August 1997

# Using Many-Sorted Inference Rules to Generate Semantic Analysis

Sabine Glesner and Wolf Zimmermann

*Institut für Programmstrukturen und Datenorganisation,*
*Universität Karlsruhe,*
*76128 Karlsruhe, Germany*
{glesner|zimmer}@ipd.info.uni-karlsruhe.de

**Abstract.** We introduce a specification language that can be used to specify semantic analysis as well as intermediate code generation. This specification language defines semantic properties by means of many-sorted inference rules. Type inference rules are just a one-sorted special case. We demonstrate that inference rules can also be used to infer other semantic information such as definitions of identifiers, scoping, inheritance relations, etc. To distinguish the different kinds of semantic information described by the rules, we use a many-sorted language to formulate the inference rules. We further show how to transform a set of inference rules algorithmically into an attribute grammar, thus proving that semantic analysis and intermediate code generation can be generated from such specifications.

**Keywords:** Inference rules, specification language, semantic analysis, intermediate code representation, compiler generators

# 1. Introduction

In the literature, there are usually two extreme ways to specify semantic analysis: Either the specification is rather operational, or the specification is declarative. In the first case, generators for the semantic analysis exist while in the second case, the specifications are only used as a support for the manual implementation of the semantic analysis. Attribute Grammars are an example of the former, type inference in functional programming languages is an example of the latter. Designing attribute grammars requires a lot of experience because it is necessary to justify that the specifying attribute grammar really defines the desired static semantics of the programming language. However, when type inference rules specify the static semantics, this justification is almost straightforward. Despite these advantages, type inference rules are mostly used to specify the static semantics of functional languages which have a rich type system but rather simple scoping rules. Also, in functional languages, each identifier and each node in the abstract syntax tree has a type. These properties do not hold for imperative and object-oriented languages: Their type systems are rather primitive compared to functional languages while their scoping rules are much richer.[1] Moreover, most of these languages are typed so that instead of type inference it is more type checking what needs to be done when performing semantic analysis.

An inference style specification has several advantages. First, it abstracts completely from particular traversals of the abstract syntax tree. It does not matter whether these are derived from local traversal specifications (as in ordered attribute grammars) or whether they are specified explicitly (as in LAG-grammars or syntax-directed translations). Secondly, a notion of consistency and completeness can be defined formally. As usual, consistency means that no contradictory information can be derived while completeness means that all information desired in a certain sense is indeed specified by the inference rules so that it could be derived in principle.

The specification method presented in this paper combines the advantages of both extreme approaches. It extends the idea of type inference rules so that other static semantic information can be defined in the same framework of inference rules without specifying the complete semantics of a programming language. Furthermore, we show how to transform such inference rules into attribute grammars. Thus, the static semantics of imperative and object-oriented languages can be specified in the declarative inference rule style while it is still possible to generate the semantic analysis from such declarative specifications. General properties of semantic information that do not depend on particular programs (such as subtype relations in imperative languages, coercibility etc.) are usually specified outside of the heart of the specification language. Attribute grammars for example require to implement functions such as coercions, balancing types etc. in the implementation language of the generated code

[1] It is specific to object-oriented languages that the programmer can specify the subtype relation. In other languages, this relation is specified in the language definition.

for the semantic analysis. In contrast to this, our approach allows us to define these properties also in the style of inference rules, i.e., there is no need to leave the formal framework when specifying such information. At this first step, it is not our goal to provide already an efficient generator. We merely show that it is possible to generate the semantic analysis from inference style specifications.

So there are the following requirements that we pose on our specification method: We want it to be declarative. It should be a closed framework in the sense that all information can be specified within the framework. It should be applicable for realistic programming languages. A notion of consistency and completeness can be defined. The semantic analysis can be generated from such specifications. And we want that the transformation into the intermediate language can also be specified in the framework. (This aspect is not in the major focus of this paper but we will argue rather shortly that it is also possible.)

The paper is organized as follows: In Section 2 we introduce the syntax and semantics of our specification language and demonstrate its use on small examples. The specification of a complete language which contains already typical properties of realistic languages – a notion of inheritance and overloading similar to Java, a non-trivial scoping rule, and automatic coercions from integers to floating point numbers – is given in the appendix. In Section 3 we show how to transform our specifications into attribute grammars. In section 4 we point out which other aspects we want to investigate in future work. Section 5 compares our approach with related work and investigates in how far they satisfy the requirements stated above. We conclude in section 6.

## 2. Specification Language

The specification language is based on the context-free grammar which defines the structure of the programming language. We assign a set of inference rules to each production rule of the context-free grammar. These inference rules describe semantic properties of the nodes in the AST. Since a node might have more than just one particular semantic information as for example its type, we need to be able to describe several semantic properties within the specification language. This is the reason why we decided to choose a many-sorted language to specify the static semantics of programming languages. The sort of a semantic information indicates the kind of property it describes. When carrying out the semantic analysis for a given program, we try to find a rule cover of the abstract syntax tree. Thereby exactly one rule has to be applicable for each node in the AST. If it is possible to find such a rule cover of the tree, then the program is correct wrt. its static semantics.

First, we describe the syntax of the specification language and show for small examples how it works. The complete specification of an example language can be found in the appendix. Then we proceed by defining the semantics of the specification lan-

guage. For that we show how to transform a specification into a first-order language so to achieve a standard representation whose semantics is well-known.

### 2.1. Syntax of the Specification Language

Since, in general, it is necessary to describe various kinds of semantic properties for nodes in the AST, we use a many-sorted specification language to describe the static semantics of programs. We assign a specific sort to each piece of semantic information. There are already certain predefined sorts in the specification language: The sort **String** denotes all strings of finite length. The sort **Bool** contains the truth values true and false. To describe the most general property of nodes in the AST, we use the sort **GenInfo**. This sort contains the three elements correct, incorrect and don't-know. For example, information that describes the properties of statements in a program will have this sort. It indicates whether the statements under consideration are correct, incorrect, or whether it is still unknown if they are correct or not. Further predefined sorts are sets, lists, and cartesian products of already existing sorts. These kinds of sorts are denoted as usual by $\{\}$, $*$, and $\times$. In this sense, the three symbols $\{\}$, $*$, and $\times$ as well as **String**, **Bool**, and **GenInfo** are sort constructor functions and sort constructor constants, resp. The set of all predefined sorts are the ground terms built from sort constructor symbols, i.e., sort constructor terms containing no variables. When specifying the static semantics of a programming language, it might be helpful to define new sorts. This is possible by stating that the new sort $s$ is built from already exisiting sorts $s_1, \ldots, s_n$ by applying the new sort constructor function $F$ of sort $s$ to them: $s := F(s_1, \ldots, s_n)$. For example, when describing object-oriented programming languages, one needs to define the type hierarchy specified in a particular program. Therefore it is necessary to define the sort **Subtyping** which basically contains tuples of strings: $\textbf{Subtyping} := \{\sqsubseteq (\textbf{String}, \textbf{String})\}$. If $A$ is a subtype of $B$, then $\sqsubseteq (A, B)$ holds.

Semantic properties are described with terms of one of the sorts. Terms are built as usual from variable, constant, and function symbols. The constant symbols [] (the empty list), $\emptyset$ (the empty set), true, false, and correct, incorrect and don't-know (the values of sort **GenInfo**) as well as the function symbols $[e \mid l]$ (adding an element $e$ to a list $l$), $\cup, \cap, \in, \subseteq, =$ (the operations for manipulating sets), and $\vee, \wedge, \neg, \rightarrow, \leftrightarrow$ (the functions for describing truth values) can be used for building terms.

The specification of a programming language consists of three major parts: the definition of new sorts, general inference rules, and special inference rules associated with particular production rules of the underlying context-free grammar. As described above, it is possible to define new sorts in a specification by defining with which new sort constructor symbols and from which already existing sorts this new sort is derived. It might be necessary to define the properties of such a new sort. This can be done by stating general inference rules. They have a similar syntactic structure as special inference rules.

An inference rule is a triple consisting of a set of assumptions $S_1, \ldots, S_m$, a set of consequences $C_1, \ldots, C_n$, and a condition $\varphi$. The assumptions and consequences are sequences, as defined below, whereas the condition $\varphi$ is a term of sort **Bool**. We write an inference rule in the following form:

$$\frac{S_1, \ldots, S_m}{C_1, \ldots, C_n} \text{ if } \varphi$$

A sequence is a triple. The first component is a context $\Gamma$ which is a semantic information. The second component is a syntactic construct $p$, and the third component is a semantic information *SemInfo*. (In general inference rules, sequences contain only a semantic information.) We write down such a sequence in the following form:

$$\Gamma \vdash p : \mathit{SemInfo}$$

A semantic information is a tupel consisting of a term $t$ and its sort $S$, denoted as $t \therefore S$.

The following two examples are intended to give an intuitive idea how a specification might look like. The complete specification for an example language is given in the appendix.

**General Inference Rule:** To define **Subtyping** as a transitive relation, one would specify the following rule whereby subtypes is assumed to be a free variable which is implicitly existence-quantified. This means that the rule can be applied when a concrete value is substituted for subtypes.

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

**Special Inference Rule:** Consider the production

$$\mathit{stat} \quad ::= \quad \mathit{des} \texttt{ := } \mathit{expr}$$

The associated special inference rule is defined as follows:

$$\frac{\begin{array}{l}\Gamma \vdash \mathit{des} : t_1 \therefore \mathbf{Type} \\ \Gamma \vdash \mathit{expr} : t_2 \therefore \mathbf{Type}\end{array}}{\Gamma \vdash \mathit{des} ::= \mathit{expr} : \mathsf{correct} \therefore \mathbf{GenInfo}} \quad \text{if } \sqsubseteq (t_2, t_1)$$

We call a specification *consistent wrt. sort S* iff, for each program, each node in the AST has at most one semantic information of sort $S$. We call a specification *consistent* iff, for each program, each node in the AST has at most one semantic information for each sort. A specification is complete wrt. sort S iff, for each program, each node in the AST has at least one semantic information of sort $S$. In particular, a specification is complete wrt. the transformation into the intermediate representation iff, for each program, each node in the AST has a semantic information specifying how to translate the node into the intermediate representation. The specifications of functional languages are typically complete wrt. the type information because each node in the AST has a type in functional programming languages.

### 2.2. Semantics of the Specification Language

We can think of the sorts in the specification language as unary functions defined on the nodes of the abstract syntax tree. Each sort is a function that assigns a value, a semantic information of this sort, to each node. The first-order meaning of a sequence

$$\Gamma \therefore \mathbf{context} \vdash \ p \ : \ \mathit{SemInfo} \therefore \mathbf{Sort}$$

would then be defined as follows:

$$\mathbf{context}(p) = \Gamma \rightarrow \mathbf{Sort}(p) = \mathit{SemInfo}.$$

## 3. Generating Algorithm

At first sight it might seem strange that there are two different pieces of semantic information in one sequence. One might think that they can be both combined in a single piece of semantic information. But we have good reason to choose this kind of information splitting: In nearly all programming languages, the AST nodes have properties that can be inferred from their successors and properties which are derived from other surrounding nodes, usually called the context information. *SemInfo* describes the semantic information which can be derived from the successors of a node. The context information is captured in the context $\Gamma$. So the difference between the two pieces of semantic information in a sequence is the direction in the AST in which they are defined. Moreover, one can imagine that the semantic information *SemInfo* can be inferred from the context. This is the reason why we call a sequence also a *semantic information judgement*.

The basic idea is to convert the sorts of the specification language into attributes. Each node in the abstract syntax tree which gets a semantic information of some sort assigned by one of the inference rules will have an attribute of the corresponding kind. The information before $\therefore$ specifies the value of the attributes. In particular, the context which is a piece of semantic information with which every AST node is equipped will be transformed into an environment attribute. Since we might have contexts of different sorts, we might also have different kinds of environment attributes. Conditions of inference rules lead to conditions in attribute grammars. Each inference rule defines a set of attribution rules:

- The contexts of the assumptions are defined based on the contexts of the consequences. The inference rule specifies the operations to be used to construct the contexts of the assumptions from the contexts of the consequences. The contexts correspond with inherited attributes in attribute grammars due to the direction in the AST in which they are defined.
- The other attributes are defined on the consequences based on the assumptions. The information before $\therefore$ specifies the values of the attributes. The inference rule specifies the operations to be used to construct the attributes in the consequences

from the attributes of the assumptions. These attributes are synthesized because they depend solely on the successors of a node.

- General inference rules specify properties of certain attributes. They can be computed by hull algorithms as explained below.

Consider the following inference rule

$$\frac{\begin{array}{rcl} f_1(\Gamma, X_{1,1}, \ldots, X_{1,n_1}) \therefore \mathbf{Context}_{f_1} & \vdash & Y_1 : v_{1,1} \therefore \mathbf{A}_{1,1} \\ & \vdots & \\ f_1(\Gamma, X_{1,1}, \ldots, X_{1,n_1}) \therefore \mathbf{Context}_{f_1} & \vdash & Y_1 : v_{1,r_1} \therefore \mathbf{A}_{1,r_1} \\ f_2(\Gamma, X_{2,1}, \ldots, X_{2,n_1}) \therefore \mathbf{Context}_{f_2} & \vdash & Y_2 : v_{2,1} \therefore \mathbf{A}_{2,1} \\ & \vdots & \\ f_2(\Gamma, X_{2,1}, \ldots, X_{2,n_1}) \therefore \mathbf{Context}_{f_2} & \vdash & Y_2 : v_{2,r_2} \therefore \mathbf{A}_{2,r_2} \\ & \vdots & \\ f_m(\Gamma, X_{m,1}, \ldots, X_{2,n_m}) \therefore \mathbf{Context}_{f_m} & \vdash & Y_m : v_{m,1} \therefore \mathbf{A}_{m,1} \\ & \vdots & \\ f_m(\Gamma, X_{m,1}, \ldots, X_{2,n_m}) \therefore \mathbf{Context}_{f_m} & \vdash & Y_m : v_{m,r_m} \therefore \mathbf{A}_{m,r_m} \end{array}}{\begin{array}{rcl} \Gamma \therefore \mathbf{Context}_j & \vdash & Y_0 : g_1(w_{1,1}, \ldots, w_{1,s_1}) \therefore \mathbf{B}_1 \\ \Gamma \therefore \mathbf{Context}_j & \vdash & Y_0 : g_2(w_{2,1}, \ldots, w_{2,s_2}) \therefore \mathbf{B}_2 \\ & \vdots & \\ \Gamma \therefore \mathbf{Context}_j & \vdash & Y_0 : g_m(w_{m,1}, \ldots, w_{m,s_m}) \therefore \mathbf{B}_m \end{array}} \quad \text{if } h(u_1, \ldots, u_l)$$

associated with the production $Y_0 ::= Y_1 \cdots Y_m$, where each $w_{i,j}$ equals to one $v_{i',j'}$, each $u_i$ and each $X_{i,j}$ may be any other information occuring in this rule (except the condition). Then:

- The grammar symbol $Y_i$, $i = 1, \ldots, m$ has the attributes $A_{i,j}$, $j = 1, \ldots, r_j$ and the attribute $env_{f_i}$.
- The grammar symbol $Y_0$ has the attributes $B_1, \ldots, B_k$ and the attribute $env_j$.
- The attribution rules associated to the inference rule are:

$$\begin{array}{rcl} Y_1.env_{f_1} & := & f_1(Y_0.env_j, X_{1,1}, \ldots, X_{1,n_1}) \\ & \vdots & \\ Y_m.env_{f_m} & := & f_m(Y_0.env_j, X_{m,1}, \ldots, X_{m,n_m}) \\ Y_0.B_1 & := & g_1(Z_{1,1}.C_{1,1}, \ldots, Z_{1,s_1}.C_{1,s_1}) \\ & \vdots & \\ Y_0.B_m & := & g_m(Z_{m,1}.C_{m,1}, \ldots, Z_{m,s_m}.C_{m,s_m}) \end{array}$$

where $Z_{i,j}$ is one of the grammar symbols $Y_1, \ldots, Y_m$ and $C_{i,j}$ is one of the sorts occuring in the assumptions associated to symbol $Z_{i,j}$, if for example $\Gamma' \vdash Z_{i,j} : w_{i,j} \therefore C_{i,j}$ were an assumption.

- $h(W_1.D_1, \ldots, W_l.D_l)$ is the condition associated to the production $Y_0 ::= Y_1 \cdots Y_m$. Here, there is a judgment $\Gamma' \vdash W_i : u_i \therefore D_i$ in an assumption or consequence for each $i = 1, \ldots, l$, or $W_i.D_i$ is an environment attribute of one of the production symbols.

General inference rules describe properties of sorts or attributes, resp. which have been defined in a given specification. Sorts or attributes can be seen as relations. A relation itself is a subset of the cartesian product built over the sets from which the elements in a single tuple are taken. The general inference rules describe properties of these subsets. If some elements are in the set, then also others have to be in the set. We can think of the general inference rules as describing a hull algorithm. As an example, think of the general inference rule describing the transitivity of a relation. It specifies how to augment a set such that the result is transitive. Since in each computation of a semantic information or attribute value, resp., there are only finitely many values, such a hull algorithm stops. When transforming a specification into an attribute grammar, for each attribute which has been introduced into the attribute grammar because of a new-defined sort in the specification, a function *update* is defined which computes exactly the convex hull as described above. As soon as an attribute has been computed, its final value needs to be determined by application of *update*.

Most of the inference rules allow for pattern matching. For example, the components of the context are enumerated in an inference rule so that the direct access to them is easily possible without any other notational overhead. When transforming a specification into an attribute grammar, it is necessary to automatically derive selector functions from the sort definitions and to transform the specification into one which does not incorporate any pattern-matching. Thereby it is not necessary to formulate the selection operators within the specification language. This is easily possible and only a minor point here.

## 4. Future Work

### 4.1. Transformation into Attribute Grammars

In the above transformation of a specification into an attribute grammar, we have assumed that the context is specified in the opposite direction than the semantic information on the right-hand side of the inference rule. Therefore, it was secured that the resulting attribute grammar is well-defined. In future work, we will try to relax the above assumptions on the definitions in a specification. Moreover, it is necessary to find efficient implementations of the sketched operations.

### 4.2. Transformation in Intermediate Language

In the same way as we have specified semantic information relevant for the semantic analysis, it is possible to define a new sort **Intermediate_Code** and specify semantic information of this sort. If we do this such that each node in the abstract syntax tree gets a semantic information of sort **Intermediate_Code**, then it is easily possible to generate the transformation into the intermediate language from such a specification. Here it makes sense to define that a specification is complete wrt. the transformation into the intermediate language. This is the case if, for each program, each node in the abstract syntax tree has a semantic information describing how to translate it. The thorough investigation of this topic is the subject of future work.

### 4.3. Static Type Safety

An important property of a programming language is its type safety. We will examine the question if we can prove, given an inference rule style specification of a language, whether this language is type-safe. Since many object-oriented languages are not statically type-safe, it is an interesting question if dynamic type checks can be generated automatically from a specification.

## 5. Related Work

We have introduced a method for specifying the semantic analysis of programming languages. Specifications written down in this formalism are declarative. We argued that realistic languages can be described by this method. Furthermore, we defined the notions of consistency and completeness. Due to the many-sortedness of our specification language, we can define different kinds of semantic information within one closed framework. Thereby, the kind of a semantic information can be everything from a type up to the (intermediate) code to which the program fragment will be translated. This means that we can in particular define the interaction of the programming language with the intemediate representation. And last but not least, we can generate the semantic analysis and also the transformation into the intermediate language from these specifications.

Research on the specification of semantic properties of programming languages and the generation of their semantic analyses was pushed forward with attribute grammars. A good survey of the obtained results can be found in [13]. The actual algorithms for the semantic analysis are simple but will fail on certain input programs if the underlying attribute grammar is not well-defined. Testing if a grammar is well-defined, however, requires exponential time [2]. A sufficient condition for being well-defined can be checked in polynomial time. This test defines the set of ordered attribute grammars as being a subset of the well-defined grammars [5]. However, there is no constructive method to design such grammars. Hence, designing an or-

dered attribute grammar remains a difficult problem. For another class of attribute grammars it is required that all attributes can be evaluated during a constant number of depth-first, left-to-right traversals of the abstract syntax tree. These are the left-ordered attribute grammars (LAG), [6], [1]. Due to their fixed traversal order, the specification of context-sensitive syntax becomes very operational, i.e. dependent on the analysis. However, because there are no alternative specification and generation methodologies, most practical tools are based on attribute grammars. Attribute grammars allow in particular that different kinds of semantic information can be specified, in particular the transformation into the intermediate representation. A notion of consistency and completeness can be defined.

A language for the specification of context-sensitive syntax which is based solely on the predicate calculus is defined in [7]. Even though this method is completely declarative, it is not intuitive due to the complexity of first-order formulas. Realistic programming languages can be specified in this framework as it is demonstrated at the example specification of Oberon. A notion of consistency and completeness is not of interest in this approach and therefore not investigated. Different kinds of semantic information can be described by first-order predicates but the interaction with the intermediate representation is not under consideration. The semantic analysis can be generated but is much too inefficient for the use in practical compilers.

In functional programming languages, type inference and checking is performed by solving systems of type equations [3]. During this computation it is necessary to unify terms denoting types. The unification method chosen is typically Robinson's [12]. Since we restrict ourselves to the checking of typed programming languages and do not require type variables, this approach is more general than necessary in our context. It cannot be used easily for non-functional programming languages. A notion of consistency and completeness is not of interest in this approach and therefore not developed. It is not possible to describe several kinds of semantic information within this framework because it was designed with emphasis on type inference. In particular, the interaction with the intermediate language cannot be defined. The generation of the semantic analysis specified with type inference rules can be generated from such descriptions.

In [11], a specification method for the semantic analysis in object-oriented languages based on constraints is given. It is declarative in the sense that it allows for the specification of constraints which are propositinal formulas that define the semantical correctness of a given program. It is not possible to describe realistic programming languages by this method because already the normal coercion in arithmetic operations which depends on both operands cannot be specified. A notion of consistency and completeness has not been further investigated. It is not possible to define several kinds of semantic information in this method since it only allows for the specification of type inference. In particular, the interaction with the intermediate language cannot be described. The semantic analysis restricted to type inference can be generated from such specifications and has time complexity $\mathcal{O}(n^3)$ where $n$ is the program size.

Plotkin's structured operational semantics (SOS) [9] describing the semantics of programming languages is based on automata. Simple rules define state transitions. Since the thereby described configuration changes happen only at the rewrite component, this way of defining semantics is also called small steps semantics. It is a declarative method but differs from ours in the sense that we do not have an underlying abstract machine whose operational semantics gives meaning to our inference rules. SOS rules can be seen as state machines while deduction systems are proof procedures. In this sense, our inference rules describe a proof procedure (and not a state machine) with which we can establish certain properties for a program (fragment).

A similar approach influenced by structured operational semantics and by Gentzen's Calculus for Natural Deduction Systems ([10]) is defined by natural semantics [4]. In this method, semantic information can be described in a logic. Inference rules describe a proof procedure with which certain semantic information can be inferred. Different logics need to be specified when different kinds of semantic information should be described. It is possible to generate an efficient semantic analysis from natural semantics specifications as it is demonstrated in [8].

Our approach is also of proof-theoretic nature: We want to prove that certain nodes in the AST have values of a specific sort. The inference rules that we provide allow us to do such reasoning within a formal system. The main difference to natural semantics is the use of a many-sorted specification language: We allow to specify different kinds of semantic values in a single specification and distinguish them by their sort tag. In natural semantics, only one kind of semantic information can be described in one specification. Different logics are necessary to describe for example the type system, the interpretation of the language by the intermediate or goal language, and other properties of the programming language. In contrast, we can describe all these different aspects in one unified framework by attaching sorts to the semantic information to distinguish the different kinds of them. The advantage of one single many-sorted logic for the specification of the semantic analysis is that information of different sorts can depend on each other and that these interdependencies can be specified.

## 6. Conclusion

We showed that semantic analysis can be specified by means of many-sorted inference rules. They generalize the type inference method commonly used to specify the type systems of functional programming languages. Since it is necessary to also define other information than just types, we need to be able to indicate the kind of information being described. This is realized by introducing a many-sorted specification language that assigns sorts to the semantic information which describes program properties. In this sense, type inference rules are a one-sorted special case of our specification language. A specification given in this specification language consists of three parts: (i) the definition of the kind of information by means of sorts, (ii)

the definition of general properties independent of particular programs by means of general many-sorted inference rules, and (iii) the definition of properties associated with productions by means of special many-sorted inference rules. We showed that it is possible to generate the semantic analysis module in compilers from such specifications. More precisely, it is possible to transform a specification based on inference rules into an attribute grammar. If we can ensure that this attribute grammar is well-defined, we can generate the semantic analysis from inference rule style specifications by using compiler generators for attribute grammars. We gave an easy sufficient condition on the specification to ensure that the resulting attribute grammar is well-defined. This sufficient condition might be too restrictive. Its generalization is subject of future work.

It was not our purpose to demonstrate already an efficient generator or an efficient generated semantic analysis. However, if the attribute grammar is well-defined, it is possible to implement semantic analysis as a kind of topological sorting of the attribute dependencies. This implies that the semantic analysis itself will be efficient as long as supporting functions (e.g. a data structure for the context) are implemented efficiently. A library of typical data structures and functions used in semantic analysis may guarantee the efficient implementation of supporting functions and data structures. Thus, the main focus of future work will be on efficient generation and on identifying typical data structures to ensure efficient semantic analysis.

**Acknowledgement:** The authors would like to thank Uwe Aßmann for many valuable discussions and comments while working on this paper.

## References

1. G. V. Bochmann. Semantic Evaluation from Left to Right. *Communications of the ACM*, 19(2):55–62, 1976.
2. M. Jazayeri. A Simpler Construction Showing the Intrinsically Exponential Complexity of the Circularity Problem for Attribute Grammars. *Journal of the ACM*, 28(4):715–720, 1981.
3. Simon L. Peyton Jones. *The Implementation of Functional Programming Languages.* Prentice-Hall International Series in Computer Science, 1987.
4. 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.
5. U. Kastens. Ordered Attribute Grammars. *Acta Informatica*, 13(3):229–256, 1980.
6. P. M. Lewis, D. J. Rosenkrantz, and R. E. Stearns. Attributed Translations. *Journal of Computer and System Sciences*, 9(3):279–307, 1974.
7. Martin Odersky. Defining context-dependent syntax without using contexts. *ACM Transactions on Programming Languages and Systems*, 15(3):535–562, July 1993.
8. Mikael Pettersson. *Compiling Natural Semantics.* PhD thesis, Department of Computer and Information Science, Linköping University, S-581 83 Linköping, Sweden, 1995.
9. Gordon D. Plotkin. A structural approach to operational semantics. Report DAIMI FN-19, Computer Science Department, Aarhus University, Denmark, September 1981.

10. Dag Prawitz. *Natural Deduction.* Almqvist & Wiksell, Stockholm, 1965.
11. Jens Palsberg and Michael I. Schwartzbach. *Object-Oriented Type Systems.* Wiley Professional Computing, 1994.
12. J.A. Robinson. A machine-oriented logic based on the resolution principle. *Journal of the ACM*, 12(1):23–41, 1965.
13. William M. Waite and Gerhard Goos. *Compiler Construction.* Springer Verlag, Berlin, New York Inc., 1984.

# Appendix Example Specification

## Context-free Grammar

| | | | |
|---|---|---|---|
| *program* | ::= | *classes* | *(1)* |
| *classes* | ::= | *class* ; *classes* \| ; | *(2,3)* |
| *class* | ::= | `class` *id* `extends` *id* ; *features* `end` ; | *(4)* |
| | \| | `class` *id* ; *features* `end` ; | *(5)* |
| *features* | ::= | *feature* ; *features* \| ; | *(6,7)* |
| *feature* | ::= | *id* : *type* | *(8)* |
| | \| | `method` *id parameters* : *type* ; *block* | *(9)* |
| *parameters* | ::= | $\varepsilon$ \| `(` *pars* `)` | *(10,11)* |
| *pars* | ::= | *id* : *type* \| *id* : *type* , *pars* | *(12,13)* |
| *type* | ::= | `INT` \| `REAL` \| `BOOL` \| *id* | *(14-17)* |
| *block* | ::= | `begin` *stats* `end` | *(18)* |
| *stats* | ::= | *( stat* \| *decl )* ; *stats* \| ; | *(19,20)* |
| *decl* | ::= | *id* : *type* | *(21)* |
| *stat* | ::= | *des* `:=` *expr* | *(22)* |
| | \| | `while` *expr* `do` *stats* `od` | *(23)* |
| *des* | ::= | *des* . *id* \| *id* \| `result` | *(24-26)* |
| | \| | *des* . *id* `(` *args* `)` \| `this` | *(27,28)* |
| *args* | ::= | *expr* \| *expr* , *args* | *(29,30)* |
| *expr* | ::= | *des* \| *int_literal* \| *real_literal* \| *bool_literal* | *(31-34)* |
| | \| | *expr* `+` *expr* \| `new` *id* \| `null` | *(35-37)* |

## Sort Definitions

$$\begin{aligned}
\textbf{Type} &:= \textbf{String} \\
\textbf{Types} &:= \{\textbf{String}\} \\
\textbf{Type_List} &:= [\textbf{Type}] \\
\textbf{Subtyping} &:= \{\sqsubseteq (\textbf{String}, \textbf{String})\} \\
\textbf{Class_Env} &:= \{\mathsf{method}, \mathsf{attribute}\} \times \textbf{Type} \times \textbf{Signature} \\
\textbf{Class_Envs} &:= \{\textbf{Class_Env}\}
\end{aligned}$$

$$
\begin{aligned}
\mathbf{Signature} &:= \mathbf{String} \times [\mathbf{Type}] \times \mathbf{Type} \cup \mathbf{String} \times \mathbf{Type} \\
\mathbf{Overriding} &:= \{\circ(\mathbf{Signature}, \mathbf{Signature})\} \\
\mathbf{List\_Equality} &:= \{=_{[]} ([\mathbf{Type}], [\mathbf{Type}])\} \\
\mathbf{List\_Coercibility} &:= \{\sqsubseteq_{[]} ([\mathbf{Type}], [\mathbf{Type}])\} \\
\mathbf{Local\_Def} &:= \{\mathsf{local}, \mathsf{ret}, \mathsf{input}\} \times \mathbf{String} \times \mathbf{Type} \\
\mathbf{Local\_Defs} &:= \{\mathbf{Local\_Def}\} \\
\mathbf{Context}_1 &:= \{\mathbf{Types}\} \times \{\mathbf{Subtyping}\} \times \{\mathbf{Class\_Envs}\} \\
\mathbf{Context}_2 &:= \mathbf{Context}_1 \times \mathbf{Type} \\
\mathbf{Context}_3 &:= \mathbf{Context}_2 \times \mathbf{Local\_Defs}
\end{aligned}
$$

## General Inference Rules

$$
\frac{\begin{array}{l}\mathsf{subtypes} \therefore \mathbf{Subtyping} \\ \sqsubseteq (A, B) \in \mathsf{subtypes} \therefore \mathbf{Bool} \\ \sqsubseteq (B, C) \in \mathsf{subtypes} \therefore \mathbf{Bool}\end{array}}{\sqsubseteq (A, C) \in \mathsf{subtypes} \therefore \mathbf{Bool}}
\qquad
\frac{\begin{array}{l}\mathsf{subtypes} \therefore \mathbf{Subtyping} \\ \mathsf{t} \therefore \mathbf{Type} \\ \mathsf{void} \therefore \mathbf{Type}\end{array}}{\sqsubseteq (\mathsf{void}, \mathsf{t}) \in \mathsf{subtypes} \therefore \mathbf{Bool}}
$$

$$
\frac{\mathsf{subtypes} \therefore \mathbf{Subtyping}}{\sqsubseteq (A, A) \in \mathsf{subtypes} \therefore \mathbf{Bool}}
\qquad
\frac{\mathsf{subtypes} \therefore \mathbf{Subtyping}}{\sqsubseteq (\mathsf{int}, \mathsf{real}) \in \mathsf{subtypes} \therefore \mathbf{Bool}}
$$

$$
\frac{\begin{array}{l}\langle \mathsf{id}_1, \mathsf{type\_list}_1, \mathsf{res\_type}_1 \rangle \therefore \mathbf{Signature} \\ \langle \mathsf{id}_2, \mathsf{type\_list}_2, \mathsf{res\_type}_2 \rangle \therefore \mathbf{Signature} \\ \mathsf{subtypes} \therefore \mathbf{Subtyping} \\ \sqsubseteq (\mathsf{res\_type}_1, \mathsf{res\_type}_2) \in \mathsf{subtypes} \\ \mathsf{list\_equals} \therefore \mathbf{List\_Equality} \\ =_{[]} (\mathsf{type\_list}_1, \mathsf{type\_list}_2) \in \mathsf{list\_equals} \\ \mathsf{overrides} \therefore \mathbf{Overriding}\end{array}}{\circ(\langle \mathsf{id}_1, \mathsf{type\_list}_1, \mathsf{res\_type}_1 \rangle, \langle \mathsf{id}_2, \mathsf{type\_list}_2, \mathsf{res\_type}_2 \rangle) \in \mathsf{overrides}}
$$

$$
\frac{\mathsf{list\_equals} \therefore \mathbf{List\_Equality}}{=_{[]} ([], []) \in \mathsf{list\_equals}}
\qquad
\frac{\begin{array}{l}\mathsf{list\_equals} \therefore \mathbf{List\_Equality} \\ =_{[]} (l_1, l_2) \in \mathsf{list\_equals}\end{array}}{=_{[]} ([t \mid l_1], [t \mid l_2]) \in \mathsf{list\_equals}}
$$

$$
\frac{\mathsf{list\_coercions} \therefore \mathbf{List\_Coercibility}}{\sqsubseteq_{[]} ([], []) \in \mathsf{list\_coercions}}
\qquad
\frac{\begin{array}{l}\mathsf{list\_coercions} \therefore \mathbf{List\_Coercibility} \\ \sqsubseteq_{[]} (l_1, l_2) \in \mathsf{list\_coercions} \\ \mathsf{subtypes} \therefore \mathbf{Subtyping} \\ \sqsubseteq (\mathsf{A}, \mathsf{B}) \in \mathsf{subtypes}\end{array}}{\sqsubseteq_{[]} ([\mathsf{A} \mid l_1], [\mathsf{B} \mid l_2]) \in \mathsf{list\_coercions}}
$$

## Special Inference Rules

**Production (1):**

$$\frac{\begin{array}{l}\langle \mathsf{Names}, \mathsf{TH}, \mathsf{Infs}\rangle \therefore \mathbf{Context}_1 \vdash \mathit{classes} : \mathsf{Names} \therefore \mathbf{Types} \\ \langle \mathsf{Names}, \mathsf{TH}, \mathsf{Infs}\rangle \therefore \mathbf{Context}_1 \vdash \mathit{classes} : \mathsf{TH} \therefore \mathbf{Subtyping} \\ \langle \mathsf{Names}, \mathsf{TH}, \mathsf{Infs}\rangle \therefore \mathbf{Context}_1 \vdash \mathit{classes} : \mathsf{Intfs} \therefore \mathbf{Class\_Envs}\end{array}}{\langle \mathsf{Names}, \mathsf{TH}, \mathsf{Infs}\rangle \therefore \mathbf{Context}_1 \vdash \mathit{program} : \mathsf{correct} \therefore \mathbf{Gen\_Info}}$$

if $\sqsubseteq (A, B) \in \mathsf{TH} \wedge A \neq B \rightarrow \neg \sqsubseteq (B, A) \in \mathsf{TH}$

**Production (2):**

$$\frac{\begin{array}{l}\langle \mathsf{Names}, \mathsf{TH}, \mathsf{Infs}\rangle \therefore \mathbf{Context}_1 \vdash \mathit{class} : \mathsf{Name}_1 \therefore \mathbf{Type} \\ \langle \mathsf{Names}, \mathsf{TH}, \mathsf{Infs}\rangle \therefore \mathbf{Context}_1 \vdash \mathit{class} : \mathsf{TH}_1 \therefore \mathbf{Subtyping} \\ \langle \mathsf{Names}, \mathsf{TH}, \mathsf{Infs}\rangle \therefore \mathbf{Context}_1 \vdash \mathit{class} : \mathsf{Intfs}_1 \therefore \mathbf{Class\_Envs} \\ \langle \mathsf{Names}, \mathsf{TH}, \mathsf{Infs}\rangle \therefore \mathbf{Context}_1 \vdash \mathit{classes} : \mathsf{Names}_2 \therefore \mathbf{Types} \\ \langle \mathsf{Names}, \mathsf{TH}, \mathsf{Infs}\rangle \therefore \mathbf{Context}_1 \vdash \mathit{classes} : \mathsf{TH}_2 \therefore \mathbf{Subtyping} \\ \langle \mathsf{Names}, \mathsf{TH}, \mathsf{Infs}\rangle \therefore \mathbf{Context}_1 \vdash \mathit{classes} : \mathsf{Intfs}_2 \therefore \mathbf{Class\_Envs}\end{array}}{\begin{array}{l}\langle \mathsf{Names}, \mathsf{TH}, \mathsf{Infs}\rangle \therefore \mathbf{Context}_1 \vdash \mathit{class}\ ;\ \mathit{classes} : \{\ \mathsf{Name}_1\} \cup \mathsf{Names}_2 \therefore \mathbf{Types} \\ \langle \mathsf{Names}, \mathsf{TH}, \mathsf{Infs}\rangle \therefore \mathbf{Context}_1 \vdash \mathit{class}\ ;\ \mathit{classes} : \mathsf{TH}_1 \cup \mathsf{TH}_2 \therefore \mathbf{Subtyping} \\ \langle \mathsf{Names}, \mathsf{TH}, \mathsf{Infs}\rangle \therefore \mathbf{Context}_1 \vdash \mathit{class}\ ;\ \mathit{classes} : \mathsf{Intfs}_1 \cup \mathsf{Intfs}_2 \therefore \mathbf{Class\_Envs}\end{array}}$$

if $\mathsf{Name}_1 \notin \mathsf{Names}_2$

**Production (3):**

$$\frac{}{\begin{array}{l}\Gamma \therefore \mathbf{Context}_1 \vdash\ ; : \emptyset \therefore \mathbf{Types} \\ \Gamma \therefore \mathbf{Context}_1 \vdash\ ; : \emptyset \therefore \mathbf{Subtyping} \\ \Gamma \therefore \mathbf{Context}_1 \vdash\ ; : \emptyset \therefore \mathbf{Class\_Env}\end{array}}$$ if true

**Production (4):**

$$\frac{\langle \mathsf{Names}, \mathsf{TH}, \mathsf{Intfs}_1, \mathsf{id}_1\rangle \therefore \mathbf{Context}_2 \vdash \mathit{features} : \mathsf{Intfs}_2 \therefore \mathbf{Class\_Envs}}{\begin{array}{l}\langle \mathsf{Names}, \mathsf{TH}, \mathsf{Intfs}_1\rangle \therefore \mathbf{Context}_1 \vdash \texttt{class}\ \mathit{id}_1\ \texttt{extends}\ \mathit{id}_2\ \texttt{;}\ \mathit{features}\ \texttt{end}\ \texttt{;} : \\ \qquad \{\sqsubseteq (\mathsf{id}_1, \mathsf{id}_2)\} \therefore \mathbf{Subtyping} \\ \langle \mathsf{Names}, \mathsf{TH}, \mathsf{Intfs}_1\rangle \therefore \mathbf{Context}_1 \vdash \texttt{class}\ \mathit{id}_1\ \texttt{extends}\ \mathit{id}_2\ \texttt{;}\ \mathit{features}\ \texttt{end}\ \texttt{;} : \\ \qquad \{\langle \mathsf{method}, \mathsf{id}_1, \mathsf{sig}\rangle \therefore \mathbf{Class\_Env} \mid \langle \mathsf{method}, \mathsf{id}_2, \mathsf{sig}\rangle \\ \qquad \in \mathsf{Intfs}_1 \wedge \forall\ \mathsf{sig'} \in \mathsf{Intfs}_2\ :\ \neg \circ (\mathsf{sig'}, \mathsf{sig})\} \\ \qquad \cup\ \mathsf{Intfs}_2 \therefore \mathbf{Class\_Envs}\end{array}}$$

if $\mathsf{id}_1 \in \mathsf{Names} \wedge\ \sqsubseteq (\mathsf{id}_1, \mathsf{id}_2) \in \mathsf{TH}\ \wedge\ \mathsf{Intfs}_2 \subseteq \mathsf{Intfs}_1$

**Production (5):**

$$\frac{\langle \mathsf{Names}, \mathsf{TH}, \mathsf{Intfs}_1, \mathsf{id}\rangle \therefore \mathbf{Context}_2 \vdash \mathit{features} : \mathsf{Intfs}_2 \therefore \mathbf{Class\_Envs}}{\begin{array}{l}\langle \mathsf{Names}, \mathsf{TH}, \mathsf{Intfs}_1\rangle \therefore \mathbf{Context}_1 \vdash \texttt{class}\ \mathit{id}\ \texttt{;}\ \mathit{features}\ \texttt{end}\ \texttt{;} : \mathsf{id} \therefore \mathbf{Type} \\ \langle \mathsf{Names}, \mathsf{TH}, \mathsf{Intfs}_1\rangle \therefore \mathbf{Context}_1 \vdash \texttt{class}\ \mathit{id}\ \texttt{;}\ \mathit{features}\ \texttt{end}\ \texttt{;} : \mathsf{Intfs}_2 \therefore \mathbf{Class\_Envs} \\ \langle \mathsf{Names}, \mathsf{TH}, \mathsf{Intfs}_1\rangle \therefore \mathbf{Context}_1 \vdash \texttt{class}\ \mathit{id}\ \texttt{;}\ \mathit{features}\ \texttt{end}\ \texttt{;} : \\ \qquad\qquad \{\sqsubseteq (\mathsf{id}, \mathsf{Supertype})\} \therefore \mathbf{Subtyping}\end{array}}$$

if $\mathsf{id} \in \mathsf{Names} \wedge \sqsubseteq (\mathsf{id}, \mathsf{Supertype}) \in \mathsf{TH} \wedge \mathsf{Intfs}_2 \subseteq \mathsf{Intfs}_1$

**Production (6):**

$$\frac{\begin{array}{r}\Gamma \therefore \mathbf{Context}_2 \vdash \mathit{feature} : \mathsf{Intf} \therefore \mathbf{Class\_Env} \\ \Gamma \therefore \mathbf{Context}_2 \vdash \mathit{features} : \mathsf{Intfs} \therefore \mathbf{Class\_Envs}\end{array}}{\Gamma \therefore \mathbf{Context}_2 \vdash \mathit{feature}\ \texttt{;}\ \mathit{features} : \{\mathsf{Intf}\} \cup \mathsf{Intfs} \therefore \mathbf{Class\_Envs}}$$

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

**Production (7):**

$$\frac{}{\Gamma \therefore \mathbf{Context}_2 \vdash\ \texttt{;} : \emptyset \therefore \mathbf{Class\_Envs}}\quad \text{if true}$$

**Production (8):**

$$\frac{\langle \mathsf{Names}, \mathsf{TH}, \mathsf{Intfs}, \mathsf{A}\rangle \therefore \mathbf{Context}_2 \vdash \mathit{type} : \mathsf{t} \therefore \mathbf{Type}}{\langle \mathsf{Names}, \mathsf{TH}, \mathsf{Intfs}, \mathsf{A}\rangle \therefore \mathbf{Context}_2 \vdash \mathit{id} : \mathit{type} : \langle \mathsf{attribute}, \mathsf{A}, \langle \mathsf{id}, \mathsf{t}\rangle\rangle \therefore \mathbf{Class\_Env}}$$

if true

**Production (9):**

$$\frac{\begin{array}{r}\langle \mathsf{Names}, \mathsf{TH}, \mathsf{Intfs}, \mathsf{A}\rangle \therefore \mathbf{Context}_2 \vdash \mathit{type} : \mathsf{t} \therefore \mathbf{Type} \\ \langle \mathsf{Names}, \mathsf{TH}, \mathsf{Intfs}, \mathsf{A}\rangle \therefore \mathbf{Context}_2 \vdash \mathit{parameters} : \mathsf{types} \therefore \mathbf{Type\_List} \\ \langle \mathsf{Names}, \mathsf{TH}, \mathsf{Intfs}, \mathsf{A}\rangle \therefore \mathbf{Context}_2 \vdash \mathit{parameters} : \mathsf{locals}_1 \therefore \mathbf{Local\_Defs} \\ \langle \mathsf{Names}, \mathsf{TH}, \mathsf{Intfs}, \mathsf{A}, \mathsf{locals}_1 \cup \mathsf{locals}_2 \\ \cup \{\langle \mathsf{ret}, \mathsf{result}, \mathsf{t}\rangle\}\rangle \therefore \mathbf{Context}_3 \vdash \mathit{block} : \mathsf{locals}_2 \therefore \mathbf{Local\_Defs}\end{array}}{\begin{array}{l}\langle \mathsf{Names}, \mathsf{TH}, \mathsf{Intfs}, \mathsf{A}\rangle \therefore \mathbf{Context}_2 \vdash \texttt{method}\ \mathit{id}\ \mathit{parameters}\ \texttt{:}\ \mathit{type}\ \texttt{;}\ \mathit{block} : \\ \qquad \mathsf{locals}_1 \cup \mathsf{locals}_2 \cup \{\langle \mathsf{ret}, \mathsf{result}, \mathsf{t}\rangle\} \therefore \mathbf{Local\_Defs} \\ \langle \mathsf{Names}, \mathsf{TH}, \mathsf{Intfs}, \mathsf{A}\rangle \therefore \mathbf{Context}_2 \vdash \texttt{method}\ \mathit{id}\ \mathit{parameters}\ \texttt{:}\ \mathit{type}\ \texttt{;}\ \mathit{block} : \\ \qquad \langle \mathsf{id}, \mathsf{types}, \mathsf{t}\rangle \therefore \mathbf{Signature} \\ \langle \mathsf{Names}, \mathsf{TH}, \mathsf{Intfs}, \mathsf{A}\rangle \therefore \mathbf{Context}_2 \vdash \texttt{method}\ \mathit{id}\ \mathit{parameters}\ \texttt{:}\ \mathit{type}\ \texttt{;}\ \mathit{block} : \\ \qquad \langle \mathsf{method}, \mathsf{A}, \langle \mathsf{id}, \mathsf{types}, \mathsf{t}\rangle\rangle \therefore \mathbf{Class\_Env}\end{array}}$$

if $\mathsf{locals}_1 \cap \mathsf{locals}_2 = \emptyset$

**Production (10):**

$$\frac{}{\Gamma \therefore \mathbf{Context}_2 \vdash \varepsilon : [] \therefore \mathbf{Type\_List} \quad \Gamma \therefore \mathbf{Context}_2 \vdash \varepsilon : \emptyset \therefore \mathbf{Local\_Defs}} \text{ if true}$$

**Production (11):**

$$\frac{\Gamma \therefore \mathbf{Context}_2 \vdash \mathit{pars} : \mathsf{types} \therefore \mathbf{Type\_List} \quad \Gamma \therefore \mathbf{Context}_2 \vdash \mathit{pars} : \mathsf{locals} \therefore \mathbf{Local\_Defs}}{\Gamma \therefore \mathbf{Context}_2 \vdash \texttt{(}\ \mathit{pars}\ \texttt{)} : \mathsf{types} \therefore \mathbf{Type\_List} \quad \Gamma \therefore \mathbf{Context}_2 \vdash \texttt{(}\ \mathit{pars}\ \texttt{)} : \mathsf{locals} \therefore \mathbf{Local\_Defs}} \text{ if true}$$

**Production (12):**

$$\frac{\Gamma \therefore \mathbf{Context}_2 \vdash \mathit{type} : \mathsf{t} \therefore \mathbf{Type}}{\Gamma \therefore \mathbf{Context}_2 \vdash \mathit{id}\ :\ \mathit{type} : \langle \mathsf{input}, \mathsf{id}, \mathsf{t} \rangle \therefore \mathbf{Local\_Defs}} \text{ if true}$$

**Production (13):**

$$\frac{\begin{array}{l}\Gamma \therefore \mathbf{Context}_2 \vdash \mathit{type} : \mathsf{t} \therefore \mathbf{Type} \\ \Gamma \therefore \mathbf{Context}_2 \vdash \mathit{pars} : \mathsf{types} \therefore \mathbf{Type\_List} \\ \Gamma \therefore \mathbf{Context}_2 \vdash \mathit{pars} : \mathsf{locals} \therefore \mathbf{Local\_Defs}\end{array}}{\begin{array}{l}\Gamma \therefore \mathbf{Context}_2 \vdash \mathit{id}\ :\ \mathit{type}\ ,\ \mathit{pars} : \\ \qquad [\mathsf{t} \mid \mathsf{types}] \therefore \mathbf{Type\_List} \\ \Gamma \therefore \mathbf{Context}_2 \vdash \mathit{id}\ :\ \mathit{type}\ ,\ \mathit{pars} : \\ \qquad \{\langle \mathsf{input}, \mathsf{id}, \mathsf{t} \rangle\} \cup \mathsf{locals} \therefore \mathbf{Local\_Defs}\end{array}}$$

if $\forall\ x, y : \neg\langle x, \mathsf{id}, y \rangle \in \mathsf{locals}$

**Production (14):**

$$\frac{}{\Gamma \therefore \mathbf{Context}_2 \vdash \texttt{INT}\ : \mathsf{int} \therefore \mathbf{Type}} \text{ if true}$$

**Production (15):**

$$\frac{}{\Gamma \therefore \mathbf{Context}_2 \vdash \texttt{REAL}\ : \mathsf{real} \therefore \mathbf{Type}} \text{ if true}$$

**Production (16):**

$$\frac{}{\Gamma \therefore \mathbf{Context}_2 \vdash \texttt{BOOL}\ : \mathsf{bool} \therefore \mathbf{Type}} \text{ if true}$$

**Production (17):**

$$\frac{}{\Gamma \therefore \mathbf{Context}_2 \vdash \mathit{id}\ : \mathsf{id} \therefore \mathbf{Type}} \text{ if true}$$

**Production (18):**

$$\frac{\Gamma \therefore \mathbf{Context}_3 \ \vdash \ \mathit{stats} : \mathsf{locals} \therefore \mathbf{Local\_Defs}}{\Gamma \therefore \mathbf{Context}_3 \ \vdash \ \texttt{begin}\ \mathit{stats}\ \texttt{end} : \mathsf{locals} \therefore \mathbf{Local\_Defs}} \quad \text{if true}$$

**Production (19.a):**

$$\frac{\Gamma \therefore \mathbf{Context}_3 \ \vdash \ \mathit{stats} : \mathsf{locals} \therefore \mathbf{Local\_Defs}}{\Gamma \therefore \mathbf{Context}_3 \ \vdash \mathit{stat}\ ;\ \mathit{stats} : \mathsf{locals} \therefore \mathbf{Local\_Defs}} \quad \text{if true}$$

**Production (19.b):**

$$\frac{\begin{array}{c}\Gamma \therefore \mathbf{Context}_3 \ \vdash \ \mathit{stats} : \mathsf{locals} \therefore \mathbf{Local\_Defs} \\ \Gamma \therefore \mathbf{Context}_3 \ \vdash \ \mathit{decl} : \mathsf{local} \therefore \mathbf{Local\_Def}\end{array}}{\Gamma \therefore \mathbf{Context}_3 \ \vdash \ \mathit{decl}\ ;\ \mathit{stats} : \{\mathsf{local}\} \cup \mathsf{locals} \therefore \mathbf{Local\_Defs}}$$
if $\exists\ x, \mathsf{id}, y\ \forall\ u, v : \langle x, \mathsf{id}, y\rangle = \mathsf{local} \wedge \neg\langle u, \mathsf{id}, v\rangle \in \mathsf{locals}$

**Production (20):**

$$\frac{}{\Gamma \therefore \mathbf{Context}_3 \ \vdash \ ; : \emptyset \therefore \mathbf{Local\_Defs}} \quad \text{if true}$$

**Production (21):**

$$\frac{\Gamma \therefore \mathbf{Context}_3 \ \vdash \ \mathit{type} : \mathsf{t} \therefore \mathbf{Type}}{\Gamma \therefore \mathbf{Context}_3 \ \vdash \ \mathit{id}\ :\ \mathit{type} : \langle \mathsf{local}, \mathsf{id}, \mathsf{t}\rangle \therefore \mathbf{Local\_Def}} \quad \text{if true}$$

**Production (22):**

$$\frac{\begin{array}{c}\langle \mathsf{Names}, \mathsf{TH}, \mathsf{Intfs}, \mathsf{A}, \mathsf{Locals}\rangle \therefore \mathbf{Context}_3 \ \vdash \ \mathit{des} : \mathsf{t}_1 \therefore \mathbf{Type} \\ \langle \mathsf{Names}, \mathsf{TH}, \mathsf{Intfs}, \mathsf{A}, \mathsf{Locals}\rangle \therefore \mathbf{Context}_3 \ \vdash \ \mathit{expr} : \mathsf{t}_2 \therefore \mathbf{Type}\end{array}}{\langle \mathsf{Names}, \mathsf{TH}, \mathsf{Intfs}, \mathsf{A}, \mathsf{Locals}\rangle \therefore \mathbf{Context}_3 \ \vdash \ \mathit{des}\ :=\ \mathit{expr} : \mathsf{correct} \therefore \mathbf{GenInfo}}$$
if $\sqsubseteq (t_2, t_1) \in \mathsf{TH}$

**Production (23):**

$$\frac{\begin{array}{c}\Gamma \therefore \mathbf{Context}_3 \ \vdash \ \mathit{expr} : \mathsf{bool} \therefore \mathbf{Type} \\ \Gamma \therefore \mathbf{Context}_3 \ \vdash \ \mathit{stats} : \mathsf{correct} \therefore \mathbf{GenInfo}\end{array}}{\Gamma \therefore \mathbf{Context}_3 \ \vdash \ \texttt{while}\ \mathit{expr}\ \texttt{do}\ \mathit{stats}\ \texttt{od} : \mathsf{correct} \therefore \mathbf{GenInfo}} \quad \text{if true}$$

**Production (24):**

$$\frac{\langle \mathsf{Names}, \mathsf{TH}, \mathsf{Intfs}, \mathsf{A}, \mathsf{Locals}\rangle \therefore \mathbf{Context}_3 \ \vdash \ \mathit{des} : \mathsf{t}_1 \therefore \mathbf{Type}}{\langle \mathsf{Names}, \mathsf{TH}, \mathsf{Intfs}, \mathsf{A}, \mathsf{Locals}\rangle \therefore \mathbf{Context}_3 \ \vdash \ \mathit{des}\ .\ \mathit{id} : \mathsf{t}_2 \therefore \mathbf{Type}}$$
if $\langle \mathsf{attribute}, \mathsf{t}_1, \langle \mathsf{id}, \mathsf{t}_2\rangle\rangle \in \mathsf{Intfs} \wedge \neg\exists\ t' : \langle \mathsf{method}, \mathsf{t}_1, \langle \mathsf{id}, [], \mathsf{t'}\rangle\rangle \in \mathsf{Intfs}$
$\vee\neg\langle \mathsf{attribute}, \mathsf{t}_1, \langle \mathsf{id}, \mathsf{t}_2\rangle\rangle \in \mathsf{Intfs} \wedge \langle \mathsf{method}, \mathsf{t}_1, \langle \mathsf{id}, [], \mathsf{t}_2\rangle\rangle \in \mathsf{Intfs}$

**Production (25):**

$$\frac{}{\langle \mathsf{Names}, \mathsf{TH}, \mathsf{Intfs}, \mathsf{A}, \mathsf{Locals}\rangle \therefore \mathbf{Context}_3 \ \vdash \ id : \mathsf{t} \therefore \mathbf{Type}}$$

if $\exists\, x : \langle x, \mathsf{id}, \mathsf{t}\rangle \in \mathsf{Locals} \vee$
$\forall\, x, t' : \neg\langle x, \mathsf{id}, t'\rangle \in \mathsf{Locals} \wedge \langle \mathsf{attribute}, \mathsf{A}, \langle \mathsf{id}, \mathsf{t}\rangle\rangle \in \mathsf{Intfs} \vee$
$\forall\, x, t' : \neg\langle x, \mathsf{id}, t'\rangle \in \mathsf{Locals} \wedge \forall\, t' : \neg\langle \mathsf{attribute}, \mathsf{A}, \langle \mathsf{id}, t'\rangle\rangle \in \mathsf{Intfs} \wedge$
$\langle \mathsf{method}, \mathsf{A}, \langle \mathsf{id}, [], \mathsf{t}\rangle\rangle \in \mathsf{Intfs}$

**Production (26):**

$$\frac{}{\langle \mathsf{Names}, \mathsf{TH}, \mathsf{Intfs}, \mathsf{A}, \mathsf{Locals}\rangle \therefore \mathbf{Context}_3 \ \vdash \ \texttt{result} : \mathsf{t} \therefore \mathbf{Type}}$$

if $\langle \mathsf{ret}, \mathsf{result}, \mathsf{t}\rangle \in \mathsf{Locals}$

**Production (27):**

$$\frac{\langle \mathsf{Names}, \mathsf{TH}, \mathsf{Intfs}, \mathsf{A}, \mathsf{Locals}\rangle \therefore \mathbf{Context}_3 \ \vdash \ des : \mathsf{t}_1 \therefore \mathbf{Type} \quad \langle \mathsf{Names}, \mathsf{TH}, \mathsf{Intfs}, \mathsf{A}, \mathsf{Locals}\rangle \therefore \mathbf{Context}_3 \ \vdash \ args : \mathsf{types} \therefore \mathbf{Type\_List}}{\langle \mathsf{Names}, \mathsf{TH}, \mathsf{Intfs}, \mathsf{A}, \mathsf{Locals}\rangle \therefore \mathbf{Context}_3 \ \vdash \ des \ . \ id \ (\ args\ ) : \mathsf{t}_2 \therefore \mathbf{Type}}$$

if $\langle \mathsf{method}, \mathsf{t}_1, \langle \mathsf{id}, \mathsf{par\_types}, \mathsf{t}_2\rangle\rangle \in \mathsf{Intfs} \wedge$
$\sqsubseteq_{[]}\ (\mathsf{types}, \mathsf{par\_types}) \in (\mathsf{lists\_coercions} \therefore \mathbf{List\_Coercibility})$

**Production (28):**

$$\frac{}{\langle \mathsf{Names}, \mathsf{TH}, \mathsf{Intfs}, \mathsf{A}, \mathsf{Locals}\rangle \therefore \mathbf{Context}_3 \ \vdash \ \texttt{this} : \mathsf{A} \therefore \mathbf{Type}} \quad \text{if true}$$

**Production (29):**

$$\frac{\Gamma \therefore \mathbf{Context}_3 \ \vdash \ expr : \mathsf{t} \therefore \mathbf{Type}}{\Gamma \therefore \mathbf{Context}_3 \ \vdash \ args : [\mathsf{t}] \therefore \mathbf{Type\_List}} \quad \text{if true}$$

**Production (30):**

$$\frac{\Gamma \therefore \mathbf{Context}_3 \ \vdash \ expr : \mathsf{t} \therefore \mathbf{Type} \quad \Gamma \therefore \mathbf{Context}_3 \ \vdash \ args : \mathsf{types} \therefore \mathbf{Type\_List}}{\Gamma \therefore \mathbf{Context}_3 \ \vdash \ expr \ , \ args : [\mathsf{t} \mid \mathsf{types}] \therefore \mathbf{Type\_List}} \quad \text{if true}$$

**Production (31):**

$$\frac{\Gamma \therefore \mathbf{Context}_3 \ \vdash \ des : \mathsf{t} \therefore \mathbf{Type}}{\Gamma \therefore \mathbf{Context}_3 \ \vdash \ expr : \mathsf{t} \therefore \mathbf{Type}} \quad \text{if true}$$

**Production (32):**

$$\frac{}{\Gamma \therefore \mathbf{Context}_3 \ \vdash \ int\_literal : \mathsf{int} \therefore \mathbf{Type}} \quad \text{if true}$$

**Production (33):**

$$\frac{}{\Gamma \therefore \mathbf{Context}_3 \ \vdash \ \mathit{real\_literal} : \mathsf{real} \therefore \mathbf{Type}} \quad \text{if true}$$

**Production (34):**

$$\frac{}{\Gamma \therefore \mathbf{Context}_3 \ \vdash \ \mathit{bool\_literal} : \mathsf{bool} \therefore \mathbf{Type}} \quad \text{if true}$$

**Production (35):**

$$\frac{\begin{array}{l}\Gamma \therefore \mathbf{Context}_3 \ \vdash \ \mathit{expr}_1 : \mathsf{t}_1 \therefore \mathbf{Type} \\ \Gamma \therefore \mathbf{Context}_3 \ \vdash \ \mathit{expr}_2 : \mathsf{t}_2 \therefore \mathbf{Type}\end{array}}{\Gamma \therefore \mathbf{Context}_3 \ \vdash \ \mathit{expr}_1 \texttt{ + } \mathit{expr}_2 : \mathsf{t}_3 \therefore \mathbf{Type}}$$

$$\begin{array}{ll}\text{if} & \mathsf{t}_1 = \mathsf{int} \wedge \mathsf{t}_2 = \mathsf{int} \wedge \mathsf{t}_3 = \mathsf{int} \vee \\ & \mathsf{t}_1 = \mathsf{int} \wedge \mathsf{t}_2 = \mathsf{real} \wedge \mathsf{t}_3 = \mathsf{real} \vee \\ & \mathsf{t}_1 = \mathsf{real} \wedge \mathsf{t}_2 = \mathsf{int} \wedge \mathsf{t}_3 = \mathsf{real} \vee \\ & \mathsf{t}_1 = \mathsf{real} \wedge \mathsf{t}_2 = \mathsf{real} \wedge \mathsf{t}_3 = \mathsf{real}\end{array}$$

**Production (36):**

$$\frac{}{\langle \mathsf{Names}, \mathsf{TH}, \mathsf{Intfs}, \mathsf{A}, \mathsf{Locals} \rangle \therefore \mathbf{Context}_3 \ \vdash \ \texttt{new} \ \mathit{id} : \mathsf{id} \therefore \mathbf{Type}}$$

if $\mathsf{id} \in \mathsf{Names}$

**Production (37):**

$$\frac{}{\Gamma \therefore \mathbf{Context}_3 \ \vdash \ \texttt{null} : \mathsf{void} \therefore \mathbf{Type}} \quad \text{if true}$$