl\_Env}} \tag{A.S7}$$

$$\begin{array}{l}\text{if } [\exists \mathsf{string} :: \mathbf{String}, \mathsf{t}_1, \mathsf{t}_2 :: \mathbf{Type}.\mathsf{Intf} = \langle \underline{\mathsf{meth}}, \langle \mathsf{string}, \mathsf{t}_1, \mathsf{t}_2 \rangle\rangle \Rightarrow \\ \quad \forall \langle \underline{\mathsf{meth}}, \langle \mathsf{string}', \mathsf{t}_1', \mathsf{t}_2' \rangle\rangle \in \mathsf{Cl\_Intfs}.\mathsf{string} \neq \mathsf{string}'] \wedge \\ \quad \exists \mathsf{string} :: \mathbf{String}, \mathsf{t} :: \mathbf{Type}.\mathsf{Intf} = \langle \underline{\mathsf{attr}}, \langle \mathsf{string}, \mathsf{t}, \mathsf{t} \rangle\rangle \Rightarrow \\ \quad \forall \langle \underline{\mathsf{attr}}, \langle \mathsf{string}', \mathsf{t}', \mathsf{t}' \rangle\rangle \in \mathsf{Cl\_Intfs}.\mathsf{string} \neq \mathsf{string}']\end{array}$$

Production (A.8): $\mathit{features} ::= ;$

$$\Gamma :: \mathbf{Context_2} \vdash \mathit{features} : \emptyset :: \mathbf{Cl\_Env} \tag{A.S8}$$

Production (A.9): $\mathit{feature} ::= \mathsf{id} : \mathit{feature\_type}$

$$\frac{\Gamma :: \mathbf{Context_2} \vdash \mathit{feature\_type} : \mathsf{t} :: \mathbf{Type}}{\Gamma :: \mathbf{Context_2} \vdash \mathit{feature} : \langle \underline{\mathsf{attr}}, \langle v(\mathsf{id}), \mathsf{t}, \mathsf{t} \rangle\rangle :: \mathbf{Cl\_Env\_Item}} \quad \text{if } \mathsf{t} \neq \mathit{void} \tag{A.S9}$$

Production (A.10) : $\mathit{feature} ::= \mathtt{method}\ \mathsf{id}_1(\mathsf{id}_2 : \mathit{feature\_type}_1) : \mathit{feature\_type}_2; \mathit{block}$

$$\frac{\begin{array}{l}\langle \mathsf{Names}, \mathsf{TH}, \mathsf{Intfs}, \mathsf{A} \rangle :: \mathbf{Context_2} \vdash \mathit{feature\_type}_1 : \mathsf{t}_1 :: \mathbf{Type} \\ \langle \mathsf{Names}, \mathsf{TH}, \mathsf{Intfs}, \mathsf{A} \rangle :: \mathbf{Context_2} \vdash \mathit{feature\_type}_2 : \mathsf{t}_2 :: \mathbf{Type} \\ \langle \mathsf{Names}, \mathsf{TH}, \mathsf{Intfs}, \mathsf{A}, \mathsf{locals} \cup \{\langle \underline{\mathsf{inp}}, v(\mathsf{id}_2), \mathsf{t}_1 \rangle, \langle \underline{\mathsf{ret}}, \underline{\mathsf{result}}, \mathsf{t}_2 \rangle\}\rangle :: \mathbf{Context_3} \\ \qquad\qquad \vdash \mathit{block} : \mathsf{locals} :: \mathbf{Locals}\end{array}}{\langle \mathsf{Names}, \mathsf{TH}, \mathsf{Intfs}, \mathsf{A} \rangle :: \mathbf{Context_2} \vdash \mathit{feature} : \langle \underline{\mathsf{meth}}, \langle v(\mathsf{id}_1), \mathsf{t}_1, \mathsf{t}_2 \rangle\rangle :: \mathbf{Cl\_Env\_Item}} \tag{A.S10}$$

$$\begin{array}{l}\text{if } v(\mathsf{id}_2) \neq \underline{\mathsf{result}} \wedge \forall \mathsf{x} :: \mathbf{Parameter\_Indicator}, \mathsf{y} :: \mathbf{String}, \mathsf{z} :: \mathbf{Type}. \\ \quad (\langle \mathsf{x}, \mathsf{y}, \mathsf{z} \rangle \in \mathsf{locals} \Rightarrow \mathsf{y} \neq v(\mathsf{id}_2))\end{array}$$

Production (A.11) : $\mathit{feature\_type} ::= \mathsf{id}$

$$\langle \mathsf{Names} \uplus \{v(\mathsf{id})\}, \mathsf{TH}, \mathsf{Intfs}, \mathsf{A} \rangle :: \mathbf{Context_2} \vdash \mathit{feature\_type} : v(\mathsf{id}) :: \mathbf{Type} \tag{A.S11}$$

Fig. 28. Inference rules for productions (A.7)–(A.11).

method Main (cf. inference rule (A.S6)). This method is executed when starting the execution of a program.

Remark A.3. The inference rules (A.S4) and (A.S5) use the function v (v = *value*) stemming from the lexical analysis. For every occurrence of nonterminal id, v returns the string of the corresponding identifier.

Figure 28 shows the inference rules for productions (A.7)–(A.11). These productions specify features of a class. Rules (A.S7) and (A.S8) collect the information from a list of features. The side condition of (A.S7) specifies that the name of a feature added to a list of features must be new. Rules (A.S9) and (A.S10) define the information of a single feature. It is a pair: the first component specifies whether the feature is an attribute or a method, the second component specifies the name of the feature and its signature. In case of methods the signature consists of the argument type and the return type. The side condition of (A.S10) specifies that the name of the parameter of a method must be different from result and from all local variables of the method. (A.S11) defines that the type of an attribute, the parameter type of a method, and the return type of a method must be available in the context, that is, occur as a class name of the program.

The remaining inference rules define the static semantics of method bodies. Figure 29 shows the inference rules for blocks, their declarations, and statements. The inference rule (A.S12) collects the local variables of a block and specifies that the context of the block is the same as the context of its declarations and its statements. The inference rules (A.S13) and (A.S14) collect the

Production (A.12) : $\mathit{block} ::= \texttt{begin}\ \mathit{decls}\ \mathit{stats}\ \texttt{end}$

$$\frac{\begin{array}{c}\Gamma :: \mathbf{Context}_3 \vdash \mathit{decls} : \mathsf{locals} : \mathbf{Locals}\\ \Gamma :: \mathbf{Context}_3 \vdash \mathit{stats} : \underline{\mathsf{correct}} :: \mathbf{GenInfo}\end{array}}{\Gamma :: \mathbf{Context}_3 \vdash \mathit{block} : \mathsf{locals} :: \mathbf{Locals}} \qquad \text{(A.S12)}$$

Production (A.13): $\mathit{decls}_0 ::= \mathsf{id} : \mathit{decl\_type}; \mathit{decls}_1$

$$\frac{\begin{array}{c}\Gamma :: \mathbf{Context}_3 \vdash \mathit{decl\_type} : \mathsf{t} :: \mathbf{Type}\\ \Gamma :: \mathbf{Context}_3 \vdash \mathit{decls}_1 : \mathsf{locals} :: \mathbf{Locals}\end{array}}{\Gamma :: \mathbf{Context}_3 \vdash \mathit{decls}_0 : \{\langle \underline{\mathsf{loc}}, v(\mathsf{id}), \mathsf{t}\rangle\} \cup \mathsf{locals} :: \mathbf{Locals}}$$
$$\text{if } \mathsf{t} \neq \mathit{void} \wedge v(\mathsf{id}) \neq \underline{\mathsf{result}} \wedge \forall \mathsf{x} :: \mathbf{Parameter\_Indicator}$$
$$\mathsf{y} :: \mathbf{String}, \mathsf{z} :: \mathbf{Type}. \langle \mathsf{x}, \mathsf{y}, \mathsf{z}\rangle \in \mathsf{locals} \Rightarrow \mathsf{y} \neq v(\mathsf{id}) \qquad \text{(A.S13)}$$

Production (A.14): $\mathit{decls} ::= ;$

$$\Gamma :: \mathbf{Context}_3 \vdash \mathit{decls} : \emptyset :: \mathbf{Locals} \qquad \text{(A.S14)}$$

Production (A.15): $\mathit{decl\_type} ::= \mathsf{id}$

$$\langle \mathsf{Names} \uplus \{v(\mathsf{id})\}, \mathsf{TH}, \mathsf{Intfs}, \mathsf{A}, \mathsf{locals}\rangle :: \mathbf{Context}_3 \vdash \mathit{decl\_type} : v(\mathsf{id}) :: \mathbf{Type} \qquad \text{(A.S15)}$$

Production (A.16): $\mathit{stats}_0 ::= \mathit{stat}; \mathit{stats}_1$

$$\frac{\begin{array}{c}\Gamma :: \mathbf{Context}_3 \vdash \mathit{stat} : \underline{\mathsf{correct}} :: \mathbf{GenInfo}\\ \Gamma :: \mathbf{Context}_3 \vdash \mathit{stats}_1 : \underline{\mathsf{correct}} :: \mathbf{GenInfo}\end{array}}{\Gamma :: \mathbf{Context}_3 \vdash \mathit{stats}_0 : \underline{\mathsf{correct}} :: \mathbf{GenInfo}} \qquad \text{(A.S16)}$$

Production (A.17): $\mathit{stats} ::= ;$

$$\Gamma :: \mathbf{Context}_3 \vdash \mathit{stats} : \underline{\mathsf{correct}} :: \mathbf{GenInfo} \qquad \text{(A.S17)}$$

Production (A.18): $\mathit{stat} ::= \mathit{des} := \mathit{expr}$

$$\frac{\begin{array}{c}\langle \mathsf{Names}, \mathsf{TH}, \mathsf{Intfs}, \mathsf{A}, \mathsf{locals}\rangle :: \mathbf{Context}_3 \vdash \mathit{des} : \mathsf{t}_1 :: \mathbf{Type}, \underline{\mathsf{lvalue}} :: \mathbf{Des\_Kind}\\ \langle \mathsf{Names}, \mathsf{TH}, \mathsf{Intfs}, \mathsf{A}, \mathsf{locals}\rangle :: \mathbf{Context}_3 \vdash \mathit{expr} : \mathsf{t}_2 :: \mathbf{Type}\end{array}}{\langle \mathsf{Names}, \mathsf{TH}, \mathsf{Intfs}, \mathsf{A}, \mathsf{locals}\rangle :: \mathbf{Context}_3 \vdash \mathit{stat} : \underline{\mathsf{correct}} :: \mathbf{GenInfo}}$$
$$\text{if } \mathsf{t}_2 \sqsubseteq \mathsf{t}_1 \in \mathsf{TH} \qquad \text{(A.S18)}$$

Production (A.19) $\mathit{stat} ::= \texttt{while}\ \mathit{expr}\ \texttt{do}\ \mathit{stats}\ \texttt{od}$

$$\frac{\begin{array}{c}\Gamma :: \mathbf{Context}_3 \vdash \mathit{expr} : \mathsf{t} :: \mathbf{Type}\\ \Gamma :: \mathbf{Context}_3 \vdash \mathit{stats} : \underline{\mathsf{correct}} :: \mathbf{GenInfo}\end{array}}{\Gamma :: \mathbf{Context}_3 \vdash \mathit{stat} : \underline{\mathsf{correct}} :: \mathbf{GenInfo}} \qquad \text{(A.S19)}$$

Fig. 29. Inference rules for productions (A.12)–(A.19).

information of declaration lists. The side condition of (A.S13) ensures that a new name is added to the list of local variables previously collected. The information on local variables consists of three components: the information that it is a local variable (distinguishing them from input parameters and return parameters), its name, and its type. Rule (A.S15) specifies that the type of a declaration must be defined in a program, that is, in the context of the declaration. Rules (A.S16) and (A.S17) specify that the context of statements in a statement list is the same as the context of a statement. Inference rule (A.S18) specifies the static semantics of an assignment, that is, the types of its left-hand side and right-hand side, respectively, and that it must be possible to assign an object to the designator of the left-hand side ($\underline{\mathsf{lvalue}}$). The side condition specifies that the type of the right-hand side of an assignment must be a subtype of the left-hand side. The inference rule (A.S19) for while loops does not specify that the control expression must be of Boolean type because Mini-Java does not offer basic types. Instead, the loop terminates when the control expression evaluates to the **nil**-object.

Figure 30 completes the specification of the static semantics of Mini-Java with the inference rules for designators and expressions. Inference rule (A.S20)

Production (A.20) $des_0 ::= des_1.\mathsf{id}$

$$\frac{\langle \mathsf{Names}, \mathsf{TH}, \mathsf{Intfs} \uplus \{\langle \mathsf{t}_1, \mathsf{Cl\_Intfs} \uplus \{\langle \underline{\mathsf{attr}}, \langle v(\mathsf{id}), \mathsf{t}_2, \mathsf{t}_2\rangle\rangle\}\rangle\}, \mathsf{A}, \mathsf{locals}\rangle :: \mathbf{Context_3} \vdash des_1 : \mathsf{t}_1 :: \mathbf{Type}, \mathsf{kind} :: \mathbf{Des\_Kind}}{\langle \mathsf{Names}, \mathsf{TH}, \mathsf{Intfs} \uplus \{\langle \mathsf{t}_1, \mathsf{Cl\_Intfs} \uplus \{\langle \underline{\mathsf{attr}}, \langle v(\mathsf{id}), \mathsf{t}_2, \mathsf{t}_2\rangle\rangle\}\rangle\}, \mathsf{A}, \mathsf{locals}\rangle :: \mathbf{Context_3} \vdash des_0 : \mathsf{t}_2 :: \mathbf{Type}, \mathsf{kind} :: \mathbf{Des\_Kind}} \qquad \text{(A.S20)}$$

Production (A.21) $des ::= \mathsf{id}$

$$\langle \mathsf{Names}, \mathsf{TH}, \mathsf{Intfs} \uplus \{\langle \mathsf{A}, \mathsf{Cl\_Intfs} \uplus \{\langle \underline{\mathsf{attr}}, \langle v(\mathsf{id}), \mathsf{t}, \mathsf{t}\rangle\rangle\}\rangle\}, \mathsf{A}, \mathsf{locals}\rangle :: \mathbf{Context_3} \vdash des : \mathsf{t} :: \mathbf{Type}, \underline{\mathsf{lvalue}} :: \mathbf{Des\_Kind}$$
$$\text{if } \forall \mathsf{x} :: \mathbf{Parameter\_Indicator}, \mathsf{y} :: \mathbf{String}, \mathsf{z} :: \mathbf{Type}. \langle \mathsf{x}, \mathsf{y}, \mathsf{z}\rangle \in \mathsf{locals} \Rightarrow \mathsf{y} \neq v(\mathsf{id}) \qquad \text{(A.S21)}$$

$$\langle \mathsf{Names}, \mathsf{TH}, \mathsf{Intfs}, \mathsf{A}, \mathsf{locals} \uplus \{\langle \mathsf{x}, v(\mathsf{id}), \mathsf{t}\rangle\}\rangle :: \mathbf{Context_3} \vdash des : \mathsf{t} :: \mathbf{Type}, \underline{\mathsf{lvalue}} :: \mathbf{Des\_Kind} \qquad \text{(A.S22)}$$

Production (A.22) $des_0 ::= des_1.\mathsf{id}(expr)$

$$\frac{\begin{array}{c}\langle \mathsf{Names}, \mathsf{TH}, \mathsf{Intfs} \uplus \{\langle \mathsf{t}_1, \mathsf{Cl\_Intfs} \uplus \{\langle \underline{\mathsf{meth}}, \langle v(\mathsf{id}), \mathsf{t}_2, \mathsf{t}\rangle\rangle\}\rangle\}, \mathsf{A}, \mathsf{locals}\rangle \\ :: \mathbf{Context_3} \vdash des_1 : \mathsf{t}_1 :: \mathbf{Type}, \mathsf{kind} :: \mathbf{Des\_Kind} \\ \langle \mathsf{Names}, \mathsf{TH}, \mathsf{Intfs} \uplus \{\langle \mathsf{t}_1, \mathsf{Cl\_Intfs} \uplus \{\langle \underline{\mathsf{meth}}, \langle v(\mathsf{id}), \mathsf{t}_2, \mathsf{t}\rangle\rangle\}\rangle\}, \mathsf{A}, \mathsf{locals}\rangle \\ :: \mathbf{Context_3} \vdash expr : \mathsf{t}_2' :: \mathbf{Type}\end{array}}{\begin{array}{c}\langle \mathsf{Names}, \mathsf{TH}, \mathsf{Intfs} \uplus \{\langle \mathsf{t}_1, \mathsf{Cl\_Intfs} \uplus \{\langle \underline{\mathsf{meth}}, \langle v(\mathsf{id}), \mathsf{t}_2, \mathsf{t}\rangle\rangle\}\rangle\}, \mathsf{A}, \mathsf{locals}\rangle \\ :: \mathbf{Context_3} \vdash des_0 : \mathsf{t} :: \mathbf{Type}, \underline{\mathsf{rvalue}} :: \mathbf{Des\_Kind}\end{array}} \quad \text{if } \mathsf{t}_2' \sqsubseteq \mathsf{t}_2 \in \mathsf{TH} \qquad \text{(A.S23)}$$

Production (A.23) $des ::= \mathsf{id}(expr)$

$$\frac{\langle \mathsf{Names}, \mathsf{TH}, \mathsf{Intfs} \uplus \{\langle \mathsf{A}, \mathsf{Cl\_Intfs} \uplus \{\langle \underline{\mathsf{meth}}, \langle v(\mathsf{id}), \mathsf{t}_1, \mathsf{t}_2\rangle\rangle\}\rangle\}, \mathsf{A}, \mathsf{locals}\rangle :: \mathbf{Context_3} \vdash expr : \mathsf{t}_1' :: \mathbf{Type}}{\langle \mathsf{Names}, \mathsf{TH}, \mathsf{Intfs} \uplus \{\langle \mathsf{A}, \mathsf{Cl\_Intfs} \uplus \{\langle \underline{\mathsf{meth}}, \langle v(\mathsf{id}), \mathsf{t}_1, \mathsf{t}_2\rangle\rangle\}\rangle\}, \mathsf{A}, \mathsf{locals}\rangle :: \mathbf{Context_3} \vdash des : \mathsf{t}_2 :: \mathbf{Type}, \underline{\mathsf{rvalue}} :: \mathbf{Des\_Kind}} \qquad \text{(A.S24)}$$
$$\text{if } \mathsf{t}_1' \sqsubseteq \mathsf{t}_1 \in \mathsf{TH}$$

Production (A.24) $expr ::= des$

$$\frac{\Gamma :: \mathbf{Context_3} \vdash des : \mathsf{t} :: \mathbf{Type}, \mathsf{kind} :: \mathbf{Des\_Kind}}{\Gamma :: \mathbf{Context_3} \vdash expr : \mathsf{t} :: \mathbf{Type}} \qquad \text{(A.S25)}$$

Production (A.25) $expr ::= \texttt{new id}$

$$\langle \mathsf{Names} \uplus \{v(\mathsf{id})\}, \mathsf{TH}, \mathsf{Intfs}, \mathsf{A}, \mathsf{locals}\rangle :: \mathbf{Context_3} \vdash expr : v(\mathsf{id}) :: \mathbf{Type} \quad \text{if } v(\mathsf{id}) \neq \mathit{void} \qquad \text{(A.S26)}$$

Fig. 30. Inference rules for productions (A.20)–(A.25).

specifies that it is possible to assign a value to an attribute of an object specified by a designator iff it is possible to assign a value to the designator. Furthermore, the identifier for the attribute must be a feature of the type of this designator. According to the grammar, a designator consisting of a single identifier is either an attribute or a local variable; it never can be a method call. There is an inference rule for each of these two cases: (A.S21) and (A.S22), respectively. In both cases, it is possible to assign values to these designators. The side condition of (A.S21) defines that a designator is an attribute only if the corresponding identifier is not hidden by a local variable. The inference rule (A.S23) describes the situation when a method is called on an object specified by a designator. It specifies that the called method is a method of the static type of the designator and (by its side condition) that the type of the argument is a subtype of the parameter type of the method. Furthermore, the type of the method call is the return type of the method, and it is not possible to assign a value to a method call. (A.S24) describes the analogous situation for a plain method call. The inference rule (A.S26) specifies that the type of a new operation must be a class name different from *void* and that the type of the expression is the class of the created object.

$$\begin{array}{rcll}
\mathit{prog} & ::= & \mathit{stats} & (\mathrm{B.1})\\
\mathit{stats}_0 & ::= & \mathit{stat}; \mathit{stats}_1 & (\mathrm{B.2})\\
\mathit{stats} & ::= & \varepsilon & (\mathrm{B.3})\\
\mathit{stat} & ::= & \mathit{var} := \mathit{expr} & (\mathrm{B.4})\\
\mathit{stat} & ::= & \mathit{var} : \mathit{type} & (\mathrm{B.5})\\
\mathit{var} & ::= & \mathsf{id} & (\mathrm{B.6})
\end{array}
\qquad
\begin{array}{rcll}
\mathit{expr} & ::= & \mathit{var} & (\mathrm{B.7})\\
\mathit{expr} & ::= & \mathit{const} & (\mathrm{B.8})\\
\mathit{expr}_0 & ::= & \mathit{expr}_1 + \mathit{expr}_2 & (\mathrm{B.9})\\
\mathit{const} & ::= & \mathsf{intconst} & (\mathrm{B.10})\\
\mathit{const} & ::= & \mathsf{realconst} & (\mathrm{B.11})\\
\mathit{type} & ::= & \underline{\mathsf{int}} & (\mathrm{B.12})\\
\mathit{type} & ::= & \underline{\mathsf{real}} & (\mathrm{B.13})
\end{array}$$

Fig. 31. Syntax of DEMO.

$$\begin{array}{rcl}
\mathbf{Type} & = & \underline{\mathsf{inttype}}|\underline{\mathsf{realtype}}\\
\mathbf{Decl} & = & \mathbf{String} \times \mathbf{Type}\\
\mathbf{Context} & = & \{\mathbf{Decl}\}
\end{array}$$

Fig. 32. Sorts used in the sorted natural semantics for DEMO.

$$\begin{array}{rll}
\sqsubseteq : & \mathbf{Type} \times \mathbf{Type} & \rightarrow \mathbf{Bool}\\
\mathit{undefined} : & \mathbf{Context} \times \mathbf{String} & \rightarrow \mathbf{Bool}\\
\sqcup : & \mathbf{Type} \times \mathbf{Type} & \rightarrow \mathbf{Type}
\end{array}$$

$$\begin{array}{cr}
\mathit{undefined}(\emptyset, \mathsf{v}). & (\mathrm{B.G1})\\
\mathit{undefined}(\Gamma \uplus \{\langle \mathsf{w}, \mathsf{t}\rangle\}, \mathsf{v}) \leftarrow \mathsf{v} \neq \mathsf{w},\ \mathit{undefined}(\Gamma, \mathsf{v}). & (\mathrm{B.G2})\\
\mathsf{t} \sqsubseteq \mathsf{t}\ . & (\mathrm{B.G3})\\
\underline{\mathsf{inttype}} \sqsubseteq \underline{\mathsf{realtype}}\ . & (\mathrm{B.G4})\\
\mathsf{t} \sqcup \mathsf{t}' = \mathsf{t}' \leftarrow \mathsf{t} \sqsubseteq \mathsf{t}' & (\mathrm{B.G5})\\
\mathsf{t}' \sqcup \mathsf{t} = \mathsf{t}' \leftarrow \mathsf{t} \sqsubseteq \mathsf{t}' & (\mathrm{B.G6})
\end{array}$$

Fig. 33. Defined functions for DEMO.

APPENDIX B. SPECIFICATION OF DEMO

This appendix specifies the language DEMO. DEMO is used in Section 3 for demonstrating the basic algorithm and in Section 5 for the comparison of natural semantics and attribute grammars. Section B.1 introduces the syntax and informal static semantics of DEMO. Section B.2 defines its static semantics by a sorted natural semantics specification, and Section B.3 gives a definition by an attribute grammar.

B.1 Syntax and Informal Introduction to DEMO

DEMO is a very simple imperative language. A DEMO-program is a list of assignments and variable declarations. Figure 31 shows the context-free grammar specifying the syntax of DEMO.

DEMO has two types, integers and reals. Integers are coercible to real numbers but not vice versa, that is, if the left-hand side of an assignment is of type integer, then the expression on the right-hand side must also be of type integers.

Production (B.1): $prog ::= stats$

$$\frac{\emptyset :: \textbf{Context} \vdash stats : \underline{\mathsf{correct}} :: \textbf{GenInfo}}{\vdash prog : \underline{\mathsf{correct}} :: \textbf{GenInfo}} \qquad \text{(B.S1)}$$

Production (B.2): $stats_0 ::= stat; stats_1$

$$\frac{\begin{array}{c}\Gamma :: \textbf{Context} \vdash stat : \langle \mathsf{x}, \mathsf{type}\rangle :: \textbf{Decl} \\ \Gamma \uplus \{\langle \mathsf{x}, \mathsf{type}\rangle\} :: \textbf{Context} \vdash stats_1 : \underline{\mathsf{correct}} :: \textbf{GenInfo}\end{array}}{\Gamma :: \textbf{Context} \vdash stats_0 : \underline{\mathsf{correct}} :: \textbf{GenInfo}} \text{ if } undefined(\Gamma, \mathsf{x}) \qquad \text{(B.S2)}$$

$$\frac{\begin{array}{c}\Gamma \uplus \{\langle \mathsf{x}, \mathsf{type}\rangle\} :: \textbf{Context} \vdash stat : \langle \mathsf{x}, \mathsf{type}\rangle :: \textbf{Decl} \\ \Gamma \uplus \{\langle \mathsf{x}, \mathsf{type}\rangle\} :: \textbf{Context} \vdash stats_1 : \underline{\mathsf{correct}} :: \textbf{GenInfo}\end{array}}{\Gamma :: \textbf{Context} \vdash stats_0 : \underline{\mathsf{correct}} :: \textbf{GenInfo}} \text{ if } undefined(\Gamma, \mathsf{x}) \qquad \text{(B.S3)}$$

Production (B.3): $stats ::= \varepsilon$

$$\Gamma :: \textbf{Context} \vdash stats : \underline{\mathsf{correct}} :: \textbf{GenInfo} \qquad \text{(B.S4)}$$

Production (B.4): $stat ::= var := expr$

$$\frac{\begin{array}{c}\vdash var : \mathsf{x} :: \textbf{String} \\ \Gamma :: \textbf{Context} \vdash expr : \mathsf{type}_1 :: \textbf{Type}\end{array}}{\Gamma :: \textbf{Context} \vdash stat : \langle \mathsf{x}, \mathsf{type}_2\rangle :: \textbf{Decl}} \text{ if } \mathsf{type}_1 \sqsubseteq \mathsf{type}_2 \qquad \text{(B.S5)}$$

Production (B.5): $stat ::= var : type$

$$\frac{\vdash var : \mathsf{x} :: \textbf{String} \qquad \vdash type : \mathsf{type} :: \textbf{Type}}{\Gamma :: \textbf{Context} \vdash stat : \langle \mathsf{x}, \mathsf{type}\rangle :: \textbf{Decl}} \qquad \text{(B.S6)}$$

Production (B.6): $var ::= \mathsf{id}$

$$\vdash var : v(\mathsf{id}) :: \textbf{String} \qquad \text{(B.S7)}$$

Production (B.7): $expr ::= var$

$$\frac{\vdash var : \mathsf{x} :: \textbf{String}}{\Gamma \uplus \{\langle \mathsf{x}, \mathsf{type}\rangle\} :: \textbf{Context} \vdash expr : \mathsf{type} :: \textbf{Type}} \qquad \text{(B.S8)}$$

Production (B.8): $expr ::= const$

$$\frac{\vdash const :: \textbf{Type}}{\Gamma :: \textbf{Context} \vdash expr : \mathsf{type} :: \textbf{Type}} \qquad \text{(B.S9)}$$

Production (B.9): $expr_0 ::= expr_1 + expr_2$

$$\frac{\Gamma :: \textbf{Context} \vdash expr_1 : \mathsf{type}_1 :: \textbf{Type} \qquad \Gamma :: \textbf{Context} \vdash expr_2 : \mathsf{type}_2 :: \textbf{Type}}{\Gamma :: \textbf{Context} \vdash expr_0 : \mathsf{type}_1 \sqcup \mathsf{type}_2 :: \textbf{Type}} \qquad \text{(B.S10)}$$

Production (B.10): $const ::= \mathsf{intconst}$

$$\vdash const : \underline{\mathsf{inttype}} :: \textbf{Type} \qquad \text{(B.S11)}$$

Production (B.11): $const ::= \mathsf{realconst}$

$$\vdash const : \underline{\mathsf{realtype}} :: \textbf{Type} \qquad \text{(B.S12)}$$

Production (B.12): $type ::= \underline{\mathsf{int}}$

$$\vdash type : \underline{\mathsf{inttype}} :: \textbf{Type} \qquad \text{(B.S13)}$$

Production (B.13): $type ::= \underline{\mathsf{real}}$

$$\vdash type : \underline{\mathsf{realtype}} :: \textbf{Type} \qquad \text{(B.S14)}$$

Fig. 34. Inference rules for DEMO.

Declarations of variables do not need to occur before their use. However, every variable used in a DEMO-program must be declared. It is possible to declare a variable twice. In this case, the declaration (i.e., the type of the variable) must be identical.

B.2 Sorted Natural Semantics for DEMO

Figure 32 shows the sorts used in the specification of the static semantics for DEMO. The context contains a set of declarations. A declaration consists of a

prod (B.1) $prog ::= stats$
attr $stats.context \leftarrow stats.defs$
cond $unambigous(stats.defs)$

prod (B.2) $stats_0 ::= stat; stats_1$
attr $stats_1.context \leftarrow stats_0.context$
$stat.context \leftarrow stats_0.context$
$stats_0.defs \leftarrow stat.decl \cup stats_1.defs$

prod (B.3) $stats ::= \varepsilon$
attr $stats.defs \leftarrow \emptyset$

prod (B.4) $stat ::= var := expr$
attr $expr.context \leftarrow stat.context$
$var.type \leftarrow gettype(stat.context, var.id)$
$stat.decl \leftarrow \emptyset$
cond $expr.type \sqsubseteq var.type \wedge$
$is\_defined(stat.context, var.id)$

prod (B.5) $stat ::= var : type$
attr $var.type \leftarrow type.type$
$stat.decl \leftarrow \{\langle var.id, type.type\rangle\}$

prod (B.6) $var ::= \mathsf{id}$
attr $var.id \leftarrow v(\mathsf{id})$

prod (B.7) $expr ::= var$
attr $expr.type \leftarrow$
$gettype(expr.context, var.id)$
$var.type \leftarrow$
$gettype(expr.context, var.id)$
cond $is\_defined(expr.context, var.id))$

prod (B.8) $expr ::= const$
attr $expr.type \leftarrow const.type$

prod (B.9) $expr_0 ::= expr_1 + expr_2$
attr $expr_1.context \leftarrow expr_0.context$
$expr_2.context \leftarrow expr_0.context$
$expr_0.type \leftarrow expr_1.type \sqcup expr_2.type$

prod (B.10) $const ::= \mathsf{intconst}$
attr $const.type \leftarrow \mathsf{inttype};$

prod (B.11) $const ::= \mathsf{realconst}$
attr $const.type \leftarrow \mathsf{realtype};$

prod (B.12) $type ::= \mathsf{int}$
attr $type.type \leftarrow \mathsf{inttype};$

prod (B.13) $type ::= \mathsf{real}$
attr $type.type \leftarrow \mathsf{realtype};$

Fig. 35. An attribute grammar for DEMO.

variable name and its type. Figure 33 shows the defined functions and their definitions. $\sqsubseteq$ is the subtype predicate. $\sqcup$ computes the least upper bound of two types with respect to $\sqsubseteq$.

Figure 34 shows the inference rules for DEMO. Each statement defines an identifier—even if it is an assignment (cf. rule B.S5). The two inference rules (B.S2) and (B.S3) distinguish the situation when a variable is not yet defined in a context and when it is already defined. In the latter case, the type in the context and the type derived from the statement must agree. Inference rule (B.S5) specifies that the type of the expression on the right-hand side of an assignment must be coercible to the left-hand side of the assignment. Rule (B.S6) specifies that the type of a variable in a declaration is the declared type. The remaining rules for typing expressions (rules (B.S8)–(B.S12)) and for obtaining types (rules (B.S13) and (B.S14)) are straightforward.

The interaction between rules (B.S2), (B.S3), (B.S5), and (B.S6) requires some explanation. Suppose that a value is assigned to an identifier x before its declaration. Then the application of rule (B.S5) introduces a new variable type for the type of the identifier x. Rule (B.S2) adds this declaration to the context for the remaining statements in the program without instantiating the variable type because the variable x may be declared later. This context is passed through (and possibly extended) by inference rules (B.S2) and (B.S3). As soon as a declaration of x is encountered, the variable type is substituted by the type of x by rule (B.S6).

If inference rule (B.S3) is applied, then variable x is declared before the examined statement *stat*. Then, the type stemming from the declaration is already in the context Γ. It must be equal to the type of the variable x in *stat*. For any of the two cases, the type information flows from the declaration of a variable to the statement. However, the flow directions are different. The unification mechanism is powerful enough such that, for both cases, the attribution can be computed from the first to the last statement in order.

B.3 An Attribute Grammar for DEMO

Since the flow-direction of context information is not uniform, we need two attributes for the context. Otherwise, the attribute grammar will not be well-defined (cf. Section 5.2). The attribute grammar in Figure 35 collects all declarations (attribute *defs*) and passes them at the root to the context. The context is simply propagated down. At the root it is checked whether the declarations are unambigous (using the function *unambigous*). The AG-condition for production (B.4) checks whether the type of the right-hand side is coercible to the left-hand side and whether the left-hand side is declared in the context. Similarly, if a variable occurs on the right-hand side, the AG-condition of production (B.7) checks whether it is declared in the context. Hence, all programs using undeclared variables are rejected. The attribution for *stat.decl* of the production (B.4) does not add a declaration for the left-hand side. This is only done in the attribution for *stat.decl* of production (B.5). The attribution rules for computing types of expressions are analogous to the specification for the sorted natural semantics. The only difference is that the type information for variables is not obtained by unification (as in inference rule (B.S8)) but by an explicit function *gettype*. All auxiliary functions need to be defined. We leave these definitions to the reader.

ACKNOWLEDGMENTS

We are grateful to the anonymous referees. Their comments and hints helped us to considerably improve our work. The authors would also like to thank Gerhard Goos for many helpful discussions and for working together on this research topic. In addition, thanks to Rubino Geiß who implemented the prototype for sorted natural semantics.

REFERENCES

ABADI, M. AND CARDELLI, L. 1996. *A Theory of Objects*. Springer-Verlag, Berlin, Germany.

ANDRÉKA, H. AND NÉMETI, I. 1978. The generalized completeness of Horn predicate logic as a programming language. *Acta Cybernet., 4*, 3–10.

ATTALI, I. 1989. *Compilation de programmes Typol par attributs sémantiques*. Ph.D. dissertation, University of Nice, Nice, France.

ATTALI, I., CAROMEL, D., AND EHMETY, S. O. 1996. A natural semantics for Eiffel dynamic binding. *ACM Trans. Program. Lang. Syst. 18*, 5 (Nov.), 711–729.

ATTALI, I. AND FRANCHI-ZANNETTACCI, P. 1988. Unification-free execution of Typol programs by semantic attribute evaluation. In *International Conference Symposium on Logic Programming*, (Seattle, Aug.). MIT Press, Cambridge, MA, 160–177.

BAADER, F. AND SCHULZ, K. U. 1998. Unification theory. In *Automated Deduction. A Basis for Applications*, W. Bibel and P. H. Schmitt, Eds. Kluwer Academic Publishers, Dordrecht, The Netherlands, vol. I, chap. 7, 225–263.

BAHLKE, R. AND SNELTING, G. 1986. The PSG system: From formal language definitions to interactive programming environments. *ACM Trans. Program. Lang. Syst., 8*, 4 (Oct.), 547–576.

BOCHMANN, G. V. 1976. Semantic evaluation from left to right. *Commun. ACM 19*, 2, 55–62.

CASTAGNA, G. 1997. *Object-Oriented Programming—A Unified Foundation*. Birkhäuser, Basel, Switzerland.

COUSOT, P. AND COUSOT, R. 1977. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proceedings of the 4th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*. ACM Press, New York, NY, 238–252.

DESPEYROUX, T. 1984. Executable specification of static semantics. In *Semantics of Data Types*, G. Kahn, Ed. Lecture Notes in Computer Science, vol. 173, Springer-Verlag, Berlin, Germany, 215–233.

DOMENJOUD, E. 1992. A technical note on AC-unification. The number of minimal unifiers of the equation $\alpha x_1 + \cdots + \alpha x_p = \beta y_1 + \cdots \beta y_q$. *J. Automat. Reason. 8*, 1, 39–44.

DROSSOPOULOU, S. AND EISENBACH, S. 1999. Describing the semantics of Java and proving type soundness. In *Formal Syntax and Semantics of Java*, J. Alves-Foss, Ed. Lecture Notes in Computer Science, vol. 1523, Springer-Verlag, Berlin, Germany, 41 ff.

ELI. n.d. Eli: Übersetzerentwicklung leicht gemacht. Available online at http://www.uni-paderborn.de/fachbereich/AG/agkastens/eli_home.html.

GEIß, R. 1998. The Sherlock-System—a prototype for many-sorted natural semantics. Term project, Universität Karlsruhe, Fakultät für Informatik, Karlsruhe, Germany.

GEIß, R. 1999. The Sherlock-System II—a prototype for many-sorted natural semantics implemented in Java. Term project. Universität Karlsruhe, Fakultät für Informatik, Karlsruhe, Germany.

GENTZEN, G. 1935. Untersuchungen über das logische Schließen. *Mathem. Zeitschrift 39*, 176–210 and 405–431.

GLESNER, S. 1998. Many-sorted natural semantics—specification and generation of the semantic analysis for imperative and object-oriented programming languages. Working paper 63. Westfälische Wilhelms-Universität Münster, Institut für Wirtschaftsinformatik, Münster, Germany. (Proceedings Workshop on Functional and Logic Programming).

GLESNER, S. 1999a. Natural semantics for imperative and object-oriented programming languages. In *Informatik '99—Informatik überwindet Grenzen, Proceedings der 29. Jahrestagung der Gesellschaft für Informatik* (Paderborn, Oct.), K. Beiersdörfer, G. Engels, and W. Schäfer, Eds. GI-Gesellschaft für Informatik e.V., Springer-Verlag, Berlin, Germany 370–379.

GLESNER, S. 1999b. *Natürliche Semantik für imperative und objektorientierte Programmiersprachen*. Ph.D. dissertation. Universität Karlsruhe, Fakultät für Informatik. Shaker Verlag, Aachen, Germany.

GLESNER, S. AND ZIMMERMANN, W. 1997. Using many-sorted inference rules to generate semantic analysis. In *Proceedings des Workshops der Informatik-Graduiertenkollegs "Promotion tut not: Innovationsmotor Graduiertenkolleg" im Rahmen der GI-Jahrestagung 1997*, Otto Spaniol, Ed. Verlag der Augustinus Buchhandlung (Aachener Beiträge zur Informatik, Band 21), Aachen, Germany.

GLESNER, S. AND ZIMMERMANN, W. 1998. Using many-sorted natural semantics to specify and generate semantic analysis. In *Proceedings of the Systems Implementation Conference* (Berlin, Germany Feb.) (SI2000), R. Nigel Horspool, Ed. (IFIP Working Group 2.4). Chapman & Hall, London, U.K., pages 249–262.

GROSCH, J. AND EMMELMANN, H. 1990. A tool box for compiler construction. In *Compiler Compilers*, D. Hammer, Ed., Lecture Notes in Computer Science, vol. 477. Springer-Verlag, Berlin, Germany, 106–116.

HANUS, M. 1994. The integration of functions into logic programming: From theory to practice. *J. Logic Program. 19*, 20, 583–628.

IGARASHI, A. PIERCE, B., AND WADLER, P. 1999. Featherweight Java: A minimal core calculus for Java and GJ. In *Proceedings of the 1999 ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*. ACM Press, New York, NY, 132–146.

JOUVELOT, P. AND GIFFORD, D. 1991. Algebraic reconstruction of types and effects. In *Proceedings of the 18th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. ACM Press, New York, NY, 303–310.

KAHN, G. 1987. Natural semantics. In *Proceedings of the 4th Annual Symposium on Theoretical Aspects of Computer Science* (Passau, Germany, Feb.) (STACS'87), F.-J. Brandenburg,

G. Vidal-Naquet, and M. Wirsing, Eds. Lecture Notes in Computer Science, vol. 247. Springer-Verlag, Berlin Germany, 22–39.

KAPUR, D. AND NARENDRAN, P. 1992. Complexity of unification problems with associative-commutative operators. *J. Automat. Reason. 9*, 261–288.

KASTENS, U. 1980. Ordered attribute grammars. *Acta Inform. 13*, 3, 229–256.

KASTENS, U. HUTT, B., AND ZIMMERMANN, E. 1982. *GAG: A Practical Compiler Generator*. Lecture Notes in Computer Science, vol. 141. Springer-Verlag, Heidelberg, Germany.

KNUTH, D. E. 1968. Semantics of context-free languages. *Math. Syst. Theor. 2*, 2, 127–146.

KNUTH, D. E. 1971. Semantics of context-free languages: Correction. *Math. Syst. Theor. 5*, 95–96.

KFOURY, A. J., TIURYN, J., AND URZYCZYN, P. 1994. An analysis of ML typability. *J. ACM 41*, 2, 368–398.

LEWIS, P. M., ROSENKRANTZ, D. J., AND STEARNS, R. E. 1974. Attributed translations. *J. Comput. Syst. Sci., 9*, 3, 279–307.

MILNER, R. AND TOFTE, M. 1991. *Commentary on Standard ML*. MIT Press, Cambridge, MA.

MILNER, R., TOFTE, M., AND HARPER, R. 1990. *The Definition of Standard ML*. MIT Press, Cambridge, MA.

MILNER, R., TOFTE, M., HARPER, R., AND MACQUEEN, D. 1997. *The Definition of Standard ML (Revised)*. MIT Press, Cambridge, MA.

MORGAN, R. 1998. *Building an Optimizing Compiler*. Butterworth-Heinemann, Oxford, U.K.

MUCHNICK, S. S. 1997. *Advanced Compiler Design and Implementation*. Morgan Kaufmann, San Francisco, CA.

NIELSON, F., RIIS, NIELSON, H., AND HANKIN, C. 1999. *Principles of Program Analysis*. Springer-Verlag, Berlin, Germany.

NIPKOW, T. AND VON OHEIMB, D. 1998. Java$_{\ell ight}$ is type-safe—definitely. In *Proceedings of the 25th ACM Symposium on Principles of Programming Languages*. ACM Press, New York, NY, 161–170.

ODERSKY, M. 1993. Defining context-dependent syntax without using contexts. *ACM Trans. Program. Lang. Syst. 15*, 3 (July), 535–562.

OZ. n.d. The Oz programming system. Programming Systems Lab, DFKI, and Universität des Saarlandes, Saarbrücken, Germany. Available online at http://www.ps.uni-sb.de/oz/.

PALSBERG, J. AND SCHWARTZBACH, M. 1994. *Object-Oriented Type System*. John Wiley & Sons, New York, NY.

PATERSON, M. S. AND WEGMAN, M. N. 1978. Linear unification. *J. Comp. Syst. Sci. 16*, 2, 158–167.

PETTERSSON, M. 1995. *Compiling natural semantics*. Ph.D. dissertation. Department of Computer and Information Science, Linköping University, Linköping, Sweden.

PETTERSSON, M. 1996. A compiler for natural semantics. In *Proceedings of the 6th International Conference on Compiler Construction* (CC'96, Linköping, Sweden, April). Lecture Notes in Computer Science, vol. 1060, Springer-Verlag, Berlin, Germany.

ROBINSON, J. A. 1965. A machine-oriented logic based on the resolution principle. *J. ACM, 12*, 1, 23–41.

SNELTING, G. AND HENHAPL, W. 1986. Unification in many-sorted algebras as a device for incremental semantic analysis. In *Proceedings of the International Symposium on Principles of Programming Languages*. ACM Press, New York, NY.

SYME, D. 1999. Proving Java type soundness. In *Formal Syntax and Semantics of Java*, J. Alves-Foss, Ed. Lecture Notes in Computer Science, vol. 1523, Springer-Verlag, Berlin, Germany, 83 ff.

VON OHEIMB, D. 2001. *Analyzing Java in Isabelle/HOL*. Ph.D. dissertation. Technische Universität München, Munich, Germany.

VON OHEIMB, D. AND NIPKOV, T. 1999. Machine-checking the Java, specification: Proving type-safety. In *Formal Syntax and Semantics of Java* J. Alves-Foss, Ed. Lecture Notes in Computer Science, vol. 1523, Springer-Verlag, Berlin, Germany, 119 ff.

WIRSING, M. 1990. Algebraic specification. In *Handbook of Theoretical Computer Science, Vol. B: Formal Models and Semantics*, J. van Leeuwen, Ed. MIT Press, Cambridge, MA, and Elsevier Science Publishers B. V., Amsterdam, The Netherlands, 675–788.

Received February 2001; revised July 2002; accepted July 2003