View metadata, citation and similar papers at core.ac.uk

brought to you by CORE

provided by KITopen

Generating Semantic Analysis Using Constraint Programming

Sabine Glesner, Andreas Heberle, and Welf Löwe

Institut für Programmstrukturen und Datenorganisation, Universität Karlsruhe, 76128 Karlsruhe, Germany, E-mail:{glesner|heberle|loewe}@ipd.info.uni-karlsruhe.de

Abstract. We describe a new approach for the specification and generation of the semantic analysis for typed programming languages. We specify context-sensitive syntactic properties of a language by a system of semantic rules. For various imperative programming language concepts, we discuss the required semantic rules and show how they can be solved efficiently, i.e., in time $\mathcal{O}(n)$ where n is the program size.

1 Introduction

Semantic analysis should check a program if it matches the conditions imposed by the context-sensitive syntactical characteristics of a language. Additionally, it computes the typing of the program which is required for further transformations, i.e., the static semantics. Writing a semantic analyzer from scratch is too expensive and error prone.

Generators have been known for years but the required specifications depends too much on the process of analysis. On one hand, the language specification should not depend on the analysis. But on the other hand such a specification cannot serve as a generator's input. This implies that, in addition to the language specification given by its designer, a second (formal) specification of the same context-sensitive syntax is needed as generator input, committing the compiler constructor to do the specification job again. Additionally, the correctness of the generated analysis must be established which remains as a proof obligation for the compiler constructor. We propose another approach that splits the specification into two parts. Name and scope rules are defined operationally by a very simple left-to-right depth-first traversal of the abstract syntax tree (AST). This is the natural way as it is usually done in programming language specifications. Furthermore, we specify semantic constraints on AST nodes in a descriptive way. The language designer does not need to specify how these constraints are solved. Especially, the computation of the solution is completely independent from the AST traversal. Therefore, our specification method does not depend on the process of analysis.

For constructing semantic analyzers the following steps are performed:

(1) The language designer defines the context-sensitive syntax by the means of semantic conditions on abstract syntax trees. Such a definition does not contain any information on how to solve the specified semantic conditions.

(2) The designer's specification serves as input for the generator of the semantic analysis. The generated analyzer extracts a system of semantic constraints.
(3) An efficient algorithm (linear in the program size) solves the extracted constraint system and computes the typing.

Since there is only one specification involved, correctness would result automatically if the generation technique and the implementation of the generator itself were correct. The first requirement is guaranteed to be fulfilled due to this paper. We consider imperative, typed programming languages with overloading and coercions, and higher-order functions.

Research on the specification of context-sensitive syntactical properties and the generation of the associated semantical analysis was enforced 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 [5]. 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 [6]. However, there is no constructive method to design such grammars. Hence, designing an ordered attribute grammar remains a difficult problem. For another class of attribute grammars it is required that all attributes can be evaluated during a single depth-first, left-to-right traversal of the abstract syntax tree. These are the left-ordered attribute grammars, [7], [1]. Due to their fixed traversal order, the specification of context-sensitive syntax becomes very operational, i.e., dependent on the analysis, and is not as easily possible as a language designer might want it to be. However, because there are no alternative specification and generation methodologies, most practical tools are based on attribute grammars.

In [12], a framework for the specification of context-sensitive syntax is given which is based on the predicate calculus and on the entity-relationship model from database theory. The specifications in this model are very complex and are not intuitive. Moreover, the generation of semantic analysis from such a specification is not always possible, as stated by the author. Therefore this approach is not widely used.

A language for the specification of context-sensitive syntax which is based solely on the predicate calculus is defined in [8]. Due to the complexity of first-order formulas, the specifications in this model may not be easy. The semantic analysis can be generated but is much too inefficient for the use in practical compilers. Another framework also based on the predicate calculus is given in [10], incorporating basically the same disadvantages.

In [9], a specification method for context-sensitive syntax in object-oriented languages based on constraints is given. In this framework the specification of context-sensitive syntax is easy to express. The semantic analysis can be generated. Their emphasis lies on the treatment of programming languages that do not require that variables are declared. So in general, type inference is performed, using an algorithm of time complexity $\mathcal{O}(n^3)$ where n is the program size.

In this paper, we restrict ourselves to type checking while allowing a richer constraint language. This gives us the possibility to describe more realistic programming languages while obtaining an $\mathcal{O}(n)$ algorithm solving the constraints where n is the program size.

We proceed in the following way: section 2 sketches the specification language. Thereby, we show how our approach works for common concepts of existing programming languages. Section 3 describes the algorithm for solving the specified semantic conditions and demonstrates it on an example. Finally, section 4 concludes the work and describes its general context.

2 The Specification

We describe our principal approach in specifying context-sensitive syntax. Specifications are given by constraints associated with each node of the abstract syntax tree (AST). Furthermore, we discuss simple imperative features and proceed by successively introducing more complex properties of the languages that we consider. For each typical language construct we show how alternative semantics can be specified.

2.1 Principal Formalism

In general, the syntax of a programming language consists of context-free as well as context-sensitive syntactic properties. Therefore, the syntax analysis of a compiler is divided into two parts. The first checks the context-independent syntactical properties and is commonly called *syntactical analysis*. Its result is the abstract syntax tree. The second part of the analysis checks the context-dependent properties and is typically called *semantic analysis*. Here we assume that a program is represented by the AST. This means that the analysis of context-free properties has already been performed. We describe context-sensitive syntactic properties inductively on the structure of programs. For each production of the language's context-free grammar we define semantic rules. These rules specify syntactical correctness of programs w.r.t. the context-sensitive syntax of the programming language.

When a program is checked, we look at it in left-to-right depth-first order. Inductively on this traversal order, we define what context-sensitive correctness means. For each node in the AST, we define a context. This context completely summarizes the context-sensitive properties belonging to the program part before (w.r.t. the left-to-right depth-first order) the actual node.

Each inner node of the AST corresponds to the left-hand side of a rule of the context-free grammar. The context-sensitive properties of such a node are described via semantic rules associated with the context-free productions. Semantic rules consist of conditions, actions, and constraints:

(1) The *condition* indicates if the particular semantic rule applies to the node in a certain context.

(2) If the semantic rule applies, the *action* defines the new context.
(3) If the semantic rule applies, the *constraints* describe the context-sensitive properties of the node.

Figure 1 shows the specification scheme for a semantic rule which is used in this paper. True-conditions and skip-actions are omitted.

Condition	Actions	Constraints
Predicate on the context	Modification of the context	Selected Constraints

Fig. 1. Specification scheme: semantic rules.

In fact, the context is represented by a definition table, the method of choice in compiler construction. The definition table defines a function *typeof* returning the *type* of a given *name*. Initially, it gives $\perp$ for each *name* which indicates that it has not been defined yet. To be able to collect all constraints arising from the use and/or declaration of the same identifier, we conceptually insert new nodes into the AST. Whenever a *name* occurs for the first time, we create such a *node* and insert it such that it is the successor of *name*, i.e., *name.succ* := *node* is then a default action. Another function *nodeof* of the definition table indicates this *node* of a *name*, depending on the context.

2.2 Types and Equivalence of Types

In principal, we can deal with all primitive types that are known from common programming languages as arithmetic, boolean, character, and string types. In this paper, w.l.o.g., we consider the basic types *int*, *real*, and *bool*, see (1). From these types we can construct complex types by applying type constructors. *type* $\rightarrow$ *type* defines function types, see (2). We consider only functions with one argument. It is obvious that this does not pose any restriction on the generality of the type system as argued by Schönfinkel [11] and later used by Curry [2]. Records can be built by joining tuples of names and types, see (3)-(6).

$$\begin{aligned} type &::= INT \mid REAL \mid BOOL && (1)\\ &\mid \; type \rightarrow type && (2)\\ &\mid \; \{\; components \;\} && (3)\\ components &::= \;; && (4)\\ &\mid \; comp \;;\; components && (5)\\ comp &::= name : type && (6) \end{aligned}$$

For structured types, different notions of type equivalence are common in programming languages. The basic distinction is between declarational and structural equivalence. For the latter, each of the four combinations (order matters, does not matter) and (names make a difference, make no difference) is possible. Nevertheless, the combination (order does not matter, names make no difference) does not seem to make sense and is therefore not considered in the following. In

No	Condition	Actions	Constraints
(1)			$[\![type]\!] = (int \vert real \vert bool)$
(2)			$[\![type0]\!] = [[\![type1]\!] \rightarrow [\![type2]\!]]$
(3)			$[\![type]\!] = [\![components]\!]$
(4)			$[\![components]\!] = \emptyset$
(4')			$[\![components]\!] = []$
(5)			$[\![components0]\!] = [\![comp]\!] \cup [\![components1]\!]$
(5')			$[\![components0]\!] = [[\![comp]\!] \vert [\![components1]\!]]$
(6)			$[\![comp]\!] = (name, [\![type]\!])$

Fig. 2. Semantic rules for type definitions.

figure 2, we show how these different notions of type equivalence can be described via semantic rules. For the syntactical rules (4) and (5), we give two alternative semantic rules describing different type equivalences. In the case that the order on the record elements matters, we represent them by a list. If the order does not matter, we choose a set description. This is described by (4), (5) and (4'), (5'), resp.[1] Furthermore, the names of the elements can make a difference between record elements. But since we need to have access to the names of the record elements whenever they are used in a program, we need to describe their names in the constraints; no matter if they are used to distinguish between different record types or not.

2.3 Imperative Features

We consider declarations, assignment and loop statements, and simple expressions. Our notation for these language constructs is assumed to be as follows:

$$decl ::= name : type \tag{7}$$
$$assign ::= des := expr \tag{8}$$
$$des ::= des \,.\, name \tag{9}$$
$$\quad \mid name \tag{10}$$
$$loop ::= \texttt{while}\ expr\ \texttt{do}\ stats\ \texttt{od} \tag{11}$$
$$expr ::= des \tag{12}$$
$$\quad \mid bool\_literal \tag{13}$$
$$\quad \mid int\_literal \tag{14}$$
$$\quad \mid real\_literal \tag{15}$$
$$\quad \mid expr\ \texttt{+}\ expr \tag{16}$$

[1] We assume [] to denote the empty list and [|] to denote concatenation of lists.

There are two different principal ways for the use of objects in programming languages. Either (i) it is required that an object is defined before it is used, or (ii) use and declaration can occur in arbitrary order. However, this distinction does not matter for the handling of declarations (7). In figure 3, we state the semantic rules for the declaration statement. We distinguish between the two possibilities that the *name* has already been declared or not. Depending on them, the corresponding actions and constraints differ. In the first case, the *name* is still undefined ($\mathit{typeof}(\mathit{name}) = \bot$). If this condition happens to be true, the resulting action defines the type entry $\mathit{typeof}(\mathit{name}) := [\![\mathit{type}]\!]$ for *name*. Furthermore, the constraint $[\![\mathit{name.succ}]\!] = [\![\mathit{type}]\!]$ describes that the type of *name* is constrained to $[\![\mathit{type}]\!]$. The node $\mathit{name.succ}$ is used to collect all constraints on this particular *name*. As described above, the node $\mathit{name.succ}$ is inserted (as a default action of the definition table) into the AST whenever *name* appears for the first time in the program. All constraints on the type of *name* are collected at this node. In the second case, *name* has already been declared before. Since we allow only one declaration per *name*, this results into an *error*: The type of *name* in the definition table is set to *error*. The corresponding constraint also restricts the type of *name* to the error type *error*. In the assignment statement (8), we require

Condition	Actions	Constraints
$\mathit{typeof}(\mathit{name}) = \bot$	$\mathit{typeof}(\mathit{name}) := [\![\mathit{type}]\!]$	$[\![\mathit{name.succ}]\!] = [\![\mathit{type}]\!]$
$\mathit{typeof}(\mathit{name}) \neq \bot$	$\mathit{typeof}(\mathit{name}) := \mathit{error}$	$[\![\mathit{name.succ}]\!] = \mathit{error}$

Fig. 3. Conditions, actions, and constraints for declarations (7).

that the type of the expression *expr* on the right-hand side is coercible to the type of the designator *des* on the left-hand side. No condition and no further action are necessary. The constraint is $[\![\mathit{expr}]\!] \leadsto [\![\mathit{des}]\!]$.

Coercibility relations, denoted by $\leadsto$, are language dependent and can be combined into a semi-lattice by introducing the error type *error* as top element. There may be different coercibility relations for different language constructs in a single programming language. They are defined by the language designer.[2]

The semantic rules for the designator are specified in figure 4. As already mentioned we distinguish if a variable has to be declared before its use (i) or not (ii). For example, in the semantic rule for no. (9), case (i), the condition asks if the type of *name* has already been declared. The semantic rule applies if this is the case. Then no action is performed which means that the state of the definition table is not changed. The constraints specify that the type of *des1* is a record type which contains an element named *name* of the same type as *des0*.

The semantic rule for the loop statement (11) is simple. We only have to require that the type of the conditioning expression is boolean, i.e. $[\![\mathit{expr}]\!] = \mathit{bool}$. In particular, there are no actions and conditions. For expressions (12)–(15), the

[2] For our example language, we assume $\mathit{int} \leadsto \mathit{real}$.

No.	Case	Condition	Actions	Constraints
(9)	(i)	$typeof(name) \neq \bot$		$("name", [\![des0]\!]) \in [\![des1]\!]$
(9)	(i)	$typeof(name) = \bot$	$typeof(name) := error$	$[\![name.succ]\!] = error \wedge$ $("name", [\![des0]\!]) \in [\![des1]\!]$
(9)	(ii)			$("name", [\![des0]\!]) \in [\![des1]\!]$
(10)	(i)	$typeof(name) \neq \bot$		$[\![des]\!] = [\![name.succ]\!]$
(10)	(i)	$typeof(name) = \bot$	$typeof(name) := error$	$[\![name.succ]\!] = error \wedge$ $[\![des]\!] = [\![name.succ]\!]$
(10)	(ii)			$[\![des]\!] = [\![name.succ]\!]$

Fig. 4. Semantic rules for designators (8).

No.	Condition	Actions	Constraints
(12)			$[\![expr]\!] = [\![des]\!]$
(13)			$[\![expr]\!] = bool$
(14)			$[\![expr]\!] = int$
(15)			$[\![expr]\!] = real$
(16)			$[\![expr0]\!] = \max_{\rightsquigarrow}([\![expr1]\!], [\![expr2]\!]) \wedge$ $[\![+]\!] = [\![[\![expr0]\!] \rightarrow [\![expr0]\!] \rightarrow [\![expr0]\!]]\!] \wedge$ $[\![expr1]\!] \rightsquigarrow [\![expr0]\!] \wedge [\![expr2]\!] \rightsquigarrow [\![expr0]\!]$

Fig. 5. Conditions, actions, and constraints for expressions (12)–(16).

constraints are obvious, cf. figure 5. Expression (16) is interesting since it may combine overloading with coercion. To demonstrate the power of our method, we assume "+" to be defined either as the boolean *or*-operator or as the common integer and real addition operator, resp. The operator is identified according to the types of its operands. The semantic rules are defined in figure 5. The first of the constraints' literals defines that the entire *expr* has as type the maximum of the operands' types in the semi-lattice $\rightsquigarrow$. Note that it is the error type if the operands are not coercible, e.g., if they are *bool* and *real* in our language[3]. The second constraint literal defines the function type for "+" dependent on the type of the entire expression. The last two constraint literals finally describe the coercibility of the operands to the types required by the operation.

2.4 Names and Scopes

Up to now, we did not talk about programming languages incorporating name spaces. In particular, when talking about name spaces as contexts we did not change between different name spaces. To be able to do so, we extend the language constructs discussed so far and allow for the declaration of methods which can be called by using their name. As a natural consequence, we get blocks defining name spaces.

[3] Here we assume that structured types are coercible only if they are equal.

$$
\begin{aligned}
\mathit{decl} &::= \texttt{function}\ \mathit{name}\ \texttt{(}\ \mathit{name}\ \texttt{:}\ \mathit{type}\ \texttt{)}\ \texttt{:}\ \mathit{type}\ \texttt{;}\ \mathit{block} && (17)\\
\mathit{name} &::= \texttt{result} && (18)\\
\mathit{expr} &::= \mathit{des}\ \texttt{(}\ \mathit{des}\ \texttt{)} && (19)\\
\mathit{block} &::= \texttt{begin}\ \mathit{stats}\ \texttt{end} && (20)\\
\mathit{stats} &::= \texttt{;} && (21)\\
&\ \ \mid\ \ (\mathit{stat} \mid \mathit{decl}\ )\ \texttt{;}\ \mathit{stats} && (22)
\end{aligned}
$$

The introduction of *blocks* requires an extended functionality of the definition table. We need to be able to create new scopes as a new block is entered and to discard them on the exit of the corresponding block. These actions are assumed to be performed by the functions *enter_scope* and *leave_scope*. The function *enter_scope* opens a new name space. In particular, this means that after the execution the function *typeof(name)* yields undefined for every identifier, until the declaration in the current block is processed. The first occurrence of an identifier *name* initiates the creation of *name.succ*. The function *leave_scope* requires a more detailed discussion.

As already explained in subsection 2.3, there are two principal ways for the use of objects in programming languages: (i) either they need to be declared before they are used, or (ii) their use and declaration can appear in arbitrary order. This distinction requires in turn that the definition table behaves differently in both cases. If we do not require that an object is declared before used (case (ii)), we do not know until the block end is reached if the *name* denotes a local object of the block or some other (global) object declared outside of the current block. I.e., before reaching the end of the block, we do not know if we eventually find a declaration for the object in the current block or if a global declaration belongs to this object. Therefore we collect all constraints for a *name* in *name.succ*. If we do not find a declaration for a name in the current block, we propagate the constraints to the enclosing scope. This is performed by the function *leave_scope*. The mechanism is as follows:

- If *name.succ* exists in the enclosing scope, we just add the local constraint set to this node. The function *typeof* remains unchanged.
- If *name.succ* does not already exist, the local identifier *name*, together with the corresponding constraints, becomes valid in the enclosing scope . In this case, executing *leave_scope* manifests *name* as an identifier of the enclosing scope.

Thereby the constraints for a yet undeclared identifier *name* can be propagated to enclosing scopes until the outermost scope is reached. In case (i), where we require that an object is declared before used, such a complex distinction is not necessary. As soon as a *name* occurs, its declaration is clear. If it does not exist, an error occurs.

Figure 6 describes semantic rules for function declarations. For simplicity, we specify only the semantic rule for case (i) with the condition that the function

Condition	Actions	Constraints
$typeof(name1) = \bot$	$typeof(name1) :=$ $[\![[\![type1]\!] \to [\![type2]\!]]\!]$; $enter\_scope$; $typeof(name2) := [\![type1]\!]$; $typeof(result) := [\![type2]\!]$	$name1 \neq name2 \wedge$ $[\![name1.succ]\!] = [\![[\![type1]\!] \to [\![type2]\!]]\!] \wedge$ $[\![name2.succ]\!] = [\![type1]\!] \wedge$ $[\![result]\!] = [\![type2]\!]$

Fig. 6. Semantic rule for function declarations (17).

name has not been declared before. This is expressed in the condition predicate *typeof*(*name1*) $= \bot$. There is no entry for *name1* in the definition table. If this condition is true, *name1* is inserted to the definition table as an object of function type. After this, a new scope of the definition table is entered. In this new scope, the definitions for the function parameter *name2* and for `result` are inserted into the definition table. We assume `result` being a predefined name denoting the result of a function. The constraints of this rule state that the name of the function and its parameter do not have to be the same. Furthermore, *name1* is specified as a function mapping arguments of *type1* to *type2*. Finally, *name2* and `result` are declared of *type1* and *type2*, resp.

Figure 7 defines the semantic rules for function calls. We describe *des1* as a function mapping objects of the type of *des2* to objects of the type of *expr*. Here, no conditions and actions are defined since the AST nodes involved in this rule do not have entries in the definition table. At the end of a block, we have to

Condition	Actions	Constraints
		$[\![des1]\!] = [\![[\![des2]\!] \to [\![expr]\!]]\!]$

Fig. 7. Constraints for function calls (19).

execute *leave_scope*. Is is associated as an action with rule (21).

3 The Analysis

Semantic conditions are associated with nodes in the abstract syntax tree, cf. section 2. It remains to show how the constraint set is organized, simplified, and checked for consistency in an efficient way.

3.1 The Algorithm

Constraints are predicates on the types $[\![n]\!]$ of nodes $n \in V_{AST}$ of the AST and the types of the programming language. E.g., the predicate $[\![n]\!] = t$ denotes that

n is of type t. We consider the following constraints:

$$t_1 = t_2 \tag{23}$$

$$t_1 \rightsquigarrow t_2 \tag{24}$$

where t_1 and t_2 are types and "=" is an equivalence and $\rightsquigarrow$ is a coercibility relation. The language designer must define both for all possible types of the language. In addition to the discussed type constructors (2) and (3), we also consider the following constructor:

$$\max_{\rightsquigarrow}(t_1, \cdots, t_k) \tag{25}$$

which denotes the maximum of types $t_1, \ldots, t_n$ in the coercibility-semi-lattice. Hence $\max_{\rightsquigarrow}$ is derived from $\rightsquigarrow$. A predicate $[\![n]\!] = t$ is called *defining* iff t is a language type.

Predicates are kept in a graph structure $C = (V, E)$ which we call the *constraint graph*[4]. The vertices V in this graph are language types and types of nodes. Edges E represent the constraints where "="-edges are undirected and "$\rightsquigarrow$"-edges are directed. Initially $C = (\emptyset, \emptyset)$. For each node n with a constraint, a vertex $[\![n]\!]$ is added to V, edges to other vertices are inserted according to the constraints. Figure 8 shows C for a defining predicate $[\![n]\!] = t$. Whenever a

Fig. 8. C for the defining predicate $[\![n]\!] = t$.

defining predicate is inserted, C is simplified as much as possible, i.e., vertices and/or edges may be removed from C. This follows the four rewriting rules. Rule (I) simply propagates definitions. Thereby, new nodes may get defining predicates such that (I) is applicable again, cf. figure 9. Equivalence of language types may be checked. If C contains an "="-edge between vertices representing language types, it may be removed. Rule (II) describes the rewriting. If both types are equivalent (a), they are melted. If they are not (b), the subgraph is replaced by a vertex which represents the *error* type, cf. figure 10. Rule (III) does the same for the $\rightsquigarrow$ constraints, cf. figure 11. Rule (IV) replaces the *max* type constructor by the result of the *max*-operator if all operands are language types, cf. figure 12.

The following theorem 1 assumes that the language specification is correct and consistent. The notion of "correctness of a program w.r.t. the specification of the context-sensitive syntax of a language" includes the following features.

- All names are declared.
- All operands are identified.

[4] Let AST$= (V_{AST}, E_{AST}), C = (V, E)$. To avoid confusion, we call the elements of the V_{AST} "nodes" and the elements of V "vertices".

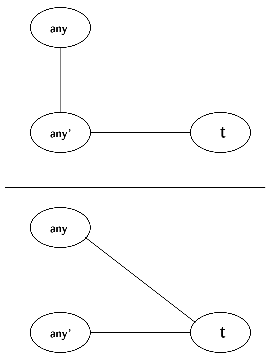


Fig. 9. Rule (I): propagation of definitions.

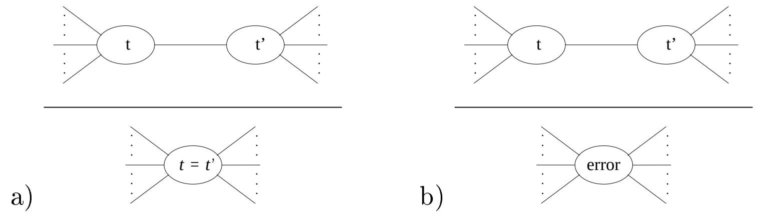


Fig. 10. Rule (II): solving equality constraints.

– The declaration of names does not contradict its application.
– All names of the same scope are unique (no overloading).

Depending on the language, it may additionally include some of the following.

– All names are declared before use.
– A name is not declared more than once in the same scope (no overwriting of declarations).

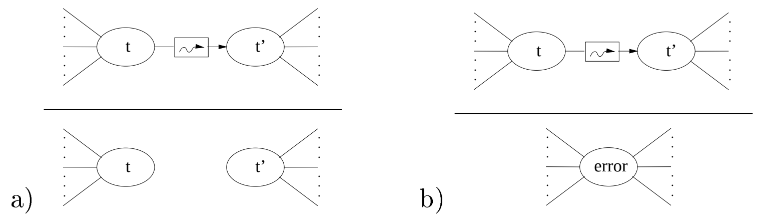


Fig. 11. Rule (III): solving coercibility constraints.

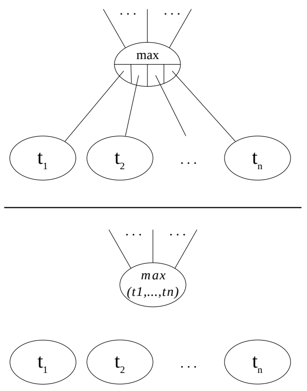


Fig. 12. RULE (IV): elimination of maximum nodes.

Theorem 1. Correctness: *A program is correct w.r.t. the specification of context-sensitive syntax of a language, iff*

(i) *all constraints, except for the defining predicates, are removed,*

(ii) *all nodes n have at most one defining predicate $[\![n]\!] = t, t \neq error$, and*

(iii) *all successor nodes of names name.succ have exactly one defining predicate $[\![name.succ]\!] = t, t \neq error$.*

Proof. First, we prove that if a program is correct w.r.t. the context-sensitive syntax then (i) – (iii) must hold. Obviously (iii) must hold for correct programs since we considered typed languages. (i) and (ii) are shown by contradiction. If (i) was false, there were constraints that cannot be resolved. This may either occur if they still depend on the types of some nodes without defining predicate or if they are equal to the error type. The former must not occur if (iii) holds because then all nodes get defining predicates by applying RULE(I) successively. If the latter occurred, the program would be obviously not correct. If (ii) did not hold, some nodes of the AST would have distinct defining types which contradicts the assumption that the program is correct.

Second, we prove that a program is correct w.r.t. the context-sensitive syntax if (i) – (iii) hold. The organization of our definition table guarantees that a name is defined and

- is not multiply defined, or
- is not multiply defined with the same type, or
- used before its definition

if this is not allowed for the considered programming language. Additionally, condition (i) guarantees that operands are identified and (ii) guarantees that they are unique. Condition (iii) guarantees that the uses of each name do not contradict each other and are not in contradiction to the definition.

Theorem 2. Complexity: *Let $AST = (V_{AST}, E_{AST})$ be the abstract syntax tree of a program. The defined algorithm performs in time*

$$\mathcal{O}(|V_{AST}|).$$

Proof. The maximum number of edges in the constraint graph is $|V_{AST}| \cdot c \cdot k$ where c is the number of constraints per node and k is there arity. The algorithm infers about constraints in the constraint graph by applying the reduction rules. Rule (I) moves an edge to its final state. Rule Rule (II)–((IV) removes edges. Each rule application processes at least one edge. We can only define a constant number of constraints for each node which must have a constant arity, This gives us the result stated in the above theorem.

3.2 An Example

We demonstrate the algorithm on a small example program which is assumed to be correct. Thus, the programming language allows that use and declaration of variables may occur in arbitrary order. Furthermore, the language requires that the right-hand side of an assignment is coercible to the left-hand side. Integer values are coercible to real values. Figure 14 shows several snapshots of the

```
a := a + 3 ;
a : REAL
```

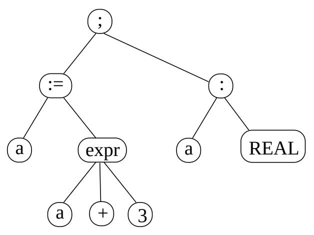
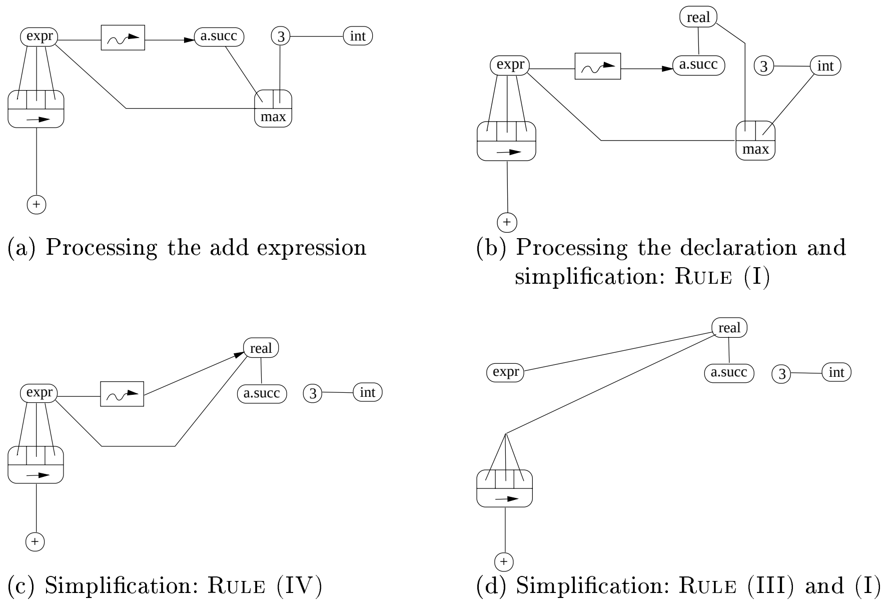


Fig. 13. An example program p and its AST representation.

constraint graph during the analysis of the program. To get a clear presentation, the pictures contain several type nodes for the same basic types. In fact, we have only one type node for each basic type. In the beginning (a), we insert the constraints for the assignment to the empty graph. In the next step (b), we process the declaration of a and propagate defining predicates . Now a has a defining predicate which can be propagated to the *max*-vertex. Then all of the arguments of the *max*-node are defined and we can apply Rule (IV) (c). Finally, we eliminate the coercible constraint (Rule III)) and propagate the definition of *expr* (Rule (I)). This leads to the consistent constraint graph for the example program (d).

4 Conclusion

We have introduced a new approach for the specification and generation of the semantic analysis in typed imperative programming languages. Our specification



(a) Processing the add expression

(b) Processing the declaration and simplification: Rule (I)

(c) Simplification: Rule (IV)

(d) Simplification: Rule (III) and (I)

Fig. 14. Snapshots of the constraint graph during the analysis of program p.

serves not only as a definition for the context-sensitive syntax of programming languages but also as an input of a generator for the semantic analysis. This is much simpler than existing techniques since we have only one specification for both the description of the language and the generator input. Double specification efforts and resulting proof obligations disappear. We demonstrated this method by defining the context-sensitive syntax for typical imperative language constructs. In particular, we showed how specifications for these constructs may vary depending on the features of the specific language. If, for example, the language allows the use of objects before their declaration is given, we can describe this easily. We are also able to express overloading of operators. This demonstrates the flexibility of our specification method. Moreover, our specifications are easy to formulate and understand, thereby appearing naturally. Our description and analysis of the context-sensitive syntax is based on abstract syntax trees. During the analysis of a program, its abstract syntax tree is traversed. We define an abstract data type “definition table” containing the names and definitions of the program objects. Specifications are given by constraints formulated according to the syntax rules of the underlying context-free grammar. The constraints collected during the traversal are managed in a data structure called “constraint graph” which allows for solving them efficiently, namely in time $\mathcal{O}(n)$ where n is the size of the program.

Current work deals with extensions of our approach to languages that allow for (1) overloading of user-defined functions, (2) subtyping and polymorphism under closed-world assumption, and (3) genericity under the assumption of separate compilation. Extension (1) requires another abstract data type "name table" since the definition table may not be constructed online but derived from the constraints. The additional constraints are handled by further rewriting rules within the same setting. This leads to a different strategy for the solution of constraints, i.e., constraints can not be solved greedy anymore, cf. [3]. (2) and (3) seem to be straight-foreward extensions of (1) since subtyping may be understood as dealing with yet another semi-lattice. Because we are already able to handle several semi-lattices for coercion, this should be possible.

The traditional compiler construction process is divided into two parts: the construction of a source language dependent frontend and the construction of a target machine dependent backend. The interface is an intermediate program representation. The work presented here is a milestone towards the more general goal to provide a framework for the generation of compiler frontends based on a formal specification of source and intermediate language semantics. In this paper we showed how the programming language specification can be given such that the corresponding analysis can be generated automatically and efficiently. A complete framework which deals with lexical, syntactic, and semantic analysis and intermediate code generation is described in [4].

References

1. G. V. Bochmann. Semantic Evaluation from Left to Right. *Communications of the ACM*, 19(2):55–62, 1976.
2. H.B. Curry and R. Feys. *Combinatory Logic*, volume 1. North-Holland, 1958.
3. A. Heberle, S. Glesner, and W. Löwe. Typing, Overloading, and Constraint Programming. In *Static Analysis Symposium, SAS '97*. Submitted, 1997.
4. A. Heberle and W. Löwe. Generierung von kompletten Compiler-Frontends. In *Arbeitstagung "Programmiersprachen", GI-Jahrestagung'97*. Submitted, 1997.
5. 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.
6. U. Kastens. Ordered Attribute Grammars. *Acta Informatica*, 13(3):229–256, 1980.
7. P. M. Lewis, D. J. Rosenkrantz, and R. E. Stearns. Attributed Translations. *Journal of Computer and System Sciences*, 9(3):279–307, 1974.
8. Martin Odersky. Defining context-dependent syntax without using contexts. *ACM Transactions on Programming Languages and Systems*, 15(3):535–562, July 1993.
9. Jens Palsberg and Michael I. Schwartzbach. *Object-Oriented Type Systems*. Wiley Professional Computing, 1994.
10. Arnd Poetzsch-Heffter. *Formale Spezifikation der kontextabhängigen Syntax von Programmiersprachen*. PhD thesis, Technische Universität München, 1991.
11. M. Schönfinkel. Über die Bausteine der mathematischen Logik. *Math. Ann.*, 92:305–316, 1924.
12. Jürgen Uhl. *Spezifikation von Programmiersprachen und Übersetzern*. PhD thesis, Universität Karlsruhe, 1986.
13. William M. Waite and Gerhard Goos. *Compiler Construction*. Springer Verlag, Berlin, New York Inc., 1984.