The PSG System: From Formal Language Definitions To Interactive Programming Environments

ROLF BAHLKE and GREGOR SNELTING
Technical University of Darmstadt

The PSG programming system generator developed at the Technical University of Darmstadt produces interactive, language-specific programming environments from formal language definitions. All language-dependent parts of the environment are generated from an entirely nonprocedural specification of the language's syntax, context conditions, and dynamic semantics. The generated environment consists of a language-based editor, supporting systematic program development by named program fragments, an interpreter, and a fragment library system. The major component of the environment is a full-screen editor, which allows both structure and text editing. In structure mode the editor guarantees prevention of both syntactic and semantic errors, whereas in textual mode it guarantees their immediate recognition. PSG editors employ a novel algorithm for incremental semantic analysis which is based on unification. The algorithm will immediately detect semantic errors even in incomplete program fragments. The dynamic semantics of the language are defined in denotational style using a functional language based on the lambda calculus. Program fragments are compiled to terms of the functional language which are executed by an interpreter. The PSG generator has been used to produce environments for Pascal, ALGOL 60, MODULA-2, and the formal language definition language itself.

Categories and Subject Descriptors: D.2.3 [**Software Engineering**]: Coding—*pretty printers, program editors*; D.2.6 [**Software Engineering**]: Programming Environments; D.3.1 [**Programming Languages**]: Formal Definitions and Theory—*semantics, syntax*; D.3.4 [**Programming Languages**]: Processors—*compilers, interpreters, parsing, translator writing systems and compiler generators*; F.3.2 [**Logics and Meanings of Programs**]: Semantics of Programming Languages—*denotational semantics*; F.4.2 [**Mathematical Logic and Formal Languages**]: Grammars and Other Rewriting Systems—*grammar types, parsing*; I.2.3 [**Artificial Intelligence**]: Deduction and Theorem Proving—*rule-based deduction*

General Terms: Algorithms, Design, Documentation, Languages, Theory, Verification

Additional Key Words and Phrases: Hybrid editor, unification-based incremental semantic analysis

1. INTRODUCTION

The Programming System Generator, PSG, developed at the Technical University of Darmstadt generates language-dependent interactive programming environments from formal language definitions. From a formal definition of a language's syntax, context conditions, denotational semantics, and additional

Preliminary versions of parts of this paper appeared in the *Proceedings of the ACM SIGPLAN 85 Symposium on Language Issues in Programming Environments*, June 1985 [7] and in the *Conference Record of the 13th ACM Symposium on Principles of Programming Languages*, Jan. 1986 [39].
This work was supported in part by the Deutsche Forschungsgemeinschaft under grants He 1170/2-3 and He 1170/3-1.
Authors' address: Technische Hochschule Darmstadt, Fachbereich Informatik, Magdalenenstr. 11c, D-6100 Darmstadt, West Germany.

information it produces an integrated software development environment. One of the major components of a PSG environment is a powerful hybrid editor which allows structure-oriented editing as well as text editing. In structure mode, the editor guarantees *prevention* of both syntactic and semantic errors, whereas in textual mode it guarantees their *immediate recognition.* The editor is generated from the language's syntax and context conditions. Furthermore, a PSG environment includes an interpreter which is generated from the language's denotational semantics. A language-independent library system is also part of the PSG environment.

The basic units for editing and executing are called *fragments*, which are internally represented as abstract syntax trees. A fragment is an arbitrary part of a program, for example, a statement, a procedure declaration, or an entire program. Fragments may be incomplete, that is, subcomponents may be missing. Missing subcomponents are called *placeholders*. Bottom-up system development is provided by combining fragments, while the fragments themselves are constructed top-down.

The editor supports two editing modes, which are fully integrated and may be mixed freely by the user. In *textual mode*, the editor behaves like a normal screen-oriented text editor with the usual facilities to enter, modify, delete, and search text. Incremental syntactic and semantic analyses are invoked by keystroke. If the input was error-free, the text will be pretty-printed according to the formatting rules of the language definition, and editing may proceed. If any syntactic or semantic errors are detected, an error message will be displayed by a menu-driven error-recovery routine. The earliest possible detection of both syntactic and semantic errors is guaranteed: As soon as a fragment cannot be embedded into a syntactically and semantically correct program, it will be classified as erroneous. For semantic errors this works even if declarations of, for instance, variables or types are still missing or incomplete. In *structured mode*, programs are developed in menu-driven refinement or modification steps. The menus are generated according to the abstract syntax of the language. The usual structure-oriented commands are offered to the user, such as refinement of a structure, selection from alternatives of a syntactic class, modification, insertion and deletion of substructures, zooming of substructures, copying of substructures, and so on. However, the menus are *filtered dynamically* by the context analysis such that only those menu items producing syntactically and semantically correct refinements or modifications after their selection will be offered to the user. Thus, in structured input mode, neither syntactic nor semantic errors can occur.

Although PSG editors guarantee the immediate detection of errors, users are not forced to correct errors on the spot. The editor is *fault-tolerant* and accepts syntactically or semantically incorrect fragments; erroneous parts of the fragment are displayed in a special font. In addition users may retrieve the context information that has already been derived. For example, they might ask the system which variables are already declared, which are still undeclared, what possible types the undeclared variables may possess and so on.[1] The user interface of the PSG editors is described in detail in [8].

[1] In our philosophy, declaration before use is not required. An undeclared variable is considered a semantic error as soon as the last placeholder offering the possibility of declaring that variable has been deleted.

Like all other system components, the interpreter is able to handle arbitrary incomplete fragments. As long as control flow in the interpreted fragment does not touch any syntactically incomplete structure, the fragment can be interpreted without difficulty. If control flow encounters a placeholder, the editor will be invoked, asking the user to enter the missing parts of the fragment. Additionally, the language definer may force the interpreter to ask the user for values of uninitialized variables or missing expressions, for instance.

A language-independent fragment library system, which stores fragments as abstract syntax trees, is also part of a generated environment. Reading, writing, and rewriting of fragments is performed automatically by the environment if necessary. Deletion of fragments requires an explicit user command. The PSG editor offers the facility of redirecting input to external text files. Furthermore, fragments may be written in pretty-printed style onto external files.

2. THE PSG LANGUAGE DEFINITION

One of the most important goals in the development of PSG was the definition of a *formal language definition language*, covering the whole spectrum of a language's syntax, context conditions, and dynamic semantics. Thus the language definer working with PSG is offered a formal, nonprocedural definition language. This is in striking contrast to most existing environment generators, which frequently support only the formal definition of the syntactic aspects of a language. It is common practice to use an abstract syntax for the generation of structure-oriented editors, together with a format syntax describing unparsing schemes for abstract syntax trees. If the editor incorporates a facility for processing textual input, a context-free grammar serves as input to a parser generator (actually, many systems use the YACC [21] parser generator). For the description of the static and dynamic semantics of a language, most environment generators are based either on the attribute grammar approach or on the action routines approach. In both cases, attribute evaluation functions or action routines have to be coded in a procedural programming language. In these approaches no distinction is usually made between static and dynamic semantics, both have to be defined using the same metalanguage.

The ALOE generator of the GANDALF system [24, 25] uses action routines written in GC, a dialect of C. To integrate an execution tool, the language definer must either produce a new code generator or modify an existing one. However, V. Ambriola and C. Montangero [3] report on the possibility of integrating an automatically derived interpreter into a GANDALF environment. A new model for action routines [14] has recently been developed. The PECAN system [30] uses a statement-oriented language to code action routines; separate specifications detailing the uses of symbols, data types, and expressions of the language also have to be given. Using the Synthesizer Generator [33], which is based on attribute grammars, the language definer has to write a specification in the synthesizer specification language, SSL. The first version of SSL [31] used the language C for coding attribute evaluation functions, the current version of the generator [34] supports the definition of semantic equations and functions within SSL. The MENTOR specification language, METAL [22], supports the generation of environments for syntax-directed editing and manipulation of

programs; a formalism to specify type-checking will be integrated in a future version [12, 13].

A PSG language definition consists of three major parts, structured as follows:

(1) *Syntax of the language*
  1.1 Lexical structure
  1.2 Abstract syntax
  1.3 Concrete syntax
  1.4 Format syntax
  1.5 Titles and menus

(2) *Context conditions of the language*
  2.1 Scope and visibility rules
  2.2 Data attribute grammar
  2.3 Format syntax of the data attributes
  2.4 Basic context relations

(3) *Dynamic semantics of the language*
  3.1 Domain definitions
  3.2 Auxiliary functions
  3.3 Meaning of executable fragments
  3.4 Meaning functions

As the core of any language definition, the syntax part is mandatory, whereas the others are optional. If the context conditions are missing, only a context-free editor will be generated, if the dynamic semantics definition is missing, an environment without an execution tool will be generated.

3. THE DEFINITION OF THE SYNTAX

The syntax definition part starts with the definition of the *lexical structure* of the language. The language definer has to specify the reserved words and the delimiters (special symbols):[2]

```
while → 'WHILE';
do    → 'DO';
if    → 'IF';
then  → 'THEN';
else  → 'ELSE';

sem   → ' ; ';
ass   → ' := ';
slp   → ' [ ';
srp   → ' ] ';
dot   → ' . ';
lp    → ' ( ';
rp    → ' ) ';
...
```

[2] In the following, all language definition examples refer to the definition of a Pascal subset called MIDIPAS [5], unless otherwise stated.

The classes of identifiers (Id) and constants (Int, Real, String) are predefined in PSG. Language-dependent tables are generated from the lexical definition to be used by the scanner and the unparser.

The abstract syntax defines the structure of the abstract syntax trees, which serve as the internal representation of any fragment in a PSG environment. Abstract syntax is specified by a collection of class and constructor rules. Sets of syntactic alternatives are described by class rules:

```
CLASS type     = integer, real, bool, arraytype, recordtype, typeid;
CLASS stat     = assign, withstat, whilestat, forstat, ifthen,
                 callstat, repeatstat, readlnstat, readstat,
                 writelnstat, writestat, compound, casestat;
CLASS var      = Id, indref, recref;
CLASS expr     = var, constant, add, sub, mult, idiv, rdiv, not,
                 andexpr, orexpr, equ, lt, gt;
CLASS constant = Int, Real, true, false, Id;
...
```

Constructor rules are either node or list rules:

```
NODE arraytype :: indrangelist type;
LIST indrangelist = indrange+;
NODE indrange :: constant constant;
...
NODE indref :: var exprlist;
NODE recref :: var Id;
...
NODE assign :: var expr;
NODE compound :: statlist;
LIST statlist = stat+;
NODE ifthen :: expr stat [stat];
NODE callstat :: Id [actplist];
...
```

Node rules define syntactic entities with a fixed number of subcomponents, which may be of different syntactic types. Subcomponents enclosed in '[' and ']' are optional (e.g., the parameterlist of a procedure call may be omitted). List rules define syntactic entities with a variable number of subcomponents of the same syntactic type.

Missing subcomponents of a node are called placeholders; they represent pending refinements. Placeholders for sublists may be moved, deleted, or inserted freely within a list.

The structure-oriented commands and menus offered to the user are generated according to the abstract syntax. For example, a menu of refinement or modification possibilities is associated with each placeholder. However, this menu is dynamically filtered with respect to context conditions (see below).

The *concrete syntax* defines the transformation from the textual representation of fragments (i.e., character strings) to abstract syntax trees. It is based on a context-free grammar augmented by transformation rules describing the

construction of the abstract syntax tree. Thus the concrete syntax is actually a string-to-tree transformation grammar:

```
statlist ::= LIST stat + − sem;
stat :: = NODE var, ass, expr ⇒ assign
    |   NODE while, expr, do, stat ⇒ whilestat
    |   NODE if, expr, then, stat, elseopt ⇒ ifthen
    |   NODE repeat, statlist, until, expr ⇒ repeatstat
    ...;
elseopt ::= [else, stat];
var ::= Id, var1;
var1 ::= var2, var1
    |   EMPTY;
var2 ::= UPDATENODE slp, exprlist, srp ⇒ indref
    |   UPDATENODE dot, Id ⇒ recref;
...
```

The list and node rules of concrete syntax resemble the corresponding rules of abstract syntax. List rules may specify a separator symbol (a semicolon separates the elements of a statement list); the right-hand side of node rules contains nonterminal as well as terminal symbols. Since top-down parsing is used in the PSG system, concrete syntax is restricted to LL(1) grammars. The above example illustrates left-factorization in order to avoid LL(1) conflicts in the definition of var. The usual left-recursive definition of variables is not top-down parsable:

```
var ::= Id
    |   NODE var, slp, exprlist, srp ⇒ indref
    |   NODE var, dot, Id ⇒ recref;
```

It must be transformed by left-factorization of the nonterminal "var" into LL(1)-form, which involves the transformation of node rules to so-called updatenode rules. As usual, parsing proceeds from left to right, construction of the abstract syntax tree is done parallel to parsing. The parser should be able to process any incomplete input entered in textual mode. So the parser has to accept arbitrary valid prefixes of any input conforming to the syntactic category of a given placeholder and to construct the corresponding (possibly incomplete) abstract syntax tree. Thus, tree construction has to be done top-down, since bottom-up tree construction will not lead to satisfactory and unambiguous results in connection with incomplete input texts. Building abstract syntax trees bottom-up, tree nodes are constructed after the recognition of the complete right-hand side of a nonterminal (i.e., when the right-hand side is reduced to the nonterminal). Building trees top-down, nodes are constructed before the processing of the right-hand side (i.e., directly after the prediction of the right-hand side). The tree construction process specified by the left-factorized nonterminal "var" in the above example is illustrated by the following: Consider '*a.b* :=' to be an input for a statement placeholder. Starting with the statement rule and recognizing the identifier '*a*', an assignment node is constructed, and a pointer to its first subcomponent (a placeholder for variable) is passed to the nonterminal "var". The right-hand side of the rule for var constructs an identifier node (with value

'*a*') by refining the placeholder for variable. A pointer to the identifier node is passed to the nontermianl "var1" (note that the abstract syntax tree for the statement so far constructed is '*a* := {expr}'). After prediction of the first alternative for "var1", the same pointer is passed to the nonterminal "var2". Using the lookahead '.', the second rule for "var2" is predicted and the up-to-now constructed tree is changed according to the updatenode rule. The current tree pointer refers to the identifier node; this node is replaced in the abstract syntax tree by a record-reference node whose first subcomponent is the just-replaced identifier node. After the change of the abstract syntax tree invoked by the updatenode rule, the abstract syntax tree looks as follows: '*a*. {Id} := {expr}'.

If any syntax errors are detected in the input text during parsing, a recovery routine will compute a menu comprising all *local correction* possibilities, which is then presented to the user. Alternatively, users may switch to *global recovery* mode, where they may correct the input by editing in textual mode or simply accept the correct part of the input.

The *format syntax* is a tree-to-string transformation grammar used to construct the external textual representation of an abstract tree:

```
arraytype ⇒ array slp indrangelist srp of type[2];
assign ⇒ ! var ass expr;
repeatstat ⇒ ! repeat statlist[2] ! until expr;
ifthen ⇒ ! if expr then stat[2] (stat[2] → ! else, );
equ ⇒ CLASS (expr=equ,lt,gt → lp,rp) eq CLASS (expr=equ,lt,gt → lp,rp);
...
```

Pretty-printing information, like insertion of newlines ('!') and indentation of substructures ('[. .]'), is part of the format definition. Conditional formatting, depending either on the existence of optional substructures or on the type of substructure, is supported. The first kind is illustrated by the else-part of the if-then-statement (if it exists, it is prefixed with a newline and the keyword 'ELSE'), the second kind by the equal expression. Here the conditional format rule is used to put left and right parentheses around the node's subexpressions if and only if they are relational expressions (note that parentheses are discarded during parsing and that operator precedences are reflected by the structure of the abstract syntax tree).

In the last part of the syntax definition, titles and menu texts have to be specified; these are used to generate the textual representation of placeholders and menu items.

4. INCREMENTAL SEMANTIC ANALYSIS WITHIN PSG EDITORS

The context analysis of PSG has been of special interest because classical concepts like attributed grammars do not work very well if arbitrary incomplete fragments have to be analyzed. Consider the following incomplete Pascal procedure fragment:

```
PROCEDURE proc1 (VAR par1: type1; par2: type2);
  BEGIN
    par1[par1[par2 + 5]] := {Expression};
    {Statementlist}
  END;
```

Although the types of 'par1' and 'par2' are undeclared within that fragment, the context analysis must derive immediately that 'par2' has type "integer" (or a subrange thereof), that 'par1' is a one-dimensional array with index and component type "integer", and that the still missing right-hand side of the assignment must also be compatible with "integer". If a user refines the missing expression of the assignment to the real constant '3.14', a semantic error must immediately be reported. In addition, the menu for the right-hand-side placeholder must be filtered in such a way that the menu items for all noninteger expressions will not be displayed. In MIDIPAS, the unfiltered menu for expressions consists of the following items:

'Variable'	'Constant'	'Addition'	
'Subtraction'	'Multiplication'	'Integer Division'	
'Real-Division'	'Not'	'And'	
'Or'	'Equal'	'Less'	'Greater'

The actual menu for the assignment's right-hand-side expression will be reduced by the context analysis to the following items:

'Variable'	'Constant'	'Addition'
'Subtraction'	'Multiplication'	'Integer Division'

After selecting the second item, only the two items 'Integer-Constant' and 'Identifier' will appear in the menu for constants, as opposed to the complete menu:

'Integer-Constant'	'Real-Constant'	
'TRUE'	'FALSE'	'Identifier'

Considering this example, the context analysis must fulfill several requirements in our setting:

—The context analysis must be able to analyze *arbitrary incomplete fragments.*
—The context analysis must guarantee the *immediate detection* of semantic errors even in incomplete fragments.
—For efficient use in an interactive programming environment, the context analysis must work in an *incremental manner.*
—Since PSG is a generating system, the context analysis must be *generated* from a formal specification of the language's context conditions.

Since the classical methods first collect the type information of variables from the declarations and then use this information to type-check expressions, these methods do not work in the above example because the declarations of 'type1' and 'type2' are not part of the fragment. Even incremental attribute-evaluation algorithms [20, 31, 32] are unable to derive type information in incomplete program fragments if attribute grammars are defined as usual (one can, however, do type inference with attribute grammars, see below).

4.1 The Concept of Context Relations

The concept of context relations has been developed to overcome difficulties with the more classical approaches. The basic idea is as follows: A fragment is correct if it is a correct program or *if it can be embedded into a correct program.* As usual, we want to use attributes for purposes of context analysis. However, in incomplete

fragments, a unique assignment of attribute values to tree nodes does not generally exist, since important information (e.g., variable declarations) may be missing.

Because of this defect we explicitly pass over from attribute values to *sets of still-possible attribute values*. The basic idea is as follows: An arbitrary 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 database 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, because 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.

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

$$\{t\colon N \to A\}.$$

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 that 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 database theory, see [1]). If a placeholder X in a fragment F is replaced by a fragment G, thus giving a new fragment H, we have

$$\mathrm{CR}(H) = \mathrm{CR}(G) \bowtie \mathrm{CR}(F).$$

This property is generally valid for all languages that do not allow the definition of overloaded or polymorphic objects. Overloaded built-in constants or functions do not destroy the property, but for user-defined objects we have to assume that they have at most one final type or attribute. The examples given later will clarify this fact.

There must of course be some relations to start with. These so-called *basic relations* have to be specified by the language definer for all terminals and all constructors of the abstract syntax of the language. Once these basic relations have been defined, all fragments may be analyzed by joining the basic relations of their components. Examples of basic relations are given later.

4.2 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), where the structure of the attributes of the language is defined. The DAG has to be specified by the language definer. For MIDIPAS, typical DAG rules look like this:

```
CLASS attribute = expr_attr, proc_attr, type, parmlist_attr, ...;
NODE expr_attr :: type class cconstval;
CLASS type = simple_type, array_type, record_type, ...;
CLASS simple_type = arithmetic, ordinal;
CLASS arithmetic = integer, real;
CLASS ordinal = integer, boolean;
NODE array_type :: ordinal type;
CLASS class = cprog, cproc, ctype, ccomp;
CLASS ccomp = cvars, cconst, cexpr;
CLASS cvars = controlvar, non_controlvar;
CLASS non_controlvar = cvariable, cppdescr, csel;
...
```

This MIDIPAS example specifies that an attribute of a syntactic object is either an expression attribute, a procedure attribute, a type attribute, or a parameterlist attribute. An expression attribute has three subcomponents, namely the type of the expression, its object-class, and a constant value (used only for constant expressions). A type may be a simple type, an array type, or a record type; a simple type is either arithmetic or ordinal, where arithmetic is integer or real and ordinal is integer or boolean. An array type attribute has two subcomponents: the ordinal index type and the component type of the array. Possible object-classes are programs, procedures, types, and computational objects, where the latter comprises variables, constants, or expressions; a variable may be either a control variable or a noncontrol variable. A noncontrol variable is either a variable, a procedure parameter, or a field selector.

Since attribute classes may contain subclasses, a DAG also includes the concept of a subtype or *inheritance* 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. Note that classes need not be disjoint: "integer" is an arithmetic type as well as an ordinal type.

A DAG describes a many-sorted free algebra with subsorts (that is, an *order-sorted algebra* [16]) 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 considered as nonnullary function symbols with arity according to the DAG. 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 $x, y \in \text{AF(DAG)}$ we write $x \stackrel{*}{\Rightarrow} y$ iff y can be derived from x by substituting an attribute form of the correct sort for a nonterminal leaf. 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 that can be derived from it.

As usual, we add variables: The algebra freely generated by the terminals, class symbols, and an infinite 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 $\text{sort}(v)$, and for $x \in \text{AFV(DAG)}$ we denote the variables in x by $\text{vars}(x)$.

An almost identical scheme has recently been proposed by Ait-Kaci and Nasr in order to extend PROLOG with inheritance concepts [2]. They propose terms of an order-sorted algebra as the basic PROLOG structure instead of the standard type-free terms; this can shorten the resolution process considerably.

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. In addition, each mapping has an environment attached, which gives values for the instantiated variables of tuple components. Of course, these values must have correct sorts with respect to the sort structure induced by the DAG. The set of all attribute-form relations is denoted by AFR. An attribute form relation $r \in \text{AFR}$ represents a possibly infinite context relation $R[r] \in \text{CR}$ as follows:

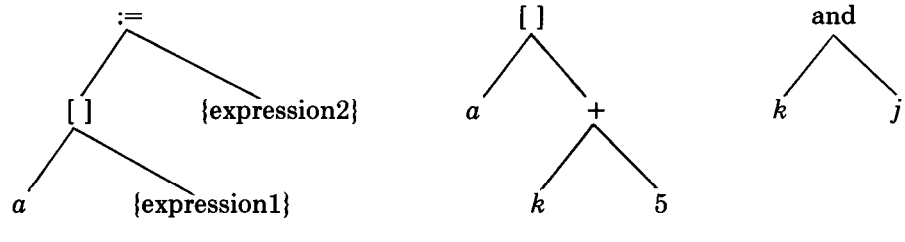
$t \in R[r]$ iff there is $(t', e') \in r$ and there is a mapping
e: $\text{vars}(t') \rightarrow \text{A(DAG)}$
such that for all $s \in \mathbf{dom}(t)$
$e^*(t'(s)) \stackrel{*}{\Rightarrow} t(s)$
and for all $v \in \text{vars}(t')$: $e'(v) \stackrel{*}{\Rightarrow} e(v)$

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

At this point, we give some examples. Consider the following MIDIPAS 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 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 that 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 must either 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*' and the index type of '*a*' must be compatible with "integer". Note that even though the addition in MIDIPAS 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 fragment nodes by attribute-form relations. For readability, we ignore the object_class and constval 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 that possess an attribute. Tuple components are attribute forms with variables. The first relation has two tuples. The first tuple contains the variables "ORDINAL" and "TYPE", which are (similar to PROLOG) written in uppercase letters. For simplicity, the names of the variables also indicate their sort (if necessary, indices will be used to distinguish several variables of the same 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. 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 accidentally 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 most important part of the context condition definition is the specification of the basic relations, which must be specified for each terminal and each node rule of the abstract syntax. A *basic relation* is simply an attribute-form relation, which defines a (possibly infinite) set of attribute assignments to the components of a node rule or a terminal, respectively. Thus the basic relations describe the context conditions local to a given abstract syntax rule. The basic relation for an assignment consists of two tuples, which use the variable "TYPE":

```
assign:
        NIL
        MK-expr_attr(TYPE, non_controlvar, cconstval)
        MK-expr_attr(TYPE, ccomp, cconstval)
      | NIL
        MK-expr_attr(real, non_controlvar, cconstval)
        MK-expr_attr(integer, ccomp, cconstval);
```

which says that in an assignment, either

—the left-hand side is a noncontrol variable of a certain "TYPE", and the right-hand side is an expression of the same "TYPE"; or
—the left-hand side is a real variable, and the right-hand side is an integer expression.

The assignment operator itself has no attribute; this is indicated by the null attribute "NIL". In contrast to the notation used above, this example shows how relations within a PSG specification actually look: each line contains one tuple component, the position of tuple components corresponds to the position of symbols in the abstract syntax rule referred to, and tuples are separated by '|'. The basic relation for an integer constant

```
Int:
    MK-expr_attr(integer, cconst, LEXID(Int));
```

says that a syntactic integer constant has type integer, is a constant, and has a constant value copied from the abstract syntax tree (this is specified by the built-in LEXID function).

The tuples occuring in an attribute-form relation can be seen as special terms in our order-sorted attribute algebra: for $i \in N$, introduce new function symbols tup_i with arity i. A tuple can then be written as a term $\text{tup}_i\ (x_1, \ldots, x_i)$, where the x_i's are the tuple components, and the argument position of an x_j corresponds to the column label of an attribute-form relation column.

4.3 Unification as a Device for Modeling the Join

It is necessary to construct an operation for attribute-form relations that exactly represents the join. This operation is *unification* in our many-sorted algebra with subsorts. Unification in order-sorted algebras works similarly to the classical Robinson unification [37]. However, as we have subsorts and nondisjoint sorts, in order to unify two variables of different sorts, it is necessary to find a sort that 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

$$(\text{AF(DAG)}; \dot{\Rightarrow}) \text{ is an upper semilattice.}$$

Thus 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". Of course, the unification of a variable of sort "array_type" and another variable of sort "real" fails, as "real" and "array_type" are disjoint sorts.

It is not a restriction that the type structure has to be a semilattice. Every partial order can be embedded into a semilattice, and if the type system of the language to be defined does not have a partial ordering at all, one can use the flat semilattice.

The unification algorithm is sketched below.

```
function unify (t1, t2: AFV(DAG); env: ENV): AFV(DAG) × ENV;
  {Unifies two terms t1, t2 of the order-sorted attribute algebra in a given environment
   env; an environment is a mapping of variables of t1 and t2 to attribute terms. Output
   is a result term as well as a new environment. If unification fails, the result is nil. }
begin
  if t1 is a constant then
    if t2 is a constant then
      begin
        s := sort(t1) ∧ sort(t2);
        if s = nil then
          unify := nil
        else
          unify := ("a new constant of sort s", env)
      end
    else if t2 is a variable then
      unify := unify(t2, t1, env)
    else if t1 and t2 are compound terms with identical functors,
            t1 = f(t11, ..., t1n),
            t2 = f(t21, ..., t2n) then
      begin
        i := 1;
        repeat
          (ai, newenv) := unify(t1i, t2i, env);
          i := i + 1;
          env := newenv
        until i = n or ai = nil;
        if ai = nil then
          unify := nil
        else
          unify := (f(a1, ..., an), newenv)
      end
  else if t1 is an uninstantiated variable and not occurs (t1, t2) then
    begin
      s := sort(t1) ∧ sort(t2);
      if s = nil then
        unify := nil
      else
        unify := (t1, env+[t1 → t2])
    end
  else if t1 is an instantiated variable then
    unify := unify(env(t1), t2, env)
  else if t2 is a Variable then
    unify := unify(t2, t1, env)
  else
    unify := nil
end;
```

Note that from a theoretical point of view it is not essential to include nondisjoint DAG classes. The special case "classes must be disjoint" is theoretically sufficient and leads to a subsort ordering which has tree structure rather than being 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 the above, including nondisjoint DAG classes is essential for performance.

THEOREM. *If the DAG induces an upper semilattice, tuple-wise unification of attribute-form relations represents the natural join exactly*:

$$R[\{(t, e) \neq \text{NIL} \mid \textit{there are } (t', e') \in r1,\ (t'', e'') \in r2,\ (t, e) = \textit{unify}(t', t'', e' + e'')\}] = R[r1] \bowtie R[r2]$$

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

PROOF. See [40].

Examples. (a) We compose fragments (1) and (2), thus obtaining the fragment:

$$a[a[k + 5]] := \{\text{expression}\}$$

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 the scope and visibility rules of the language in question must be obeyed when determining which columns match (see below).

Unifying the attributes "array_type(ORDINAL, TYPE)" and "array_type(integer, TYPE)" results in a new sort for "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 for 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 unifiable, (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 the still missing right-hand side of the assignment, as well as the 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:

$$a[a[k + 5]] := k \text{ AND } j.$$

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:

$$\frac{a \mid [\ ] \mid [\ ] \mid k \mid + \mid 5 \mid \text{and} \mid j}{\mid \quad \mid \quad \mid \quad \mid \quad \mid \quad \mid \quad \mid},$$

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 that did not match are known, it is also possible to locate semantic errors exactly and to produce appropriate error messages.

In general, the scope and visibility rules of the language in question have to be obeyed when determining which columns match. The language definer has therefore to specify these rules for identifiers as part of the context-condition specifications. This information is used to determine whether the different occurrences of the lexically same identifier in a fragment denote the same *abstract* identifier. If so, their columns in two context relations match, and the attribute terms in these columns have to be unified.

In the current implementation, the scope analysis specification language is rather simple-minded. It offers various built-in concepts to the language definer, but is not a general mechanism. The concepts that are supported at the moment are the block concept and a concept of named-scopes (record concept). A more general scheme is currently under development.

The process of scope and visibility analysis must be done before any relations are joined. This analysis computes an equivalence relation on the nodes of the abstract syntax tree. Since this is done before the computation of attributes actually starts (the join needs the *column-matching* information first), the analysis of user-defined overloading is not possible within the current system. In addition to scope rules, overloading resolution requires type information too, which, however, has not been computed at the time of scope analysis. It seems that for overloading resolution, declarations are actually required. If overloaded procedures are declared within a fragment, one can build up a basic relation for each overloaded object which contains one tuple for each overloaded variant, and then use this relation instead of the standard basic relation for procedure calls. Thus context relations can also be used as a natural tool for overloading resolution. An experimental implementation of our unification-based semantic analysis using the Synthesizer Generator uses this approach [35].

4.4 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 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 nonempty equational theory. The unification algorithm in our example must know that $3 - 3 = 0$ and $3 \leq 0 =$ false. Arbitrarily complicated examples such as this may be constructed. However, in his well-known paper [29], Plotkin showed that finite most-general unifiers for algebras with nonempty equational theory in general do not exist. There are equational theories where finite unifiers exist; but we do not want to force the language definer to look for a correct unification algorithm for his specific langauge.

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 correctly with almost any input in the sense of open problem 8 in [38]. 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 instance, can determine that $3 - 3 = 0$. Thus, if we assume that constants are described by their type and value:

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

where Int_value, and so on, are assumed to be predefined DAG classes. The basic relation for constant addition in MODULA-2 might look as follows:

```
const_add:
    MK-const_attr(ARITHMETIC, VALUE1 + VALUE2)
    MK-const_attr(ARITHMETIC, VALUE1)
    MK-const_attr(ARITHMETIC, VALUE2);
```

During analysis, unification and evaluation are *interleaved*. The system keeps track of unevaluated expressions. Once the necessary arguments of as yet unevaluated expressions are known (this might be a consequence of unification), the expressions are evaluated at once. This is known as *data-driven evaluation*: Unevaluated expressions are waiting as demons; they are always evaluated as soon as possible. Thus, unification calls evaluation if possible; however, the results of evaluation must again be considered for unification: evaluation calls unification if necessary. This concept does not work in every case: there may be

unevaluated expressions that in case of evaluation would cause subsequent unifications to fail—they, however, never get evaluated. As an example consider:

```
CONST a = {constant expression};
      b = a − 3;
VAR x: ARRAY [a .. b] OF {type}
```

This fragment is already incorrect, but that will not be detected until the '{constant expression}' placeholder is replaced by a constant or complete constant expression. 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 this approach to be an alternative to narrowing algorithms [36].

The interpreter used in the current implementation is able to handle arithmetic, relational, and boolean operators as well as operations on lists. The concept itself, however, can be used with a full-scale interpreter for a functional language. Therefore, in addition to the unification mechanism, arbitrarily complicated functions may be evaluated during the analysis.

4.5 The Incremental Analysis Algorithms

We have seen that unification, interleaved 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 attributes 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 analyze 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 reanalysis 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 subordinate to that node. In a local relation attached to node X, it is not necessary to include columns for attributes of objects that, according to the scope rules of the language, are not visible at X. For example, consider the MIDIPAS fragment:

```
(1) PROCEDURE p (x: t);
(2)   VAR a: t2;
(3)   PROCEDURE q;
(4)     VAR b: REAL;
(5)       BEGIN
(6)         b := 5
          END;
(7) BEGIN
(8)   a[x] := {expression};
      {statements}
    END;
```

If we assume that each statement, declaration, procedure, and program has a local relation attached, the local relations for this fragment have columns for the

following syntactic objects:

(1) *p*, *x*, *t*, *a*, *t*2, *q*, {expression}
(2) *a*, *t*2
(3) *q*, *b*
(4) *b*, REAL
(5) *b*
(6) *b*, 5
(7) *a*, *x*, {expression}
(8) *a*, [ ], *x*, {expression}

In order to see that no more information is needed, consider the local variable '*b*': since its declaration is complete, '*b*' cannot influence relations outside '*q*'. Semantic information about '*q*' is part of the attributes of '*q*', nothing else is needed outside '*q*'. As a consequence, the complete bottom-up analysis of a fragment can use a variant of the join, which does not copy all columns of the second relation and is therefore more efficient: given a position in the tree, the join will determine via scope analysis which columns of its second argument are actually needed at that position. The join operation is sketched below:

```
function join(r1, r2: AFR; x: tree_node): AFR;
  {Joins two context relations by unifying corresponding tuple components. x is used to
   check which columns of r2 have to be included in the result}
  begin
    r := empty_relation;
    for all tuples t1 in r1 do
      for all tuples t2 in r2 do
        begin
          e := env(t1) + env(t2);
          t := empty_tuple;
          for all components c1 of t1 do
            for all components c2 of t2 do
              if c1 and c2 have to be matched then
                begin
                  (c, e1) := unify(c1, c2, e);
                  e := e1;
                  if c = nil then
                    t := nil
                  else
                    add c as a tuple component to t
                end
              else
                begin
                  add c1 as a tuple component to t;
                  if c2 is needed at x then
                    add c2 as a tuple component to t
                end;
          if t ≠ nil then
            add tuple t with environment e to relation r
        end;
    join := r
  end;
```

The data-driven evaluation of expressions is not shown in this algorithm.

After an editing step there is usually only a small number of local relations to be updated, which is far more efficient than to update a single global relation.

Only local relations attached to fragment nodes on the path from the modified subtree to the fragment root must be considered, since local relations describe local semantic information and are therefore independent from siblings or parents of the tree node they are attached to. After a subtree insertion these relations have to be joined with the relation of the new subtree (which has to be analyzed 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 a local relation on the path from the modification point to the fragment root does not change, updating of local relations may be aborted. A sketch of the analysis algorithms is given below:

```
function complete_analysis(t: fragment): AFR;
  {Complete bottom_up analysis of a fragment}
  if t is a terminal leaf then
    complete_analysis := basicrelation(t)
  else
    begin
      r := basicrelation(t);
      for all sons s of t, s not a placeholder do
        r := join(r, complete_analysis(s), t);
      if t has a local relation then
        relation(t) := r;
      complete_analysis := r
    end;

function partial_analysis(t: fragment): AFR;
  {Analyzes t, using local relations that might be attached to nodes of t}
  if t is a terminal leaf then
    partial_analysis := basicrelation(t)
  else
    begin
      r := basicrelation(t);
      for all sons s of t, s not a placeholder do
        if s has a local relation attached then
          r := join(r, relation(s), t)
        else
          r := join(r, partial_analysis(s), t);
      partial_analysis := r
    end;

procedure analyze_refinement(t: fragment; x: tree_node; n: fragment);
  {Analysis of replacing placeholder x in fragment f by fragment n}
  begin
    r := complete_analysis(n);
    s := x;
    repeat
      s := parent(s);
      if s has a local relation attached then
        relation(s) := join(relation(s), r, s);
    until relation(s) did not change or s = root of t
  end;
```

```
procedure analyze_deletion(t: fragment; x: tree_node);
  {Analysis for deleting the subtree starting at x from fragment t}
  begin
    s := x;
    repeat
      s := parent(s);
      if s has a local relation attached then
        relation(s) := partial_analysis(s)
    until relation(s) did not change or s = root of t
  end;
```

The complete analysis of a fragment of size n requires $O(n)$ unifications. In an incremental setting, however, where local relations have to be computed and updated, 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. On a SIEMENS 7551 (a machine comparable to a VAX 780), the complete analysis of a 90-line MIDIPAS program needs 1.4 CPU seconds, whereas deletion of a statement requires 0.4 seconds and reinsertion of that statement needs 0.1 seconds. These results, however, are correct only if the number of tuples within relations is bounded. This is usually the case; one can, however, give examples where combinatorial explosion occurs. In such a situation the method becomes unusable. It depends on the language definition whether the number of tuples in local relations is always bounded or not.

The incremental analysis algorithms can be further improved by splitting up local relations into several smaller relations. The idea is as follows: A relation needs columns only for those tree nodes whose attributes are mutually dependent owing to variables within tuples. Thus a local relation may be split up in such a way that each subrelation has columns for mutually dependent nodes only; attributes in different subrelations are independent. After a modification of the tree, only the affected subrelations of local relations have to be considered, which reduces the complexity of incremental analysis. In order to see how combinatorial explosion can be avoided, consider the fragment:

```
BEGIN
  a := b;
  c := d;
  e := f
END;
```

Its local relation has 6 columns, 8 tuples, and 48 attributes. However, since the attributes of $\{a, b\}$, $\{c, d\}$, and $\{e, f\}$ are independent, it can be split up into three subrelations with two columns and two tuples each, resulting in 12 attributes. Thus, dynamic analysis of attribute dependencies together with the concept of splitting local relations may result in a substantial performance improvement and reduced memory requirements.

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 may always have a look at the still-possible 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. If any semantic error is detected, the user is not forced to correct it immediately. The method is fault-tolerant: inconsistent parts of a fragment are displayed in a different font, but editing may continue. The analysis algorithms simply ignore empty local relations.

In order to perform *dynamic context-sensitive menu filtering*, it suffices to check for each menu item whether its selection would result in an empty local relation. This is done by testing whether the join of the basic relation of a menu item and the local relation containing its attributes will become empty, which is easier than actually computing the join. Note that since a tree node may possess columns in more than one local relation, the one nearest to the fragment root has to be used, as it contains the most precise information. In order to avoid unnecessary searching, all tree nodes have a pointer attached to their "outermost" local relation.

Dynamic context-sensitive menu filtering is one of the most important features of the PSG editor. No other programming environment known to the authors provides an equally powerful method to prevent semantic errors in structured input mode. The guarantee that programs are correct at every stage of their development and the prevention of syntactic and semantic errors in structured input mode has turned out to be very helpful, particularly for beginners.

4.6 Comparison with Related Work

Several techniques for incremental semantic analysis in language-specific editors have been developed. The most well-known concepts are probably semantic action routines in GANDALF and incremental attribute evaluation within the Synthesizer Generator; there are also variations on the attribute grammar theme (e.g., [20]). It is possible to implement our concept using these techniques. In fact, context relations and unification have been experimentally implemented using the Synthesizer Generator [35]. However, the concept of inferring sets of still-possible attributes within incomplete fragments seems to be new. All the known language-independent concepts have always obeyed the classical scheme: first inspect the declarations, then use the collected information for the analysis of statements. It was a direct consequence of the PSG fragment concept that we had to do it another way.

The Milner-style analysis [27] of type-free lambda calculus expressions (including a **let** construct) computes the most general polymorphic type of a given lambda term. It also uses unification, and is in some sense similar to our scheme: Milner's notion of a most-general polymorphic type corresponds to our notion of sets of still-possible attributes of a fragment. However, the original approach is language-dependent and is not incremental.

Meertens extended Milner's approach with incremental algorithms [26]. He used the incremental polymorphic type inference algorithms within an editor for the language B. The incremental analysis concepts are similar to our algorithms. His scheme, however, is also a language-specific concept.

More recently, the MENTOR group has developed a generator for semantic analysis, which is based on inference rules and unification [12]. The context

conditions are specified in a special language for inference rules, called TYPOL, and then translated into PROLOG programs, which are executed during editing. Their scheme, however, is not fully incremental, since the necessity for change propagation is not checked dynamically. After a program modification, all inference rules that depend statically (that is, are program-independent) on the modification are reconsidered.

Finally, let us discuss some limitations of the relational approach. The concept has been developed for the semantic analysis of Pascal-like languages, primarily. It can, however, be used for other purposes. For example, we have generated a proof editor for propositional calculus, where the semantic analysis checks proofs for correctness [40]. The main restrictions of the current specification language can be found in the scope analysis. As mentioned above, the current version is not very flexible and cannot be used to specify certain complicated language features (e.g., module interfaces in Ada). Also, the algorithm cannot analyze polymorphism since it assumes that every object has just one final type. In order to analyze polymorphisms (as in Milner's algorithm), a careful and language-specific analysis of variable bindings is necessary; this cannot be specified within PSG.

From a practical viewpoint, the problem of combinatorial explosion is more difficult. If the language has many constructs that are overloaded, the number of tuples within local relations tends to increase quickly. There are two techniques for coping with this problem. The first is to decrease the number of local relations. Each local relation thus describes a bigger part of the abstract tree, and therefore contains more information, that is, fewer tuples. Incremental behavior will however not benefit from such a change. The second technique is to replace basic relations with more than one tuple by basic relations with only one tuple and additional functional dependencies, as described in Section 4.4. This is always possible, but, since expression evaluation may be delayed, immediate error detection can no longer be guaranteed—a semantic error may not be detected until the declarations of the objects involved are complete.

5. SEMANTICS DEFINITION AND FRAGMENT EXECUTION

5.1 The Denotational Semantics Definition

In the PSG system, *denotational semantics* [41, 42] are used to define the dynamic semantics of a language. As usual, the language definer must specify a *semantic function* for each syntactic construct, defining the meaning of that construct within a *semantic domain*, depending on the meaning of its subcomponents. Semantic functions are written in a *functional language* based on a type-free lambda calculus. This language supports the basic data types integer, real, boolean, string, and identifier, the structured high-level data types list/tuple and map, and higher-order functionals of arbitrary rank. The common operations on these data types are supported as well as function application, control constructs (**if-then-else**, McCarthy conditional), various combinators for list applications (similar to, e.g., **mapcar** in InterLisp), and the usual **let** and **letrec** constructs, where the latter allows the definition of recursive functions and recursive maps. The definition language has been inspired by the applicative

parts of META-IV [9, 10]. Function application evaluates its arguments *call-by-need*, whereas the elements of the structured data types are evaluated *call-by-value*. It is possible, however, to specify that selected range elements of maps be evaluated call-by-need. Call-by-need (i.e., delayed parameter evaluation) has been chosen because it combines the advantages of both call-by-value and call-by-name: Arguments are evaluated only once as in call-by-value, but only when needed, as in call-by-name, so unnecessary computations of function arguments which are never used are avoided. Call-by-need is a correct implementation of recursion, allowing nonstrict functions [43]. Actually, the PSG environment forces a call-by-need strategy, since the evaluation of some language constructs may have visible side effects within the environment (note that the language itself is side-effect-free). For example, the **answer** construct behaves like the identity function, but within the PSG programming environment its argument will be output interactively on the screen. However, a simple strictness analysis is performed to find those parameters of functions that are evaluated in all calls of the function. Strict parameters are evaluated call-by-value to avoid the unnecessary overhead of building closures. More complex strictness analysis algorithms are described in [11] and [19].

Syntactic domains correspond to the abstract syntax; semantic domains are explicitly specified in the semantics definition. Domain definitions are used to type-check semantic functions. The semantics definition specifies a direct mapping from syntactic objects (the constructs of the abstract syntax) to their denotations (i.e., semantic objects). Consider as an illustrative example the semantics of the assignment command in the (very elementary) language LOOP [42]. The relevant parts of the domain definition are given first:

```
State = [Id → Int];
...
| [Cmd] | : State → State;
| [Expr] | : State → Int;
...
{Abstract Syntax: NODE Assignment :: Id Expr;}
Assignment: LAM st. MAPADD st, [ | [Id] | → ( | [Expr] | st)];
```

The meaning of any LOOP command is a function from states to states. A state defines a mapping from variables (i.e., identifiers) to their current values, which are integers. The symbols '| [' and '] |' are the so-called *denotational brackets*. '| [Id] |' denotes the meaning of the assignment's first subcomponent, and '| [Expr] |' of its second subcomponent. The meaning of an identifier is the identifier itself (i.e., its character representation); the meaning of an expression is a function from states to values. Thus the meaning of an assignment is a function which takes as its argument a state and returns a new state: This is the same as the argument state, except that the target value of the identifer denoted by '| [Id] |' is changed to the value, resulting from the application of the function denoted by '| [Expr] |' to the old state.

Note that our definition method does not require the definition of explicit evaluation or interpretation functions as opposed to, for example, META-IV, MELA [3], or SIS. In SIS [28], the above would read as follows:

```
WITH cc(cmd0) (s): S =
CASE cmd0
  ...
  /[var ":=" expr] → LET n = ee(exp)(s) IN   update(var, n)(s)
  ...
ESAC
```

where 'cc' is the command evaluation function and 'ee' the expression evaluation function; the abstract syntax tree is passed as a parameter to the evaluation functions ('cmd0', 'exp', and 'var' in the above example), and this parameter passing has to be specified explicitly by the language definer.

Actually, a semantics definition in SIS is a higher order function, mapping an abstract syntax tree to its corresponding semantic function. Execution of a program in SIS is done by expensive, successive applications: first the complete semantics definition is applied to the program's abstract syntax tree and then the resulting function is applied to the input value(s) to obtain the output value(s). Both applications are handled by the same complex interpreter and their evaluation is interleaved. The PSG semantics definition serves, on the contrary, as a specification of a (simple) *compiler*, translating an abstract syntax tree to a term of a functional language. For example, the generated compiler for LOOP will translate the command '$x := 5$' to the following term (ignoring any optimizations done by the compiler):

LAM st. MAPADD st, [$x \rightarrow$ (LAM st.5 st)]

Note that the representation of the semantic functions corresponding to the assignment's subcomponents (e.g., '| [Expr] |') are not part of the functional language, substitution of the actual semantic functions is performed by the compiler.

The current version of the semantics definition language consists of four parts: the domain definitions, the definition of auxiliary functions, the semantic functions for each syntactic construct, and a third part describing the meaning of each executable fragment.

5.2 Fragment Compilation and Execution

Execution of fragments is based on a three-step process: *generation* of the language-specific compiler by the PSG generator, *compilation* of the fragment to a term of the functional language by the generated compiler, and *execution* of the compiled fragment by the PSG interpreter. Compilation of a fragment is performed by a top-down processing of its abstract syntax tree. The meaning functions specified in the language definition can be considered as *code templates* where the variables enclosed in denotational brackets serve as placeholders. Starting with the term that represents the meaning of the fragment, the occurrences of all placeholders are replaced subsequently by their appropriate terms. Going back to the previous example (the command fragment '$x := 5$'), compilation proceeds as follows:

```
| [Cmd] | ⇒ ANSWER ( | [Cmd] | [ ])
        ⇒ ANSWER (LAM st.MAPADD st, [ | [Id] | → ( | [Expr] | st)] [ ])
        ⇒ ANSWER (LAM st.MAPADD st, [x → (LAM st.5 st)] [ ])
```

Since the compiled terms are never used during editing, compilation is deferred until the fragment is to be executed, thus avoiding the overhead of incremental compilation. The generated compilers are however able to perform *incremental compilation*, but only in cases where the abstract syntax tree is modified by refinement steps. Incremental compilation is essential to allow the execution of incomplete fragments, thus giving the user the possibility of *interleaving fragment editing with fragment execution.* If placeholders (i.e., representations of unexpanded tree nodes) or fragment leaves (i.e., references to other fragments) occur in the abstract syntax tree, the compiler generates a special term containing a reference to the abstract syntax tree, which serves as a code stub. During execution, if control flow reaches such a code stub, execution is suspended. If the suspension is due to an unexpanded tree node, the PSG editor is invoked to allow the refinement of the specific unexpanded tree. Note that the editor is invoked in a special read-only mode to prevent the user from changing other parts of the executed fragment. In the other case, where the code stub points to a fragment leaf, the referenced fragment is loaded from the fragment library (if the fragment is not found in the library, a dummy fragment of the appropriate type is created). That part of the abstract syntax tree that has been changed as a result of the modification is incrementally compiled, the resulting term replaces the code stub and execution continues.

5.3 The Interpreter

Since the functional language is based on lambda calculus, the core of the interpreter consists of the reduction rules of lambda calculus. Reductions are implemented using *closures* and *environments* (as in the SECD machine [23]). Function abstractions as well as unreduced function arguments (due to the call-by-need evaluation strategy) are represented by closures, a pair of a term and an associated environment. Environments store bindings of the variables (reduced or unreduced) to terms represented by closures. Reduced terms are represented by closures with an empty environment.

The following evaluation function **eval** implements the reduction rules for a restricted set of terms:

```
{Domain Definitions}
Closure = Term × Env
Env     = [Var → Closure]
Term = x                        {Variable}
     | λx. T                    {Abstraction}
     | T1(T2)                   {Application}

eval: Closure → Closure

  eval(⟨λx. T, E⟩)    = ⟨λx. T, E⟩
  eval(⟨x, E⟩)        = eval(E(x))
  eval(⟨T1(T2), E⟩) = eval(apply(eval(⟨T1, E⟩), ⟨T2, E⟩))
        where apply(⟨λx. T1, E1⟩, cl) = ⟨T1, E1 + [x → cl]⟩
```

Obviously, **eval** uses a call-by-name reduction strategy instead of the intended call-by-need strategy. With call-by-need, things are getting a bit more complicated, since the evaluation of variables has side effects on the environment. In the call-by-need evaluation function, **evaln**, which is the core of the implementation of the interpreter, we model the environment as a state to allow side

effects on the environment:

```
NClosure = Term × Loc
State    = [Loc → Var × NClosure × Loc]
Loc      = "unspecified set of labels"

evaln: NClosure × State → NClosure × State

evaln(⟨λx. T, l⟩, S) = ⟨⟨λx. T, l⟩, S⟩
evaln(⟨x, l⟩, S)     = let loc = search(x, l, S) in
                         let ⟨x', cl, l'⟩ = S(loc) in
                           if is_reduced(cl) then
                             ⟨cl, S⟩
                           else
                             let ⟨cl', S'⟩ = evaln(cl, S) in
                               ⟨cl', S' + [loc → ⟨x, cl', l'⟩]⟩
    where search(x, l, S) = let ⟨x', cl', l'⟩ = S(l) in
                              if x = x' then
                                l
                              else
                                search(x, l', S)
    where is_reduced(cl) = "true, if cl is a reduced closure"
evaln(⟨T1(T2), l⟩, S) = evaln(applyn(evaln(⟨T1, l⟩, S), ⟨T2, l⟩))
    where applyn(⟨⟨λx. T, l⟩, S⟩, cl) = let l' = new_loc in
                                          ⟨⟨T, l'⟩, S + [l' → ⟨x, cl, l⟩]⟩
    where new_loc = "returns a new, unused location"
```

Since direct interpretation of terms is rather inefficient, terms may be compiled to a machine-oriented language (for more details see [6] and [18]).

6. EXPERIENCE WITH PSG

Until now, environments have been generated for ALGOL 60, Pascal, MODULA-2, C, LISP, the language-definition language itself, and some experimental specification languages. The language definition environment has been used extensively, not only by the members of the project team but also by students. The Pascal environment was used to implement other parts of the PSG system, as well as in some introductory courses on programming.

Our experience with PSG has shown that all language dependent parts of an environment can be formally described and automatically generated, at least for languages of a complexity not greater than that of Pascal or MODULA-2. The use of a formal-language definition language has many advantages:

—PSG language definitions are *safe*, since all inconsistencies in a definition are detected at generation time.
—Considering the power and complexity of the generated environments, PSG language definitions are *very short*. Typically, they vary in size between 240 lines for an ALGOL 60 environment, without context conditions and semantics, and 3600 lines for a MODULA-2 environment, including specification of context conditions and denotational semantics.
—The expressive power of the language-definition language allows concentration on the relevant aspects of a language definition. The language definer does not have to deal with minor details such as the organization of symbol tables, and so on.

—The modular design of the language-definition language improves *readability* and *reliability*. It allows the independent definition of the syntactic, context-dependent, and semantic aspects of a language, once the abstract syntax has been defined.
—A formal language-definition language in conjunction with a generator allows rapid prototyping of new languages. In a *language design lab*, language definitions are easily modified and tested.

As a consequence, the amount of human effort required to generate an environment is small: Having some initial knowledge of the PSG system, it is possible to specify and debug a definition for an ALGOL-like language without context conditions and semantics within two weeks. The MODULA-2 language definition including context conditions and denotational semantics was written as part of a diploma thesis within eight months. Thus, the use of a formal language-definition language allows the quick generation of correct, reliable, and powerful programming environments.

7. FINAL REMARKS

Work on the PSG system started in 1980; the first design considerations were published in 1978 [17]. A prototype of the PSG system, colloquially known as the BLKS system [4], has been in operation since late 1981. It combined early versions of the editor's context-free component and the interpreter. The complete PSG system implemented in Pascal on Siemens BS2000 machines has been running since 1983. In order to utilize modern personal workstations and hardware with raster graphics and pointing devices, we redesigned the user interface completely. Retargetting PSG to UNIX workstations (PERQ under PNX, CADMUS under MUNIX) is in progress; the PERQ version is almost complete. Recently, PSG has been chosen as the basis for the program constructor of the SUPRENUM supercomputer [15].

ACKNOWLEDGMENTS

The authors would like to acknowledge the work of all persons involved in the PSG project. Special thanks are due to W. Henhapl and T. Letschert, without whose inspiring ideas the development of PSG would not have been possible.

M. Hunkel, T. Reps, and the referees provided valuable comments on earlier versions of this paper.

REFERENCES

1. AHO, A. V., BEERI, C., AND ULLMAN, J. D. The theory of joins in relational databases. *ACM Trans. Database Syst. 4*, 3 (1979), 297–314.
2. AIT-KACI, H., AND NASR, R. Logic and inheritance. In *Conference Record of the 13th ACM Symposium on Principles of Programming Languages* (St. Petersburg, Fla., Jan. 1986), ACM, New York, 219–228.
3. AMBRIOLA, V., AND MONTANGERO, C. Automatic generation of execution tools in a GANDALF environment. *J. Syst. Softw. 5*, 2 (May 1985), 155–171.
4. BAHLKE, R., AND LETSCHERT, T. The BLKS system: Towards the generation of programming environments. In *Proceedings 3. GI-Fachgespräch Compiler-Compiler* (Munich, Mar. 1982), W. Henhapl, Ed., 153–173.

5. BAHLKE, R., AND SNELTING, G. Programmiersystemgenerator—Arbeitsbericht 1984, Rep. PU2R2/84, TH Darmstadt, Feb. 1984.
6. BAHLKE, R, AND LETSCHERT, T. Ausführbare Denotationale Semantik. In *Proceedings 4. GI-Fachgespräch Implementierung von Programmiersprachen* (Zurich, Mar. 1984), H. Ganzinger, Ed., 3–19.
7. BAHLKE, R., AND SNELTING, G. The PSG programming system generator. *ACM SIGPLAN Not. 20*, 7 (July 1985), 28–33.
8. BAHLKE, R., AND SNELTING, G. Context-sensitive editing with PSG environments. In *Proceedings of the International Workshop on Advanced Programming Environments: Lecture Notes in Computer Science*. Springer Verlag, New York (to appear June 1986).
9. BJØRNER, D., AND JONES, C. B. (Eds.) *The Vienna Development Method: The Metalanguage. Lecture Notes in Computer Science, 61*. Springer Verlag, New York, 1978.
10. BJØRNER, D., AND JONES, C. B. *Formal Specification and Software Development*. Prentice-Hall, Englewood Cliffs, N.J., 1982.
11. CLACK, C., AND PEYTON JONES, S. L. Strictness analysis—a practical approach. In *Lecture Notes in Computer Science, 201*. Springer Verlag, New York, 1985, 35–49.
12. DESPEYROUX, T. Executable specification of static semantics. In *Lecture Notes in Computer Science, 173*. Springer Verlag, New York, 1984, 215–233.
13. DONZEAU-GOUGE, V., KAHN, G., LANG, B., AND MÉLÈSE, B. Document structure and modularity in MENTOR. *ACM SIGPLAN Not. 19*, 5 (May 1984), 141–148.
14. ELLISON, R. J., AND STAUDT, B. J. The evolution of the GANDALF system. *J. Syst. Softw. 5*, 2 (May 1985), 107–119.
15. GILOI, W. K., AND MÜHLENBEIN, H. Rationale and concepts for the SUPRENUM supercomputer architecture. Internal Rep., Gesellschaft für Mathematik und Datenverarbeitung, 1985.
16. GOGUEN, J. A. Order-sorted algebras: Exception and error sorts, coercions and overloaded operators. Semantics and Theory of Computation Rep. 14, UCLA, 1978.
17. HENHAPL, W. Von der Compiler-Generierung zu der Programmiersystemgenerierung. In *Proceedings 1. GI-Fachgespräch Compiler-Compiler* (Berlin, Sept. 1978), 160–169.
18. HENHAPL, W., AND LETSCHERT, T. VDM in research, development, and education: Local experiences. In *Formal Models in Programming*. North-Holland, Amsterdam, 1985, 157–180.
19. HUDAK, P., AND YOUNG, J. Higher-order strictness analysis in untyped lambda calculus. In *Conference Record of the 13th ACM Symposium on Principles of Programming Languages* (St. Petersburg, Fla., Jan. 1986), ACM, New York, 97–109.
20. JOHNSON, G. F., AND FISCHER, C. N. A metalanguage and system for nonlocal incremental attribute evaluation in language-based editors. In *Conference Record of the 12th ACM Symposium on Principles of Programming Languages* (New Orleans, La., Jan. 1985), ACM, New York, 141–151.
21. JOHNSON, S. C. YACC: Yet another compiler-compiler. In *UNIX Programmer's Manual Vol. 2B*, Bell Labs., July 1978.
22. KAHN, G., LANG, B., MÉLÈSE, B., AND MORCOS, E. METAL: A formalism to specify formalisms. *Sci. Comput. Program. 3* (1983), 151–188.
23. LANDIN, P. J. The mechanical evaluation of expressions. *Comput. J. 6*, 4 (1964), 308–320.
24. MEDINA-MORA, R. Syntax-directed editing: Towards integrated programming environments. Ph.D. thesis, Carnegie-Mellon Univ., Mar. 1982.
25. MEDINA-MORA, R., NOTKIN, D. S., AND ELLISON, R. J. ALOE user's and implementor's guide. Carnegie-Mellon Univ., May 1982.
26. MEERTENS, L. Incremental polymorphic typechecking in B. In *Conference Record of the 10th ACM Symposium on Principles of Programming Languages* (Austin, Tex., Jan. 1983), ACM, New York, 265–275.
27. MILNER, R. A theory of type polymorphism in programming. *J. Comput. Syst. Sci. 17*, 3 (1978), 348–375.
28. MOSSES, P. SIS—Semantics Implementation System, reference manual and user guide. Rep. DAIMI MD-30, Aarhus Univ., Aug. 1979.
29. PLOTKIN, G. Building in equational theories. *Mach. Intell. 7* (1972), 73–90.
30. REISS, S. An approach to incremental compilation. *ACM SIGPLAN Not. 19*, 6 (June 1984), 144–151.
31. REPS, T. Generating language-based environments. Rep. TR 82-514, Cornell Univ., Aug. 1982.

32. REPS, T., TEITELBAUM, T., AND DEMERS, A. Incremental context-dependent analysis for language-based editors. *ACM Trans. Program. Lang. Syst. 5*, 3 (July 1983), 449–477.
33. REPS, T., AND TEITELBAUM, T. The Synthesizer Generator. *ACM SIGPLAN Not. 19*, 5 (May 1984), 42–48.
34. REPS, T., AND TEITELBAUM, T. *The Synthesizer Generator Reference Manual.* Cornell Univ., Aug. 1985.
35. REPS, T., AND SNELTING, G. Context relations implemented with attribute grammars. Cornell Univ., Jan. 1986.
36. RETY, P., KIRCHNER, C., KIRCHNER, H., AND LESCANNE, P. NARROWER: A new algorithm for unification and its application to logic programming. In *Lecture Notes in Computer Science, 202.* Springer Verlag, New York, 1985, 139–157.
37. ROBINSON, J. A. A machine-oriented logic based on the resolution principle. *J. ACM 12*, 1 (1965), 23–41.
38. SIEKMANN, J. H. Universal unification. In *Lecture Notes in Computer Science, 170.* Springer Verlag, New York, 1984, 1–42.
39. SNELTING, G., AND HENHAPL, W. Unification in many-sorted algebras as a device for incremental semantic analysis. In *Conference Record of the 13th ACM Symposium on Principles of Programming Languages* (St. Petersburg, Fla., Jan. 1986), ACM, New York, 229–235.
40. SNELTING, G. Inkrementelle semantische Analyse in unvollständigen Programmfragmenten mit Kontextrelationen. Doctoral thesis, TH Darmstadt, Mar. 1986.
41. STOY, J. E. *Denotational Semantics: The Scott–Strachey Approach to Programming Language Theory.* The MIT Press, Cambridge, Mass., 1977.
42. TENNENT, R. D. The denotational semantics of programming languages. *Commun. ACM 19*, 8 (Aug. 1976), 437–453.
43. VUILLEMIN, J. Correct and optimal implementations of recursion in a simple programming language. *J. Comput. Syst. Sci. 9* (1974), 332–354.

Received September 1985; revised March and April 1986; accepted April 1986