Names, Types, and Static Semantic Analysis

Andreas Heberle, Sabine Glesner and Welf Löwe

Institut für Programmstrukturen und Datenorganisation, Universität Karlsruhe,
76128 Karlsruhe, Germany, E-mail:{heberle|glesner|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. In particular, we show how overloading of programmer-defined identifiers can be handled.
We propose an algorithm to solve these constraint systems efficiently, i.e., in time $\mathcal{O}(n^2)$ 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 depend too much on the process of analysis. On the one hand, the language specification should not depend on the analysis. But on the other hand such an analysis-independent 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). In a name table, we keep and update all context-sensitive information arising from the declarations and use of identifiers when traversing the tree. This is the natural way as it is usually done in programming language specifications. Depending on the content of the name table, we define operationally how the name table is updated. Furthermore, we specify, also depending on the content of the name table, 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 their solution is postponed and therefore 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 of programmer-defined functions and identifiers and coercions, and higher-order functions. This work is an extension of [GHL97] where we introduced our method for simple imperative languages. In these languages we did not allow for overloading of programmer-defined functions. The extension for arbitrary overloading considered in this paper needs a more sophisticated algorithm in the sense that not all equal identifiers also denote the same object. Therefore the work presented here is a clear continuation and extension of [GHL97].

2 Related Work

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 [WG84]. 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 [Jaz81]. 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 [Kas80]. 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 constant number of depth-first, left-to-right traversals of the abstract syntax tree. These are the left-ordered attribute grammars (LAG), [LRS74], [Boc76]. 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 [Uhl86], 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 [Ode93]. 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 [PH91], incorporating basically the same disadvantages.

In [PS94], 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 functional languages, type inference and checking is performed by solving systems of type equations [Jon87]. During this computation it is necessary to unify terms denoting types. The unification method chosen is typically Robinson's [Rob65] which needs exponential time in the worst case. Since we do not require type variables, this approach is more general than necessary in our context.

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 $\mathcal{O}(n^2)$ algorithm solving the constraints where n is the program size. During the process of semantic analysis, we assign a type to each node of the abstract syntax tree, thereby giving a meaning to the nodes. This approach can be viewed as carrying out an *abstract interpretation* [CC79]. In our approach, a type is either a basic type or built from already existing types by the application of certain type constructors. We assign the standard type, i.e., the type of which they have been declared, to objects being explicitly declared as for example variables, formal parameters, or constants. Then we define and infer the types of other nodes in the abstract syntax tree inductively, starting at its leaves. This gives us an operational formulation of types and in turn an abstract interpretation based on operational semantics.

We proceed in the following way: section 3 sketches the specification language. Thereby, we show how our approach works for common concepts of existing programming languages. Section 4 describes the algorithm for solving the specified semantic conditions. Additionally, we establish the correctness of this algorithm and discuss details of an efficient implementation. Finally, section 5 concludes the work and describes its general context.

3 The Specification

There are two principal ways for describing the context-sensitive syntax in programming languages: either an operational or a descriptive approach may be chosen. Depending on this choice, the context information of nodes in the AST is treated differently. In an operational specification, context information is expressed and collected directly. Descriptive methods are more subtle and describe,

for each node in the AST, what has to be true for certain predecessors and successors of the node so that the entire program is conform to the context-sensitive syntax.

Attribute grammars are a good example for an operational specification of context-sensitive syntax, at least in the sense that local dependencies of attributes and thereby the associated computation have to be specified. Context-sensitive information for example is specified by environment attributes. For each node, an associated environment attribute is defined by specifying how it is computed from the environments of surrounding nodes.

Descriptive methods are based on the predicate calculus. The context is modeled by a first-order description, as everything else as well. All context-sensitive properties of an AST can be described with a first-order formula φ. To check the context-sensitive properties of a program, it is necessary to prove that the program is a model of this formula φ. Since the program is finite, this question is decidable: The formula φ is transformed into a propositional formula by eliminating all quantifiers. Each forall-quantified formula $\forall x.\varphi$ is (w.r.t. the program as model) equivalent to the conjunction $\varphi[x := u_1] \wedge \cdots \wedge \varphi[x := u_n]$ where $u_1, \ldots, u_n$ are all program nodes. Exists-quantifiers are eliminated analogically. After this transformation, we have a propositional formula whose validity can be tested. Since this test is as hard as deciding whether a formula is satisfiable (φ valid $\leftrightarrow$ $\neg\varphi$ not satisfiable), this problem is NP-complete [Coo71]. It is an open question if the problem of checking whether a program is conform to a specification of context-sensitive properties is also NP-complete. However, a partial solution to the above described inefficiency problem can be given by eliminating all negations in the formula to be checked. Each negated predicate $\neg P$ is replaced by $\overline{P}$. This preserves the correctness of the decision procedure (formulas recognized as valid are indeed valid) but its completeness does not exist any more. Valid formulas may not be recognized. Furthermore, for some inputs, the computation might not stop [Rob79, Ode93]. For any practical use, such a behavior does not suffice at all.

Our approach combines the advantages of both operational and descriptive methods. In a name table, we keep and update all context-sensitive information arising from the declarations and use of identifiers when traversing the AST in left-to-right depth-first order. Depending on the content of the name table, we define constraints for each node in the AST. These constraints are positive propositional formulas (i.e., contain no negations) and describe context-sensitive correctness denotationally. Their validity can be checked in efficient time. We developed a special data structure, a constraint graph, which is especially designed for constraint systems arising from context-sensitive program analysis.

In a certain sense, we use an LAG(1) grammar to specify and compute context information and to collect the set of constraints. Whenever a computation would be too complicated for an LAG(1) description, we define constraints which are only collected during the tree traversal while their solution is postponed. In particular, we take care that only positive constraints (without any negation) are defined.

In this section, we describe how specifications are given by semantic rules 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.

3.1 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 name table	Modification of the name table	Selected Constraint

Fig. 1. Specification scheme: semantic rules

The name table provides access to declaration nodes and to use nodes of "names"[1]. The function *Decls* associates a set of declaration nodes with the corresponding *"name"*. The function *Use* relates *"names"* with their different uses in the program. This means that it associates "names" with those nodes in the AST that represent one particular use of the "name". We assume *Use*("*name*") to return a set of nodes. In section 4, we discuss efficient implementations for the representation of sets of types.

3.2 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 [Sch24] and later used by Curry [CF58]. Records can be built by joining tuples of names and types, see (3)-(6).

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

To process record declarations, we require the name table to handle identifiers which are concatenated by the "."-operator, much in the same way as record names are built. We assume that the name table stores the actual record name in some variable. This variable consists of a list of identifiers, concatenated by the "."-operator. In addition, a pointer shows the dot which corresponds to the actual record context. When a record definition is entered, the action *enter_record* is performed. It moves the pointer one "name" to the right. On the exit of a record definition, the action *leave_record* is executed. It moves the pointer one "name" to the left. Type constructs appear only in the context of declarations. When a declaration of some variable occurs, we have to assume that it could be a record definition. Therefore we update the actual record name by "."-concatenating the name of the declared variable to it. This is performed by the function *update*("*name*"). This function removes all names at the right from the pointer of the actual record context. Then it appends the "name" to its right but does not move the pointer. Only if a record type declaration is entered, the pointer is moved, performed with the functions *enter_record* and *leave_record*.

[1] We use *"name"* to identify the key to the string representation of a name defined by the lexical analysis. We distinguish this from *name* which an identifies AST node.

For structured types, different notions of type equivalence are common in programming languages. The basic distinction is between declarational and structural equivalence. In 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.[2] 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. In principle, 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. Therefore we did not mention this case in the above enumeration of possible type equivalences.

No	Condition	Actions	Constraints
(1)			$[\![type]\!] = (int \mid real \mid bool)$
(2)			$[\![type0]\!] = [\![[\![type1]\!] \rightarrow [\![type2]\!]]\!]$
(3)		$enter\_record$	$[\![type]\!] = [\![components]\!]$
(4)		$leave\_record$	$[\![components]\!] = \emptyset$
(4')		$leave\_record$	$[\![components]\!] = []$
(5)			$[\![components0]\!] = [\![comp]\!] \cup [\![components1]\!]$
(5')			$[\![components0]\!] = [[\![comp]\!] \mid [\![components1]\!]]$
(6)		$Use(comp\_name) :=$ $Use(comp\_name) \cup name$	$[\![comp]\!] = (\text{"}name\text{"}, [\![type]\!]) \wedge$ $[\![name]\!] = [\![type]\!]$

Fig. 2. Semantic rules for type definitions

3.3 Imperative Features

We consider declarations, assignment and loop statements, and simple expressions where we especially discuss overloading. Our notation for these language

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

constructs is assumed to be as follows:

$$
\begin{aligned}
\mathit{decl} &::= \mathit{name} : \mathit{type} && (7)\\
\mathit{assign} &::= \mathit{des} \ := \ \mathit{expr} && (8)\\
\mathit{des} &::= \mathit{des} \ . \ \mathit{name} && (9)\\
&\mid \ \mathit{name} && (10)\\
\mathit{loop} &::= \texttt{while}\ \mathit{expr}\ \texttt{do}\ \mathit{stats}\ \texttt{od} && (11)\\
\mathit{expr} &::= \mathit{des} && (12)\\
&\mid \ \mathit{bool\_literal} && (13)\\
&\mid \ \mathit{int\_literal} && (14)\\
&\mid \ \mathit{real\_literal} && (15)\\
&\mid \ \mathit{expr}\ \texttt{+}\ \mathit{expr} && (16)
\end{aligned}
$$

There are two different principal ways for the use of objects in programming languages:

(i) Either it is required that an object is defined before it is used,
(ii) or use and declaration can occur in arbitrary order.

However, this distinction does not matter for the handling of declarations (7). Furthermore, we distinguish between

- languages that allow for overloading and those that do not, and
- languages that allow for re-declarations (multiple declarations for a variable with exactly the same type) and those that do not.

In principal, arbitrary combinations of these concepts are possible in programming languages. We demonstrate the power of our method by showing how these different concepts can be specified, cf. figures 3, 4, 5.

Condition	Actions	Constraints
$\mathit{Decls}(\text{"}\mathit{name}\text{"}) = \emptyset$	$\mathit{Decls}(\text{"}\mathit{name}\text{"}) := \mathit{Decls}(\text{"}\mathit{name}\text{"}) \cup \{[\![\mathit{type}]\!]\}$ $\mathit{update}(\text{"}\mathit{name}\text{"})$	$[\![\mathit{name}]\!] = [\![\mathit{type}]\!]$
$\mathit{Decls}(\text{"}\mathit{name}\text{"}) \neq \emptyset$	$\mathit{update}(\text{"}\mathit{name}\text{"})$	$[\![\mathit{name}]\!] = \mathit{error}$

Fig. 3. Semantic rules for declarations (7), overloading and re-declaration not allowed

In the assignment statement (8), for a first try, we require that the type of the expression *expr* on the right-hand side is of the same type as the designator *des* on the left-hand side. This would result in the semantic rules as specified in figure 6 where $\leadsto$ is replaced by $=$. Note that here is a clear distinction between (i) and (ii). For our example in section A we assume (ii).

Condition	Actions	Constraints
$[\![type]\!] \notin Decls("name")$	$Decls("name") :=$ $Decls("name") \cup \{[\![type]\!]\}$ $update("name")$	$[\![name]\!] = [\![type]\!]$
$[\![type]\!] \in Decls("name")$	$update("name")$	$[\![name]\!] = error$

Fig. 4. Semantic rules for declarations (7), re-declarations not allowed

Condition	Actions	Constraints
$type \in Decls("name")$	$update("name")$	$[\![name]\!] = [\![type]\!]$
$type \notin Decls("name")$	$update("name")$ $Decls("name") := Decls("name") \cup \{[\![type]\!]\}$	$[\![name]\!] = [\![type]\!]$

Fig. 5. Semantic rules for declarations (7), overloading and re-declaration allowed

Of course, to require the equality of left- and right-hand side types is much too restrictive. In general, only coercibility is needed. Coercibility relations, denoted by $\rightsquigarrow$, are language dependent and can be combined into a semi-lattice by introducing the error type *error* as top element, see figure 7. There may be different coercibility relations for different language constructs in a single programming language. They are defined by the language designer. For our example language, we get the semi-lattice from figure 7. The semantic rules for the assignment statement are summarized in figure 6.

The semantic rules for the designator are specified in figure 8. As in the case of declarations we distinguish if a variable has to be declared before its use (i) or not. In all possible cases we remember this particular use of a *name* by defining an entry for the set $Use("name")$ containing all nodes of the AST where a variable has been used. The semantic rule for the loop statement (11) is simple. We only have to require that the type of the conditioning expression is boolean. In particular, there are no actions and conditions, see figure 9.

For expressions (12)–(15), the constraints are obvious, cf. figure 10. 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 10. The first of the constraints' literals defines that

Condition	Actions	Constraints
		$[\![expr]\!] \rightsquigarrow [\![des]\!]$

Fig. 6. Semantic rules for assignments (8)

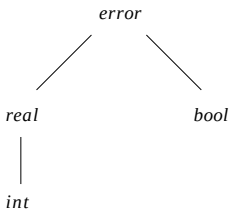


Fig. 7. Coercibility-Semi-Lattice of the example

No.	Condition	Actions	Constraints
(9)(i)	$Decls(\text{"}name\text{"}) \neq \emptyset$	$comp\_name := comp\_name.\text{"}name\text{"}$ $Use(comp\_name) :=$ $\quad Use(comp\_name) \cup \{name\}$	$[\![des0]\!] = [\![name]\!]$
(9)(i)	$Decls(\text{"}name\text{"}) = \emptyset$	$comp\_name := comp\_name.\text{"}name\text{"}$ $Use(comp\_name) :=$ $\quad Use(comp\_name) \cup \{name\}$	$[\![name]\!] = error \wedge$ $[\![des0]\!] = [\![name]\!]$
(9)(ii)		$comp\_name := comp\_name.\text{"}name\text{"}$ $Use(comp\_name) :=$ $\quad Use(comp\_name) \cup \{name\}$	$[\![des0]\!] = [\![name]\!]$
(10)(i)	$Decls(\text{"}name\text{"}) \neq \emptyset$	$comp\_name := \text{"}name\text{"}$ $Use(comp\_name) :=$ $\quad Use(comp\_name) \cup \{name\}$	$[\![des]\!] = [\![name]\!]$
(10)(i)	$Decls(\text{"}name\text{"}) = \emptyset$	$comp\_name := \text{"}name\text{"}$ $Use(comp\_name) :=$ $\quad Use(comp\_name) \cup \{name\}$	$[\![name]\!] = error \wedge$ $[\![des]\!] = [\![name]\!]$
(10)(ii)		$comp\_name := \text{"}name\text{"}$ $Use(comp\_name) :=$ $\quad Use(comp\_name) \cup \{name\}$	$[\![des]\!] = [\![name]\!]$

Fig. 8. Semantic rules for designators (9) and (10)

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.

3.4 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

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

Condition	Actions	Constraints
		$[\![expr]\!] = bool$

Fig. 9. Constraints for loops (11)

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

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.

$$decl ::= \texttt{function}\ name\ \texttt{(}\ name\ \texttt{:}\ type\ \texttt{)}\ \texttt{:}\ type\ \texttt{;}\ block \tag{17}$$

$$name ::= \texttt{result} \tag{18}$$

$$expr ::= des\ \texttt{(}\ des\ \texttt{)} \tag{19}$$

$$block ::= \texttt{begin}\ stats\ \texttt{end} \tag{20}$$

$$stats ::= \texttt{;} \tag{21}$$

$$\mid\ (stat \mid decl)\ \texttt{;}\ stats \tag{22}$$

The introduction of *blocks* requires an extended functionality of the name 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 blocks. These actions are assumed to be performed by the functions *enter_scope* and *leave_scope*.

As already explained in subsection 3.3, 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 name 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*. If we do not find a declaration for a name in the current block, we process the constraints to the

outside scope where we deal with them in the same way. Thereby the constraints for a yet undeclared *name* can be processed this way 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. To simplify further discussions, we assume that these different functionalities are performed automatically as a default action by the name table. We assume `result` being a predefined name

Condition	Actions	Constraints
	$Decls(\text{"}name1\text{"}) :=$ $\quad Decls(\text{"}name1\text{"}) \cup [\![[\![type1]\!] \rightarrow [\![type2]\!]]\!]$ $enter\_scope$ $Decls(\text{"}name2\text{"}) := [\![type1]\!]$ $Decls(\texttt{result}) := [\![type2]\!]$	$[\![name1]\!] = [\![[\![type1]\!] \rightarrow [\![type2]\!]]\!]$ $[\![name2]\!] = [\![type1]\!]$ $[\![result]\!] = [\![type2]\!]$

Fig. 11. Semantic rules for function declarations (17)

denoting the result of a function. For simplicity we assume that there is also such a `result`-parameter in the outermost block, denoting the result of the entire program. 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.

Condition	Actions	Constraints
	$Use(\texttt{result}) := Use(\texttt{result}) \cup \texttt{result}$	$[\![des]\!] = [\![\texttt{result}]\!]$

Fig. 12. Semantic rules for result parameters (18)

Figure 13 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 name table.

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

Fig. 13. Constraints for function calls (19)

At the end of a block, we have to update the current scope. This is performed by the execution of *leave_scope*, see figure 14. This rule applies always at the end of blocks. Therefore no condition is stated. Also no constraints result from this semantic rule.

Condition	Actions	Constraints
	leave_scope	

Fig. 14. Actions for block ends (21)

4 The Analysis

Semantic conditions are associated with nodes in the abstract syntax tree, cf. section 3. It remains to show how the constraint set is organized, simplified, and checked for consistency in an efficient way.

4.1 The Analysis 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:

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

$$\vee(t_1, \ldots, t_k) \tag{26}$$

which denotes the maximum of types $t_1, \ldots, t_n$ in the coercibility-semi-lattice and the set of types $\{t_1, \ldots, t_n\}$, respectively. $\max_{\rightsquigarrow}$ is derived from $\rightsquigarrow$. A predicate $[\![n]\!] = t$ is a *definition* of n iff t is a language type. A predicate $[\![n]\!] = \{t_1, \ldots, t_n\}$ determines the set of *possible definitions* of n.

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

[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".

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 15 shows C for a defining predicate $[\![n]\!] = t$ and possible definitions $[\![n]\!] = \{t_1, \ldots, t_n\}$. Initially, the set of possible definitions for each name

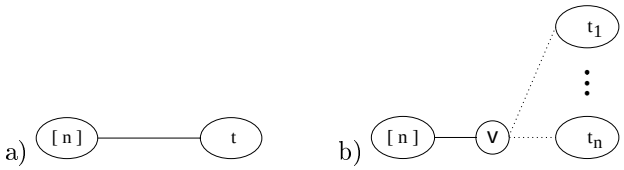


Fig. 15. C for the defining predicate $[\![n]\!] = t$ (a) and the possible definitions $[\![n]\!] = \{t_1, \ldots, t_n\}$ (b).

is determined by the name table. These sets are propagated along the equality and coercibility edges in C, resp., and simplified if possible. Thereby, vertices and/or edges may be removed from C. This follows the four principle rewriting rules. RULE (I) simply propagates definitions, cf. figure 16. It means that if a node must have the same type as another node (two vertices in C are connected by an equality edge) then the definitions or the possible definitions for one node must be a definition or a possible definition of the other node.

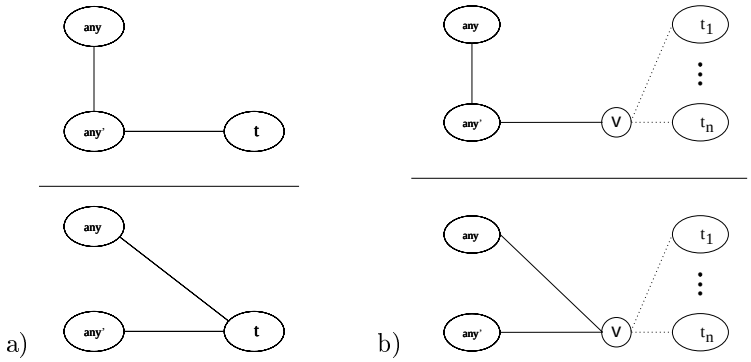


Fig. 16. RULE (I): propagation of definitions (a) and possible definitions (b).

Equivalence of language types may be checked. If C contains an "="-edge between vertices representing language types or sets of 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 17. If they are sets of types they are replaced by the intersection of the both sets (c)[5]. If this intersection is empty the subgraph is replaced by the *error* vertex (d).

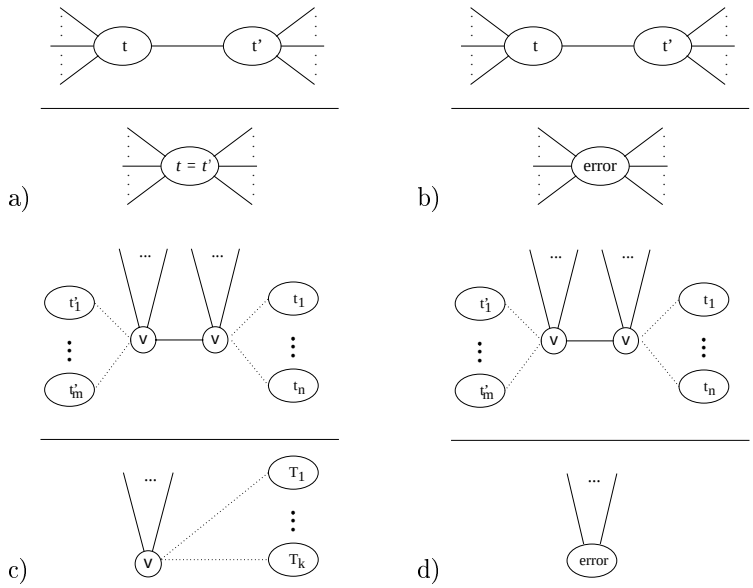


Fig. 17. RULE (II): solving equality constraints.

A special case of RULE (II) occurs if one of the types sets is the set of possible definitions. This set is treated differently because it will never be removed. In this case, cf. figure 18, the intersection is computed as before. If the resulting type is unique then an edge to the appropriate definition is inserted. In figure 18: $\{t'_x\} = \{t'_1, \ldots, t'_m\} \cap \{t_1, \ldots, t_n\}$.

After application of RULE (II) a special case of RULE (I) may occur. Assume that one of the compared types is a function type where the argument type and/or the result type is not yet defined. This situation arises from the constraint for calls, cf. 13. Anyway, we compute the intersection and reduce the set of possible definitions to the possible function types by applying RULE (II). If the resulting set is empty, an error has been detected. If the set contains only one definition, new constraints for the argument and/or result types of a use are inserted according to the argument and/or result types of the definition, cf. figure 19.

Basically, RULE (III) performs the same for the $\rightsquigarrow$ constraints as RULE (II) for the equivalence. However, if RULE (III) is applied, the vertices are not melted

[5] A single defining type may be seen as a set containing only one element.

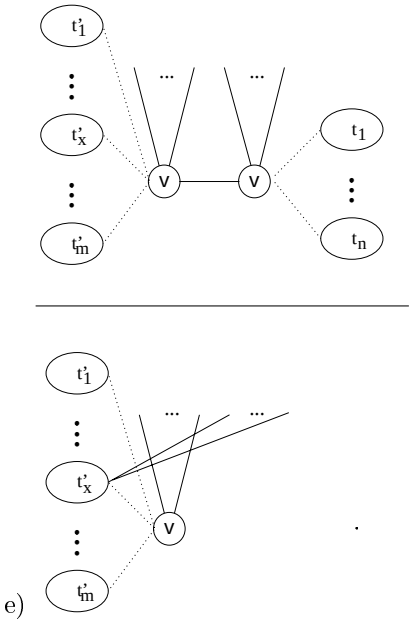


Fig. 18. RULE (II): (e) selecting the proper definition.

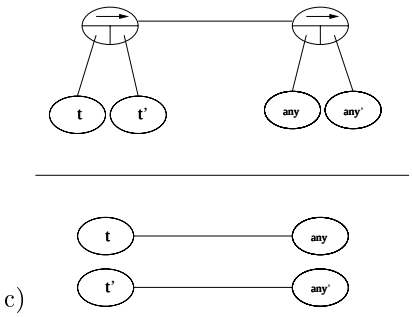


Fig. 19. RULE (I): (c) propagation of definitions to function parameter and result type.

but replaced by a pair of types for which coercion is defined, cf. figure 20. An error type is inserted if both types or type sets are not coercible, i.e. cases (b) and (d) are identical for equivalence and coercibility constraints.

RULE (IV) replaces the *max* type constructor by the result of the *max*-operator if all operands have basic types, cf. figure 21.

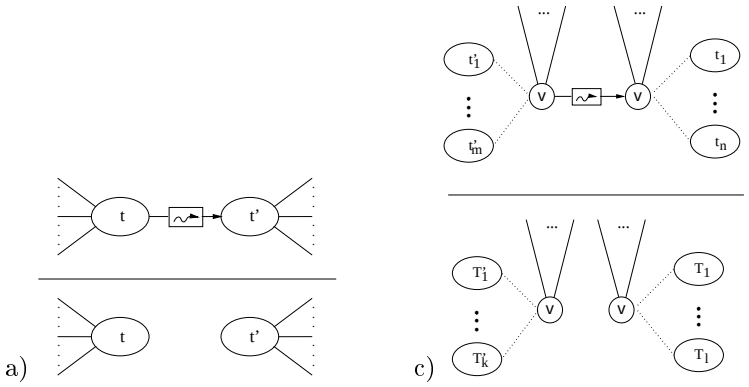


Fig. 20. RULE (III): solving coercibility constraints.

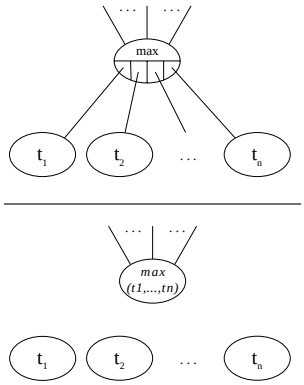


Fig. 21. RULE (IV): elimination of maximum nodes.

4.2 Correctness of the Analysis

Now we establish the correctness of the rewriting rules. Therefore, we assume 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.
- The declarations of names do not contradict their application.

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

- All names of the same scope are unique (no overloading).

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

Theorem 1. Correctness: *A program is correct w.r.t. the specification of the context-sensitive syntax of a language, iff there is no further application of* RULE(I) – RULE(IV) *possible and*

(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 names have exactly one defining predicate* $[\![name]\!] = t, t \neq error$.

Proof. First, we prove that if a program is correct w.r.t. the context-sensitive syntax then (i) – (iii) must hold after the termination of the graph rewriting. 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) – RULE(IV) 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 the rewriting has been applied completely and (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.

4.3 The Implementation of the Analysis

It remains to show how the sets of types and the operations equivalence, maximum and coercion of type sets are implemented.

Sets of types are represented by finite ordered lists[6]. This is possible because all types of all scopes are defined after the AST traversal. The set of all types in a given program is finite and a lexicographic sorting of the types is always possible. Consider a string representation of the types where each type t_i is represented by a string of length $|t_i|$. Lexicographic sorting of all types t_i takes time

$$\mathcal{O}(\sum_{t_i \in T} |t_i|), \tag{27}$$

[6] A bit vector implementation is preferable.

where T is the set of all types [AHU87]. Obviously

$$\sum_{t_i \in T} |t_i| = \mathcal{O}(n), \tag{28}$$

where n is the program length.

The intersection of sets represented by ordered lists of size n can be performed in $\mathcal{O}(n)$. Therefore, the intersection of the type sets takes time $\mathcal{O}(n)$.

Coercion and maximum operation are only defined on basic types. The number of basic types b is a constant. Hence, there are at most 2^b sets of basic types in any program. Coercibility of two sets of basic types may therefore be stored in a table of size 4^b, which is a constant, and computed in time $\mathcal{O}(1)$. The maximum of two types is pre-computed in the same way. It follows that the maximum of k types can be computed in time $\mathcal{O}(k)$.

The vertices of the constraint graph are pairs of node pointers and type sets. For every node we insert constraints in terms of other nodes of the same syntactic rule which we can find in time $\mathcal{O}(1)$. Additionally, we insert constraints according to possible definitions of names. Because of the hash table organization of our name table, we find all these possible definitions for a name in expected time of $\mathcal{O}(1)$ and in worst-case time $\mathcal{O}(n)$.

The rewriting rules could be applied in an arbitrary order. But, we determine the order for reasons of efficiency. We start with propagating definitions and possible definitions by RULE(I) until RULE(II), RULE(III), or RULE(IV) is applicable. Then these rules are applied which results in new situations for the application of RULE(I). The solution terminates if no rule is applicable. Due to this approach, we always find a new application of a rule in time $\mathcal{O}(1)$ if it exists.

From the above observations we conclude

Theorem 2. Complexity: *The defined algorithm for semantic analysis performs in time*

$$\mathcal{O}(n^2),$$

where n is the size of the program.

Proof. Traversing the AST takes time $\mathcal{O}(n)$. Insertion of the constraints for each AST node takes time $\mathcal{O}(n)$. Constructing the sets of types takes time $\mathcal{O}(n)$ for each of the $\mathcal{O}(n)$ AST nodes. Hence, constructing the constraint graph is performed in time $\mathcal{O}(n^2)$.

Each application of RULE(I) removes an equality constraint between vertices representing types of AST nodes and thus cannot applied twice to the same subgraph of C. Application of RULE(I) (c) adds a new edge but three edges are removed instead. Cyclic application cannot occur. Hence, RULE(I) is applicable only $|E|$ times where E is the set of edges in the constraint graph. Each application takes time $\mathcal{O}(1)$.

Each application of RULE(II) removes an equality constraint between vertices representing language types. Again, cyclic application cannot occur. Hence

RULE(II) is applicable only $|E|$ times. Each application takes time $\mathcal{O}(n)$. By the same argumentation, RULE(III) and RULE(IV) cannot be applied more than $|E|$ times. Each application of RULE(III) takes time $\mathcal{O}(1)$ for basic types and time $\mathcal{O}(n)$ for structured types. Each application of RULE(IV) takes time $\mathcal{O}(k)$ where k is the arity of the maximum operation. k is obviously a constant since we can only define constraints with constant arity.

A possible application of a rule is detected in time $\mathcal{O}(1)$ if it exists. Otherwise, the algorithm terminates. The constraint solving algorithm therefore performs in time $\mathcal{O}(n * |E|)$. The maximum number of edges in the constraint graph is $\mathcal{O}(|V_{AST}| * c * k)$ where c is the number of constraints per node and k is their arity. c and k are constants, $|V_{AST}|$ is $\mathcal{O}(n)$. Therefore, the constraint solving is done in $\mathcal{O}(n^2)$. Since constructing and solving the constraint graph is done sequentially and only once the theorem follows.

Remark. The program size n may be measured e.g. in the number of characters of a program. If, as a further restriction, the length of all names is bounded by a constant, then theorem 2 continues to hold for $n = |V_{AST}|$.

5 Conclusion

We have introduced a new approach for the specification of context-sensitive syntax and the generation of the semantic analysis in typed imperative programming languages. Our specification 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 a simplification in comparison to the state of the art since we have only one specification for both the description of the language and the generator input. Double specification efforts and resulting proof obligations become superfluous.

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 especially able to express overloading of arbitrary (programmer-defined) operators. This demonstrates the flexibility and power of our specification method. Moreover, our specifications are easy to formulate and understand, thereby appearing naturally.

In our approach we have combined the advantages of operational and descriptive methods. The 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 "name table" containing the names and definitions of the program objects. Specifications are given by semantic rules formulated according to the syntactic rules of the underlying context-free grammar. These semantic rules consist of three parts: conditions, actions, and constraints. Conditions indicate when a semantic rule is applicable, the actions describe how the internal state of the name table has to be changed,

and the constraints define which semantic conditions must be fulfilled. In particular, the updating of the name table and the selected constraints depend only on the actual state of the name table. The constraints collected during the traversal are managed in a data structure called "constraint graph" which allows for solving them efficiently, namely in time $\mathcal{O}(n^2)$ where n is the size of the program.

Current work deals with the stepwise extension of our approach to languages that allow for:

(1) different kinds of parameter passing,
(2) subtyping and polymorphism under closed-world assumption, and
(3) genericity under the assumption of separate compilation.

(1) seems to be straight-forward by introducing two different function types. Depending on the kind of parameter passing one of the two function types is chosen. This choice is already determined by the context-free syntax. Remember that we look only at functions with one argument so that two different function types are sufficient. (2) is a direct extension of the work presented here since subtyping may be understood as dealing with yet another lattice. Because we are already able to handle several coercibility-semi-lattices 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 our 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 [HL97].

References

[AHU87] A.V. Aho, J.E. Hopcroft, and J.D. Ullman. *Data Structures and Algorithms.* Addison-Wesley, 1987.

[Boc76] G. V. Bochmann. Semantic Evaluation from Left to Right. *Communications of the ACM*, 19(2):55–62, 1976.

[CC79] P. Cousot and R. Cousot. Systematic design of program analysis frameworks. In *Proceedings of the Sixth Annual ACM Symposium on Principles of Programming Langugaes*, pages 269–282, January 1979.

[CF58] H.B. Curry and R. Feys. *Combinatory Logic*, volume 1. North-Holland, 1958.

[Coo71] S. A. Cook. The complexity of theorem-proving procedures. In *Proc. 3rd Ann. ACM Symp. on Theory of Computing*, pages 151–158. Association for Computing Machinery, New York, 1971.

[GHL97] Sabine Glesner, Andreas Heberle, and Welf Löwe. Generating Semantic Analysis Using Constraint Programming. Submitted to the Third International Conference on Principles and Practice of Constraint Programming (CP97), 1997.

[HL97] A. Heberle and W. Löwe. Generierung von kompletten Compiler-Frontends. In *Arbeitstagung "Programmiersprachen", GI-Jahrestagung'97*. Submitted, 1997.

[Jaz81] 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.

[Jon87] Simon L. Peyton Jones. *The Implementation of Functional Programming Languages*. Prentice-Hall International Series in Computer Science, 1987.

[Kas80] U. Kastens. Ordered Attribute Grammars. *Acta Informatica*, 13(3):229–256, 1980.

[LRS74] P. M. Lewis, D. J. Rosenkrantz, and R. E. Stearns. Attributed Translations. *Journal of Computer and System Sciences*, 9(3):279–307, 1974.

[Ode93] Martin Odersky. Defining context-dependent syntax without using contexts. *ACM Transactions on Programming Languages and Systems*, 15(3):535–562, July 1993.

[PH91] Arnd Poetzsch-Heffter. *Formale Spezifikation der kontextabhängigen Syntax von Programmiersprachen*. PhD thesis, Technische Universität München, 1991.

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

[Rob65] J.A. Robinson. A machine-oriented logic based on the resolution principle. *Journal of the ACM*, 12(1):23–41, 1965.

[Rob79] J. Robinson. *Logic: Form and Function - The Mechanization of Deductive Reasoning*. North-Holland, New York, 1979.

[Sch24] M. Schönfinkel. Über die Bausteine der mathematischen Logik. *Math. Ann.*, 92:305–316, 1924.

[Uhl86] Jürgen Uhl. *Spezifikation von Programmiersprachen und Übersetzern*. PhD thesis, Universität Karlsruhe, 1986.

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

A 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.

The following figures show several snapshots of the 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. Additionally, we represent the function + by one function, instead of representing currying explicitly.

In the beginning, we insert the constraint for the declaration of a, visit the conditional statement and add the corresponding constraint, see fig. 23(a). Then we extend the graph by constraint for assignments, 23(b). In the following we visit the children nodes of the expression, process the declarations for a and the

```
a : REAL
if a then
    a := succ(a) + 3.1
endif
...
a : BOOL
a : INT
...
succ(x : INT) : INT
    result := x + 1 ;
end
```

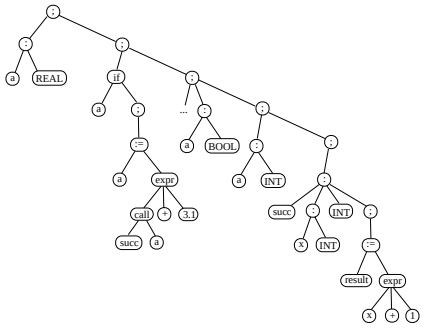
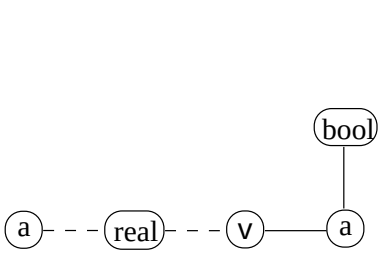
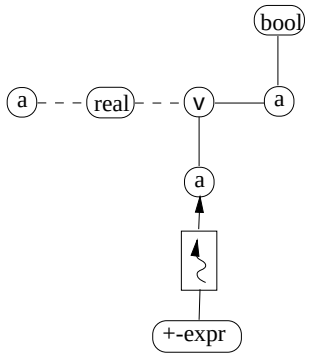


Fig. 22. An example program p and its AST representation

definition of *succ*, and add the according constraints. This leads to the global constraint graph in fig. 24. Figure 25 describes the constraints for the function *succ*.



(a) Processing the declaration and the conditional



(b) Processing the assignment

Fig. 23. Snapshots during the creation of the global constraint graph

After finishing the traversal of the AST, we start simplifying the constraint graph. Figure 26 shows the simplification of the subgraph which corresponds to the function *succ*. First we propagate the type definition of x (rule 17e), eliminate the *max*-vertex (rule 21) and remove the coercions $x \rightsquigarrow +- expr$, $1 \rightsquigarrow +- expr$ (rule 20a). Then, we determine the type of the +-operator by intersecting the two corresponding type sets (rule 18). Figure 26 shows the final subgraph for

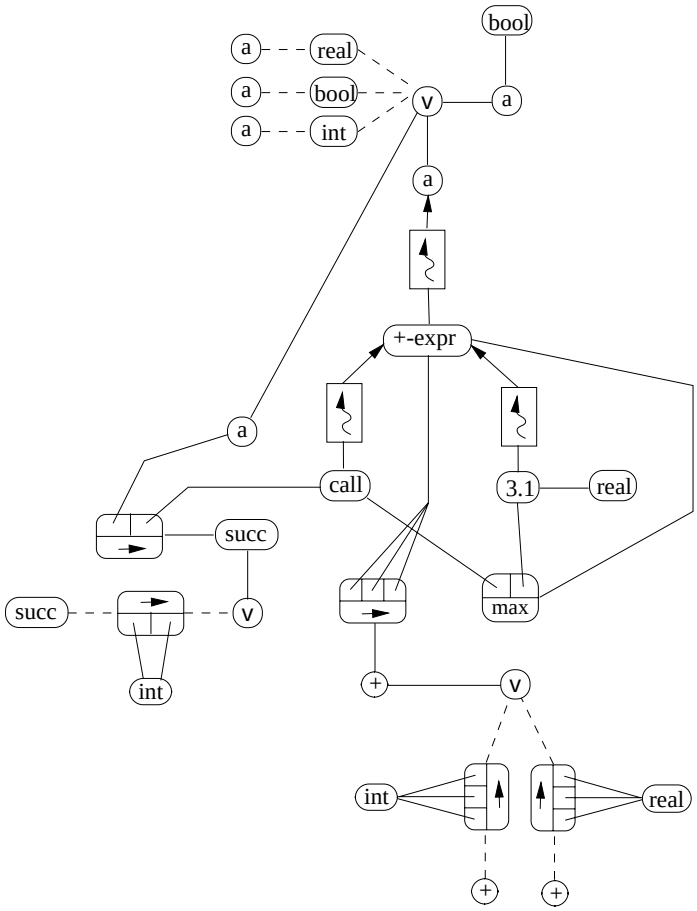


Fig. 24. The constraint graph of the global scope

succ.

During the simplification of the global program part, figure 27 describes the situation after propagating the declaration information of names.

Figure 28 shows the constraint graph after removing the *max*-vertex (rule 21), after eliminating the coercions $call \rightsquigarrow +- expr$ and $3.1 \rightsquigarrow +- expr$ (rule 20a), and after determining the type of + (rule 18).

Finally, we determine the type of a by solving the coercion constraint (rule 20b). The result of applying rule 20b is that a and $+- expr$ are of type *real*. Now, we can apply rule 16a and then rule 18. This leads to the consistent graph in figure 29.

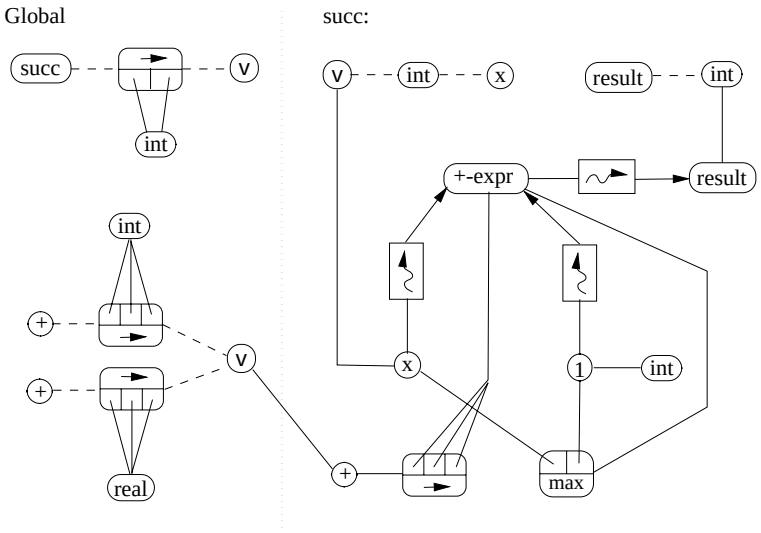


Fig. 25. (e): Local and global constraints for *succ*

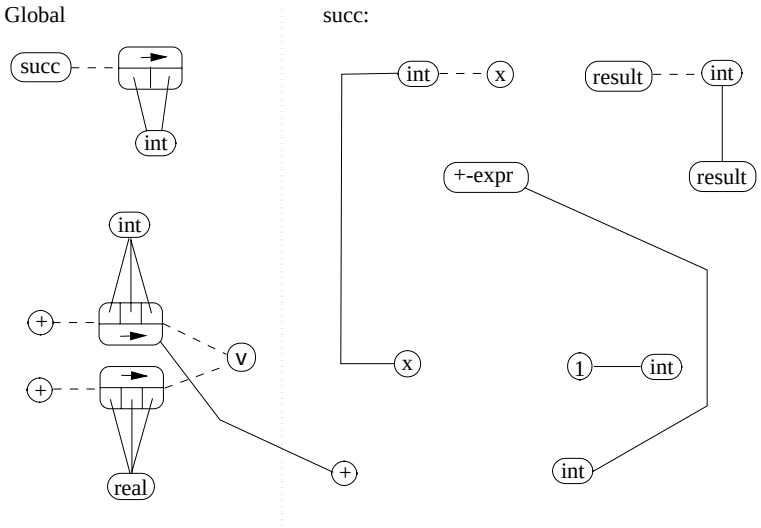


Fig. 26. Simplification of *succ*

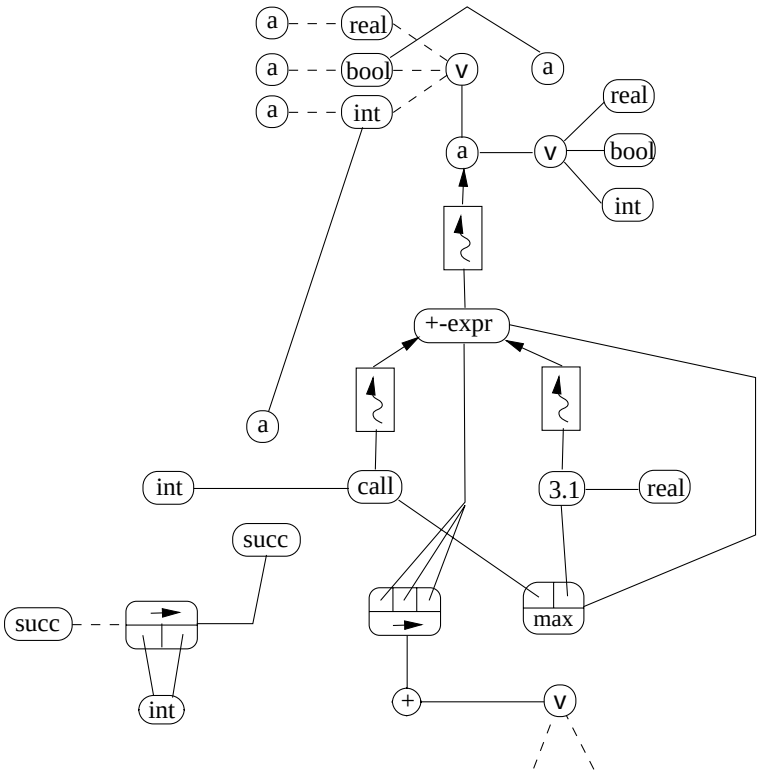


Fig. 27. Simplification of global

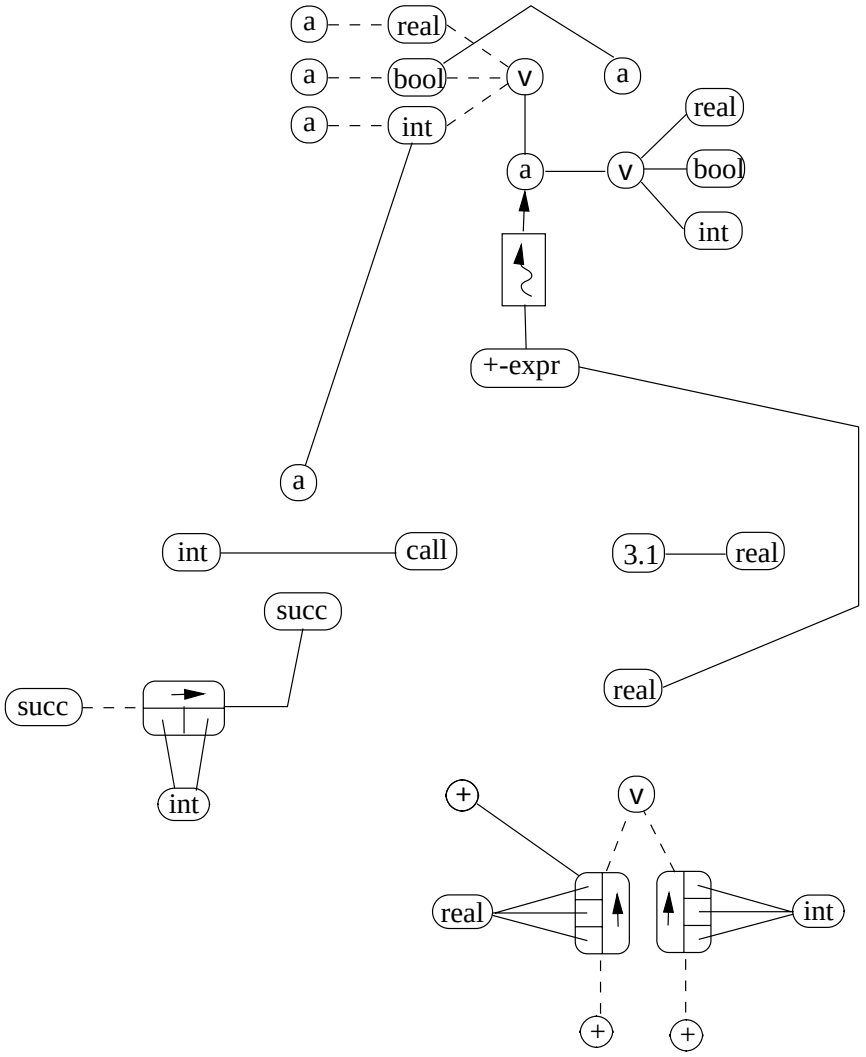


Fig. 28. Further simplification of global

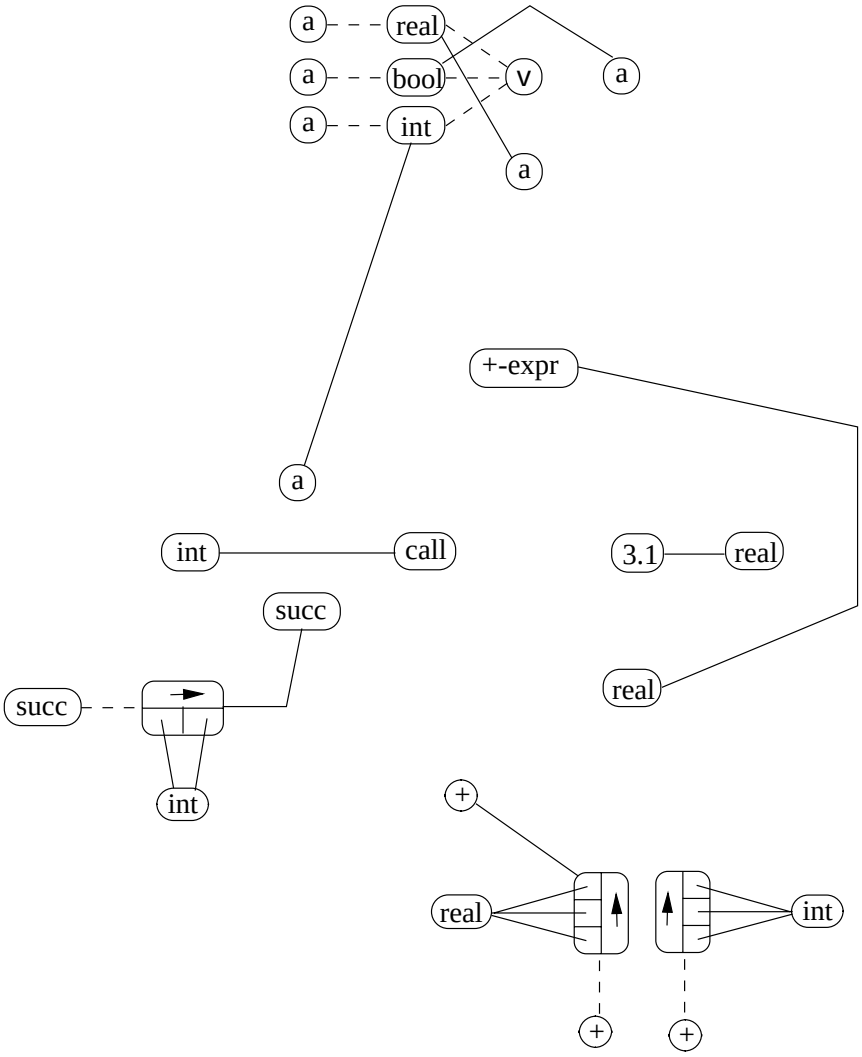


Fig. 29. Final constraint graph of the global scope