Using many-sorted natural semantics to specify and generate semantic analysis

Sabine Glesner and Wolf Zimmermann
Institut für Programmstrukturen und Datenorganisation
Universität Karlsruhe, 76128 Karlsruhe, Germany
Tel.: +49 - 721 608 {7399|4759}, Fax: +49 - 721 30047
E-mail:{glesner|zimmer}@ipd.info.uni-karlsruhe.de

Abstract
We present an extension of natural semantics which can be used to describe the static semantics of imperative and object-oriented programming languages. Furthermore we show that the semantic analysis can be generated from these descriptions. As a side effect, we get a precise definition of which properties of a programming language are statically decidable and which properties can only be checked dynamically during run-time. As an example, we show how a subset of the Java programming language incorporating the full notion of inheritance can be specified within our mechanism.

Keywords
Natural Semantics, Semantic Analysis, Specification, Generator, Imperative and Object-Oriented Programming Languages

1 INTRODUCTION

Natural semantics [Kah87] has been established as a declarative framework to specify the static and dynamic semantics of functional programming languages. Its applicability for imperative languages with simple block structure has been shown (e.g. the specifications of simple while-languages). The reason for the wide-spread use of natural semantics lies in its declarative self-contained style which has been proved useful especially for the description of functional languages. For these languages, the implementation of the analysis for the static properties is simply type inference. Yet, as far as to the authors' knowledge, no specifications of real imperative and object-oriented languages exist.

In this paper, we investigate which extensions of natural semantics are necessary to apply the formalism to imperative and object-oriented programming languages. We are particularly interested in the description of the context-

©IFIP 1996. Published by Chapman & Hall

sensitive properties of these programming languages. Moreover, we want that the analysis of the described static semantics can be generated from the specification.

Compared to the rather complicated scope rules of imperative and object-oriented programming languages, their type systems are simple. In contrast, functional programming languages have fairly simple scope rules but rich type systems. An inference style specification is well-suited to describe the properties of functional languages. Moreover, it has several advantages since it abstracts completely from particular traversals of the abstract syntax tree. Secondly, a notion of consistency and completeness can be defined formally.

Our approach extends the framework of natural semantics so that many-sorted inference rules can be used to specify the static semantics of imperative and object-oriented programming languages. In particular, we show how this extension can be used to define structured context information. It is also possible to define a variety of semantic information for each point in a program. We demonstrate the applicability for imperative and object-oriented languages by showing how a small example language incorporating the full Java type system (without overloading) can be specified. Furthermore, we define static type safety. Finally, we show that the semantic analysis can be generated from many-sorted natural semantics specifications.

This paper is organized as follows: Section 2 introduces natural semantics and its many-sorted extension. Section 3 introduces an algorithm to perform semantic analysis using natural semantics. Section 4 compares many-sorted natural semantics with related approaches. The appendix gives an example specification for a Java subset.

2 MANY-SORTED NATURAL SEMANTICS

Natural semantics is typically used as a framework to specify the static and dynamic semantics of functional programming languages by means of inference rules. In particular, the static semantics description consists usually of type inference rules. In this section, we first introduce natural semantics according to [Kah87] and sketch its use in the description of the static semantics of functional languages. Then we proceed by extending natural semantics with sort annotations so to be able to describe the static semantics of imperative and object-oriented languages.

2.1 Natural semantics

A natural semantics specification consists of a set of axioms and inference rules. An *inference rule* has the form

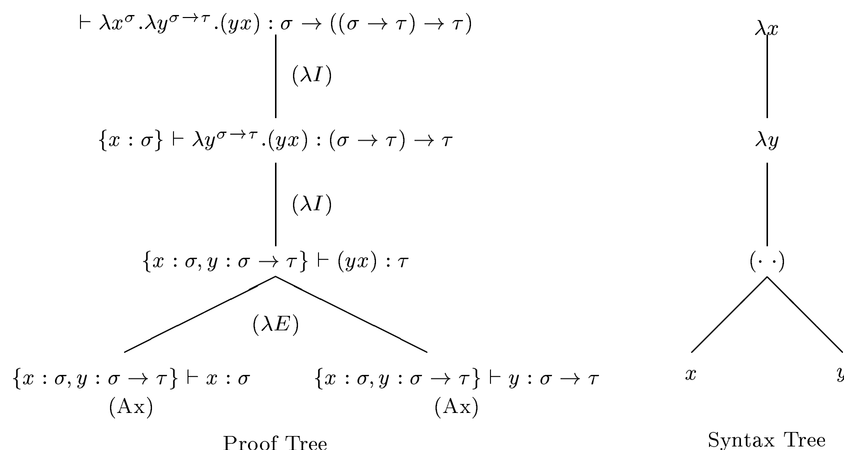
$$\frac{\Gamma_1 \vdash s_1 : \mathsf{t}_1, \ldots, \Gamma_n \vdash s_n : \mathsf{t}_n}{\Gamma \vdash f(s_1, \ldots, s_n) : \mathsf{t}} \quad \text{if } \varphi$$

$\Gamma_i \vdash s_i : \mathtt{t}_i$ is a *judgement*. The informal meaning is that under the *assumptions* Γ_i, it can be *concluded* that the term s_i has type $\mathtt{t}_i$. Here s_i is a term which may contain variables. The assumptions are a set of sequences of pairs $x_j : \mathtt{tt}_j$ where x_j is a variable and $\mathtt{tt}_j$ is a type. In the case of programming languages, the assumptions are called *context*. $x_j : \mathtt{tt}_j \in \Gamma$ means that x_j is assumed to have type $\mathtt{tt}_j$ in the assumptions (or context, resp.) Γ. The judgements in the nominator of the rule are called *premises*, the judgement in the denominator of the rule is the *consequence*. An *axiom* is a rule without premises. An inference rule is applicable only if the condition φ is satisfied. Such a condition is a first-order formula. Only those variables may appear in the formula of a rule that are contained in the judgements of the rule.
Example: The following natural semantics rules define the simple typed lambda calculus (see e.g. [Mit90]):

(Ax)	$\Gamma \cup \{x : \tau\} \vdash x : \tau$	(λI)	$\frac{\Gamma \uplus \{x:\tau\} \vdash t:\sigma}{\Gamma \vdash \lambda x^{\tau}.t:\tau \rightarrow \sigma}$
(λE)	$\frac{\Gamma \vdash t:\tau \rightarrow \sigma \quad \Gamma' \vdash t':\tau}{\Gamma \cup \Gamma' \vdash (t\ t'):\sigma}$	(CI)	$\frac{\Gamma \vdash t:\sigma}{\Gamma \cup \Gamma' \vdash t:\sigma}$

For showing that a λ-term s is well-typed, it is sufficient to prove that there is a type $\mathtt{t}$ such that $\emptyset \vdash s : \mathtt{t}$ using the above inference rules. The proof is analogous to Gentzen's natural deduction [Gen69].

The following picture shows the proof tree for $\emptyset \vdash \lambda x^{\sigma}.\lambda y^{\sigma \rightarrow \tau}.(yx) : \sigma \rightarrow ((\sigma \rightarrow \tau) \rightarrow \tau)$ and the abstract syntax tree for the λ-term.



Proof Tree Syntax Tree

The proof structure is determined by the structure of the λ-term. Thus, λ-terms can be considered as proofs of their types. ■

This analogy can be generalized straightforwardly to functional languages and simple imperative (while-) languages: programs are proofs of their types.

2.2 Many-sorted extension

Imperative and object-oriented languages cannot be specified directly by natural semantics for two reasons: First, due to the scope nesting in imperative programs and the subtype relation in object-oriented programming languages, the assumption needs to be more structured. The kind of context information often differs in different parts of the program. Secondly, it is necessary to derive several semantic information for a program or fragments of it different from types.

Our goal is to generalize natural semantics such that the above restrictions do not apply and the "programs as proofs"-analogy is kept. The basic difference to natural semantics is that judgements are generalized such that structured and also differently structured contexts and different kinds of semantic information are expressible. A *sort* defines the structure of the contexts and the kind of semantic information. $\mathtt{t} \therefore \mathbf{Sort}$ denotes that the semantic information or context $\mathtt{t}$ has the sort **Sort**. A *judgement* has the form:

$$\Gamma_1 \therefore \mathbf{Sort}_1, \ldots, \Gamma_k \therefore \mathbf{Sort}_k \vdash s : \mathrm{t}_1 \therefore \mathbf{Sort}_{k+1}, \ldots, \mathrm{t}_n \therefore \mathbf{Sort}_{k+n}$$

Sorts are defined by extended context-free grammars (i.e. using additionally sets and lists on the right-hand side of productions). The meaning of terminal symbols used to define a new sort can be specified by axioms and inference rules as well. The only difference to the inference rules above is that these rules are applicable under each assumption. Therefore, we omit the context information here and call these inference rules *general inference rules*. The other kind of inference rules containing specific context information are called *special inference rules*. We associate a set of special inference rules with each production of the abstract syntax. All special inference rules must use only symbols of the right hand side of their production in its premises and the symbol on the left hand side in their consequence.

Example: Consider the specification of Mini-Java defined in the appendix. There is a sort **GenInfo** that distinguishes correct programs (correct) from incorrect ones wrt. the static semantics. The sort **Subtyping** is a directed graph, expressed by the relation $\sqsubseteq$. The sort **Cl_Env** contains a function *Thru*. The appendix contains the inference rules for $\sqsubseteq$ and *Thru*. The relation $\sqsubseteq$ is defined by $\in$, the function *Thru* by the equality $=$. ■

The informal meaning of a judgement is generalized in the straightforward way: Assume that the context information specified in the assumption of the judgement is valid. Then we can conclude that the syntactic construct (i.e. the program fragment) has the semantic information of the specified sorts.

A proof in the many-sorted natural semantics is defined as in natural semantics with the addition that the sorts have to be respected. Again, for the special inference rules, the structure of the proof is the same as the ab-

stract syntax tree for the program. However, the proof tree is larger if general inference rules are applied.

2.3 Definition of language properties

There are three important properties of languages and language specifications that we define in this section. First, we state a definition for *consistency* of specifications. Then we define under which circumstances a specification is called *well-defined*. Thirdly, we give a definition for statical checkability wrt. a natural semantics specification. Thereby, we distinguish between checkability of a program and checkability of a whole language. Furthermore, we show how to retrieve the necessary run-time checks whenever a single program or a programming language is not statically checkable.

Whenever different rules can be applied which do not infer the same judgement, different information for the particular program point can be derived. In this case, the specification is not consistent. This leads to the following definition: A specification is *consistent* iff there exists no program whose abstract syntax tree contains a node for which more than one rule is applicable.

With the notion of well-definedness, we make sure that everything that is supposed to be statically computable can indeed be determined statically from the specification: A specification is *well-defined* iff there is a special inference rule for each production and the judgements for each nonterminal have the same sorting. The first requirement makes sure that, in principle, a proof exists for each program. The second requirement means that each special inference rule describing the same production has a context of the same sort and that the sorts of the specified semantic information are equal. Speaking in the language of attribute grammars, with the second requirement we guarantee that each nonterminal symbol has been assigned the same inherited and synthesized attributes by each judgement describing it.

We can think of semantic analysis as proof checking and proof completion. The structure of the proof tree is partly known a priori because it contains the structure of the abstract syntax tree. When checking the static semantics of a program, we need to find a rule cover for the abstract syntax tree such that the semantic information of the nodes can be computed and such that the conditions of the applied rules can be verified. When verifying the conditions and computing the semantic information, it might be necessary to extend the proof tree beyond the abstract syntax tree because general inference rules need to be applied. Thus, semantic analysis means proof checking (Does there exist a rule cover for the abstract syntax tree?) and proof completion (How can the specified semantic information be computed? And are the conditions true?).

In a statically checkable language, we require that, for each program, all conditions can be checked at compile-time. This is not true in general. It

might be the case that for some programs some conditions are neither *true* nor *false* but just satisfiable. These conditions remain as run-time checks. Thus, a program is *statically checkable* iff a rule cover (i.e., for each node there is an applicable rule) for its abstract syntax tree exists, the context and semantic information of all of its nodes can be computed, and all conditions stemming from inference rule applications are valid. A programming language is *statically checkable* iff all programs contained in it are statically checkable.

The dynamic checks necessary to perform at run-time are exactly those conditions in the abstract syntax proof tree which are satisfiable but not valid at compile-time.

3 GENERATING SEMANTIC ANALYSIS

In Section 2, we demonstrated already the correspondence between proofs of static correctness and programs. In particular, we showed that there is an embedding of the abstract syntax tree into the proof tree if general inference rules need to be applied. The basic idea of a generator is therefore first to construct the proof tree corresponding to the abstract syntax tree of the program and then computing the necessary semantic information and checking the conditions, both by applying general inference rules. For the following discussion, we consider only consistent specifications.

After checking the conditions, the following three situations might happen for any applied inference rule r: First, the condition of r is true. Secondly, the condition of r is false. Thirdly, the condition of r is satisfiable, but not true. For the first case, everything is fine. For the second case, if there is another applicable inference rule, then there might be another correctness proof. For the third case, the specification is not statically checkable. These conditions must be checked at runtime.

For simplicity, we assume that there is a sort **GenInfo** $= \{\mathsf{correct}\}$ and static semantic correctness is equivalent to proving $\vdash prog : \mathsf{correct} \therefore \mathbf{GenInfo}$.

The basis is a recursive algorithm $prove(\Gamma \vdash t : \mathsf{seminfos})$ which constructs a proof tree according to the abstract syntax tree of the program fragment t. Algorithm *prove* requires the definition of judgement unification and substitution. The unifications and substitutions are many-sorted. The variables of the judgements which need to be considered are in contexts on the left of $\vdash$ and the semantic information on the right of $\vdash$. For simplicity, we assume that there is a total order on the sorts, and the contexts and semantic information in the judgements are ordered according to their sorts (usually, these are not sequences but sets). The *most general unifier* of two judgments $\Gamma \vdash t : \mathsf{seminfo}$ and $\Gamma' \vdash t : \mathsf{seminfo}'$ is a substitution σ respecting the sorting of the terms (i.e., variables are replaced with terms of the same sort) with the following properties:

(i) $\sigma\Gamma = \sigma\Gamma'$

(ii) $\sigma\mathsf{seminfo} = \sigma\mathsf{seminfo}'$

(iii) For every substitution σ' respecting the sorting and satisfying (i) and (ii), there is a substition ρ respecting the sorting such that $\sigma' = \rho \circ \sigma$.

In order to prove a judgment $\Gamma \vdash t : \mathsf{seminfos}$, it has to be unified with a consequence (or axiom) of a special inference rule. These may contain function symbols (e.g. *Thru*). Therefore, an inference rule or axiom may not be applicable although it should be applied. To solve this problem, a unification (Algorithm *unify*) returns a substitution and a set of equalities to be proved in order to unify the consequence of the selected inference rule with the goal to be proved.

Algorithm *check* checks whether a condition is true. This algorithm extends the abstract syntax tree with the proof tree for $\sigma\varphi$ and the equalities to be proved using general inference rules. We assume for simplicity that all conditions and functions can be computed when the inference rule is applied.

Algorithm $prove(\Gamma \vdash t : \mathsf{seminfos})$

(1) suppose the abstract syntax tree t is constructed according $X_0 ::= X_1 \dots X_n$;
(2) **while** there is an unused special inference rule
(3) $\dfrac{\Gamma_1 \vdash X_{i_1} : \mathsf{sem}_1, \dots, \Gamma_k \vdash X_{i_k} : \mathsf{sem}_k}{\Gamma' \vdash X_0 : \mathsf{sem}'}$ if φ at the root of t
-- $X_{i_1}, \dots, X_{i_k} \in \{X_1, \dots, X_n\}$
(4) $(\sigma, Eq) := unify(\Gamma' \vdash X_0 : \mathsf{sem}', \Gamma \vdash t : \mathsf{seminfos})$
(5) **do**
(6) mark this inference rule as used at the root of t
(7) let t_i be the subtree of t with root X_i, $i = 1, \dots, n$
(8) **for** $j = 1, \dots, k$ **do**
(9) $prove\sigma(\Gamma_j \vdash t_{i_j} : \mathsf{sem}_j)$
(10) $\sigma := \sigma' \circ \sigma$ where σ' is the substitution associated with the root of t_{i_j};
(11) **end**;
(12) **if** none of these proofs failed, $check(\sigma Eq)$, and $check(\sigma\varphi)$
(13) **then**
(14) associate σ with the root of t;
(15) exit **while loop** successfully;
(16) **else for** $j = 1, \dots, k$ **do**
(17) unmark all special inference rules used in t_j
(18) **end**;
(19) **end**;
(20) **if** the while loop is not exited successfully **then** fail;

Basically, Algorithm *prove* is a backtracking algorithm which tries to apply all possible special inference rules. After applying a special inference rule (line (4)), the judgements in the premises have to be proved. These are proved recursively according to the production rule (line (8)–(11)). Observe that such a proof may substitute more variables (line (10)). If all proofs are successful, then the condition of the inference rule and the equations are checked (line (12)). The substitution associated with the root of t instantiates the contexts and semantic information of the judgement to be proved. Thus, Algorithm $prove(\Gamma \vdash t : \mathsf{seminfo})$ constructs a proof of the judgement $\sigma\Gamma \vdash t : \sigma\mathsf{seminfo}$ if it successfully terminates. This is one part of the correctness.

Theorem 1 *$prove(\Gamma \vdash t : \mathsf{seminfo})$ constructs a proof of the judgement $\sigma\Gamma \vdash t : \sigma\mathsf{seminfo}$ iff there exists one, where σ is the substitution associated with the root of t.*

Proof: We first show by induction that if $prove(\Gamma \vdash t : \mathsf{seminfo})$ terminates successfully, then there is a proof of $\sigma\Gamma \vdash t : \sigma\mathsf{seminfo}$. Consider the last iteration of the while loop. By induction hypothesis, this iteration constructs proofs for the judgements $\sigma\Gamma_j : t_{i_j} : \sigma\mathsf{seminfo}_j$, $j = 1, \ldots, k$ since it is necessary for successful termination that there are proofs of these judgements, and by line (10) there is a substitution ρ such that $\sigma = \rho \circ \sigma_{i_j}$ where σ_{i_j} is the substitution associated with the root of t_{i_j}. Applying the inference rule chosen in the last iteration leads to the desired proof, since the condition is also satisfied by line (12) and successful termination.

Now, suppose Algorithm $prove(\Gamma \vdash t : \mathsf{seminfo})$ fails to construct a proof of $\sigma\Gamma \vdash t : \sigma\mathsf{seminfo}$. For correctness, it is sufficient and necessary that there is no proof of $\Gamma \vdash t : \mathsf{seminfo}$. Suppose there would be a proof and let r be the last applied inference rule (this is applied at the root of t). Then, there is an iteration of the while loop where r is considered. By induction hypothesis, the recursive calls in line (8)–(11) cannot fail. Thus, the only reason for failure would be that the condition φ is not satisfied or the equalities Eq cannot be proved. However, then r cannot be the last inference rule applied in the proof of $\sigma\Gamma \vdash t : \sigma\mathsf{seminfo}$. ■

In general, Algorithm *prove* is inefficient and may lead to exponential execution time. However, it seems that there are not many special inference rules per production (e.g., see the Appendix). It is even not necessary to require that every production has exactly one inference rule to ensure at most $O(n)$ recursive calls of *prove*, where n is the number of nodes in the abstract syntax tree of the program. Instead, it is sufficient to require that every (direct or indirect) *recursive* production has exactly one inference rule. In this case, when Algorithm *prove* backtracks, then it does not backtrack in the recursive calls of *prove* which ensures at most $O(n)$ calls of *prove*. The other performance bottleneck is the proof completion with general inference rules in line (12). These rules are used to prove equalities on functions and relations on sorts. A practical implementation would avoid the proof completion and implement these sorts and functions by data structures (e.g. environments by definition tables, type hierarchies by directed acyclic graphs using adjacency lists).

Algorithm *prove* assumes that the conditions and equalities can be checked at the node where an inference rule is applied. If this assumption is dropped, then the condition or equality can be marked as open and associated with the node. If it cannot be proved or disproved, then the check is postponed, i.e. moved to the parent of the node. After termination of Algorithm *prove* all conditions must be true. If there remain conditions which are not true but satisfiable, these conditions have to be checked at runtime and the specification cannot be checked statically.

4 RELATED WORK

Attribute grammars are a well-known specification and generation method for static semantics [WG84]. Each attribute grammar can be transformed into a many-sorted natural semantics specification. The attributes in the attribute grammar correspond to the sorts in the judgements of a many-sorted natural semantics specification. The attribute values correspond with the semantic information in the judgements. Each inherited attribute becomes part of the context information while synthesized attributes are mapped to pieces of semantic information. Conditions of the attribute rules are transformed directly into conditions of the corresponding special inference rules. Functions on attributes are transformed into general inference rules. The inverse transformation, natural semantics into attribute grammars, is discussed in [Att88, AP94, GZ97]. These works show that this transformation requires structural conditions on the inference rules. Therefore, many-sorted natural semantics specifications are at least as powerful as attribute grammars.

TYPOL is a specification language for common (one-sorted) natural semantics, implemented in the CENTAUR system [BCD+88, Des88]. The CENTAUR system searches for a proof using a Prolog implementation. Our search algorithm is a generalization of their search algorithm. [Pet95, Pet96] introduces an alternative approach to implement natural semantics. The main goal is to improve the performance of the CENTAUR system.

An alternative approach to specify and generate type analysis in object-oriented programming languages is type inference by constraint-solving [PS94]. However, the specification mechanism is very restricted and would not suffice for the specification of languages like e.g. Mini-Java and Pascal.

5 CONCLUSION

We have shown that natural semantics can be extended such that it is applicable for imperative and object-oriented programming languages. This extension is achieved by defining a many-sorted version of natural semantics. In particular, this allows us to define structured context information. We are also able to specify different semantic information for each node in a program. This extension allows us to define arbitrary semantic information instead of only type information as it is the case in functional programming languages and their specifications. It turns out that the program is part of the proof of its static correctness. We presented an algorithm that checks whether the program can be completed to a proof or not.

Our specification language is designed such that each specification consists of three parts: The first contains all sort definitions that are valid for the particular specification. The second part comprises general inference rules that define the properties of the introduced new sorts. In the third part, we define the context-sensitive properties of nodes in a program by specifying special

inference rules that are associated with production rules of the underlying context-free grammar. Furthermore, we have identified which properties of a programming language or a single program are static and which can only be checked dynamically during run time. We showed that many-sorted natural semantics is at least as expressive as attribute grammars.

There remain some open problems with which we want to deal in future work: The question is how we can recognize which properties in a specification can be checked statically and which properties we can only check during run time. Up to now we have implemented the semantic analysis as a search algorithm. The question here is how we can cut down the search space to make the analysis more efficient. One source of inefficiency is the proof search arising from the application of general inference rules. These specify standard functions typically used in compilers (e.g. symbol table, subtype relations, etc.). Therefore replacing the search by an explicit implementation of these functions would improve the efficiency of the semantic analysis.

Acknowledgments: We thank the anonymous referees for their valuable comments. This work is partially supported by DFG project *Verifix* and by the Graduiertenkolleg *Beherrschbarkeit komplexer Systeme*.

REFERENCES

[AP94] Isabelle Attali and Didier Parigot. Integrating Natural Semantics and Attribute Grammars: the Minotaur System. Technical Report 2339, Institut National de Recherche en Informatique et en Automatique (INRIA), September 1994.

[Att88] Isabelle Attali. Compiling Typol with Attribute Grammars. In Pierre Deransart, Bernard Lorho, and Jan Maluszynski, editors, *Programming Language Implementation and Logic Programming, 1st International Workshop PLILP'88*, pages 252–272, Orléans, France, May 16-18 1988. Springer, Lecture Notes in Computer Science, Vol. 348.

[BCD+88] P. Borras, D. Clément, Th. Despeyroux, J. Incerpi, G. Kahn, B. Lang, and V. Pascual. CENTAUR: the system. In *Proceedings of the Third Symposium on Software Development Environments (SDE3), ACM Sigsoft '88*, Boston, December 1988. also appears as INRIA research report no. 777, Dec. 1987.

[Des88] T. Despeyroux. Typol: a formalism to implement Natural Semantics. INRIA research report 94, INRIA, 1988.

[Gen69] G. Gentzen. Investigation into Logical Deduction (Thesis 1935). Reprinted in "The collected papers of Gerhard Gentzen" E. Szabo, North-Holland, Amsterdam, 1969.

[GZ97] Sabine Glesner and Wolf Zimmermann. Using Many-Sorted Inference Rules to Generate Semantic Analysis. In Otto

Spaniol, editor, *Proceedings des Workshops der Informatik-Graduiertenkollegs "Promotion tut not: Innovationsmotor Graduiertenkolleg" im Rahmen der GI-Jahrestagung 1997*. Verlag der Augustinus Buchhandlung (Aachener Beitrge zur Informatik, Band 21), 1997.

[Kah87] Gilles Kahn. Natural Semantics. In Franz-Josef Brandenburg, Guy Vidal-Naquet, and Martin Wirsing, editors, *Proceedings of the 4th Annual Symposium on Theoretical Aspects of Computer Science (STACS'87)*, pages 22–39, Passau, Germany, February 1987. Springer, LNCS 247.

[Mit90] John C. Mitchell. *Type Systems for Programming Languages*, volume B of *Handbook of Theoretical Computer Science*, chapter 8, pages 365–458. MIT Press/Elsevier Science Publishers B.V., 1990.

[Pet95] Mikael Pettersson. *Compiling Natural Semantics*. PhD thesis, Department of Computer and Information Science, Linköping University, S-581 83 Linköping, Sweden, 1995.

[Pet96] Mikael Pettersson. A Compiler for Natural Semantics. In *Proceedings of the 6th International Conference on Compiler Construction,, CC'96*, Linköping, Sweden, April 1996. Springer, Lecture Notes in Computer Science, Vol. 1060.

[PS94] Jens Palsberg and Michael I. Schwartzbach. *Object-Oriented Type Systems*. Wiley Professional Computing, 1994.

[WG84] William M. Waite and Gerhard Goos. *Compiler Construction*. Springer Verlag, Berlin, New York Inc., 1984.

APPENDIX 1 SPECIFICATION OF THE STATIC SEMANTICS OF MINI-JAVA

Mini-Java is a Java subset taking into account inheritance, subclassing, and polymorphism of Java (overloading is not included). This appendix gives a definition of the static semantics by many-sorted natural semantics. First we show the sorts used in the specification. In particular, there are sorts for subtyping, class environments, overriding of methods, and local definitions. The sort **Cl_Env** is partly defined by the function *Thru* which is used to override class environments by another class environment. Furthermore, there are the predefined sorts **GenInfo** = {correct}, sets, lists, and strings which are lists of characters.

$$\begin{aligned}
\textbf{Type} &= \textbf{String} \\
\textbf{Types} &= \{\textbf{Type}\} \\
\textbf{Subtyping} &= \{\sqsubseteq(\textbf{Type}, \textbf{Type})\} \\
\textbf{Cl_Env_Item} &= \{\mathsf{meth}, \mathsf{attr}\} \times \textbf{Sig}
\end{aligned}$$

$$\begin{array}{rcl}
\mathbf{Cl_Env} & = & \{\mathbf{Cl_Env_Item}\} \uplus \mathit{Thru}(\mathbf{Subtyping}, \mathbf{Cl_Env}, \mathbf{Cl_Env}) \\
\mathbf{Env} & = & \{\mathbf{Type} \times \mathbf{Cl_Env}\} \\
\mathbf{Sig} & = & \mathbf{String} \times \mathbf{Type} \times \mathbf{Type} \uplus \mathbf{String} \times \mathbf{Type} \\
\mathbf{Overriding} & = & \{\circ(\mathbf{Subtyping}, \mathbf{Sig}, \mathbf{Sig})\} \\
\mathbf{Local} & = & \{\mathsf{loc}, \mathsf{ret}, \mathsf{inp}\} \times \mathbf{String} \times \mathbf{Type} \\
\mathbf{Locals} & = & \{\mathbf{Local}\}
\end{array}$$

For compactness reasons, we introduce the three sorts

$$\begin{array}{lcl}
\mathbf{Context}_1 & := & \mathbf{Types} \times \mathbf{Subtyping} \times \mathbf{Env} \\
\mathbf{Context}_2 & := & \mathbf{Context}_1 \times \mathbf{Type} \\
\mathbf{Context}_3 & := & \mathbf{Context}_2 \times \mathbf{Locals}
\end{array}$$

The general inference rules define the transitivity of subtyping and overriding of signatures (determined by the result type).

$$\frac{\mathsf{subtypes} \therefore \mathbf{Subtyping}; \quad A \sqsubseteq B \in \mathsf{subtypes} \therefore \mathbf{BOOL}; \quad B \sqsubseteq C \in \mathsf{subtypes} \therefore \mathbf{BOOL}}{A \sqsubseteq C \in \mathsf{subtypes} \therefore \mathbf{BOOL}}$$

$$\frac{\mathsf{subtypes} \therefore \mathbf{Subtyping}}{A \sqsubseteq A \in \mathsf{subtypes} \therefore \mathbf{BOOL}}$$

$$\frac{\begin{array}{c}\langle \mathsf{id}, \mathsf{type}, \mathsf{res_type}_1 \rangle \therefore \mathbf{Sig}; \quad \langle \mathsf{id}, \mathsf{type}, \mathsf{res_type}_2 \rangle \therefore \mathbf{Sig} \\ \mathsf{subtypes} \therefore \mathbf{Subtyping}; \quad \mathsf{res_type}_1 \sqsubseteq \mathsf{res_type}_2 \in \mathsf{subtypes} \therefore \mathbf{Bool} \\ \mathsf{overrides} \therefore \mathbf{Overriding}\end{array}}{\circ(\mathsf{subtypes}, \langle \mathsf{id}, \mathsf{type}, \mathsf{res_type}_1 \rangle, \langle \mathsf{id}, \mathsf{type}, \mathsf{res_type}_2 \rangle) \in \mathsf{overrides} \therefore \mathbf{Bool}}$$

$$\frac{\mathsf{subtypes} \therefore \mathbf{Subtyping}; \quad X \therefore \mathbf{Sig}}{\mathit{Thru}(\mathsf{subtypes}, \emptyset, X) = \emptyset \therefore \mathbf{Bool}} \qquad \frac{\mathsf{subtypes} \therefore \mathbf{Subtyping}; \quad X \therefore \mathbf{Sig}}{\mathit{Thru}(\mathsf{subtypes}, X, \emptyset) = X \therefore \mathbf{Bool}}$$

$$\frac{\begin{array}{c}\mathsf{subs} \therefore \mathbf{Subtyping}; \quad \mathsf{Intf} \therefore \mathbf{Sig}; \quad \mathsf{Intf}' \therefore \mathbf{Sig} \\ \mathsf{Cl_Env} \therefore \mathbf{Cl_Env}; \quad X \therefore \mathbf{Cl_Env} \\ \neg \circ (\mathsf{subs}, \mathsf{Intf}, \mathsf{Intf}') \therefore \mathbf{Bool}\end{array}}{\mathit{Thru}(\mathsf{subs}, \mathsf{Cl_Env} \uplus \{\mathsf{Intf}\}, X \uplus \{\mathsf{Intf}'\}) = \mathit{Thru}(\mathsf{subs}, \mathsf{Cl_Env} \uplus \{\mathsf{Intf}\}, X) \therefore \mathbf{Bool}}$$

if $\forall \mathsf{Intf}'' \in \mathsf{Cl_Env} : \neg \circ (\mathsf{subs}, \mathsf{Intf}'', \mathsf{Intf}')$

$$\frac{\begin{array}{c}\mathsf{subs} \therefore \mathbf{Subtyping}; \quad \mathsf{Intf} \therefore \mathbf{Sig}; \quad \mathsf{Intf}' \therefore \mathbf{Sig}; \\ \mathsf{Cl_Env} \therefore \mathbf{Cl_Env}; \quad X \therefore \mathbf{Cl_Env} \\ \circ(\mathsf{subs}, \mathsf{Intf}, \mathsf{Intf}') \therefore \mathbf{Bool}\end{array}}{\mathit{Thru}(\mathsf{subs}, \mathsf{Cl_Env} \uplus \{\mathsf{Intf}\}, X \uplus \{\mathsf{Intf}'\}) = \mathit{Thru}(\mathsf{subs}, \mathsf{Cl_Env}, X) \therefore \mathbf{Bool}}$$

All other inference rules are special inference rules.

Production (1): $\mathit{prog} ::= \mathit{classes}$

$$\frac{\langle \mathsf{Names}, \mathsf{TH}, \mathsf{Intfs} \rangle \therefore \mathbf{Context}_1 \vdash \mathit{classes} : \mathsf{Names} \therefore \mathbf{Types}, \mathsf{TH} \therefore \mathbf{Subtyping}, \mathsf{Intfs} \therefore \mathbf{Env}}{\vdash \mathit{prog} : \mathsf{correct} \therefore \mathbf{GenInfo}}$$

if $\forall A \sqsubseteq B \in \mathsf{TH} \Rightarrow A = B \vee B \sqsubseteq A \notin \mathsf{TH}$

Production (2): $\mathit{classes}_0 ::= \mathit{class}; \mathit{classes}_1$

$$\frac{\begin{array}{c}\Gamma \therefore \mathbf{Context}_1 \vdash \mathit{class} : \mathsf{Name} \therefore \mathbf{Type}, \mathsf{TH}_1 \therefore \mathbf{Subtyping}, \mathsf{Intfs}_1 \therefore \mathbf{Env} \\ \Gamma \therefore \mathbf{Context}_1 \vdash \mathit{classes}_1 : \mathsf{Names} \therefore \mathbf{Types}, \mathsf{TH}_2 \therefore \mathbf{Subtyping}, \mathsf{Intfs}_2 \therefore \mathbf{Env}\end{array}}{\Gamma \therefore \mathbf{Context}_1 \vdash \mathit{classes}_0 : \begin{array}{l}\mathsf{Names} \cup \{\mathsf{Name}\} \therefore \mathbf{Types}, \mathsf{TH}_1 \cup \mathsf{TH}_2 \therefore \mathbf{Subtyping}, \\ \mathsf{Intfs}_1 \cup \mathsf{Intfs}_2 \therefore \mathbf{Env}\end{array}}$$

if $\mathsf{Name} \notin \mathsf{Names}$

Production (3): $classes ::= ;$ $\quad \Gamma \therefore \mathbf{Context}_1 \vdash classes : \emptyset \therefore \mathbf{Types}, \emptyset \therefore \mathbf{Subtyping}, \emptyset \therefore \mathbf{Env}$

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

$$\frac{\langle \mathsf{Names}, \mathsf{TH}, \mathsf{Intfs} \uplus \{\langle \mathsf{id}_2, \mathsf{Cl_Intfs}_1\rangle\}, \mathsf{id}_1\rangle \therefore \mathbf{Context}_2 \vdash features : \mathsf{Cl_Intfs}_2 \therefore \mathbf{Cl_Env}}{\begin{array}{c}\langle \mathsf{Names}, \mathsf{TH}, \mathsf{Intfs} \uplus \{\langle \mathsf{id}_2, \mathsf{Cl_Intfs}_1\rangle\}\rangle \therefore \mathbf{Context}_1 \vdash class : \mathsf{id}_1 \therefore \mathbf{Type}, \\ \{\mathsf{id}_1 \sqsubseteq \mathsf{id}_2\} \therefore \mathbf{Subtyping}, \{\langle \mathsf{id}_1, Thru(\mathsf{TH}, \mathsf{Cl_Intfs}_1, \mathsf{Cl_Intfs}_2) \cup \mathsf{Cl_Intfs}_2\rangle\} \therefore \mathbf{Env}\end{array}}$$

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

$$\frac{\langle \mathsf{Names}, \mathsf{TH}, \mathsf{Intfs}, \mathsf{id}\rangle \therefore \mathbf{Context}_2 \vdash features : \mathsf{Cl_Intfs} \therefore \mathbf{Cl_Env}}{\langle \mathsf{Names}, \mathsf{TH}, \mathsf{Intfs}\rangle \therefore \mathbf{Context}_1 \vdash class : \begin{array}{l}\mathsf{id} \therefore \mathbf{Type}, \{\langle \mathsf{id}, \mathsf{Cl_Intfs}\rangle\} \therefore \mathbf{Env}, \\ \emptyset \therefore \mathbf{Subtyping}\end{array}}$$

Production (6): $features_0 ::= feature; features_1$

$$\frac{\begin{array}{l}\Gamma \therefore \mathbf{Context}_2 \vdash feature : \mathsf{Intf} \therefore \mathbf{Cl_Env_Item} \\ \Gamma \therefore \mathbf{Context}_2 \vdash features_1 : \mathsf{Cl_Intfs} \therefore \mathbf{Cl_Env}\end{array}}{\Gamma \therefore \mathbf{Context}_2 \vdash features_0 : \{\mathsf{Intf}\} \cup \mathsf{Cl_Intfs} \therefore \mathbf{Cl_Env}}$$

if $[\exists string \therefore \mathbf{String}, t_1, t_2 \therefore \mathbf{Type} : \mathsf{Intf} = \langle \mathsf{meth}, \langle string, t_1, t_2\rangle\rangle \Rightarrow$
$\forall \langle \mathsf{meth}, \langle string', t_1', t_2'\rangle\rangle \in \mathsf{Cl_Intfs} : string \neq string'] \wedge$
$[\exists string \therefore \mathbf{String}, t \therefore \mathbf{Type} : \mathsf{Intf} = \langle \mathsf{attr}, \langle string, t\rangle\rangle \Rightarrow$
$\forall \langle \mathsf{attr}, \langle string', t'\rangle\rangle \in \mathsf{Cl_Intfs} : string \neq string']$

Production (7): $features ::= ;$ $\quad \Gamma \therefore \mathbf{Context}_2 \vdash features : \emptyset \therefore \mathbf{Cl_Env}$

Production (8) : $feature ::= \mathsf{id} : type$

$$\frac{\Gamma \therefore \mathbf{Context}_2 \vdash type : \mathsf{t} \therefore \mathbf{Type}}{\Gamma \therefore \mathbf{Context}_2 \vdash feature : \langle \mathsf{attr}, \langle \mathsf{id}, \mathsf{t}\rangle\rangle \therefore \mathbf{Cl_Env_Item}}$$

Production (9) : $feature ::= \texttt{method}\ \mathsf{id}_1(\mathsf{id}_2 : type_1) : type_2; block$

$$\frac{\begin{array}{l}\langle \mathsf{Names}, \mathsf{TH}, \mathsf{Intfs}, \mathsf{A}\rangle \therefore \mathbf{Context}_2 \vdash type_1 : \mathsf{t}_1 \therefore \mathbf{Type} \\ \langle \mathsf{Names}, \mathsf{TH}, \mathsf{Intfs}, \mathsf{A}\rangle \therefore \mathbf{Context}_2 \vdash type_2 : \mathsf{t}_2 \therefore \mathbf{Type} \\ \langle \mathsf{Names}, \mathsf{TH}, \mathsf{Intfs}, \mathsf{A}, \mathsf{locals} \cup \{\langle \mathsf{inp}, \mathsf{id}_2, \mathsf{t}_1\rangle, \langle \mathsf{ret}, \mathsf{result}, \mathsf{t}_2\rangle\}\rangle \therefore \mathbf{Context}_3 \\ \qquad \vdash block : \mathsf{locals} \therefore \mathbf{Locals}\end{array}}{\langle \mathsf{Names}, \mathsf{TH}, \mathsf{Intfs}, \mathsf{A}\rangle \therefore \mathbf{Context}_2 \vdash feature : \langle \mathsf{meth}, \langle \mathsf{id}_1, \mathsf{t}_1, \mathsf{t}_2\rangle\rangle \therefore \mathbf{Cl_Env_Item}}$$

if $id_2 \notin \{y | \exists x, z : \langle x, y, z\rangle \in \mathsf{locals}\} \cup \{\mathsf{result}\}$

Production (10) : $type ::= \mathsf{id}$

$\langle \mathsf{Names} \uplus \{\mathsf{id}\}, \mathsf{TH}, \mathsf{Intfs}, \mathsf{A}\rangle \therefore \mathbf{Context}_2 \vdash type : \mathsf{id} \therefore \mathbf{Type}$
$\langle \mathsf{Names} \uplus \{\mathsf{id}\}, \mathsf{TH}, \mathsf{Intfs}, \mathsf{A}, \mathsf{locals}\rangle \therefore \mathbf{Context}_3 \vdash type : \mathsf{id} \therefore \mathbf{Type}$

Production (11) : $block ::= \texttt{begin}\ decls\ stats\ \texttt{end}$

$$\frac{\Gamma \therefore \mathbf{Context}_3 \vdash decls : \mathsf{locals} : \mathbf{Locals}; \quad \Gamma \therefore \mathbf{Context}_3 \vdash stats : \mathsf{correct} \therefore \mathbf{GenInfo}}{\Gamma \therefore \mathbf{Context}_3 \vdash block : \mathsf{locals} \therefore \mathbf{Locals}}$$

Production (12) : $stats_0 ::= stat; stats_1$

$$\frac{\Gamma \therefore \mathbf{Context}_3 \vdash stat : \mathsf{correct} \therefore \mathbf{GenInfo}; \quad \Gamma \therefore \mathbf{Context}_3 \vdash stats_1 : \mathsf{correct} \therefore \mathbf{GenInfo}}{\Gamma \therefore \mathbf{Context}_3 \vdash stats_0 : \mathsf{correct} \therefore \mathbf{GenInfo}}$$

Production (13): $stats ::= ;$ $\quad \Gamma \therefore \mathbf{Context}_3 \vdash stats : \mathsf{correct} \therefore \mathbf{GenInfo}$

Production (14): $decls_0 ::= \mathsf{id} : type; decls_1$

$$\frac{\Gamma \therefore \mathbf{Context}_3 \vdash type : \mathsf{t} \therefore \mathbf{Type}; \quad \Gamma \therefore \mathbf{Context}_3 \vdash decls_1 : \mathsf{locals} \therefore \mathbf{Locals}}{\Gamma \therefore \mathbf{Context}_3 \vdash decls_0 : \{\langle \mathsf{loc}, \mathsf{id}, \mathsf{t}\rangle\} \cup \mathsf{locals} \therefore \mathbf{Locals}}$$

if $\mathsf{id} \notin \{y | \exists x, z : \langle x, y, z\rangle \in \mathsf{locals}\} \cup \{\mathsf{result}\}$

Production (15): $decls ::= ;$ $\quad \Gamma \therefore \mathbf{Context}_3 \vdash decls : \emptyset \therefore \mathbf{Locals}$

Production (16): $stat ::= des := expr$

$$\frac{\begin{array}{l}\langle \mathsf{Names}, \mathsf{TH}, \mathsf{Intfs}, \mathsf{A}, \mathsf{locals}\rangle \therefore \mathbf{Context}_3 \vdash des : \mathsf{t}_1 \therefore \mathbf{Type} \\ \langle \mathsf{Names}, \mathsf{TH}, \mathsf{Intfs}, \mathsf{A}, \mathsf{locals}\rangle \therefore \mathbf{Context}_3 \vdash expr : \mathsf{t}_2 \therefore \mathbf{Type}\end{array}}{\langle \mathsf{Names}, \mathsf{TH}, \mathsf{Intfs}, \mathsf{A}, \mathsf{locals}\rangle \therefore \mathbf{Context}_3 \vdash stat : \mathsf{correct} \therefore \mathbf{GenInfo}}$$

if $\mathsf{t}_2 \sqsubseteq \mathsf{t}_1 \in \mathsf{TH}$

Production (17) $des_0 ::= des_1.\mathsf{id}$

$$\frac{\begin{array}{c}\langle \mathsf{Names}, \mathsf{TH}, \mathsf{Intfs} \uplus \{\langle \mathsf{t}_1, \mathsf{Cl_Intfs} \uplus \{\langle \mathsf{attr}, \langle \mathsf{id}, \mathsf{t}_2\rangle\rangle\}\rangle\}, \mathsf{A}, \mathsf{locals}\rangle \therefore \mathbf{Context_3} \\ \vdash des_1 : \mathsf{t}_1 \therefore \mathbf{Type}\end{array}}{\begin{array}{c}\langle \mathsf{Names}, \mathsf{TH}, \mathsf{Intfs} \uplus \{\langle \mathsf{t}_1, \mathsf{Cl_Intfs} \uplus \{\langle \mathsf{attr}, \langle \mathsf{id}, \mathsf{t}_2\rangle\rangle\}\rangle\}, \mathsf{A}, \mathsf{locals}\rangle \therefore \mathbf{Context_3} \\ \vdash des_0 : \mathsf{t}_2 \therefore \mathbf{Type}\end{array}}$$

Production (18) $des ::= \mathsf{id}$

$$\langle \mathsf{Names}, \mathsf{TH}, \mathsf{Intfs} \uplus \{\langle \mathsf{A}, \mathsf{Cl_Intfs} \uplus \{\langle \mathsf{attr}, \langle \mathsf{id}, \mathsf{t}\rangle\rangle\}\rangle\}, \mathsf{A}, \mathsf{locals}\rangle \therefore \mathbf{Context_3} \vdash des : \mathsf{t} \therefore \mathbf{Type}$$
$$\text{if } \mathsf{id} \notin \{y \mid \exists x, z : \langle x, y, z\rangle \in \mathsf{locals}\}$$

$$\langle \mathsf{Names}, \mathsf{TH}, \mathsf{Intfs}, \mathsf{A}, \mathsf{locals} \uplus \{\langle \mathsf{x}, \mathsf{id}, \mathsf{t}\rangle\}\rangle \therefore \mathbf{Context_3} \vdash des : \mathsf{t} \therefore \mathbf{Type}$$

Production (19) $des ::= \mathsf{result}$

$$\langle \mathsf{Names}, \mathsf{TH}, \mathsf{Intfs}, \mathsf{A}, \mathsf{locals} \uplus \{\langle \mathsf{x}, \mathsf{result}, \mathsf{t}\rangle\}\rangle \therefore \mathbf{Context_3} \vdash des : \mathsf{t} \therefore \mathbf{Type}$$

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

$$\frac{\begin{array}{c}\langle \mathsf{Names}, \mathsf{TH}, \mathsf{Intfs} \uplus \{\langle \mathsf{t}_1, \mathsf{Cl_Intfs} \uplus \{\langle \mathsf{meth}, \langle \mathsf{id}, \mathsf{t}, \mathsf{ret}\rangle\rangle\}\rangle\}, \mathsf{A}, \mathsf{locals}\rangle \therefore \mathbf{Context_3} \\ \vdash des_1 : \mathsf{t}_1 \therefore \mathbf{Type} \\ \langle \mathsf{Names}, \mathsf{TH}, \mathsf{Intfs} \uplus \{\langle \mathsf{t}_1, \mathsf{Cl_Intfs} \uplus \{\langle \mathsf{meth}, \langle \mathsf{id}, \mathsf{t}, \mathsf{ret}\rangle\rangle\}\rangle\}, \mathsf{A}, \mathsf{locals}\rangle \therefore \mathbf{Context_3} \\ \vdash expr : \mathsf{t}_2 \therefore \mathbf{Type}\end{array}}{\begin{array}{c}\langle \mathsf{Names}, \mathsf{TH}, \mathsf{Intfs} \uplus \{\langle \mathsf{t}_1, \mathsf{Cl_Intfs} \uplus \{\langle \mathsf{meth}, \langle \mathsf{id}, \mathsf{t}, \mathsf{ret}\rangle\rangle\}\rangle\}, \mathsf{A}, \mathsf{locals}\rangle \therefore \mathbf{Context_3} \\ \vdash des_0 : \mathsf{ret} \therefore \mathbf{Type}\end{array}}$$
$$\text{if } \mathsf{t}_2 \sqsubseteq \mathsf{t} \in \mathsf{TH}$$

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

$$\frac{\begin{array}{c}\langle \mathsf{Names}, \mathsf{TH}, \mathsf{Intfs} \uplus \{\langle \mathsf{A}, \mathsf{Cl_Intfs} \uplus \{\langle \mathsf{meth}, \langle \mathsf{id}, \mathsf{t}, \mathsf{ret}\rangle\rangle\}\rangle\}, \mathsf{A}, \mathsf{locals}\rangle \therefore \mathbf{Context_3} \\ \vdash expr : \mathsf{t}_1 \therefore \mathbf{Type}\end{array}}{\begin{array}{c}\langle \mathsf{Names}, \mathsf{TH}, \mathsf{Intfs} \uplus \{\langle \mathsf{A}, \mathsf{Cl_Intfs} \uplus \{\langle \mathsf{meth}, \langle \mathsf{id}, \mathsf{t}, \mathsf{ret}\rangle\rangle\}\rangle\}, \mathsf{A}, \mathsf{locals}\rangle \therefore \mathbf{Context_3} \\ \vdash des : \mathsf{ret} \therefore \mathbf{Type}\end{array}}$$
$$\text{if } \mathsf{t}_1 \sqsubseteq \mathsf{t} \in \mathsf{TH} \wedge \mathsf{id} \notin \{y \mid \exists x, z : \langle x, y, z\rangle \in \mathsf{locals}\}$$

Production (22) $expr ::= des$

$$\frac{\Gamma \therefore \mathbf{Context_3} \vdash des : \mathsf{t} \therefore \mathbf{Type}}{\Gamma \therefore \mathbf{Context_3} \vdash expr : \mathsf{t} \therefore \mathbf{Type}}$$

Production (23) $expr ::= \mathtt{new}\ \mathsf{id}$

$$\langle \mathsf{Names} \uplus \{\mathsf{id}\}, \mathsf{TH}, \mathsf{Intfs}, \mathsf{A}, \mathsf{locals}\rangle \therefore \mathbf{Context_3} \vdash expr : \mathsf{id} \therefore \mathbf{Type}$$

About the authors:

Sabine Glesner is a PhD student in the Computer Science Department at the University of Karlsruhe in Prof. Goos' research group. Currently, she is working on semantic analysis. Under a Fulbright grant, she received her M.S. in Computer Science from the University of California, Berkeley, in 1994. Her master's thesis is about representation and inference of uncertain knowledge. In 1996, she received an *Informatik-Diplom* from the University of Darmstadt. Her diploma thesis deals with automated theorem proving. From 1991 to 1996, she was a member of the *Studienstiftung des deutschen Volkes*, the German National Scholarship Foundation.

Wolf Zimmermann studied computer science at the University of Karlsruhe from 1982 to 1987 where he received his diploma. From 1987 to 1988 he was staff member at GMD Forschungsstelle at Karlsruhe. He joined the University of Karlsruhe as a staff member from 1988 to 1992 where he received the PhD in 1990 awarded by the "Preis des Fördervereins des Forschungszentrum Informatik". From 1990 to 1991, he was postdoctoral fellow at the International Computer Science Institute in Berkeley. Since 1992 he is senior scientist at the University of Karlsruhe. His main research fields are parallel computing, construction of correct compilers, software libraries, and foundations of object-oriented computing.