Unification in many-sorted algebras as a device for incremental semantic analysis

Gregor Snelting
Wolfgang Henhapl

Programmiersprachen und Übersetzer II
Fachbereich Informatik
Technische Hochschule Darmstadt
Magdalenenstr. 11
D-61 Darmstadt
West Germany

ABSTRACT

Language-specific editors for typed programming languages must contain a subsystem for semantic analysis in order to guarantee correctness of programs with respect to the context conditions of the language. As programs are usually incomplete during development, the semantic analysis must be able to cope with missing context information, e.g. incomplete variable declarations or calls to procedures imported from still missing modules. In this paper we present an algorithm for incremental semantic analysis, which guarantees immediate detection of semantic errors even in arbitrary incomplete program fragments. The algorithm is generated from the language's context conditions, which are described by inference rules. During editing, these rules are evaluated using a unification algorithm for many-sorted algebras with semi-lattice ordered subsorts and non-empty equational theories. The method has been implemented as part of the PSG system, which generates interactive programming environments from formal language definitions, and has been successfully used to generate an incremental semantic analysis for PASCAL and MODULA-2.

1. Introduction

Programming environments for a specific programming language should support the interactive construction of correct programs. Correctness for typed programming languages includes wellformedness according to the context conditions of the language. Therefore, a semantic analyser must be part of a programming environment which checks context conditions during program construction. For use within a language-specific editor, the analysis algorithm must fulfill several requirements:

1. Programs are usually incomplete during development. Program parts which are important with respect to semantic analysis (e.g. declarations) may be still missing or incomplete. The most general form of an incomplete program is a sentential form of some nonterminal of the language's syntax. We call such forms (and their representations as abstract trees) fragments. If the basic units for editing are incomplete fragments which may be edited separately, in a PASCAL environment the following situation could arise: A programmer types the incomplete procedure fragment

```
procedure p(a:t1; k:t2);
begin
   a[a[k+5]] := 3.14;
```

Although t1 and t2 are global objects the definition of which is not part of the fragment, the semantic analysis must be able to handle this fragment as a separate unit. Therefore, we require that the semantic analysis must be able to analyse arbitrary incomplete program fragments.

2. Fragments are correct, if they are correct programs or if they can be embedded into a correct program. Therefore, the semantic analysis must report semantic errors as soon as a fragment can no longer be embedded into a correct program. Our above example must immediately considered semantically incorrect, as left and right hand sides of the assignment have incompatible types, regardless of the missing global declarations.

3. For efficient use in an interactive environment, the semantic analysis must work in an incremental manner.

4. It is useful not to implement a language-specific environment by hand, but to use a generator. Therefore, it must be possible to generate the semantic analysis from a formal definition of

the context conditions of a language.

In the following, we present the concept, theory and algorithms of a semantic analysis satisfying the above requirements. The method has been implemented as part of the PSG Programming System Generator, which generates interactive programming environments from formal language definitions /BaSn85/. Within PSG environments, fragments are the basic units for editing and execution. The semantic analysis is part of the hybrid editor generated by PSG. Analysers for ALGOL60, PASCAL, MODULA-2 and the formal language definition language itself have been generated successfully.

2. The concept of context relations.

We assume that program fragments are internally represented as abstract syntax trees during editing. After each modification, the tree is subjected to the incremental semantic analysis. As usual, we associate attributes with the nodes of an abstract tree. However, it will be impossible to compute uniquely determined attribute values for tree nodes, because in an incomplete fragment important information (e.g. declarations) may be missing. A well-known method to handle this problem is to use classical attribute grammars together with special "default" attribute values associated with completing productions, and to use an incremental attribute evaluation algorithm /Reps83/. However, classical-style attribute grammars always follow the scheme: first inspect the declarations, building up an environment, then use this environment to perform e.g. type checking. Therefore, using classical-style attributed grammars, incomplete fragments cannot be type checked if declarations are missing.

Because of this defect, we explicitely pass over from attribute values to sets of "still possible" attribute values. The basic idea is as follows: A correct fragment can be embedded into a (usually infinite) set of correct and complete programs. These programs can be attributed, yielding a set of attribute assignments to tree nodes. The restriction of all these assignments onto the fragment in question results in a set of attribute assignments for the fragment, which represents exactly the context information corresponding to the fragment. Instead of using several attributes for a tree node, we use at most one attribute for each node, which however may be structured. As attribute values are associated with tree nodes, a collection of attribute assignments can then be seen as a relation in the sense of relational data base theory: the columns of such a relation are labelled with the tree nodes, tuple elements are attribute values, and each tuple represents a possible attribute assignment for the fragment. Such a relation is called a context relation. A context relation associated with a fragment contains exactly the still possible attribute assignments of the fragment. If the fragment is complete and correct, the relation will contain exactly one tuple, as there is only one possible attribute assignment for complete programs. In case of a semantic error, the relation will become empty, bebause no correct assignment of attribute values to tree nodes exists. Note that a context relation may be of infinite size, if the set of underlying attribute values is infinite. For certain languages (not for PASCAL or C) context relations may even not be recursively enumerable.

Formally, let A be the set of possible attribute values of the language, N the nodes of a fragment F. The context relation CR(F) associated with F is a set of mappings

$$\{t: N \to A\}$$

The set of all context relations is denoted by CR.

During editing, a fragment is produced step by step by composing a bigger tree from smaller trees: subtree placeholders (unexpanded nonterminals) will be replaced by subtrees, or subtrees of a fragment will be deleted and replaced by subtree placeholders. As a basis for incremental analysis, we therefore need an operation which computes the relation of a fragment from the relations of its components. Actually, this operation is just the natural join of relations (as known from data base theory, see /AhoB79/). If a placeholder X in a fragment F is replaced by a fragment G, thus giving a new fragment H, we therefore have

$$CR(H) = CR(G) \bowtie CR(F)$$

For examples, see /HeSn84/ and the following sections. However, there must be some relations to start with! At this point, the language definer enters the scene: In a syntax-oriented manner, he has to specify so-called basic relations for all terminals and all constructors of the abstract syntax of the language /PSG85/. Once these basic relations have been defined, all fragments may be analysed by joining the basic relations of their components. Again, basic relations may contain an infinite number of tuples: In PASCAL, an isolated identifier may have the whole set of PASCAL types as still possible attribute. Therefore, the basic relation for identifiers contains at least one tuple for each PASCAL type.

3. The representation of context relations

As context relations are usually infinite, we have to construct a finite representation. The basic idea is to use a grammar: The set of all attribute values is described by an abstract syntax, a so-called data attribute grammar (DAG). Here, the structure of the attributes of the language is defined. For PASCAL, typical DAG rules might look like this:

```
attribute :: type object_class
type = simple_type, array_type, set_type, ...
simple_type = arithmetic, ordinal
arithmetic = integer, real
ordinal = integer, boolean, char, enumeration, ..
set_type :: ordinal
arraytype :: ordinal type
object_class = variable, ctype, constant,
                        procedure, function, ...
...
```

Here, :: indicates a so-called node rule, that is, the subcomponents of a given structure are described. In our example an attribute consists of two subcomponents, namely the type of an object as well as its object class. On the other hand, =

indicates a class rule, that is, alternatives within attributes are described. For example, a type may be a simple type, an array type, a set type and so on. Note that classes need not be disjoint: integer is an arithmetic type as well as an ordinal type. An array type has again subcomponents, namely an ordinal index type and a component type, which may be an arbitrary type. Since attribute classes may contain subclasses, a DAG also includes the concept of a subtype in a natural way: integer is also an ordinal type, and each ordinal type is a simple type, each simple type is a type. DAG symbols not occuring on the left hand side of a rule are considered to be terminals.

A DAG describes a many sorted free algebra with subsorts as follows: Each symbol of the DAG gives rise to a sort. The terminal symbols are considered as nullary constants of their own sort, and the left hand sides of node rules are considerd as non-nullary function symbols with arity according to the node rule. The terms freely generated by all terminal symbols are exactly the possible attribute values, denoted by A(DAG). As an abstract syntax also describes a set of trees, A(DAG) can also be seen as the tree language generated from the DAG. The terms freely generated by the terminal symbols and the class names (which also are considered nullary constants) are just the incomplete derivation trees (sentential forms) generated by the DAG; they are called attribute forms and are denoted by AF(DAG). Thus, an attribute form may contain nonterminal leaves. As usual, we also use the notion of derivation: for x,y ∈ AF(DAG) we write x ⇒ y iff y may be derived from x by substituting an attribute form of the correct sort for a nonterminal leave. In this case, we also consider the sort associated with y to be a subsort of the sort associated with x. An attribute form can be used to represent an infinite set of attributes, namely all those attributes which can be derived from it.

As usual, we add variables: The algebra freely generated by the terminals, class symbols and an inifite set of sorted variables is called the algebra of attribute forms with variables and denoted by AFV(DAG). The sort of a variable v is denoted by sort(v), and for x ∈ AFV(DAG) we denote the variables in x by vars(x).

We now define the notion of attribute form relations: Given a fragment F, an attribute form relation describing F is a finite set of mappings from the tree nodes N of F to attribute forms with variables. The set of all attribute form relations is denoted by AFR. An attribute form relation r∈AFR represents a possibly infinite context relation R[r] ∈ CR as follows:

t ∈ R[r] iff there is t' ∈ r and there is a mapping

$$e: vars(t') \to A(DAG)$$

such that for all s ∈ dom(t)

$$e^*(t'(s)) \to t(s)$$

where e* is the homomorphic extension of e to attribute forms.

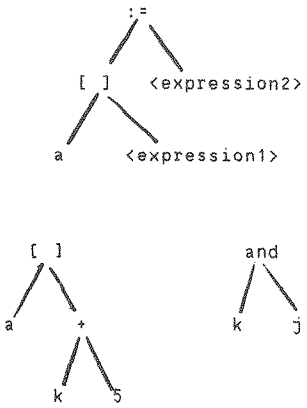
At this point, we will give some examples. Consider the PASCAL fragments

1) a[<expression1>] := <expression2>
(an incomplete assignment)

2) a[k+5] (a variable)

3) k and j (an expression)

with corresponding abstract syntax trees



For the sake of simplicity, we do not distinguish between a subrange type and its base type, and we assume, that within an assignment both sides must have the same type or the left hand side has type real and the right hand side has type integer (these simplifications are not essential). Therefore, in fragment one, we do not know the component type of the array, but it is clear that the still missing index must be of ordinal type. Furthermore, the still missing right hand side of the assignment either must have the same type as the component type of array a, or a has component type real and the right hand side has type integer. In the second fragment, k must be a variable, function or constant of type integer, and a therefore has index type compatible with integer. Note that even though the addition in PASCAL is overloaded, k cannot be real, as it is used within an array index. The fragment itself has the same type as the component type of array a. In the third fragment, k, j and the fragment itself must have type boolean.
These inferences are valid regardless of the programs into which the fragments can be hypothetically embedded and can be done without looking at any declarations, but more cannot be said. We now describe the possible attribute assignments to fragments nodes by attribute form relations. For the sake of readability, we will ignore the object_class component of attributes and simply concentrate on the types of the objects involved. The attribute form relations corresponding to the fragments are

(1)

a	<expression1>	[ ]	<expression2>
array_type(ORDINAL, TYPE)	ORDINAL	TYPE	TYPE
array_type(ORDINAL, real)	ORDINAL	real	integer

(2)

a	[ ]	k	+	5
array_type(integer, TYPE)	TYPE	integer	integer	integer

(3)

k	and	j
boolean	boolean	boolean

The column labels of these relations are the nodes of the corresponding fragments which possess an attribute. Tupel components are attribute forms with variables. The first relation has two tupels. The first tuple contains the variables ORDINAL and TYPE, which are (similar to PROLOG) written in Upper case letters. For the sake of simplicity, the names of the variables also indicate their sort. Thus, the first tuple states that 'a' may be an array of unknown index and component type and that the still missing index expression is of the same ordinal type; the right hand side must be of the same type as the array component type. The second tuple states that alternatively 'a' may have component type real and the right hand side type integer; again the variable ORDINAL describes that the unknown index type must be the same as the type of the still missing index. A similar interpretation applies to the two other relations. Note that the scope of a variable is always the tuple it occurs in. The first two relations represent infinite context relations, whereas the third relation represents a one-tuple context relation (which is accidently identical with the basic relation for the logical and operator). Note that still possible attribute assignments different from those represented by the given relations do not exist for our fragments, regardless of global context: the relations describe exactly the sets of possible attribute assignments to fragment nodes. As example three shows, such a set can be unique even in the absence of declarations.

Unification as a device for modelling the join

It is now neccessary to construct an operation for attribute form relations which exactly represents the join. This operation is unification in our many-sorted algebra with subsorts. Unification in many-sorted algebras works similar to the classical Robinson unification /Robi65/. However as we have subsorts and non-disjoint sorts, in order to unify two variables of different sorts it is neccessary to find a sort which describes exactly the intersection of the original sorts. Therefore, we require for two sorts that their intersection is either empty or again a sort, which is equivalent to

(AF(DAG); $\subseteq$) is an upper semilattice.

Thus, the unification has to compute suprema in this lattice from time to time. For our sample DAG, if we have a variable of sort ordinal and a variable of sort arithmetic, their unification is a variable of sort integer. On the other hand, the unification of a variable of sort array_type and another variable of sort real fails, as real and array_type are disjoint sorts. Note that from a theoretical point of view it is not essential to include non-disjoint DAG classes. The special case "classes must be disjoint" is theoretically sufficient and leads to a subsort ordering which has tree structure rather than to be an upper semilattice. From a practical point of view, however, it is essential that relations contain as few tuples as possible. Therefore, any attribute subset relevant in a language should be represented by a DAG class rather than by different tuples within a relation. Considering that, including non-disjoint DAG classes is essential for performance.

Theorem Considering tuples as special terms, if the DAG induces an upper semilattice, tuple-wise unification of attribute form relations represents the natural join exactly:

$$R[\{t \mid \text{there are } t' \in r1,\ t'' \in r2,\ t = univ(t', t'')\}] = R[r1] \bowtie R[r2]$$

Furthermore, the unification as scetched above will produce a correct and unique most general unifier for many-sorted algebras with semilattice ordered subsorts.

Proof see /Snel83/.

Examples

a) We compose fragments 1) and 2), thus obtaining the fragment

a[a[k+5]] := <expression>

We have to unify tuple components in corresponding columns of our two relations. In the example, the column for 'a' in relation (1) has to be matched against the corresponding column for 'a' in relation (2), and the column for '<expression1>' in relation (1) has to be matched against the column for '[ ]' in relation (2). Note that in general, scope and visibility rules of the language in question must be obeyed when determining which columns match; this process is not described here (see /HuSc83/). Unification of the attributes array_type(ORDINAL, TYPE) and array_type(integer, TYPE) results in a new sort for the variable ORDINAL, namely integer, as integer is a subsort of ordinal. Furthermore, the two TYPE variables are unified. Next, considering the columns for '<expression1>' in relation (1) and '[ ]' in relation (2), we have to unify the variables ORDINAL and TYPE. But ORDINAL has already been substituted by integer. Therefore, TYPE also changes its sort and becomes integer (note that in our setting for a variable "to change sort" and "to get a new value" are somewhat equivalent). Now, the second tuple of the first relation must be considered. Here, we unify array_type(ORDINAL, real) and array_type(integer, TYPE) resulting in a new sort for ORDINAL, namely integer, and a new sort for TYPE, namely real. Next, ORDINAL and TYPE have to

be unified; however, because the constants integer and real are not unifyable (the intersection of the corresponding sorts is empty) the whole unification fails. Thus, we obtain a new relation consisting of one tuple

a	[ ]	[ ]	k	+	5	<expression>
array_type(integer,integer)	integer	integer	integer	integer	integer	integer

that is, we have inferred that in the new fragment the still missing right hand side of the assignment, as well as index and component type of array a, must be of type integer.

b) We compose our newly derived fragment and our original fragment (3) to form the assignment

```
a[a[k+5]] := k and j
```

Here, we have to unify the attributes for <expression> in fragment (1) and the "and" node in fragment (3), as well as the attributes for k. However, unification of integer and boolean fails at once. We therefore obtain the empty relation

a	[ ]	[ ]	k	+	5	and	j

indicating a semantic error: Type conflict in an assignment. This example illustrates how the method guarantees immediate detection of semantic errors even in incomplete fragments. Furthermore, since the columns and attributes which did not match are known, it is also possible to locate semantic errors exactly.

4. Building in equational theories

The use of sorted variables allows the specification of equality in certain (sub)attributes of a tuple together with an indication of admissible substitutions. However, for practical purposes this is not enough. We will give an example: MODULA-2 allows the use of constant expressions within constant declarations. For purposes of semantic analysis, it is important to evaluate these constant expressions: The fragment

```
CONST a = 3;
      b = a-3;
VAR x: ARRAY [a..b] OF <type>
```

is obviously incorrect. Expression evaluation within semantic analysis based on unification is equivalent to unification in algebras with non-empty equational theory. The unfication algorithm in our example must know that 3-3 = 0 and 3 <=0 = false. Arbitrary complicated examples like this may be constructed. However, in his well-known paper /Plo72/ Plotkin showed that finite most general unifiers for algebras with non-empty equational theory in general do not exist. As our concept is language independent, we do not want to look for correct unification algorithms for each language (there are certain equational theories where finite most general unifiers exist). Since the general problem is not solvable, we have developed an extension of our unification in order to be able to handle arbitrary equational theories, which works correct with "almost any" input in the sense of open problem 8 in /Siek84/.

The basic idea is as follows: We extend our attribute algebra with sorts and terms for which an interpreter is assumed to exist, that is, we mark certain attribute forms as evaluable. In our example, we introduce integer values and arithmetic and assume that an interpreter for arithmetic and relational expressions exists which for example can determine that 3-3=0. Thus, if we assume that constants are described by their type and value

```
const_attr :: simple_type value
value = Int_value, Real_value, Bool_value
```

where Int_value etc. are assumed to be predefined DAG classes, the basic relation for constant addition in MODULA-2 might look as follows (const_add is a node with two sons,const_expr1 and const_expr2):*

const add	const attr(ARITHMETIC,VALUE1+VALUE2)
const expr1	const attr(ARITHMETIC,VALUE1)
const_expr2	const_attr(ARITHMETIC,VALUE2)

During analysis, unification and evaluation are intertwined. The system keeps track of unevaluated expressions. Once the neccesary arguments of as yet unevaluated expressions are known (this might be a consequence of unifications), the expressions are evaluated at once. This concept is known as data driven evaluation strategy: Unevaluated expressions are waiting as demons; they are always evaluated as soon as possible. Thus, unification calls evaluation if possible, however the results of evaluations must again be considered for unification: evaluation calls unification if neccessary. This concept does not work in every case: there might be unevaluated expressions, which, in case of evaluation, would cause subsequent unifications to fail; however, they never get evaluated. Fortunately, this does not happen very often, and as mentioned above, something better will probably not exist for arbitrary equational theories. Note that extending unification by data-driven evaluation is also considered a useful extension of PROLOG. We consider the approach to be an alternative to narrowing algorithms.

5. The incremental analysis concept

We have seen that unification, intertwined with evaluation, gives a useful basis for incremental semantic analysis. Conceptually, it would be sufficient to store with each fragment one big "global relation" which contains all the attri-

* This relation has been rotated for layout reasons

butes of the fragment. During editing, this relation must then be modified after each editing step. If, for example, an unexpanded nonterminal is replaced by a new subtree, it would be sufficient to analyse the new subtree by joining the basic relations of its components and then to use one join to update the global relation. This scheme, however, is not very appropriate, because it might require a complete re-analysis of fragments after subtree deletions. It is far better to distribute the global relation within the syntax tree: Some fragment nodes have a "local relation" attached, which describes part of the subtree beginning at that node. It is not neccessary to include into such a local relation e.g. attributes of objects which are not visible at the corresponding fragment node, according to the scope rules of the language. After an editing step, there is usually only a small number of small relations to be updated, which is far more efficient than to update one big relation. Only local relations attached to fragment nodes on the path from the modified subtree to the fragment root must be considered. After a subtree insertion,these relations have to be joined with the relation of the new subtree (which has to be analysed first). After a subtree deletion, these relations must be recomputed from basic relations and other local relations which are not affected by the subtree deletion and therefore need not be recomputed. The analysis can often be stopped after considering just one or two local relations: as soon as the scope rules guarantee that inserted or deleted syntactic objects cannot have any influence on surrounding parts of the fragment, updating of local relations on the path from the modified subtree to the fragment root may be aborted. It is even possible to implement a Reps-style change propagation algorithm for local relations: updating local relations is stopped as soon as no more changes occur.

In general, the complete analysis of a fragment of size n requires O(n*ln n) unifications, whereas the incremental analysis after one editing step requires typically O(ln n) unifications. For a detailed description of the incremental analysis algorithms including complexity analysis, see /Snel85/.

During editing, the context relations are primarily used to detect semantic errors. Of course, relations associated with fragments can also be used as symbol tables: within a PSG environment, the user always may have a look at the attributes of syntactic objects. Note that relational analysis does not require any objects to be declared, scope analysis will however detect missing declarations as soon as the last possibility of declaring that object has been deleted and there is no possibility of declaring that object outside the fragment in question.

Furthermore, the relational approach can be used to guarantee not only immediate detection of semantic errors, but also their prevention: A PSG editor always offers users the possibility to manipulate their fragments by selection from language-specific menu items. These menus are generated according to the abstract syntax of the language. However, they are additionally filtered dynamically with respect to context conditions. Thus, if PSG editors are simply used as structure editors, they guarantee the prevention of both, syntactic and semantic errors.

6. Comparison with related work

Several techniques for incremental semantic analysis in language-specific editors have been developed. The probably most well-known concepts are semantic action routines in GANDALF /Hab82/ and incremental attribute evaluation within the Cornell Program Synthesizer /Reps83/; there are also variations on the attribute grammar theme e.g. /JoFi82/. It is possible to implement our concept using these techniques. In fact, context relations and unification have experimentally been implemented using the Cornell synthesizer generator. However, the concept of inferring sets of still possible attributes within incomplete fragments seems to be new. All the known concepts have always obeyed the classical scheme: First inspect the declarations, then use the collected information for the analysis of statements etc. It was a direct consequence of the PSG fragment concept that we had to do it another way.

The Milner-Style analysis of type-free lambda calculus expressions /Miln78/ computes the most general polymorphic type of a given lambda term. It also uses unification and is in some sense similar to our scheme. However, Milner and the workers who extended the approach never tried to implement incremental algorithms, and they also did not use the concept within an environment generator.

7. Final remarks

The incremental semantic analysis with context relations is in operation as part of the PSG system since 1984. It is implemented in PASCAL, all in all about 10000 lines of code. For a 100 line PASCAL program, the complete analysis requires 1.2 CPU seconds on a SIEMENS 7551 machine; incremental analysis of modifications requires between 0.02 and 0.3 seconds per editing step. The definition of the PASCAL context conditions (that is, the specification of the basic relations) consists of about 600 lines of meta language, which we consider to be not very much. Note that we did not present the specification of scope rules as well as several useful extensions like e.g. operations on lists.

8. References

/AhoB73/ Aho, A.V., Beeri,C., Ullman,J.D: The theory of joins in relational databases. ACM TODS 4 (1979), 3, pp. 297-314

/Aust83/ Austermühl, B.: Ein relationaler Ansatz zur Beschreibung der statischen Semantik von Programmiersprachen. PhD thesis, Technische Hochschule Darmstadt 1983

/BaSn85/ Bahlke, R, Snelting, G.: The PSG - Programming System Generator. Proc. ACM SIGPLAN 85, Language issues in programming environments. SIGPLAN notices 20 (1985), 7, pp. 28-33

/JoFi82/ Johnson, G.F, Fisher, C.N.: Non-syntactic attribute flow in language-based editors. Proc. 9th ACM symposium on principles of programming languages, 1982, pp.196-206

/Hab82/ Habermann, N. et al: The second compendium of Gandalf documentation. Dept. of Computer science, Carnegie Mellon University 1982

/HeSn84/ Henhapl, W., Snelting, G.: Context relations: a concept for incremental context analysis in program fragments. Proc. GI Fachtagung Programmiersprachen und Programmentwicklung, Springer Verlag 1984, Informatik Fachberichte 77, pp. 128-143

/HuSc83/ Hunkel, M., Schmitt, H.: Ein System zur Bezeichneridentifikation und dessen Integration in ein strukturorientiertes Ediersystem. Diploma thesis, Technische Hochschule Darmstadt 1983.

/Miln78/ Milner,R.: A theory of type polymorphism in programming. J. Computer and system sciences 17 (1978), pp. 348-375

/PSG85/ Bahlke,R., Hunkel,M., Klug,M., Snelting,G.: Language definers guide to PSG. Report PU2R3/85, Technische Hochschule Darmstadt, 1985.

/Plo72/ Plotkin, G.: Building in equational theories. Machine intelligence 7 (1972), pp. 73-90

/Reps83/ Reps, T., Teitelbaum, T., Demers, A.: Incremental context-dependent analysis for language-based editors. ACM TOPLAS 5 (1983), 3, pp. 449-477

/Robi65/ Robinson, J.A.: A machine oriented logic based on the resolution principle. JACM 12 (1965), 1, pp. 23-41

/Siek84/ Siekmann, J.: Universal unification. Proc. 7th international conference on automated deduction, Springer Verlag, LNCS 170, pp. 1-42

/Snel85/ Snelting, G.: Inkrementelle semantische Analyse in unvollstandigen Programmfragmenten mit Kontextrelationen. PhD thesis, Technische Hochschule Darmstadt 1985.