Proceedings of the 3rd International Symposium on
Programming Language Implementation and Logic
Programming
1991, Springer-Verlag. LNCS 528

Logic–Based Specification of Visibility Rules

Arnd Poetzsch–Heffter
Institut für Informatik der TU München
Arcisstrasse 21, D–8000 München 2
poetzsch@lan.informatik.tu-muenchen.de

Abstract

The paper describes a new, declarative method for the formal specification of visibility rules. In contrast to common methods that are based on the specification of a symboltable (or environment), of appropriate update operations and of passing rules, the presented method is related to visibility descriptions in language reports. It consists of three steps: First, the program entities are specified, i.e. the candidates for the meaning of an identifier; the next step defines the ranges where the program entities are valid or hidden; finally, visibility is expressed in terms of these ranges. To formally define the semantics of these visibility specifications, a modeltheoretic view of abstract syntaxtrees is sketched. Using this framework, we give a fixpoint semantics for such specifications.

1 Introduction

Formal specifications play an increasingly important role in the design and definition of programming languages (PL's). This has two main reasons: The PL's become more and more complex, and the requirements concerning portability and standardization increase. A programming language definition usually consists of four more or less seperated parts:

- the lexical syntax (regular expressions)
- the context–free syntax (context–free grammars)
- the context–dependent syntax (attribute grammars or functional specifications)
- the semantics (e.g. denotational semantics)

By context–dependent syntax, we mean the visibility and type rules and similar contextual constraints (cf.[Wat84]). This paper concentrates on specifications of visibility rules. It proposes a new specification technique that allows considerable shorter and better to read formal specifications than e.g. attribute grammars.

1.1 State of the Art

There are two problems with specification techniques for context–dependent syntax:

- How can we get efficient implementations from specifications?
- How can we decrease the size of these often voluminous specifications?

A lot of work has been done to solve the first problem, especially in the field of compiler generation ([Jon80]). Most of the developed methodes make use of the same concept: They gather declaration information by constructing an appropriate data structure (symboltable, environment,..) and pass this data structure to the applications of program entities in the syntaxtree. Thus, the visibility definition consists of three parts: The specification of the often very complex data structure, the specification of the update operations, and the passing rules (cf. [Ua82]). Unfortenately, such specifications are hard to read, are a bad basis to prove language properties, and give a rare support for error handling and treatment of program fragments. Recent approaches tried to meet these requirements by using predicate logic or specially designed specification languages. Let us briefly sketch the so far proposed methods:

- S. Reiss (see [Rei83]) presented a specification language for the definition of very elaborate symboltable modules. These modules are developed for visibility rules like those in Ada. The symboltable modules provide functions that have to be used to define the visibility rules in an attribute grammar like framework.

- The work of G. Snelting (see [SH86]) concentrates on type rules and similar constraints and their checking in incremental structure editors. The visibility rules are defined in a specially designed specification language based on a fixed scope and visibility concept like that in PASCAL.

- J. Uhl (see [Uhl86]) and M. Odersky (see [Ode89]) provide a general framework for the specification of context–dependent syntax. As in this paper, they regard a syntax tree of the programming language being specified as a first–order logical structure and define the contextual constraints via first–order formulae. In contrast to our approach, they do not provide special means to define visibility rules. This is less comfortable, leads to less structured specifications, and does not allow special implementation techniques for the identification process, that are indispensable to generate sufficiently efficient context analysers for realistic validation purposes.

1.2 A New Approach to Visibility Specification

This paper presents a new method for the specification of visibility rules. The method is part of a general framework for the specification of context–dependent syntax. It provides a formal logical basis and is related to visibility descriptions in language reports. Visibility specifictions have to answer the following question:

What identifiers are visible at a program point and what is their meaning there?

A specification according to the proposed method, answers this question in three steps:

i) The first step specifies what the meaning of an identifier can be; this is what we call a *program entity*, i.e. a variable, procedure, function, type, label, selector, etc..

ii) The second step specifies the program constructs influencing the visibility, specifies the ranges of this influence, and specifies, how these constructs influence the visibility: A construct can make valid identifier–entity–bindings and/or it can hide such bindings in the specified range.

iii) Finally, visibility is defined in terms of the ranges for validity and hiding.

The main advantage of the method compared to attribute grammar or functional specifications is that symboltable mechanisms — the crucial aspect of those techniques — can be avoided. The method is more flexible than the fixed visibility models of Reiss [Rei83] and Snelting [SH86]. On the other hand, it is sufficiently restricted to get much better implementations than [Ode89].

1.3 Paper Overview

The paper is organized as follows: Section 2 presents the specification method in more detail, describes the underlying model, and explains the visibility clauses, i.e. the construct used for the visibility specification. Section 3 provides the formal semantics for the visibility clauses by mapping them into first–order predicate definitions. As these definitions are recursive, a fixpoint–semantics is given. Section 4 contains a sketch of the whole framework for syntax specification.

2 Specification Method

This section presents the 3-step specification method sketched in the introduction. The aim of such specifications is to define a predicate *is_visible* for given abstract syntax trees (AST). The predicate takes two arguments: A so–called binding consisting of an identifier string ID and a program entity PE (see above), and as second argument a program point PP. It yields true iff the program entity PE is visible under ID at PP. To illustrate this, let us consider the following PASCAL–fragment:

```
(1)        procedure P;
(2)             type T1  =  record ... end;
(3)                  T2  =  T1;

(4)             procedure P;
(5)                  type T1  =  T2;
(6)             begin  ...  end;

(7)        begin  ...  end;
```

We are interested in the effects of the type declarations: Line (2) introduces a record type under the identifier T1. In line (3), this type additionally gets the name T2. What is the effect of the pathological[1] type renaming in line (5) ? It hides the binding between T1 and the record type of line (2) from the beginning of line (5) up to the end of line (6). And it makes just the same binding valid from the end of line (5) to the end of line (6) (cf. [ANS83]). Before we show how these informal statements can be described more precisely, we have to say some words about the representation of programs.

As we need the program structure for the visibility specification, we use abstract syntax trees to represent programs. To model the relation between the tree nodes, we provide functions like *father, firstson*, etc. and selection by names of nonterminals or terminals.

[1] We use a somewhat pathological program to demonstrate some problems with visibility specifications by a tiny example.

Let us for instance consider the AST of the above PASCAL–program (figure 1). If R denotes the root of the tree, *R.DfId.string* denotes the name of the outermost procedure, i.e. "P". The dotted notation is used throughout this paper as a convenient equivalent to unary function application. So an equivalent notation for *R.DfId.string* would be $string(DfId(R))$. To express the grammatical properties, we consider the nonterminals as types and provide type predicates of the form <*nonterminal_name*> [_] (cf. section 3).

figure 1

```
ProcDcl    ::   DfId  Dcls  Block
Dcls       :*   Dcl
Dcl        ==   ProcDcl  |  TypeDcl  |  TypeRenm
TypeDcl    ::   DfId  TypeSpec
TypeSpec   ::    ....
TypeRenm   ::   DfId  UsId
Block      ::    ....
DfId       :-   string
UsId       :-   string
```

figure 2

2.1 Specification of Program Entities

To keep our example specification small, we consider PASCAL–programs consisting only of parameterless procedure and type declarations as in the example above. The corresponding grammar is given in figure 2. In this PASCAL–subset, we have two kinds of program entities, namely procedures and types. Given an AST, we can represent the program entities by tree nodes. For procedures, we use the corresponding *ProcDecl*–nodes,

for types the *TypeSpec*–nodes; i.e. a program entity is a procedure declaration node or type specification node. The entities of the example are surrounded by a box in figure1. In our framework, we express this by the class production

```
ProgEntity  ==   ProcDcl  |  TypeSpec   .
```

2.2 Specification of Visibility Ranges

Certain constructs in a programming language influence the visibility. These are typically the declarations making the declared entities valid and hiding others. But there are other such constructs as well: e.g. the with– and use–clauses in Ada, renaming constructs, selections, etc. The influence usually ranges over a certain part of the program called a "range". To model ranges, we introduce program points. For each node N of an AST, there are two program points, which we denote by "*before*(N)" and "*after*(N)". The program points are linearly ordered as shown in figure 1 by the numbering. A range is specified by its starting and end point.

As the method should be strong enough to define overloading and renaming, the following situations can occur at a given program point: Several entities are visible under the same identifier; an entity is visible under several identifiers; an entity is not visible. That's why we describe the visibility by bindings between identifiers and program entities. Thereby, the basic idea of the specification method can be put as follows: Define the ranges where bindings are valid (called v-ranges) and where they are hidden (called h-ranges); then combine these ranges to get the points where an binding is visible (s. next section). The distinction between v-ranges and h-ranges is necessary, because in most languages they are different and independent, i.e. it is usually not possible to derive h-ranges from corresponding v-ranges. Let us e.g. consider the visibility in PASCAL. A declared program entity is valid (under its identifier) from the end of its declaration to the end of the directly enclosing block (except for procedures and pointer types), but hides bindings with the same identifier in the entire block (cf. [ANS83]). This has the following consequence for our program fragment: If there was a type visible under the name T2 outside the outermost procedure, it would not be allowed to reference this type in the type specification of line (2), because it is hidden there by the type renaming in line (3).

To specify visibility, we use a special language construct, called "visibility clause". A visibility clause consists of three parts: The first part specifies the program construct influencing the visibility (starting with keyword `vis`); the second part describes the bindings that become valid or are hidden (starting with keywords `valid` or `hidden`); and the third part specifies the range (indicated by the keywords `from` and `to`). E.g. the first visibility clause in figure 3 can be read as follows: A procedure declaration makes valid the binding between the procedure identifier and the procedure itself in the specified range; this range starts after the `DfId`–node and ends after the procedure node, if the procedure node is the root of the AST, otherwise it ends after the block of the enclosing procedure. The visibility clause for type renamings makes valid the binding between the lefthandside identifier and the type that is visible under the identifier on the righthandside of the equation in the specified range. As the hiding rules are the same for all declarations, we can specifiy them by one visibility clause as shown by the last one of figure 3: A declaration (except the outermost procedure) hides all bindings with the same identifier as the declared iden-

tifier in the range extending from before the corresponding declaration list to after the corresponding block.

```
vis    ProcDcl PD {
valid  Binding( PD.DfId.string, PD )
from   after( PD.DfId )
to     if is_root[PD]   then  after(PD)
                        else  after(PD.father.father.Block) fi }

vis    TypeDcl TD {
valid  Binding( TD.DfId.string, TD.TypeSpec )
from   after( TD )        to  after(TD.father.father.Block)  }

vis    TypeRenm TR {
valid  Binding( TR.DfId.string, TS ){
         TypeSpec[TS]
       ∧  is_visible[ Binding(TR.UsId.string,TS), before(TR.UsId) ] }
from   after( TR )        to  after(TR.father.father.Block)  }


vis    Dcl D :   ¬ is_root[D] {
hidden Binding( D.DfId.string, PE )
from   before(D.father)   to  after(D.father.father.Block)  }
```

figure 3

That is all we need to specifiy visibility. Especially, we do not need any symboltable data structure with update routines. The semantics for visibility clauses is given in section 3. Up to now, it should be noticed that the predicate *is_visible* is used in the visibility clauses, although it will be defined by them; i.e. we have a recursive definition.

2.3 Specification of Visibility

Finally, we define visibility in terms of the specified ranges: A binding BD is said to be visible at a program point PP, if

- there is a range $\langle SP, EP \rangle$ containing PP, where BD is valid, and
- there is no range containing PP that is part of $\langle SP, EP \rangle$, where BD is hidden.

This is a more precise formulation of language report statements like "an entity is visible at a given place, if it is valid and not hidden". It covers as well cases in which a binding is made visible in a range where it is hidden. This is illustrated by our example: The type renaming in line (5) hides all bindings with identifier T1 that are defined outside the procedure P in line (4), but not the binding that is made valid by the declaration itself.

Even though the presented specification method is rather simple, it is very powerful: It can capture the different scope rules of block structured languages as well as named scopes and mutual recursive declarations. For example, if we want to allow mutual recursive

definitions of functions in a declaration sequence or in a `letrec`–expression of a functional language, we only have to make valid the bindings for the functions from before the declaration sequence or the `letrec`–expression respectively. The main advantage of the method is that it leads to smaller and more natural specifications. In an experience with a PASCAL–subset, the specification with our method was about five times smaller than the corresponding attribute grammar, mainly because we need not specify symboltable data structure, update routines, and passing rules, and because we can handle many productions by one visibility clause as shown by the hiding clause in figure 3.

3 Semantics

The given tiny example specification consists of three parts: the abstract syntax, the class production for `ProgEntity`, and the visibility clauses. The following subsections define the semantics of these parts.

3.1 Syntax Trees as First–Order Structures

A *signature* of a first–order structure consists of two families of finite sets of symbols, the predicate symbols $(PRED_s)_{s\in\mathbb{N}}$ and the fuction symbols $(FUNC_s)_{s\in\mathbb{N}}$. A *first–order structure* S with signature Σ is given by a set U called the universe of S and two families of mappings $(\varphi_s)_{s\in\mathbb{N}}$ and $(\pi_s)_{s\in\mathbb{N}}$,

$$\varphi_s : FUNC_s \to \mathcal{F}(U^s, U) \quad \text{and} \quad \pi_s : PRED_s \to \mathcal{P}(U^s) \ ,$$

where $\mathcal{F}(U^s, U)$ denotes the functions from U^s to U and $\mathcal{P}(U^s)$ denotes the power–set of U^s. For more details about first–order structures see [End72], p. 79.
The semantics of a grammar as shown in figure 2 is given by a class of first–order structures. The signature of these structures consists of

- a fixed part containing the function symbols *father*(_), *firstson*(_), *rightbrother*(_), *root*(), *after*(_), *before*(_), and the predicate symbols *is_root*[_], *Node*[_], *Point*[_], [_≤_];
- a grammar–dependent part with the predicate symbols for the nonterminals, like *ProcDcl*[_], *Dcl*[_] in our example, and the function symbols to denote son–selection via nonterminal or terminal name, like *DfId*(_), *TypeSpec*(_), *string*(_).

The class of first–order structures for a grammar can be regarded as a representation of the set of abstract syntax trees. Each abstract syntax tree is modelled by one structure. The universe of such a structure is the union of the tree nodes, the terminal values, and the program points, where we have two program points for each tree node, as indicated in figure 1. Additionally, the universe contains an extra element called *undef* to handle partial functions. The interpretation of predicate and function symbols is defined as follows:

- *father*(_), *firstson*(_), *rightbrother*(_), *root*() are interpreted according to the structure of the syntax tree; in cases where their evident meaning is not defined, they yield *undef*;

- if the argument is a node, *after*(_) and *before*(_) yield the program points after and before the node; otherwise they yield *undef*;
- whether an element of the universe is the root, a node, or a program point is expressed by the predicates *is_root*[_], *Node*[_], *Point*[_]; the order on the program points is modelled by the predicate [_≤_];
- whether a node is marked by a certain nonterminal, is expressed by the corresponding predicate; e.g. the predicate *ProcDcl*[_] is exactly true for all ProcDcl–nodes, and the predicate *Dcl*[_] is true, iff a node is a ProcDcl–node, a TypeDcl–node, or a TypeRenm–node;
- the interpretation of the selection functions, like *DfId*(_), *TypeSpec*(_), *string*(_), is as follows: They are only defined for nodes that have exactly one son of the corresponding terminal or nonterminal type; if they are defined, they yield this son, otherwise *undef*.

Thus, the abstract syntax trees with program points are represented by a class of first order structures with the same signature. We call these structures *program models*. The class production for `ProgEntity` enriches each program model by the predicate *ProgEntity*[_] defined by

$$\forall N : \mathit{ProgEntity}[N] \leftrightarrow \mathit{ProcDcl}[N] \vee \mathit{TypeSpec}[N]$$

Finally, a binding is a pair that has a string as first and a program entity as second component. For bindings, we provide the constructor *Binding*(_,_) and the predicate *is_binding*[_] that test whether an element is a binding. For details, how such classes of enriched program models can be formally defined and implemented see [PH91].

3.2 Semantics for Visibility Clauses

As already mentioned, the visibility clauses enrich each program model by the visibility predicate *is_visible*[_,_]. Their semantics will be given by transforming them into a recursive definition for this predicate. This is done in four steps:

1. Define an auxiliary predicate *is_valid* that corresponds to the visibility clauses with the keyword `valid`, the so–called *v–clauses*.
2. Define an auxiliary predicate *is_hidden* that corresponds to the visibility clauses with the keyword `hidden`, the so–called *h–clauses*.
3. Express the predicate *is_visible* in terms of *is_valid* and *is_hidden*.
4. Expand the definition of *is_visible* by the auxiliary definitions.

We demonstrate this transformation by our example. For the three v–clauses, we get the following definition:

$$
\begin{array}{l}
is\_valid[\,BD,\ SP,\ EP\,] \ \Leftrightarrow_{def} \\
\quad is\_binding[BD] \ \wedge\ Point[SP] \ \wedge\ Point[EP] \\
\wedge\ (\ (\exists PD:\ ProcDcl[PD] \\
\qquad \wedge\ BD\ =\ Binding(PD.DfId.string, PD) \\
\qquad \wedge\ SP\ =\ after(PD.DfId) \\
\qquad \wedge\ EP\ =\ \textbf{if}\ is\_root[PD]\ \textbf{then}\ after(PD) \\
\qquad\qquad\qquad\qquad\qquad \textbf{else}\ after(PD.father.father.Block)\ \textbf{fi} \\
\quad) \\
\quad \vee\ (\exists TD:\ TypeDcl[TD] \\
\qquad \wedge\ BD\ =\ Binding(TD.DfId.string, TD.TypeSpec) \\
\qquad \wedge\ SP\ =\ after(TD)\ \wedge\ EP\ =\ after(TD.father.father.Block) \\
\quad) \\
\quad \vee\ (\exists TR:\ TypeRenm[TR] \\
\qquad \wedge\ (\exists TS:\ BD\ =\ Binding(TR.DfId.string, TS) \\
\qquad\quad \wedge\ TypeSpec[TS] \\
\qquad\quad \wedge\ is\_visible[\,Binding(TR.DfId.string, TS),\ before(TR.UsId)\,]\ ) \\
\qquad \wedge\ SP\ =\ after(TR)\ \wedge\ EP\ =\ after(TR.father.father.Block) \\
\quad) \\
\quad)
\end{array}
$$

In just the same way, we get the definition for the auxiliary predicate *is_hidden* :

$$
\begin{array}{l}
is\_hidden[\,BD,\ SP,\ EP\,] \ \Leftrightarrow_{def} \\
\quad is\_binding[BD] \ \wedge\ Point[SP] \ \wedge\ Point[EP] \\
\wedge\ \exists D:\ Dcl[D] \ \wedge\ \neg\, is\_root[D] \\
\qquad \wedge\ (\exists PE:\ BD\ =\ Binding(D.DfId.string, PE)\ ) \\
\qquad \wedge\ SP\ =\ before(D.father) \\
\qquad \wedge\ EP\ =\ after(D.father.father.Block)
\end{array}
$$

Then, the predicate *is_visible* is defined in terms of *is_valid* and *is_hidden* following exactly the informal description in section 2.3:

$$
\begin{array}{l}
is\_visible[\,BD,\ PP\,] \ \Leftrightarrow_{def} \\
\quad is\_binding[BD] \ \wedge\ Point[PP] \\
\wedge\ (\exists\, SP,\ EP:\ Point[SP] \ \wedge\ Point[EP] \\
\quad \wedge\ SP\ \leq\ PP \ \wedge\ PP\ \leq\ EP \\
\quad \wedge\ is\_valid[\,BD,\ SP,\ EP\,] \\
\quad \wedge\ (\nexists\, SPH,\ EPH:\ Point[SPH] \ \wedge\ Point[EPH] \\
\qquad \wedge\ (\ (SP < SPH \ \wedge\ EPH \leq EP) \ \vee\ (SPH \leq PP \ \wedge\ PP < EPH)\ ) \\
\qquad \wedge\ is\_hidden[\,BD,\ SPH,\ EPH\,] \\
\quad\ ) \\
\quad)
\end{array}
$$

Finally, we expand the occurrences of *is_valid* and *is_hidden* in the above equivalence. The result is a recursive definition of *is_visible*.

We will give a fixpoint–semantics for such definitions. We call a predicate occurrence in a formula *positive* (*negative*), if there is an even (odd) number of negations on the path from the predicate occurrence to the root in the abstract syntax tree of the formula. If we claim that all occurrences of *is_visible* in the v–clauses are positive and those in the h–clauses are negative, then all occurrences of *is_visible* in the defining equivalence are positive. This restriction is fulfilled by nearly all visibility rules of existing programming languages; a detailled discussion of this aspect and a semantics for visibility clauses violating this restriction can be found in [PH91]. With this restriction, we get the following fixpoint–definition.

Let $PROG$ be a program model with universe U and let us denote the righthandside of the defining equivalence for *is_visible* by $\alpha[BD, PP]$ (the visibility clauses must guarantee that $\alpha[BD, PP]$ has no free variables except BD and PP). Considering 2–ary predicates as subsets of U^2, we define a mapping

$$\tau \ : \ \mathcal{P}(\, U^2 \,) \rightarrow \mathcal{P}(\, U^2 \,)$$

as follows: Let $Q \subseteq U^2$ and $PROG_Q$ be the enrichment of $PROG$ by the predicate *is_visible* such that the interpretation of *is_visible* is given by Q. Then:

$$\tau(Q) \ =_{def} \ \{\, (v, w) \in U^2 \mid \alpha[v, w] \, \textit{is valid in} \, PROG_Q \} \ .$$

It is not hard to show that the positivity of $\alpha[BD, PP]$ with respect to *is_visible* implies the monotonicity of τ. As $(\mathcal{P}(U^2), \subseteq)$ is a complete lattice, the Knaster–Tarski theorem [Tar55] ensures that τ has a least fixpoint. As this holds for every program model, we can define the semantics of *is_visible* by the least fixpoint of the corresponding τ. (For a more formal treatment of fixpoint definitions in first–order logic and further references to related problems see e.g. [GS86].)

4 Application

As already pointed out, the visibility clauses are only part of a comprehensive method for syntax specification. In this section, we sketch the rest of this method and shortly discuss implementation aspects of visibility clauses.

Comprehensive Specification Framework A specification consists of five parts:

- the specification of the abstract syntax defining the program models;
- the specification of the concrete syntax defining the relation between program texts and program models;
- the visibility specification defining the meaning of used identifier occurrences;
- the type rules;
- further contextual constraints.

A visibility specification itself consists of three parts. The specification of the program entities as shown in section 2.1, the visibility clauses, and the specification of a visibility function *meaning* taking a UsId–node as argument and yielding the corresponding program entity; for languages without overloading, we would have specifications like

```
function  meaning ( UID: UsId ) ProgEntity:
  that  ProgEntity PE: is_visible[ Binding(UID.string, PE), before(UID) ]
```

Implementation Aspects In the discussion of Odersky's approach in section 1.1, we critized specification methods that do not even have implementations for realistic validation purposes. What is the advantage of the presented approach in this respect? The visibility clauses are a specialized specification construct for the definition of visibility rules. They are sufficiently powerful to precisely describe the visibility of common programming languages in a very natural way. On the other hand, the restrictions compared with arbitrary specification in first–order logic (as in [Uhl86] and [Ode89]) permit specialized and therefore more efficient implementations.

The visibility clauses give strong hints how to implement the corresponding part of a context checker:

- generate a matching mechanism that finds the constructs influencing the visibility according to the part after the keyword `vis`;
- provide a general and global datatype that manage the visibility information: For each identifier–entity–binding, we have to know, where it is valid and where it is hidden;
- a global function is then able to extract all program entities that are visible at a given program point.

Even if such generated symboltable mechanisms will probably be less efficient than hand-coded ones, they are certainly much better than general implementations of pure first–order logic. And there is another advantage of the proposed approach. After the identification process is correctly finished, we have a simple and formal representation of the identified abstract syntax tree: The function *meaning* is sufficient to get the declaration information; it can be simply implemented by pointers from the applications to the corresponding declarations.

5 Conclusions

A new, declarative method to formally specify visibility rules of programming languages was developed. This method is related to visibility descriptions in language reports and does not use complex data structures to pass information through the abstract syntax tree. We presented a logic–based fixpoint semantics for such specifications. As a side–effect, the paper reveals the intrinsic recursion hidden in visibility rules.

The presented method is only part of a project for the specification of context–dependent syntax. We view the context–dependent analysis as a partial mapping from abstract syntax trees to syntax DAG's containing arcs from used program entities to their declaration. We described this mapping only for programming languages where the

visibility rules are independent of the typing rules. For languages in which overloading resolution depends on user–defined types, we will get mutual recursive definitions for the predicates expressing the visibility and typing. Of course, this is no problem for the presented approach, because we only have to generalize the mapping τ, so that it can handle several predicates.

References

[ANS83] ANSI. *Pascal Computer Programming Language*, ansi/ieee 770 x3.97–1983 edition, 1983.

[End72] H. B. Enderton. *A Mathematical Introduction to Logic*. Academic Press, 1972.

[GS86] Y. Gurevich and S. Shelah. Fixed–point extensions of first–order logic. *Annals of pure and applied logic*, 32, 1986.

[Jon80] N. D. Jones, editor. *Semantics–Directed Compiler Generation*, volume 94 of *Lecture Notes in Computer Science*. Springer Verlag, 1980.

[Ode89] M. Odersky. *A New Approach to Formal Language Definition and its Application to Oberon*. PhD thesis, Swiss Federal Institute of Technology (ETH) Zürich, 1989. Diss. ETH No. 8938.

[PH91] A. Poetzsch-Heffter. *Formale Spezifikation kontextabhängiger Syntax von Programmiersprachen*. PhD thesis, Technische Universität München, July 1991.

[Rei83] S. Reiss. Generation of compiler symbol processing mechanisms from specifications. *ACM Transactions on Programming Languages and Systems*, 5(2), 1983.

[SH86] G. Snelting and W. Henhapl. Unification in many–sorted algebras as a device for incremental semantic analysis. *Conference Record of the Thirteenth ACM Symposium on Principles of Programming Languages*, 1986.

[Tar55] A. Tarski. A lattice–theoretical fixpoint theorem and its application. *Pacific Journal of Mathematics*, 5, 1955.

[Ua82] J. Uhl and andere. An Attribute Grammar for the Semantic Analysis of Ada. *Lecture Notes in Computer Science 139*, 1982.

[Uhl86] J. Uhl. *Spezifikation von Programmiersprachen und Übersetzern*, volume 161 of *GMD–Bericht*. R. Oldenbourg Verlag, 1986.

[Wat84] D. A. Watt. Contextual constraints. In B. Lorho, editor, *Methods and Tools for Compiler Construction*, pages 45–80. Cambridge University Press, 1984.