Acta Informatica 34, 737–772 (1997)

Prototyping realistic programming languages based on formal specifications

Arnd Poetzsch-Heffter

Praktische Informatik V, FernUniversität, D-58084 Hagen, Germany
(e-mail: poetzsch@fernuni-hagen.de)

Received: 20 November 1995 / 20 January 1997

Abstract. The specification of realistic programming languages is difficult and expensive. One approach to make language specification more attractive is the development of techniques and systems for the generation of language–specific software from specifications. To contribute to this approach, a tool–based framework with the following features is presented: It supports new techniques to specify more language aspects in a static fashion. This improves the efficiency of generated software. It provides powerful interfaces to generated software components. This facilitates the use of these components as parts of language–specific software. It has a rather simple formal semantics. In the framework, static semantics is defined by a very general attribution technique enabling e.g. the specification of flow graphs. The dynamic semantics is defined by evolving algebra rules, a technique that has been successfully applied to realistic programming languages.

After providing the formal background of the framework, an object–oriented programming language is specified to illustrate the central specification features. In particular, it is shown how parallelism can be handled. The relationship to attribute grammar extensions is discussed using a non-trivial compiler problem. Finally, the paper describes new techniques for implementing the framework and reports on experiences made so far with the implemented system.

1 Introduction

Modern programming languages become more and more complex. Besides the classical language constructs, they should support powerful and flexible modularization and programming concepts like object–orientation, genericity, exception handling, features for parallel programming, assertions, etc. While the complexity and size of languages increases, the need and demand for formal language specification grows. This has the following reasons:

- Portability, standardization: Programs should run on different computing systems. Therefore they need system independent semantics.
- Correct language implementation: High level languages are used to build safety critical systems. This makes even expensive attempts worthwhile to eliminate errors from language implementations.

– Formal methods: To keep the growing software complexity tractable, formal methods play an increasingly important rôle in software development. To interface them with programming languages without losing formal strength, programming languages need appropriate formal specifications.

The research in programming language specification can be structured into three threads focusing on different applications: a) formal documentation (central aspects: readability, completeness, and extensibility of specifications; cf. [Mos92]); b) tool generation (cf. [GHL+92]); c) verification of language or program properties. The last years have seen a growing interest in bringing these threads closer together and integrate developed techniques. One reason for this is to increase the benefits gained from language specifications by exploiting them for several of the above purposes. Different methods have been used to lessen the gap between the threads. In [AP94], attribute grammar technology is used to generate practical tools from restricted natural semantics specifications. [Ørb94] and [Pet94] report on special implementation techniques for compiler and interpreter generation based on action semantics and natural semantics respectively. In e.g. [BF95] and [PH94a], the relation between language specification techniques and logical frameworks is investigated. The presented work extends, adapts, and integrates specification techniques from data type definition, attribute grammars, and evolving algebras. The integration is done in such a way that the essential features of the integrated techniques are retained. The design of the integrated framework was mainly led by the following goals:

Goal 1: Generation of efficient interpreters for full size programming languages: Complete specifications of realistic programming languages are several thousand lines long. To ensure that specifications reflect the ideas of language designers, validation is very important. An interpreter is one of the best means to validate language specifications.

Goal 2: Flexible and powerful interfaces to generated software components: To use software components generated from a language specification as part of language–specific software, the components have to provide flexible interfaces. The application of the components should not require knowledge about the generation technology.

Goal 3: Simple semantics for the integrated framework: A simple semantics facilitates to use the framework and to embed specifications into logical frameworks.

The presented work can be understood as a practical, tool–based compromise between techniques developed in the three research threads sketched above (cf. Sect. 4 for a discussion).

General approach

One way to classify language specification methods is to study the rôle that programs play in the specifications (cf. Sect. 4). We distinguish the following methods: The *translational* method specifies the semantics of a language by defining appropriate semantic domains and by mapping syntax trees to elements of these domains (in denotational semantics (e.g. [Sch86]), the domains consist of functions; in action semantics (cf. [Mos92]), they consist of actions). The *structural* method specifies the semantics as relations between configurations; these relations are inductively defined over the syntax and have syntax trees as arguments (cf. [Ast91]). The *global state*

method specifies the semantics as a relation between global states and possible successor states; in this method, the whole program is part of the global states. For our above goals, the last method has two advantages:

- Having the whole program as part of the state easily allows us to use richer program representations. Instead of "naked" syntax trees, we can easily use attributed trees or graphs as basis for program interpretation. I.e. static information about programs can be computed once and stored for quick access so that execution is sped up (important for goal 1).
- Richer program representations and the distinction between static and dynamic information on the specification level leads to more practical interfaces for software generated from such specifications (important for goal 2).

To illustrate these advantages let us consider a small example (a more detailed discussion is given in Sect. 4). In classical imperative languages, the link between a procedure call and the procedure declaration is static program information[1]. An interpreter that only has to follow links for calling procedures (dereferencing a pointer) is faster than an interpreter that has to administer dynamically changing environments and has to look up the procedure each time it reaches a call. There are two approaches to obtain the fast interpreter: 1) Specify static information such as the described links explicitly in the language specification (e.g. by specifying an attribute for call sites yielding the corresponding procedure declaration node). 2) Specify static information implicitly by an environment mechanism and extract it by optimizations showing that parts of the environment at a program point do not change during execution. We decided to support the first approach, because

- not all static aspects can be discovered automatically;
- it can be helpful for the reader of a language specification to know which parts of a specification are static and which parts are dynamic;
- many language–specific tools only need the static part of a language specification; thus, it should be easy to separate static from dynamic specification parts;
- making dependencies between nodes of the syntax tree explicit can simplify the interface of generated components.

As an example for the last argument let us reconsider the link between call sites and corresponding procedure declarations. Such a link is not only useful for fast interpretation, but as well very helpful e.g. to implement program browsers or to compute dependencies between procedures (e.g. in reengineering tools). The interface for such links is simply a function from call site nodes to declaration nodes. The specification of this function by a possibly complex symbol table mechanism can be hidden from the user.

Technical approach

To express static language aspects, we looked for an extension of attribute grammars. This extension had to be sufficiently powerful to specify links in syntax trees in a declarative way. The challenge was to integrate the developed techniques into a simple formal framework (cf. goal 3 above). To illustrate this challenge, consider e.g. the notions used in attribute grammars: context–free grammar (productions, nonterminals,

[1] This is different in object–oriented languages; see Sect. 3 for an example.

terminals), semantic domains, semantic functions, attributes, attribute equations. For the needed extensions and the dynamic aspects, we would have to add even new notions. To avoid this, we developed a slim extension of term algebras called *occurrence algebras*. Their universes contain not only all freely generated (sorted) terms over a signature, but as well all (sorted) occurrences of these terms. Thereby, an attribute can be defined as a unary function having an occurrence sort as domain. As the range sort of the attribute may contain occurrences, links in trees can be expressed as well.

To specify dynamic semantics, we use evolving algebra rules. Evolving algebras are developed by Gurevich to specify computation in the presence of bounded resources (cf. [Gur95]). They were successfully applied for different kinds of specification purposes, in particular for the specification of dynamic programming language semantics. In evolving algebras, a state is modeled by an algebra. Transitions are modeled by updating functions of the algebras. Evolving algebras are well suited for our goals: 1. They were designed for the specification of global state transitions. 2. A combination with occurrence algebras is straightforward (cf. Sect. 2). 3. Evolving algebra rules provide a good basis for efficient implementation, because they do not rely on unification or matching. 4. Evolving algebras have a simple semantics, i.e. a semantics that is easy to learn and can be embedded in most verification frameworks (cf. [HDP96]).

Overview

The rest of the paper is structured as follows. Section 2 defines occurrence algebras and introduces the needed features of evolving algebras. Section 3 presents techniques for specifying programming languages and related aspects within the framework. As specification techniques developed for attribute grammars can be directly applied in the presented framework (cf. our specification of the C programming language [Bic95] and the discussion in Sect. 4.2), Sect. 3 concentrates on dynamic language aspects and those static aspects that are difficult to handle by attribution frameworks. A small object–oriented language with recursive methods and parallelism is specified. To illustrate special attribution techniques, a specification for common subexpression elimination, a typical compiler problem, is presented. Section 4 discusses the relation to other specification frameworks. Section 5 describes our implementation of the framework, the MAX System; in particular, it explains the used attribute evaluation techniques. Finally, it reports on experiences made with MAX.

The presented results are based on and extend those described in [PH93] and [PH94b]. New are specification techniques for object–oriented languages, parallelism, and optimization tasks. In addition to that, the implementation described in Sect. 5 generalizes that of [PH93] by supporting more flexible attribute coupling techniques and nondeterminism in evolving algebras via external functions.

2 Formal background

This section introduces the formal concepts on which our specifications are based. The first subsection defines occurrence algebras, an extension of term algebras. In these algebras, occurrences of subterms within terms are elements of the algebras (e.g. variable declaration occurrences in program syntax trees). Having occurrences as algebra elements allows to specify functions with occurrences as domain and/or

range. In particular, such functions can be used to express links in a tree. The second subsection gives a short introduction to evolving algebras that are used to specify dynamic semantics. Both concepts use the notion of "Σ–algebras" where Σ denotes a signature:

Definition 2.1 *A* **signature** Σ *is a family of sets SYMB*$_s$ *of function symbols where* $s \in \mathbb{N}$ *is the arity of the function symbols in SYMB*$_s$*. A* Σ**–algebra** *is given by a set* U*, called the* **universe** *of the algebra, and an* **interpretation** φ *mapping function symbols in SYMB*$_s$ *to functions from* U^s *to* U*.*
□

In the above definition, Σ–algebras are not sorted, i.e. the universe is not (explicitly) partitioned into several sorts and the functions are defined on the whole universe. We consider sorts as subsets of the universe. Sorts are formally modeled by unary predicates yielding true for all elements of the sort. As we follow a model-oriented approach in this paper, proceeding this way allows to avoid dispensable technical overhead[2].

Notation As usual, we write $f(a_1, \ldots, a_n)$ for applying function f to arguments $a_1, \ldots, a_n$. For n = 1, we write as well $a_1.f$ to mimic the usual attribute notation and to make application sequences more readable. For n = 0, the parentheses may be omitted.

2.1 Occurrence algebras

Occurrence algebras can be considered as an extension of term algebras. The universe of an occurrence algebra contains not only all terms that can be built by a set of constructors, but as well all occurrences of subterms within these constructor terms. Before the formal definition of occurrence algebras, we give an informal introduction. As an example, we consider part of the expression syntax of the object–oriented language treated in Sect. 3:

```
Exp          =  Send | UsedId | ...
Send         (  Exp  Ident  ExpList  )
ExpList      *  Exp
UsedId       (  Ident  )
```

An occurrence algebra is defined by a list of *productions*. There are three kinds of productions. (a) Variant productions: E.g. the first line defines an expression to be a send expression or a used identifier (other alternatives will be given in Sect. 3). (b) Tuple productions: E.g. a *Send*-expression is defined to be a triple with an expression component (yielding the *receiver* of the send), an identifier component (denoting the name of the message), and an expression list component (providing the actual parameter list of the send). (c) List productions: E.g. that for `ExpList`. The sort *Ident* is predefined.

For many purposes[3] one has to refer to the "outer context" of a syntax tree node, i.e. in particular its parent node. This is not possible if a tree node is represented by an element of a term algebra. An element of a term algebra does not incorporate

[2] In axiomatic approaches to algebra specification, living without explicit sorting is much harder.
[3] E.g. to define links in syntax trees or control flow graphs; cf. Sect. 3.

information about the context in which it occurs. Let us consider an example: A send expression for concatenating a list *lv1* with itself and another list *lv2* by using a message *conc3*, in concrete syntax `lv1.conc3(lv1,lv2)` , is represented by the term t_{Send}:

$$t_{Send} \equiv \textit{Send}(\ \textit{UsedId}(\text{“lv1”}),\ \text{“conc3”},\ \textit{ExpList}(\ \textit{UsedId}(\text{“lv1”}),\ \textit{UsedId}(\text{“lv2”})\ )\ )$$

If we consider syntax tree nodes simply as elements of a term algebra, the two occurrences of *UsedId*(“lv1”) in t_{Send} are indistinguishable. In particular, we cannot define a function *parent* for them. That is why we consider as well occurrences in terms. Occurrences are represented using the constructors *occ* and $c1, c2, \ldots$: *occ(t)* denotes the *root occurrence* of term t and $ci(o)$ yields the i-th child of occurrence o. Using the dotted notation[4] for function application, the first occurrence of *UsedId*(“lv1”) in t_{Send} is $t_{Send}.\textit{occ}.c1$, whereas the second is $t_{Send}.\textit{occ}.c3.c1$. This notation closely corresponds to the usual meta notation for occurrences[5] where an occurrence is denoted as a pair of a term and a list of positive integers. Thus, the contribution is to make a meta notation available as part of our algebraic framework. In particular, occurrences are ordinary algebra elements and *occ* and the ci’s are ordinary functions of the algebras.

It is useful to extend the sorting on terms to occurrences: An occurrence of a subterm s of sort *S* is said to be of sort *S@*; e.g. the occurrences given above are of sort *UsedId@*. Based on occurrence sorts, we can define new sorts. For example execution points before and after expression occurrences can be defined by the following productions:

```
ExecPoint  =  before  |  after
before     (  Exp@  )
after      (  Exp@  )
```

In the following we define what the occurrence algebra for a finite set of productions is.

Definition 2.2 *Let PRODSORTS and PRIMSORTS be two disjoint sets of symbols called production sort symbols and primitive sort symbols*[6]*. A* **sort symbol** *is a production or primitive sort symbol followed by a possibly empty sequence of @-characters, i.e. has the form S@***, where S* $\in$ *PRODSORTS* $\cup$ *PRIMSORTS. The set of sort symbols is denoted by SORTS. In most applications, we only deal with sort symbols of the form S or S@ (cf. comment below).*

- *A* **production** *has one of the following forms: S* ($S_1 \ldots S_m$) *, called* **tuple production***, or S*$*$*T, called* **list production***, or S = S*$_0|\ldots|$*S*$_n$*, called* **variant production***, where S* $\in$ *PRODSORTS, S*$_i$*, T* $\in$ *SORTS, and* $m, n \in \mathbb{N}$*; S is called the* **left hand side** *of the respective production.*
- *Let* Π *be a finite set of productions such that for each S* $\in PRODSORTS$ *there is exactly one production with left hand side S. For each sort in SORTS\PRIMSORTS we inductively define a set of* **elements** *assuming that the sets for the primitive sorts are given:*

[4] Recall that $a.f$ is equivalent to $f(a)$; see above.

[5] In some communities, occurrences are called *positions*.

[6] The MAX System currently provides the primitive sorts *Ident*, *Int*, *Bool*, *Char*, and *String*.

1. $S(t_1, \ldots, t_n)$ *is an element of sort* S *if* $S\ (S_1 \ldots S_n) \in \Pi$ *and each* t_i *is of sort* S_i *;*
2. $S(t_1, \ldots, t_l)$ *is of sort* S *if* $S * T \in \Pi$, $l \in \mathbb{N}$, *and each* t_i *is of sort* T*;*
3. t *is of sort* S *if* $S = S_0 | \ldots | S_n \in \Pi$ *and* t *is of sort* S_i *for one* i, $0 \leq i \leq n$*;*
4. $t.occ$ *is of sort* $S@$ *if* t *is of sort* S*;*
5. $t.occ.c\mathrm{i}_1. \ldots .c\mathrm{i}_m$ *is of sort* $S@$ *if* $t \equiv T(t_1, .., t_{i_1}, .., t_n)$ *is of sort* T *and* $t_{i_1}.occ.c\mathrm{i}_2. \ldots .c\mathrm{i}_m$ *is of sort* $S@$*.*

Elements generated by the first two rules are called **constructor terms***; elements generated by the last two rules are called* **occurrence terms** *or simply* **occurrences**.

– *The universe of the* **occurrence algebra** *defined by* Π *consists of the elements as defined above, the elements of the primitive sorts, and the extra element nil; the functions of the occurrence algebra defined by* Π *are:*
  – *functions defined on occurrences: parent*(x)*, lsib*(x)*, rsib*(x)*, child*(i, x)*, the parent, left/right sibling,* i*-th child of occurrence* x*; term*(x)*, the subterm that corresponds to occurrence* x*; root*(x)*, the root occurrence of the term that* x *is an occurrence of; numchilds*(x)*, yielding the number of children of* x*.*
  – *functions defined on terms: subterm*(i, t)*, the* i*-th subterm of* t*; append*(e, l)*, first*(l)*, rest*(l)*, isempty*(l) *with the usual meaning on list terms; for each list production* $S * T$ *the empty list constructor* $S()$*; and for each tuple production* $S\ (S_1 \ldots S_n)$ *the constructor* $S(t_1, \ldots, t_n)$*. (For convenience, we use the same name for the sort (slanted font; e.g. Send) and the corresponding constructor (italic font; e.g. Send).)*
  – *for each sort* S *a boolean function is*$S(x)$ *testing whether* x *is of sort* S*.*
  – *the equality and the functions defined on primitive sorts (inequality will be written by #)*

  Whenever a function is applied to elements where the meaning as described above is not defined, it yields the extra element nil; i.e. all functions are total.

□

Occurrence algebras can be considered as an extension of term algebras, because their universes contain not only all terms generated by the constructor functions (rule 1 and 2 above), but as well all occurrences in these terms (rule 4 and 5 above). They support a restricted ordering on sorts, i.e. sort *S* may be a subsort of a sort *T*. This subsort ordering is defined by the variant productions (cf. rule 3 above). In Sect. 3, we show how occurrence algebras can be exploited for language specification. In specification contexts, we will often use "node" as a synonym for occurrence in a syntax tree. Finally, we like to comment on sorts like *Exp@@* that might seem somewhat strange. Indeed, elements of such sorts are rarely needed in practice. But, the used generality simplifies the above definition, because otherwise terms like $t_{Send}.occ.before.occ.c1$ where t_{Send} is the term defined above, have to be ruled out explicitly, although they do not cause any problems.

2.2 Evolving algebras

Evolving algebras provide an elaborate specification framework (see [Gur95]) used for a fairly broad range of applications. In particular, they were successfully applied

for specifying the dynamic semantics of several different programming languages[7] including PROLOG ([BR92]), and Occam ([GM89]). In this subsection, we summarize those aspects of evolving algebras that will be needed for the rest of the paper.

In an evolving algebra specification, computation states are modeled by algebras. The evolving algebra specifies how states are related to possible successor states: Given an algebra A, it describes how the functions of A may be "updated" to get a successor algebra — reflecting the intuition that in a computation step only a small part of the state is changed. As an introductory example let us specify the semantics of lists of assignments of the form $\langle variable \rangle$:= $\langle variable \rangle$, i.e.:

```
AssignList  *  Assign
Assign      (  Ident  Ident  )
```

The state information that changes during execution is represented by the unary function *cont* mapping variables to values and the "0-ary" function *al* holding the rest of the assignment list. Here is an evolving algebra specifying the dynamic semantics of assignment lists (the initial state is not specified by these rules; see 2.3):

```
IF NOT isempty(al) THEN  al := rest(al)                FI
IF NOT isempty(al) THEN
       cont( subterm(1,first(al)) )
         := cont( subterm(2,first(al)) )               FI
```

The evolving algebra has two rules that are executed in parallel: If the current assignment list *al* is not empty, simultaneously update *al* to its rest and the contents of the left hand side variable of the first assignment in the list to the contents of the corresponding right hand side variable. In general, rules are executed as follows: Evaluate the guards, the right hand side expressions of the updates, and the arguments on the left hand sides of the updates (here *subterm*$(1, \mathit{first}(\mathit{al}))$). Then, simultaneously perform those updates with a true guard. The following definition makes this more precise:

Definition 2.3 *An* **evolving algebra** *EA over a signature Σ is given by a finite set of rules of the form* **IF** *guard* **THEN** $f(\mathit{arg}_1, \ldots, \mathit{arg}_n)$:= *rhsterm* **FI** *where guard, arg_i, and rhsterm are ground Σ–terms and n is the arity of f. "$f(\mathit{arg}_1, \ldots, \mathit{arg}_n)$:= rhsterm" is called an* **update**. *Functions like f occurring on the outermost position of the left hand side of an update are called* **dynamic functions**, *or in case of $n = 0$* **dynamic variables**. *Other functions are called* **static**.

- *Let A be a Σ–algebra containing the boolean values. A Σ–algebra A' is an EA–***successor** *of A if updating the functions of A according to the following procedure yields A'* [8]*:*

  Let UPDATES be the set of updates of those rules the guards of which evaluate to true in A. Evaluate all arguments and right hand side terms occurring in UPDATES and replace them by their values. If the resulting set contains subsets of contradicting updates, e.g. $g(b_1, \ldots, b_n) := r$ and $g(b_1, \ldots, b_n) := q$ with $r \neq q$, no updates are performed, i.e. $\varphi = \varphi'$. Otherwise, change the functions of A according to this set of updates, i.e. change the interpretation of function f at point $a_1, \ldots, a_n$ to r if there is an update $f(a_1, \ldots, a_n) := r$.
- *A sequence $(A_i)_{i \in \mathbb{N}}$ of algebras such that A_{i+1} is an EA–successor of A_i for all i is called a* **computation** *of EA for initial algebra A_0.*

[7] In most of these specifications, syntax and static semantics are only informally described.

[8] In more technical terms: if the pair of interpretations (φ, φ') is included in the relation described by the given procedure.

□

For practical purposes, the restricted syntax of evolving algebras as shown above is rather inconvenient. Assuming that the considered algebras always contain the constant *true* and conjunction and negation as (static) boolean functions, we can introduce the following four syntactical extensions[9] — the associated rules show how to transform the extended syntax into the restricted syntax above:

- unguarded updates as rules: $f(\vec{a})$:= *rhsterm*
  ⇒ **IF** *true* **THEN** $f(\vec{a})$:= *rhsterm* **FI**,
- nested if-rules: **IF** *guard1* **THEN IF** *guard2* **THEN** *body* **FI FI**
  ⇒ **IF** *guard1* $\wedge$ *guard2* **THEN** *body* **FI**
- rule lists: **IF** *guard* **THEN** $R_1 \ldots R_n$ **FI**
  ⇒ **IF** *guard* **THEN** R_1 **FI** $\ldots$ **IF** *guard* **THEN** R_n **FI**
- if-then-else-rules: **IF** *guard* **THEN** *body1* **ELSE** *body2* **FI**
  ⇒ **IF** *guard* **THEN** *body1* **FI** **IF** $\neg$ *guard* **THEN** *body2* **FI**

Nondeterminism So far, evolving algebra rules can only describe deterministic computations. For the specification of nondeterminism, we use so–called *external functions/variables*. The interpretation of function symbols being declared as external may change during execution like that of dynamic functions, but the change is nondeterministic and not controlled by the evolving algebra rules. Such functions are called external, because we can consider the changes to be caused by the external environment our computation process is embedded in (like e.g. inputs from outside or scheduling decisions). External functions are sufficient for expressing nondeterminism in the context of MAX specifications. For a more elaborate treatment of nondeterminism in evolving algebras, the reader is refered to [Gur95].

2.3 Specifications

A specification in our framework consists of a set of productions, a set of recursive function definitions (see Sect. 3), a set of external function symbols with arity, and a set of evolving algebra rules. The productions, function definitions and symbols for external functions specify a set of algebras. These algebras

- have the universe of the occurrence algebra defined by the productions;
- interpret the recursive functions according to their definitions;
- differ in the interpretation of the external function symbols.

They specify the possible initial execution states. The evolving algebra specifies the possible computations starting in these states.

3 Specifying with occurrence and evolving algebras

This section gives an introduction to the specification with occurrence and evolving algebras. It shows the basic techniques that we use for specifying programming

[9] In addition to that, the MAX System provides a case construct for more efficient implementation.

languages and related tasks in our framework. These specification techniques are illustrated using two examples: (a) the specification of BOPL, a **b**asic **o**bject–oriented **p**rogramming **l**anguage; (b) the specification of common subexpression elimination to show advanced attribution techniques. The specifications are given in the syntax of the MAX System that generates attribute evaluators and interpreters out of it (cf. Sect. 5). Whereas this section concentrates on explaining the specifications, Sect. 4 will compare the techniques to related frameworks.

BOPL is taken from [PS94] where it is used to illustrate typing of object–oriented languages. Even though necessarily small, BOPL contains many features that are difficult to handle in other language specification frameworks, for example method selection (illustrating how to handle exceptions and jumps), recursive procedures/methods, side-effects in expressions, and parallel interleaved evaluation of binary expressions[10]. It should be noted here that all of these aspects occur in some form in realistic programming languages. The BOPL specification is structured into syntax, static semantics, and dynamic semantics. The syntax specification comprises context free and context dependent aspects. The static semantics specification defines the data types of the programming language and other semantical aspects that can (easily) be treated in a static way.

3.1 Specifying syntax

This section provides the abstract syntax of BOPL and specifies identification, the task of relating used and declared identifier occurrences (often called name analysis as well).

Abstract syntax We use abstract syntax to represent programs. The specification of concrete syntax and the mapping to abstract syntax is assumed here; in our implementations, we use conventional parser generators (like e.g. YACC) for that purpose. The abstract syntax of BOPL is given by the following productions:

```
(1)   Program   (  ClassList.classlist  Exp.mainexp  )
(2)   ClassList *  Class
(3)   Class     (  Ident  VarList.varlist  MethodList  )
(4)   Method    (  Ident.methid  ParList.formpars  Exp.body  )
(5)   VarList   *  Var
(6)   ParList   *  Par
(7)   Var       (  Ident  )
(8)   Par       (  Ident  )

(9)   Exp       = Send | UsedId | Constant | Self | Assign | Seq | If
                    | While | New
(10)  Constant  = Int | Bool | Null
(11)  Null      ()
(12)  Self      ()
(13)  Assign    (  UsedId  Exp  )
(14)  Seq       (  Exp  Exp  )
(15)  If        (  Exp  Exp  Exp  )
(16)  While     (  Exp  Exp  )
(17)  New       (  Exp  )
```

[10] Not considered in [PS94].

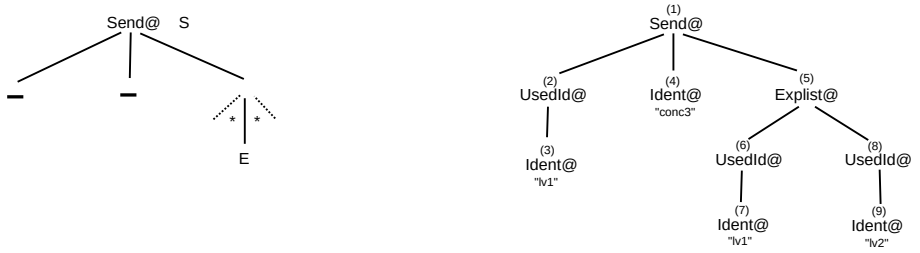
A BOPL program consists of a list of classes and a main expression. A class is given by a list of (instance) variables and a list of methods. A method is given by an identifier, a list of parameters, and an expression as body. The production for *Method* (line 4) illustrates as well how selector functions can be defined: For the sort *Method* the selectors *methid*, *formpars*, and *body* are defined. Selectors are used to make specifications more readable by substituting the polymorphic selectors *subterm* and *child*. E.g. the selector *body* applied to a constructor term of sort *Method* yields the third subterm; applied to an occurrence of sort *Method@* it yields the third child; applied to other elements it yields *nil*.

Besides *Send*- and *UsedId*-expressions already introduced in 2.1, BOPL provides constants, the *Self*-expression denoting the receiver object in the body of a method (in C++ denoted by *this*), assignments, sequential execution, conditional and loop expressions, and *New*-expressions that yield, given an object of class D, a new object of class D.

Identification Identification is the task of relating used and declared occurrences of identifiers. The identification specification for BOPL is presented here as an introduction to function and attribute definitions based on occurrence algebras. As first example, let us consider the function *encl_method* yielding for each expression node E the enclosing method, if any: If E is the body of method M, yield M; else if E occurs in an actual parameter list of a send expression S, yield the enclosing method of S; else if E is the main expression of a program yield *nil*; else yield the enclosing method of E's parent:

```
FCT  encl_method( Exp@ E ) Method@:
  IF  Method@<_,_,E> M      THEN  M
   |  Send@<_,_,<*,E,*>> S  THEN  S.encl_method
   |  Program@<_,E>         THEN  nil
  ELSE  E.parent.encl_method
```

The function *encl_method* demonstrates several aspects: The use of occurrences enables to define functions yielding nodes in the outer context of the argument node. This is not possible if trees are represented by constructor terms. To ease specification MAX provides an elaborate pattern notation; these patterns allow in particular to refer to the outer context of a node. As this does not make sense in existing functional programming languages, it may deserve some explanation. Consider e.g. the pattern "`Send@<_,_,<*,E,*>> S`" illustrated in the following figure. The figure shows as well the tree of occurrences for the term t_{Send} from Sect. 2.1; occurrences are numbered to refer to them:



The pattern contains two kinds of wildcards: the underscore standing for exactly one element, the asterisk standing for an arbitrary number of elements. And it contains

three kinds of identifiers: sort identifiers like *Send@*, free identifiers like S, and bound identifiers like the formal parameter E. The pattern specifies a condition on the bound identifier E, namely that E is some child of the third child of a *Send*-node. This would e.g. be true, if E is bound to occurrence (6) or (8) of the given occurrence tree. If the condition holds, the free identifiers of the pattern are bound to the corresponding nodes. In the example, S would be bound to occurrence (1). In general, a pattern consists of three parts, each of which is optional: a sort identifier, a list of patterns enclosed in angle brackets possibly containing wildcards, and a (free or bound) identifier. Notice that the children of a list node are the elements of the list (cf. Definition 2.2).

An attribute in MAX is a unary function. Its domain sort must be an occurrence sort or a sort like e.g. *ExecPoint* that is defined by a unary constructor on an occurrence sort. Specifying a function as an attribute is merely a hint to the system to compute the attribute value once and store it for later use (cf. Sect. 5). As an example, we explain the attribute *decl* that links each used identifier occurrence to the corresponding declaration:

```
Declaration    =  Par@ | Var@ | Class@

ATT  decl( UsedId@ U ) Declaration:
  LET  PAR == lookup( U, U.encl_method.formpars.first ) IN
  IF   PAR # nil   THEN  PAR
  ELSE LET  VAR
         == lookup( U, U.encl_method.parent.lsib.first ) IN
       IF   VAR # nil  THEN  VAR
       ELSE  lookup( U, U.root.classlist.first )
```

A BOPL declaration is a parameter, variable, or class declaration. Reflecting the hiding rules of identifiers in BOPL, attribute *decl* looks up[11] the declaration of the used identifier

- first in the parameter list of the enclosing method,
- if this fails, in the variable list of the enclosing class,
- and if this fails, in the class list.

In order to keep the specification compact and to illustrate some unconventional features of MAX, we specified identification by a search in the tree. We could have used classical symbol table methods as well. A more complex identification example using a mixture of both methods and mutually recursive attributes is contained in [PH93].

3.2 Specifying static semantics

This subsection presents the static semantics for BOPL, i.e. the data types of the programming language and other semantical aspects that can (easily) be treated in a static way. According to our specification technique, the static semantics comes in two parts: The first part specifies the values and locations of the language, the second makes control flow information explicit.

[11] The definition of *lookup* is given in Appendix A.

Values, objects, and variables A value in BOPL is an integer, a boolean value, the value *Null*, or an object. An object is represented by a pair (*class*, *objid*) where *class* is the class of the object and *objid* is an object identifier; to keep things simple, integers are used as object identifiers:

```
Value       =  Int | Bool | Null | Object
Object      (  Class@.class  ObjId.objid  )
ObjId       =  Int
```

A *runtime variable* (*RtVar*) is an entity that can hold a value during execution. Runtime variables play the rôle of locations. Whereas locations are simply addresses into an abstract memory, runtime variables incorporate information about the program construct that caused their creation. E.g. in BOPL, we have the following four kinds of runtime variables:

1. Instance variables: If V is a variable declaration of class C, the component of the object *Object*(C, OID) corresponding to V is called an instance variable. It will be represented by the pair *InstVar*(V, OID).
2. Actual parameters: As we need them for each method incarnation, they are represented as pairs consisting of a parameter declaration and an incarnation identifier (as with object identifiers, we use integers for incarnation identifiers).
3. Runtime variables for the anonymous parameter *self*: For each method incarnation the self object needs a runtime variable.
4. Temporaries: They are used to store intermediate values during expression evaluation.

Runtime variables are specified as follows:

```
InstVar     (  Var@.static  ObjId.objid  )
ActuPar     (  Par@.static  IncarId  )
SelfVar     (  IncarId  )
TempVar     (  Exp@.static  IncarId  )
RtVar       =  InstVar | ActuPar | SelfVar | TempVar
IncarId     =  Int
```

The usual technique for specifying actual parameters (or local variables) is to dynamically allocate them at procedure entry, i.e. to bind them to some new location. Here, actual parameters are represented by tuples consisting of the parameter declaration and a procedure incarnation. I.e. the sort *ActuPar* can be considered as a statically defined set of locations where each location "knows" to which parameter and which incarnation it belongs. Putting it the other way round, the constructor *ActuPar* is a static function mapping parameters and incarnations to locations. Thus, it is not necessary to bind parameter identifiers to newly allocated locations each time a procedure is called. The location of a given parameter for a given procedure incarnation is statically determined. Another interesting aspect is to compare the relation between classes and objects to the relation between procedures and procedure incarnations: In both cases, the former can be considered as a scheme the instantiation of which produces the latter.

The value of a variable at runtime is captured by the function

```
DYN  cont( RtVar V ) Value:  ...
```

The keyword `DYN` indicates that *cont* is a dynamic function[12]. The interpretation of dynamic functions in the initial execution states can be specified by an ordinary

[12] The property of being dynamic can be easily extracted from the evolving algebra rules (cf. Sect. 2.2), but we prefer to make it explicit.

MAX–expression. Appendix A shows the initialization for *cont*. Dynamic functions may be only used in evolving algebra rules. It should be noted here that *cont* captures in particular the states of objects by providing the values of the instance variables. E.g. the value of the first instance variable of an object $Object(C, OID)$ is

```
InstVar( C.varlist.first, OID ) . cont
```

Tasks and static control flow The combination of occurrence and evolving algebras offers a lot of flexibility in specifying language semantics. In particular, the dynamic semantics can be specified over different data structure: It can be specified (a) directly and exclusively over tree representations of programs, (b) over flow graph representations, or (c) using a mixture of both. Option (a) is the basis for classical approaches. The semantics specification of C given by Gurevich and Huggins (see [GH92]) uses option (c) by using links in the syntax tree only for specifying jumps. For the following reasons, we demonstrate option (b) here:

1. It allows to keep the *dynamic* semantics of realistic programming languages small by separating static from dynamic aspects and by focusing on the primitive dynamic concepts of the considered programming language.
2. It provides a useful formal background for areas where language specifications play an essential rôle (data flow analysis, programming logic, and interpreter generation).
3. It illustrates the power of occurrence algebras (that may be otherwise not obvious from the small BOPL specification).

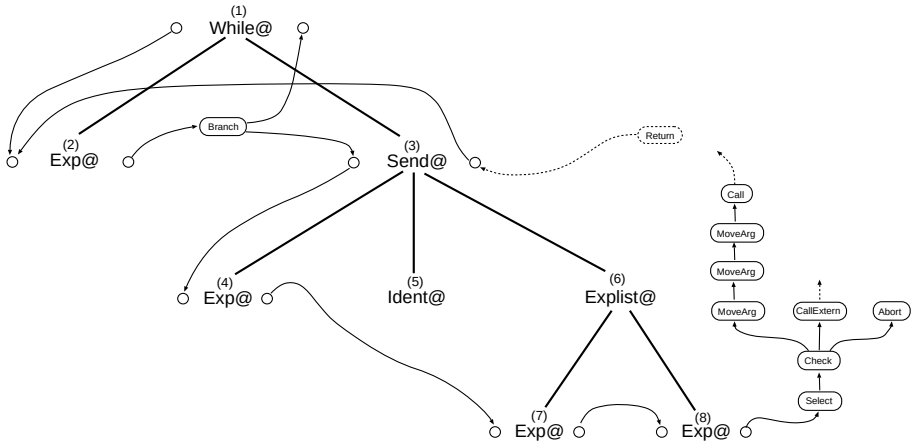
The idea of such a flow graph based specification is (i) to design a set of atomic *tasks*, (ii) to construct a flow graph the nodes of which represent tasks, and (iii) to specify the state transitions corresponding to a task. This subsection provides most BOPL specification parts for (i) and (ii). Step (iii) is given in 3.3.

For BOPL, we use eleven different tasks. Each task corresponds to one primitive semantic action: to branch; to select a method; to check the outcome of the method selection; to abort in case a method selection failed; to call a method; to call procedures that are predefined in BOPL (*CallExtern*); to move an actual parameter to the corresponding runtime variable; to move a value from one runtime variable to another; to create a new object; to return from a method; to terminate. In order to relate tasks to the syntax tree, their successors, and needed arguments, they are specified as recursive data types. E.g. a branch task needs the expression node corresponding to the condition of the branch and the successor tasks for true and false case. In this section, the following tasks are needed (Appendix A provides the omitted productions):

```
Branch       (  Exp@.cond  Task.ttsucc  Task.ffsucc )
Select       (  Exp@.site  Task.succ  )
Check        (  Task.succ )
Abort        ()
Terminate    ()
Call         (  Send@.site  )
CallExtern   (  Send@.site  )
MoveArg      (  StaticVar.src  Int.parpos  Task.succ )

StaticVar    =  Var@    | Par@    | Self    | Exp@
SnglSuccTask =  ExecPoint | Select | MoveArg | ...
Task         =  SnglSuccTask | Branch | Check | Call
             | CallExtern | Abort | ...
```

Those tasks having exactly one static successor are called single successor tasks. As shown in the production for *SnglSuccTask*, execution points before and after BOPL expressions (cf. Sect. 2.1) are considered as single successor tasks as well. They are only used to define control flow graphs over syntax trees. In the following figure,



execution points are depicted as small circles before and after expression occurrences. The upper part of the figure shows the control flow for while expressions using a branch task; the lower part illustrates the flow for a send with two parameters. The successor(s) of a task cannot always be determined statically. E.g. in BOPL, the same send expression may invoke different methods depending on the class of the current receiver object (dynamic method binding). In the flow graph, this is indicated by the dotted arrows.

Flow graphs are specified by an attribute that yields for each execution point its successor task. The successor of an execution point P depends on the context of the corresponding expression. Let N denote the node corresponding[13] to P and let us consider the case that P is the execution point after N. If N is the condition of a while (like occurrence (2) in the figure above), then the successor of P is a branch task with N as first argument, the execution point before the loop body as second and the execution point after the while as third argument. This paraphrases the first four lines of the following attribute specification:

```
(1) ATT  succ( ExecPoint P ) Task:
(2)   LET  N == P.node   IN
(3)   IF   P = N.after  THEN
(4)     IF While@<N,B> W    THEN
            Branch( N, B.before, W.after )
(5)      | While@<C,N>      THEN  C.before
           ....
(6)      | Send@<N,_,<E,*>> THEN  E.before
(7)      | Send@< RCV, ID, <*,N> EL > S  THEN
(8)         Select( S,  Check( MoveArg( RCV, 0,
(9)         moveargtasks( N, numchilds(EL), Call(S) ) ) ) )
        ...
```

The most interesting case is a send with parameters (lines 7–9), because it demonstrates how to construct task structures that are not directly related to the syntax tree:

[13] We write *P.node* instead of *subterm*$(1, P)$.

The successor of P is a *Select*-task selecting the method to be called, followed by a *Check*-task checking the outcome of the selection (see below). If the selection is successful, the actual parameter values are moved to the actual parameter variables. This is done by a sequence of *MoveArg*-tasks, the first element of which moves the receiver object. The rest of the task sequence moving the other parameters is generated by the function *moveargtasks*:

```
FCT  moveargtasks( Exp@ ACTP, Int POS, Task SUCC ) Task:
  IF  POS = 1  THEN   MoveArg( ACTP, 1, SUCC )
  ELSE   moveargtasks( ACTP.lsib, POS-1,
                       MoveArg(ACTP,POS,SUCC) )
```

The full specification of *succ* is given in Appendix A.

3.3 Specifying dynamic semantics

This section illustrates the specification of dynamic semantics for sequential programs by evolving algebra rules. Nondeterminism and parallelism are discussed in the following section.

States The states of program executions are defined by a couple of dynamic functions (cf. 2.2). The input is described via external functions/variables. In BOPL, the only input to execution is the program itself. It is used to initialize the dynamic variable *ct* to the execution point before the main expression of the program. *ct* records the **c**urrent **t**ask, i.e. the task to be executed next:

```
(1)   EXT  prog     ()  Program@
(2)   DYN  ct       ()  Task:               prog.mainexp.before

(3)   DYN  ci       ()  IncarId:            0
(4)   DYN  rtrnto   ( IncarId IC ) RtTask:  nil

(5)   DYN  resofslct()  MethodElse:         nil
(6)   DYN  newincar ()  IncarId:            1
(7)   DYN  newobjid ()  ObjId:              1

(8)   RtTask        (  Task.task  IncarId.incar  )
(9)   MethodElse    =  Method@  |  CallExtern  |  Abort
```

The dynamic variable *ci* (line 3) records the **c**urrent **i**ncarnation number. The dynamic function *rtrnto* (line 4) records for each method incarnation the (task,incarnation)-pair it has to **re**t**urn to**; such a pair is called a runtime task (see line 8). The other dynamic variables play an subordinate rôle: *resofslct* records the **res**ult **of** the method **se**l**ect**ion; this is either a declared method, a call to an external procedure, or an abort, if method selection fails. *newincar* and *newobjid* are counters used to generate new incarnation and object identifers where needed.

Dynamics State transitions are defined by evolving algebra rules. Essentially, there is at least one rule for each task having a successor task. In addition to this, there may be rules specifying updates for classes of tasks; e.g.:

```
IF  isSglSuccTask(ct) THEN   ct :=  ct.succ   FI
```

A classical rule is that for a branch:

```
IF  isBranch( ct )    THEN
  IF  cont( TempVar(ct.cond,ci) ) = true  THEN  ct := ct.ttsucc
                                          ELSE  ct := ct.ffsucc  FI FI
```

Finally, we discuss the most difficult rule, the rule for the call task. The called method has to return to the execution point after the send expression with the current incarnation. The current task is set to the execution point before the body of the selected method; the current incarnation to the new incarnation:

```
IF  isCall( ct )      THEN
  rtrnto(newincar) :=  RtTask( ct.site.after, ci)
  ct               :=  resofslct.body.before
  ci               :=  newincar
  newincar         :=  newincar + 1                    FI
```

This completes the introduction to the BOPL specification. The remaining rules can be found in Appendix A.

3.4 Specifying nondeterminism and parallelism

This section illustrates how nondeterminism and parallelism can be specified in our framework. As example, we extend BOPL by a construct that provides parallel execution of BOPL expressions; i.e. we add

```
Parallel     (  Exp  Exp  )
```

to the abstract expression syntax. As the *Parallel*-expression can occur within recursive methods, there may be a statically unbounded number of execution threads of a program. In summary, we obtain a fairly complex language with unbounded parallelism, recursive procedures, and common object space.

Syntactically tiny language extensions can cause large changes to the semantics specification whereby the extent of the needed changes strongly depends on the applied specification technique. E.g if a sequential language is specified in a structural operational style, adding parallelism can usually not be specified as a mere extension, but entails various changes of the sequential specification. In contrast to this, [MM94] demonstrates that an action semantics specification of a simple ML–like sequential language only needs to be *extended* when concurrency is added to the language; the specification of the sequential language remains untouched. As shown below, the specification of BOPL with *Parallel*-expressions can as well be obtained without modifying the described sequential specification. The extension is based on the specification of a scheduler, i.e. it depends on the control over global execution states. Compared to the elegant communicative facet of action semantics, this is certainly a fairly primitive technique (in particular because the missing module concept for evolving algebras makes it necessary to use the whole sequential specification in a rule of the extended specification). On the other hand, the global and direct control of execution can be helpful in the development of tools (e.g. for debuggers).

We describe parallelism as interleaved execution controlled by a scheduler. The scheduler manages a **s**uspended **t**ask **l**ist recorded by the dynamic variable *stl* which is initially empty. Execution of tasks and the scheduler is alternated. Alternation is controlled by the dynamic boolean variable *schedule_turn* (*schedule_turn = true* means that it is the turn of the scheduler).

```
DYN  stl          ()  RtTaskList:   RtTaskList()
DYN  schedule_turn()  Bool:         false
```

The specification of sequential BOPL is extended in two steps. In the first step, the tasks for forking execution threads and synchronizing them are specified. The second step specifies the scheduler. Figure 1 shows the control flow for a *Parallel*-expression.

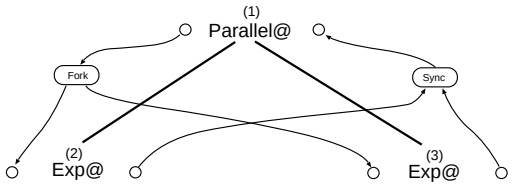


Fig. 1. Control flow for expression *Parallel*

It uses the tasks *Fork* and *Sync*. The *Fork*-task simply sets the current task to its first successor and appends its second successor to the list of suspended runtime tasks (the scheduler chooses the next task to be executed from the current task or the suspended tasks; see below). Synchronization is handled via a dynamic boolean function *waiting*: If a *Sync*-task is reached in state waiting (i.e. *waiting*(*SYNC*) = *true*), it resets its waiting state to *false* and sets the current task to its successor. Otherwise it sets its waiting state to *true* and sets *ct* to the *Skip*-task which will be ignored during scheduling.

```
Fork       (  Task.firstsucc  Task.secondsucc )
Sync       (  Task.succ  )
Skip       ()

DYN  waiting( RtTask RT ) Bool: false

IF  isFork( ct )      THEN
  ct  := ct.firstsucc
  stl := append( RtTask(ct.secondsucc,ci), stl )      FI

IF  isSync( ct )      THEN
  IF  waiting( RtTask(ct,ci) ) = true  THEN
        waiting( RtTask(ct,ci) ) := false
        ct := ct.succ
  ELSE  waiting( RtTask(ct,ci) ) := true
        ct := Skip()                          FI  FI
```

Altogether the dynamic semantics rule for task execution looks as follows:

```
IF  NOT schedule_turn    THEN
  schedule_turn  := true
  Rules of sequential BOPL
  Fork–Rule
  Sync–Rule
FI
```

In the specification of the scheduler one aspect needs some care. Task sequences from a select to a call use the auxiliary dynamic variable *resofslct*. As *resofslct* may be overwritten by other tasks, such *sequences* have to be considered as atomic; i.e. tasks in these sequences are executed *immediately* without scheduling a suspended task in between:

```
Immediate   =  Check | MoveArg | Call

EXT  choose( RtTaskList ) RtTask

IF  schedule_turn  AND  NOT isTerminate(ct) AND  NOT isAbort(ct)  THEN
  schedule_turn := false
  IF  NOT isImmediate(ct)  THEN
      ct    :=  choose( newtl(ct,ci,stl) ).task
      ci    :=  choose( newtl(ct,ci,stl) ).incar
      stl   :=  delete( choose( newtl(ct,ci,stl) ), newtl(ct,ci,stl) )
  FI
FI
```

The function *newtl* adds the runtime task (ct, ci) to the suspended task list *stl* if *ct* is not the *Skip*-task; otherwise it returns *stl*. The external function *choose* expresses the choice from a list. In one execution step, *choose* yields the same result if applied to the same list; but in different execution steps the choices may vary nondeterministically (cf. Sect. 2.2 for the semantics of external functions). In general, the MAX System supports two kinds of external functions. An external function is either an external variable of sort *S* or a unary function from a list sort to the corresponding element sort. An external function of the first kind yields in each computation step an arbitrary element of *S*, if *S* is nonempty, otherwise *nil*. An external function of the second kind yields in each computation step an arbitrary element of the actual list parameter l, if l is nonempty, otherwise *nil*. (In addition to that, external functions can be provided by the user via the interface to C; cf. Sect. 5.)

3.5 Specifying optimizations

The presented framework is not restricted to language specification, but can be used as well for other purposes. This section investigates a nontrivial optimization problem from compiler construction: common subexpression elimination in basic blocks. With classical attribution mechanisms, common subexpression elimination is difficult to handle, because it entails the transition from trees to graphical structures. We use the example to demonstrate some more advanced attribution techniques. The following simple syntax for basic blocks is sufficient for the purposes here (extensions to more complex expressions is straightforward):

```
BasicBlock *  Assign
Assign     (  Ident  Exp  )
Exp        =  Var  |  Add
Var        (  Ident  )
Add        (  Exp.left  Exp.right )
```

The basic idea of common subexpression elimination is to determine equivalent expression occurrences. The value of an expression occurrence during execution depends on the values of the variables at the beginning of the execution of the basic block. Thus, it can be represented by a term of sort *Exp* where variables in such terms represent their values at the beginning of the basic block. Calling such terms *value expressions*, two expression occurrences are equivalent if they have the same value expression. Consider e.g. the following basic block (Fig. 2 shows this basic block in graphical representation):

b := (a + a) ; c := a ; a := ((c+c) + b)

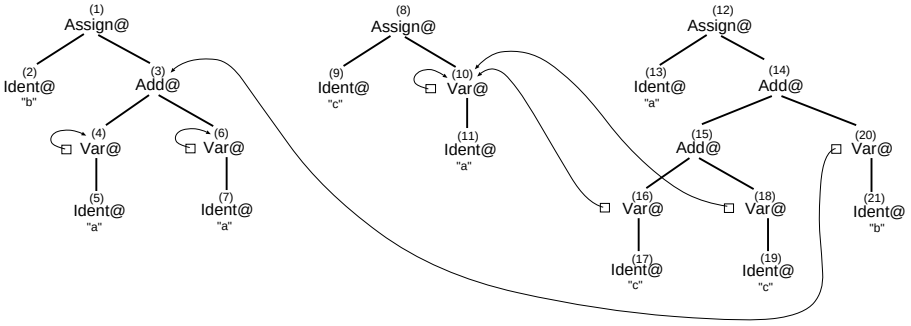


Fig. 2. Basic block with attribute *def_exp* depicted as □

The value expression of occurrence (15) is “a+a”; “c” is not used in this value expression, because the value of variable c is changed by the second assignment. As the value expression of (3) and (20) is “a+a” too, occurrences (3), (15) and (20) are equivalent.

To focus on the central techniques, we assume here an attribute *def_exp* that maps each variable occurrence to the right hand side of the defining assignment or to itself if no such assignment exists in the basic block. This is illustrated in Fig. 2. Attribute *def_exp* can be specified with the same technique as *decl* in 3.1. Based on *def_exp*, we can specify the value expression corresponding to an expression occurrence as follows:

```
(1) ATT  val_exp( Exp@ E ) Exp:
(2)   IF  Add@<LE,RE> E     THEN  Add( LE.val_exp, RE.val_exp )
(3)    |  Var@<ID> E        THEN
(4)      IF  E.def_exp = E  THEN  Var(ID.term) ELSE E.def_exp.val_exp
(5)   ELSE  nil
```

For each class of equivalent expression occurrences, we select one occurrence as *representative*. This can be done as follows: 1. Specify some linear order on all expression occurrences of a basic block; we assume an attribute *pred* yielding for each expression occurrence its predecessor if it exists; otherwise nil. 2. Specify two mutually recursive attributes *repr* and *available_reprs* where *repr(E)* yields the selected representative for expression occurrence E and *available_reprs(E)* yields the list of selected representatives for the predecessors of E:

```
(6)   ExpOccList  *  Exp@

(7)   ATT  repr( Exp@ E ) Exp@:
(8)     LET  REPR == lookup( E, E.available_reprs )  IN
(9)     IF   REPR = nil    THEN  E
(10)                       ELSE  REPR

(11)  FCT  lookup( Exp@ E, ExpOccList AVREPR ) Exp@:
(12)    IF  isempty( AVREPR )                 THEN  nil
(13)     |  AVREPR.first.val_exp = E.val_exp  THEN  AVREPR.first
(14)   ELSE  lookup( E, AVREPR.rest )

(15)  ATT  available_reprs( Exp@ E ) ExpOccList:
(16)    LET  EP ==  E.pred  IN
(17)    IF  EP = nil        THEN   ExpOccList()
```

```
(18)     |  EP = EP.repr    THEN   append( EP, EP.available_reprs )
(19)    ELSE  EP.available_reprs
```

The common subexpression information is used for code generation. E.g. to generate three-address code, let us assume that the function *mktmp* yields for each expression occurrence a temporary name. If an *Add*-expression occurrence E is a representative, generate the instruction "E.mktmp := E.left.repr.mktmp + E.right.repr.mktmp"; otherwise no code has to be generated for E. If abstract syntax is as well used to represent code, further optimizations can be specified by attributing the code trees. E.g. based on life variable information, redundant assignments can be eliminated.

4 Comparing language specification frameworks

This section compares the presented techniques to other frameworks that are based on language specifications. It is divided into two parts. The first part discusses the relation to frameworks in which language specification aspects are central; the second part focuses on attribute grammar based frameworks aiming mainly on generation aspects.

4.1 General language specification methods

This subsection compares the presented approach to action semantics ([Mos92]) and natural semantics ([Kah87]). W.r.t. this comparison, denotational semantics behaves similar to action semantics, because both approaches specify language semantics as a mapping from syntax trees to some appropriate semantic domains. Thus, we mention denotational semantics only together with aspects where it behaves differently from action semantics. Natural semantics is considered as a representative for structural operational methods. For brevity, we do not consider methods based on conditional equations here (cf. [Kli90] for a powerful realization); for most points of the comparison, they behave similar to structural operational methods. The comparison starts with a general discussion of specification properties. Then it considers the three design goals stated in the introduction.

General comparison Before the comparison, we shortly sketch action and natural semantics. An action semantic description consists of three parts: specification of context–free syntax, of semantic entities, and of inductively defined semantic functions mapping syntax trees to semantic entities. In contrast to denotational semantics, the semantic entities are specified by composing and extending predefined actions (primitive actions and action combinators). These predefined actions encapsulate basic semantic language concepts.

A natural semantics description essentially consists of sort and constructor definitions for abstract syntax and auxiliary data types and a set of natural deduction inference rules. Inference rules specify semantic relations structurally over the abstract syntax, i.e. the semantics of a program part P in a given context is specified based on the semantics of the components of P. Different semantic relations can be specified: e.g. computation relations where each inference step corresponds to one

computation step; or evaluation relations formalizing the relation between a configuration and a final result; or translation relations expressing program translations. Moreover, natural semantics supports additional flexibility by logical variables and unification.

Action semantics, natural semantics, and the presented approach represent three different classes of operational specification methods (cf. introduction). Action and natural semantics inductively specify the semantics over the syntax trees. In particular, they provide a semantics of program parts. Our approach specifies a relation between global states where a global state contains the whole program. Inductive methods are advantageous if only simple environment information is necessary and control flow essentially follows the tree structure. In our approach we have to explicitly specify control flow. This advantage gets smaller or vanishes if environments become complex. E.g. in BOPL the environment has to contain information about all classes and all methods to handle method selection. I.e. the environment is essentially a recoding of the whole program which can become more complex than using an appropriate enrichment of the program itself (e.g. self–referential bindings are needed in environments whereas in our approach an attribute like *decl* given in Sect. 3.1 is sufficient). As proof techniques, inductive methods support structural and computational induction whereas global state methods are restricted to computational induction.

Action semantics provides an elegant notation, a large number of predefined actions, and a powerful module concept. In natural semantics and our approach there are much less predefined entities. The language specifier has to work essentially with the basic constructs offered by these frameworks. The tradeoff here is between

- powerful predefined constructs with a fairly elaborate semantics enabling reusability and a kind of standardization and
- simple basic constructs that are quicker to learn and to understand in all details.

Due to a well–structured semantics of actions and a nice organization into different facets, action semantic specifications are very extensible. In our approach, extensibility w.r.t. parallelism comes from the use of evolving algebras (i.e. global state and simultaneous, independent transition rules). In natural semantics, adding parallelism to a language can cause larger modifications, because the signature of the semantic relation and with it most rules may change. A similar situation can occur in denotational semantics with the semantic domains.

Validation Validation means to check whether a specification corresponds to the intentions of the specifier. For language specifications, essentially two validation techniques exist: consistency checking and testing by generated interpreters. In the three approaches compared here, consistency checking is mainly based on sort checking. In addition to that, the presented approach tries to test whether dependencies between attribute instances are acyclic. Of course, this test is essentially restricted to cases where specified attributes can be translated to classical attribute grammars (cf. 4.2); e.g. the absence of cycles is automatically checkable for BOPL, but not for the optimization example. Moreover, half–automatic consistency checking techniques can be applied; e.g. proving that a semantic relation of a natural semantics specification is a function, or that an evolving algebra specification never reaches a contradicting update (in most cases such a proof is as simple as for BOPL). However, these checks can guarantee neither that a specification is formally sensible (e.g. in action semantics,

data definitions may be logically inconsistent) nor that it reflects the intentions of the specifier. In particular for the second aspect, flexible and efficient interpreters can be very helpful.

Comparing the three approaches w.r.t. interpreter generation, it is important to keep the original intentions of the approaches in mind. Action semantics focuses on providing optimal language specification properties (interpretation of actions is discussed in [Wat94]). Natural semantics provides a general and flexible mechanism to inductively specify relations over abstract syntax and make them operational. Natural semantics specifications are supported by the Centaur system (cf. [BCD+89]). Our framework was designed in such a way that specification techniques are supported enabling or simplifying the generation of efficient interpreters. The basic idea is to distinguish between static aspects (like identification, typing, context checking, control flow) and dynamic aspects and to include into the static part all aspects that can (easily) be done once and for all during program interpretation. To illustrate this, let us consider the interpretation of the following block structured program[14]:

```
(1)       var int a := 0;
(2)       var int b := 0;
            ...
(10)      while a + b < 10000  do
(11)           var int a := 1;
               ...
(15)           if ...  then  goto LABEL  end;
               ...
(20)           b := b+1
(21)      end
(22)      LABEL: ...
```

If the interpretation is based on an inductive language specification (like action or natural semantics), each time the loop body is entered a new location for the local variable a in line (11) is allocated, initialized to 1, and the environment is updated accordingly. In the presented framework, an attribute *decl* from used variable occurrences to declared occurrences would be specified (cf. Sect. 3.1); as this is static information, the generated interpreter evaluates the attribute values once before starting the execution of the program (just as it parses the program once before execution). The dynamic semantics would be based on a dynamic function from variables to values; i.e. dynamic allocation of locations is not necessary. Thus, each time the loop body is entered, the interpreter only has to initialize a in line (11) to 1 (in our implementation, this corresponds to dereferencing a pointer and updating an array).

Another example where the distinction between static and dynamic aspects on the specification level and the global state method contribute to the efficiency of generated interpreters is control flow. E.g. the relation between the goto statement in line (15) above and its target in line (22) is static. Specified as an attribute it is computed once before execution. During execution a jump simply means to dereference the computed link (not even a change of environment is necessary).

Development of language–specific software Translational specification methods like action and denotational semantics are good for semantics–directed compiler generation. The translation of the source programs to the semantic entities eliminates the

[14] A more interesting example would be to compare what operations are necessary to interpret a method call in BOPL according to our specification and according to an inductive specification; but this is beyond the scope of this paper.

peculiarities of the specified language. Optimization and code generation is based on the constructs of the specification language. Whereas this is advantageous for semantics–directed compiler generation, it is bad for the development of interactive tools or in cases where computed information has to be related back to source programs (browsers, debuggers, simulators, pretty printers, profilers, etc.). In structural specification methods, at least part of the program belongs to an execution state. And additional techniques based on occurrences (cf. [Ber90]) allow to relate the current execution state to the whole original program.

In our framework, the access to the whole program and all attributes in any phase of execution is directly supported. The generated software components provide interfaces for all specified data types, functions, and attributes. This simplifies e.g. the development of animation tools. In particular, the clear distinction between static and dynamic specification parts enables the generation of program browsers supporting the inspection of attributes and allowing to follow links in the tree, e.g. from a used occurrence to a declared occurrence. By adding some dynamic functions and evolving algebra rules (similar to the scheduler in Sect. 3.4), simple source level debuggers can be generated. Other applications are discussed in Sect. 5.2.

Simple semantics Quite a number of frameworks and techniques have been developed for the specification of programming languages and for supporting tool generation. Most of these frameworks are based on some kind of formal models. But, the degree of formality is very different: e.g. action semantics specifications are founded in all details by unified algebras and structured operational semantics, whereas in attribute grammar based specifications the formal underpinning is restricted to the attribute grammar part, leaving out the data types, semantic functions, and the notions to specify dynamic semantics. From a formal point of view, our framework supports only three constructs: productions to define occurrence algebras, function definitions, and evolving algebra rules. An embedding of these constructs into logical frameworks to support formal proofs of program properties should be fairly straightforward, although the amount of technical work should not be underestimated (for an axiomatic characterization of occurrence algebras, one has to consider *occ* and the ci's as constructors and define an appropriate equivalence relation on the generated terms; the embedding of evolving algebras is considered in [HDP96]). In [BF95], an embedding of natural semantics into the specification language of the Coq proof assistant is investigated.

4.2 Attribute grammar based specifications

Attribute grammars are a well developed technique for the specification of static aspects of programming languages and widely used for the generation of language–specific tools (cf. [DJL88]). This subsection explains the relation between classical attribute grammars and our attribution technique. Then, it compares our framework with attribute grammar extensions, in particular with those supporting the specification of dynamic semantics.

Relation to classical attribute grammars Each attribute grammar can be mechanically translated into a MAX specification. The context–free grammar is expressed by a set of productions. Attributes are translated to MAX attributes such that the domain sorts

correspond to the attributed nonterminals and the range sorts correspond to the types of the attributes in the AG. Attribute equations are translated to the bodies of MAX attributes such that each equation corresponds to one case in a conditional expression. The conditions are used to express the different contexts, in which an attribute can occur (cf. the attribute specification of *val_exp* in 3.5). The main structural difference between the specification techniques is that in attribute grammars all equations for one production are grouped together, whereas in MAX all right hand sides of the equations for one attribute are grouped together in the attribute body. An example of such a translation is given in Appendix B.

A mechanical translation from MAX attribute specifications to attribute grammars is in general not possible. An attribute grammar does not allow to refer to syntax tree nodes in attribute equations; thus, attributes like *decl* or *repr* cannot be specified. In particular, the declarative specification of graphical structures as demonstrated by the control flow graphs in Sect. 3.2 is not possible.

Relation to attribute grammar extensions

This paragraph discusses the relation of the presented framework to other attribute grammar extensions.

Remote attribute access Almost all modern attribute grammar based systems provide remote attribute access in some form. Copy rules were already used in [KHZ82]. The Cornell Synthesizer Generator [RT89] provides specification techniques for upward remote attribute access. In addition, it allows references to attribute instances; but such references can be used only in a very restricted way and the system does not guarantee that attribute values are computed before accessed through such a reference. The ability of specifying references to other nodes of the syntax tree is also available in Door Attribute Grammars, an amalgamation of attribute grammars and object–oriented techniques (cf. [Hed91]). In MAX remote attributes can be accessed by using nested patterns, the standard tree functions (*parent* etc.), or through arbitrary functions or attributes yielding remote occurrences (e.g. in 3.5, attribute *val_exp* uses in line (4) a remote reference through attribute *def_exp*). Furthermore, MAX allows to pass nodes/occurrences as values and reference their attributes whenever needed: E.g. in the specification of *lookup* in 3.5, line (13), the attribute *val_exp* is accessed.

Coupling attributions Conceptually, attribute coupling means to use values provided by one attribution as syntax trees in other attributions (cf. [Gie88]). E.g. the intermediate code generation sketched at the end of 3.5 can be specified by an attribution, the result of which is used as input for an attribution eliminating redundant assignments, the result of which is used as input to machine code generation. Most modern systems provide attribute coupling in some way (cf. [JP88, RT89]). In its most general form (cf. [VSK89]), one even can intertwine syntax tree construction and attribution by using attribute values to define parts of the syntax tree of the same phase. In MAX, attribute coupling reduces to function/attribute composition.

Attribute grammars and dynamic semantics Whereas the above extensions to attribute grammars were only invented to increase flexibility and modularity for the specifi-

cation of static semantics, other extensions add the capability to specify dynamic aspects. Most of such work aimed to provide a basis for the generation of interactive editors: In [Kai89] dynamic semantics is described by so–called action equations that are formulated based on a fixed set of actions and events. Walz and Johnson (cf. [WJ95]) developed so–called inductive attribute grammars. Both approaches focus on incremental evaluation. This leads to language specification techniques that are difficult to handle, in particular if it comes to nondeterminism or parallelism.

In summary, the presented framework provides a high degree of specification flexibility and expressibility using a fairly simple formal model. Of course, there is a price to pay: 1. If the flexibility is exploited, the system can no longer guarantee that the attribute dependencies are acyclic. In such cases, it will report cycles at runtime. 2. The framework in the presented form does not provide specialized features to support e.g. compiler subtasks like identification (compare this to [GHL+92]). On the other hand, occurrence algebras allow to solve many problems in a direct and elegant way based on the provided rich data types. E.g. for the specification of common subexpression information it is very helpful that constructor terms (for value expressions) and occurrences (for representatives) are available.

5 Implementation aspects and experiences

This section describes the MAX System, an implementation of the presented framework, and summarizes experiences gained with MAX.

5.1 Implementation

After a short implementation overview, this section sketches attribute evaluation and the implementation of evolving algebras.

Implementation overview

From specifications like that of Sect. 3, the MAX System generates interpreters. In the simplest case, a generated interpreter takes a term representing the abstract syntax tree of the input program, performs attribute evaluation, and starts execution of the program according to the dynamic semantics. The abstract syntax term is usually provided by a parser, but may as well be the result of a previous attribution phase. This way, attribute coupling is supported. In order to generate interpreters, the MAX System essentially performs three tasks:

1. It type-checks the specification.
2. It performs some analyses on attributes.
3. It generates code for the specified data types (including occurrences), for functions, for attribute evaluation and access, and for execution of the evolving algebra rules.

The type checker determines the least sort for each expression in the specification. Sort information is used to report specification errors and warnings, and to generate dynamic sort checks where needed. In the following paragraphs we focus on the implementation of attribute evaluation and evolving algebra rules.

Attribute evaluation

In specifications, we distinguish functions and attributes (e.g. *succ* was specified as an attribute). Whereas the semantics of attributes and (unary) functions are the same, the implementation is different. Needed attribute values are computed once and cached for later use, e.g. during interpretation. Extending classical attribute grammars, our framework allows attribute dependencies that arise during attribute evaluation, i.e. we cannot even determine all attribute dependencies knowing the syntax tree. We call such attribute dependencies *dynamic*. Consider e.g. line 4 of attribute *val_exp* (Sect. 3.5): If a variable occurrence *VO* has a defining expression in the basic block, the value expression of *VO* depends on the value expression of *VO.def_exp*. And *VO.def_exp* is determined by attribute evaluation and cannot be found out by looking at static attribute dependencies. Attributes like *def_exp* and *val_exp* may be even mutually recursive like e.g. the attributes *available_reprs* and *repr*.

Essentially, attributes are implemented as recursive functions with result caching: If attribute *attr* is called with argument *N*, it checks whether *attr(N)* is already computed; if not, it computes, caches, and returns the value; otherwise, it returns the cached value. This technique is enhanced by a mechanism detecting cyclic dependencies between attribute instances. This algorithm is essentially an adaption of the one proposed in [Jou84]. Notice that it is different from that of Katayama [Kat84]. The recursion is not controlled by the tree structure, but directly by the attribute dependencies.

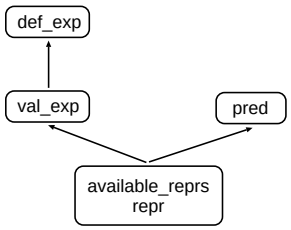
If we compare the time needed to manage the control flow per evaluated attribute value, two reasons cause the MAX attribute evaluator to perform slower than a statically generated tree-walk evaluator: 1. A function call is needed for each attribute evaluation. 2. Whereas the tree-walk evaluator "knows" its context in the tree, the first task in evaluating an attribute instance in MAX is to determine its context. This is usually done via patterns (cf. the specification examples above). By using an optimizing implementation of conditional pattern matching, the resulting overhead can be neglected.

Of course, we have to pay a certain price for the simple, but very powerful attribute evaluation technique: In the worst case, e.g. if we start attribute evaluation with an attribute instance that depends on every other attribute instance, we would need stack space proportional to the depth of the attribute (instance) dependency DAG. In order to make this evaluation technique feasible[15] for large programs with hundred thousands of attribute instances, we manage the attribute evaluation stack on the heap and reduce the (temporarily) needed space by partitioning the attributes and approximating the dependencies.

Partitioning attributes The described evaluation technique leads to call–by–need–computation of attribute values. In order to reduce space consumption for the attribute evaluation stack, we *refined* the call-by-need strategy as follows:

– During generation time: The attributes (not the attribute instances) are grouped into a partial order of *partitions* so that an attribute only depends on attributes in its own partition P or in partitions being before P in the partial order. E.g. for the common subexpression example in 3.5, we get the following partitions where $P_1 \rightarrow P_2$ means that P_1 depends on P_2:

[15] As target machines, we consider here modern PC's and workstations.



– During evaluation time: Let *attr* be an attribute in partition P, $P_1, .., P_q$ the partitions on which P depends, n be an occurrence, and let t abbreviate *term*(*root*(n)). If the value of *attr*(n) is needed, but not yet computed, do the following: compute the values of all attributes in partitions $P_1, .., P_q$ for all occurrences of t; then compute *attr*(n). The worst case size of the attribute evaluation stack in number of frames is then bound by the maximum of the sizes for each partition; the size for a partition Q with attributes $a_1, ..., a_z$ having parameter sorts $ps_1, ..., ps_z$ is given by $\Sigma_{j=1}^{z} a_j * \#(t, ps_j)$ where $\#(t, ps_j)$ denotes the number of occurrences of t with sort ps_j. As the number of attributes in one partition is typically one or two, rarely greater than 5, the size of a partition is rarely greater than the number of tree nodes.

The rational for evaluating all attributes in partitions on which the current partition depends is based on the observation that in most cases the need of one attribute value implies the need of all other attribute values for the tree under consideration.

Approximating dependencies Using the stack mechanism, any attribute evaluation order is correct, but consumes a different amount of stack space. If the evaluation order respects the dependencies of attribute instances, no stack space is needed, whereas a bad evaluation order may lead to the worst case size from above. The better the evaluation order approximates the attribute dependencies, the smaller the needed stack space. By a rather complex attribute dependence analysis (cf. [PHM94]), the MAX System determines dependence information. This information is used to generate for each tuple or list production a plan that describes the order in which children nodes are visited and attribute instances of the parent node are evaluated. If e.g. attribute instances can be evaluated in a left–to–right depth–first tree traversal (L–attribution), the information is sufficient to generate an optimal evaluation order. In general, the generated evaluation order only approximates the real attribute dependencies. The strength of the attribute dependence analysis compared to classical attribute analysis is that it can handle attribute access through node–valued functions/attributes. E.g. for the common subexpression example in Sect. 3.5, the system generates optimal evaluation orders for all partitions except for the one containing *val_exp*.

Evolving algebra implementation The implementation of evolving algebras has two aspects: the representation of the dynamic functions and the stepwise execution of the rules. In our current implementation, we only allow dynamic variables and unary[16] dynamic functions. Dynamic functions are implemented by arrays. Depending on the

[16] Functions with greater arity can be handled by defining a tuple sort for their parameter sort tuple.

parameter sort we use different kinds of encoding and hashing techniques. In particular, occurrences are represented by indices, a by-product of their implementation.

As all rules of an evolving algebra have to be executed in parallel, we divide each computation step in two substeps: Compute all arguments and right hand sides of updates with true guard and then perform the updates in some order. To avoid testing the guards twice, we keep track of the updates that have to be performed in a list of pointers to parameterless procedures; each of these procedures performs the set of updates corresponding to one node in the tree of conditionals that reflects the nesting of rules in the evolving algebra.

5.2 Pragmatics and experiences

The MAX System is implemented in ANSI C and runs under different UNIX versions including Linux, HPUX, Solaris, and SunOS. A central design aspect of the MAX implementation was the development of simple and flexible interfaces to C. Whereas this is of minor importance from a specification point of view, it is of great importance for the construction of realistic systems from specifications:

- Interaction with environment: E.g. it is rather straightforward to use MAX specifications as a basis for visualization and testing tools (class browsers, source level debuggers). But this requires a close interaction between the data structures managed by MAX and graphical user interfaces. In particular, attributes and the execution states must be accessible.
- Refinement: Our approach to language implementation is to start with a specification designed for good readability and then stepwise refining it towards efficiency (cf. [PH94b]). Whereas many steps can be made in MAX itself, in the end efficiency critical aspects have often to be implemented in languages closer to the machine. E.g. the MAX type checker was developed in MAX; for the final version, we substituted the handling of sets of sorts by a C implementation based on bit vectors.

MAX provides for every function and attribute a corresponding C function procedure and allows to use C function procedures in specifications.

As another pragmatic feature, MAX supports the specification of context conditions in a natural and convenient way based on predicate logic. Especially during language design time, such high–level executable specifications of context conditions proved to be very useful. A context condition consists of a universal quantification part and a boolean expression. The quantification is described using patterns. E.g. the following context condition specifies that a used identifier in BOPL has to be declared:

```
CND  UsedId@ U:  U.decl # nil
```

A more sophisticated use of patterns in context conditions is shown in Appendix A. In order to issue error messages, string expressions can be attached to context conditions.

Until now, the MAX System has been used for four realistic size applications, for a number of small up to mid-size applications, and in compiler construction courses. The realistic applications are: (a) the bootstrap of the system itself; (b) a front-end for a PASCAL subset; (c) an object–oriented programming environment; (d) a C specification. During bootstrapping, we developed and refined the MAX language; for that the prototyping facilities proved to be very helpful for language design and documentation. The PASCAL front-end was mainly written in order to compare MAX with

a conventional AG system. We compared it to the CMUG System, a slim successor of MUG2 (cf. [GGMW82]). In summary, the MAX specification is about 2.4 times smaller than the CMUG specification; on the other hand, the front-end generated by MAX is about 2 times slower than the CMUG counterpart (1.1s for a thousand line PASCAL program measured on a HP9700[17]). The programming environment demonstrates how language–specific software can be developed in MAX based on language specifications (see [Mül95]). It supports a subset of the object–oriented language SATHER extended by executable annotations. The annotations provide a good documentation facility and proved to be very helpful for testing programs. The most interesting application from a language specification view point is the C specification (see [Bic95]). With the exception of floats, unions, and variable initialization, the current version completely covers ANSI C: syntax and semantics. The design of the specification was led by three goals:

1. We tried to keep the dynamic semantics as small as possible resulting in an evolving algebra specification of less than two pages.
2. The semantics expresses interleaved, nondeterministic execution of expressions and captures de-/allocation of local variables correctly, even for jumps into blocks.
3. We factored out the implementation dependent aspects of C. In particular, the specification uses an abstract memory model that is specified as an abstract data type (not within MAX).

6 Conclusions

We proposed a combination of occurrence algebras, recursive first–order functions, and evolving algebras as a language specification framework. The framework is supported by the MAX system that generates interpreters from specifications. This way, languages can be prototyped during design time and language–specific software can be developed based on generated components.

Occurrence algebras were developed by the author as a result of integrating attribute grammars and term algebras; evolving algebras were introduced by Y. Gurevich. Even though the formal foundations are comparably simple, the framework is sufficiently expressive to specify all aspects of modern programming languages including nondeterminism and parallelism. To illustrate this, we presented the specification of an object–oriented language with recursive methods. By extending this specification, we showed how parallelism can be handled. Some advanced attribution techniques were demonstrated with a specification of common subexpression elimination.

Based on the framework, we implemented the MAX System for prototyping specifications. From language specifications, MAX generates interpreters, but it can be used as well to develop compilers or other language–specific programming tools. We made encouraging experiences with refinement of specifications. This is mainly due to the flexibility of the framework and the system. Specification refinement can be used to improve the efficiency of the generated code. Future development steps include the design of a module concept for the MAX specification language and the embedding of specifications into a logical framework.

[17] The timings given in [PH93] are measured with an older MAX version.

Acknowledgements. We gratefully acknowledge the helpful comments of the anonymous referees. Thanks go to P. Müller, B. Bauer, S. Schreiber, and B. Reichel for their critique on earlier versions of this paper.

References

[AP94] I. Attali, D. Parigot: Integrating natural semantics and attribute grammars: The minotaur system. Technical Report No. 2339, INRIA, September 1994.

[Ast91] E. Astesiano: Inductive and operational semantics. In *Formal Description of Programming Concepts*, (IFIP State–of–the–Art Report), pp. 51–136. Berlin, Heidelberg, New York: Springer 1991.

[BCD+89] P. Borras, D. Clement, T. Despeyroux, J. Incerpi, G. Kahn, B. Lang, V. Pascual: CENTAUR: The system. In *ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments*, SIGPLAN Notices 24(2), 1989.

[Ber90] Y. Bertot: Occurrences in debugger specifications. Technical Report No. 1350, INRIA, December 1990.

[BF95] Y. Bertot, R. Fraer: Reasoning with executable specifications. In P. D. Mosses, M. Nielsen, M. I. Schwartzbach, editors, *TAPSOFT'95*, LNCS 915, pp. 531–545. Berlin, Heidelberg, New York: Springer 1995.

[Bic95] K. Bichler: Specification of the C programming language. Master's thesis, Technische Universität München, 1995. (In German).

[BR92] E. Börger, D. Rosenzweig: A simple mathematical model for full Prolog. Tech. Rep. TR-33/92, Dipartimento di Informatica, Universita di Pisa, 1992.

[DJL88] P. Deransart, M. Jourdan, B. Lorho: *Attribute Grammars*. LNCS 323. Berlin, Heidelberg, New York: Springer 1988.

[GGMW82] H. Ganzinger, R. Giegerich, U. Möncke, R. Wilhelm: A truly generative semantics-directed compiler generator. In *SIGPLAN '82 Symposium on Compiler Construction*, SIGPLAN Notices 17(6), pp. 172–184. ACM Press, 1982.

[GH92] Y. Gurevich, J. Huggins: The semantics of the C programming language. In E. Börger et al., editor, *Computer Science Logic*, LNCS 702, pp. 274–308. Berlin, Heidelberg, New York: Springer 1992.

[GHL+92] R. W. Gray, V. P. Heuring, S. P. Levi, A. M. Sloane, W. M. Waite: Eli: A complete, flexible compiler construction system. *Communications of the ACM*, 35(2):121–131, February 1992.

[Gie88] R. Giegerich: Composition and evaluation of attribute coupled grammars. *Acta Inf.*, 25:355–423, 1988.

[GM89] Y. Gurevich, L. Moss: Algebraic operational semantics and Occam. In E. Börger et al., editor, *Computer Science Logic*, LNCS 440, pp. 176–192. Berlin, Heidelberg, New York: Springer 1989.

[Gur95] Y. Gurevich: Evolving algebras 1993: Lipari guide. In E. Börger, editor, *Specification and Validation Methods*. Oxford University Press, 1995.

[HDP96] F. v. Henke, A. Dold, H. Pfeifer: Towards mechanization of evolving algebras in PVS. Workshop on Evolving Algebras, May 1996. (Unpublished).

[Hed91] G. Hedin: Incremental static semantics analysis for object–oriented languages using Door attribute grammars. In H. Alblas, B. Melichar, editors, *International Sommer School on Attribute Grammars, Applications, and Systems*, LNCS 545, pp. 374–379. Berlin, Heidelberg, New York: Springer 1991.

[Jou84] M. Jourdan: An optimal-time recursive evaluator for attribute grammars. In M. Paul, B. Robinet, editors, *International Symposium on Programming*, LNCS 167, pp. 167–178. Berlin, Heidelberg, New York: Springer 1984.

[JP88] M. Jourdan, D. Parigot: The FNC–2 system: Advances in attribute grammars technology. Technical Report No. 834, INRIA, April 1988.

[Kah87] G. Kahn: Natural semantics. In *STACS'87*, LNCS 247, pp. 22–39. Berlin, Heidelberg, New York: Springer 1987.

[Kai89] G. Kaiser: Incremental dynamic semantics for language–based programming environments. *ACM Transactions on Programming Languages and Systems*, 11(2):169–193, 1989.

[Kat84] T. Katayama: Translation of attribute grammars into procedures. *ACM Transactions on Programming Languages and Systems*, 6:345–369, 1984.

[KHZ82] U. Kastens, B. Hutt, E. Zimmermann: *GAG: A Practical Compiler Generator*. LNCS 141. Berlin, Heidelberg, New York: Springer 1982.
[Kli90] P. Klint: A META–environment for generating program environments. Technical Report CS-R9064, CWI, 1990.
[Mül95] P. Müller: Specification and implementation of an annotation language for an object–oriented programming language. Master's thesis, Technische Universität München, 1995. (In German).
[MM94] P. D. Mosses, M. Musicante: An action semantics for ML concurrency primitives. In M. Naftalin, T. Denvir, M. Bertran, editors, *FME '94: Industrial Benefit of Formal Methods*, LNCS 873, pp. 461–479. Berlin, Heidelberg, New York: Springer 1994.
[Mos92] P. D. Mosses: *Action Semantics*. Cambridge Tracts in Theoretical Computer Science 26. Cambridge University Press, 1992.
[Ørb94] P. Ørbaek: OASIS: An optimizing action–based compiler generator. In P. Fritzson, editor, *Compiler Construction*, LNCS 786, pp. 1–15. Berlin, Heidelberg, New York: Springer 1994.
[Pet94] M. Pettersson: Rml — a new language and implementation for natural semantics. In M. Hermenegildo, J. Penjam, editors, *Programming Language Implemenation and Logic Programming, 6th International Symposium, PLILP '94*, LNCS 844, pp. 117–131. Berlin, Heidelberg, New York: Springer 1994.
[PH93] A. Poetzsch-Heffter: Programming language specification and prototyping using the MAX System. In M. Bruynooghe, J. Penjam, editors, *Programming Language Implementation and Logic Programming*, LNCS 714, pp. 137–150. Berlin, Heidelberg, New York: Springer 1993.
[PH94a] A. Poetzsch-Heffter: Deriving partial correctness logics from evolving algebras. In B. Pehrson, I. Simon, editors, *IFIP 13th World Computer Congress '94*, pp. 434–439. Amsterdam: Elsevier 1994.
[PH94b] A. Poetzsch-Heffter: Developing efficient interpreters based on formal language specifications. In P. Fritzson, editor, *Compiler Construction*, LNCS 786, pp. 233–247. Berlin, Heidelberg, New York: Springer 1994.
[PHM94] A. Poetzsch-Heffter, R. Merk: Complex remote attribute dependencies do not exclude the generation of efficient evaluators. Technischer Bericht TUM-I9432, Technische Universität München, August 1994.
[PS94] J. Palsberg, M. I. Schwartzbach: *Object–Oriented Type Systems*. John Wiley & Sons, 1994.
[RT89] T. Reps, T. Teitelbaum: *The Syntheziser Generator Reference Manual*, 3rd edn. Berlin, Heidelberg, New York: Springer 1989. (3rd edition).
[Sch86] D. A. Schmidt: *Denotational Semantics: A Methodology for Language Development*. Allyn & Bacon, 1986.
[VSK89] H. Vogt, S. Swierstra, M. Kuiper: Higher order attribute grammars. In *SIGPLAN '89 Conference on Progamming Language Design and Implementation*, SIGPLAN Notices 24(7), pp. 131–145. ACM Press, 1989.
[Wat94] D. A. Watt: Using ASF+SDF to interpret and transform actions. In P. Mosses, editor, *First International Workshop on Action Semantics*, BRICS Notes Series NS-94-1, Department of CS, University of Aarhus, Denmark, May 1994.
[WJ95] J. Walz, G. Johnson: Inductive attribute grammars. *Acta Inf.*, 32:117–144, 1995.

Appendix

A Completing the BOPL specification

In this appendix, we provide parts of the BOPL specification that were omitted in Sect. 3. With each part we illustrate further aspects/details of MAX specifications.

Strictness w.r.t. nil Attribute *decl* in Sect. 3.1 uses a function *lookup*:

```
FCT  lookup( UsedId@ U, Declaration D ) Declaration:
 IF  D.first.term = U.first.term THEN D
                                 ELSE lookup( U, D.rsib )
```

The specification of *lookup* demonstrates two aspects: First, it shows the general scheme for recursively searching in an occurrence list. Second, it demonstrates a detail of the MAX semantics, namely that all functions and attributes return *nil* if one of their arguments is *nil*. Here, *lookup* returns *nil* if it fails.

More about context conditions To illustrate a more sophisticated use of patterns in context conditions, we specify that two identifiers occurring in the same list of declarations have to be different:

```
CND <*, Declaration<ID1,*>,*, Declaration<ID2,*>,*>:
        ID1.term # ID2.term
| '"*** Identifier " ID1.term.idtos " declared twice\n"'
```

The context condition reads as follows: For all pairs of declarations with identifier occurrences ID1 and ID2 having the same parent (a class, parameter, or variable list) it must hold that the identifiers corresponding to ID1 and ID2 are different.

Initialization of dynamic functions In Sect. 3.3 we illustrated the initialization of dynamic functions only by some trivial examples. An interesting example is the initialization of the function *cont*:

```
DYN  cont( RtVar V ) Value:
  IF  isConstant@ (V.static)
   |  isUsedId@ (V.static)
      AND  isClass@ (V.static.decl)      THEN
      IF  V.static.first.term.idtos = "Int"   THEN  0
       |  V.static.first.term.idtos = "Bool"  THEN  true
      ELSE  Object( V.static.decl, 0 )
  ELSE  Null
```

If the static part of a runtime variable V is a constant expression occurrence C, i.e. V is a temporary, then *cont*(V) is initialized to the value of the term corresponding to C. The next lines specify that a class identifier in an expression denotes a particular object of that class: For class "Int" this object is 0, for class "Bool" this is *true*, and for all other classes this is the object with object identifier 0. This way, a special new-expression that takes the class name as argument is dispensable. The last line initializes all other runtime variables to *Null*, in particular all instance variables. This makes especially object creation so simple.

Completing the control flow specification Here, we give the task sorts omitted in 3.2 and the complete specification of the attribute *succ*:

```
NewObj       ( Exp@.src  Exp@.dst  Task.succ )
Move         ( StaticVar.src  StaticVar.dst  Task.succ )
Return       ( Exp@.body )
SnglSuccTask =  ExpPoint | Select | MoveArg | NewObj
             | Move
Task         =  SnglSuccTask | Branch | Check | Call
             | CallExtern | Abort | Return | Terminate
```

```
              | Abort

 ATT  succ( ExecPoint P ) Task:
   LET  N == P.node   IN
   IF   P = N.after  THEN
     IF Assign@<ID,N> E
         THEN  Move( N, ID.decl, Move(N,E,E.after) )
      | Seq@<_,N> E      THEN  Move( N, E, E.after )
      | If@<N,T,E>
          THEN  Branch( N, T.before, E.before )
      | If@<_,N,_> E     THEN  Move( N, E, E.after )
      | If@<_,_,N> E     THEN  Move( N, E, E.after )
      | While@<N,B> E
          THEN  Branch( N, B.before, E.after )
      | While@<C,N>      THEN  C.before
      | Send@<N,_,<E,*>> THEN  E.before
      | Send@<N,ID,<>> E THEN
         Select( E,  Check( MoveArg( N, 0, Call(E) ) ) )
      | Send@< RCV, ID, <*,N> EL > E  THEN
         Select( E,  Check( MoveArg( RCV, 0,
         moveargtasks( N, numchilds(EL), Call(E) ) ) ) )
      | New@<N> E        THEN  NewObj( N, E, E.after )
      | Method@<_,_,N>   THEN  Return( N )
      | Program@<_,N>    THEN  Terminate
     ELSE  P.next
    |   P = N.before THEN
     IF Self@ N         THEN  Move( Self, N, N.after )
      | UsedId@ N       THEN
        IF NOT is[ N.decl,_Class@ ]
          THEN  Move( N.decl, N, N.after )
          ELSE  N.after
      | Assign@<_,E> N   THEN  E.before
     ELSE  P.next
   ELSE nil
```

Assignments in BOPL are expressions just as in C. Thus, the value of the right hand side is moved to the left hand side variable and the assignment expression itself.

Evolving algebra rules for BOPL (continued) In addition to the three rules given in Sect. 3.3, the BOPL specification contains the following evolving algebra rules; they are mainly given to show the compactness of such specifications:

```
IF  isSelect( ct )      THEN
  resofslct := meth_sel( cont(TempVar(ct.site.first,ci)),
                         ct.site )  FI

IF  isCheck( ct )       THEN
  IF  isMethod@( resofslct )
      THEN   ct :=  ct.succ
      ELSE   ct :=  resofslct   FI FI
```

```
IF  isNewObj( ct )      THEN
  IF  isObject[ cont( TempVar(ct.src,ci) ) ]   THEN
        nextobjid :=  plus( nextobjid, 1 )
        cont( TempVar(ct.dst,ci) ) :=
                  Object( cont(TempVar(ct.src,ci)).class,
                  nextobjid )
  ELSE  cont( TempVar(ct.dst,ci) ) :=  Null        FI  FI

IF  isMoveArg( ct )     THEN
  IF  ct.parpos = 0     THEN
        cont( SelfVar(nextincar) )
          :=  cont( TempVar(ct.src,ci) )
  ELSE  cont( ActuPar(child(ct.parpos,resofslct.formpars),
              nextincar) )
          :=  cont( TempVar(ct.src,ci) ) FI  FI
IF  isMove( ct )        THEN
  cont( rtvar( ct.dst, ci, cont(SelfVar(ci)) ) )  :=
        cont( rtvar( ct.src, ci, cont(SelfVar(ci)) ) )  FI

IF  isReturn( ct )      THEN
  ct                :=  rtrnto(ci).task
  cont( TempVar( rtrnto(ci).task.node,
        rtrnto(ci).incar ) ) :=
        cont( TempVar(ct.body,ci) )
  ci                :=  rtrnto(ci).incar               FI

IF  isCallExtern( ct )  THEN
  ct := ct.site.after
  cont( TempVar(ct.site,ci) ) :=
     eval( ct.site.methid.term,
       cont( TempVar(ct.site.rcv,ci) ),
       cont( TempVar(ct.site.actpars.first,ci) ),
       cont( TempVar(ct.site.actpars.first.rsib,ci) ) ) FI
```

The rules use three functions omitted here: *meth_sel* selects a method given an object *O* and a *Send*-node by looking it up in the method list of *O.class*; if successful, it returns a method or an external call; otherwise, the *Abort*-task. The auxiliary function *rtvar* returns a runtime variable given the static part *SV* of it, the current incarnation *INC*, and the current self object *SELF*. Finally, the function *eval* evaluates external calls; to keep things simple external calls are restricted to at most three parameters.

B Translating attribute grammars to MAX

By a small example it is shown how attribute grammars can be translated to MAX specifications. Let us consider the following AG with nonterminals *M* and *N*, terminal *T*, inherited attribute *niatt*, and synthesized attributes *nsatt* and *msatt*:

```
M  attributed by   msatt: MsattType   synthesized
N  attributed by   nsatt: NsattType   synthesized
                   niatt: NiattType   inherited
```

```
R: M -> N    { N.niatt    = init()
               M.msatt    = h2( N.nsatt )              }

P: N -> N N  { N[1].niatt = f1( N[0].niatt )
               N[2].niatt = f2( N[0].niatt )
               N[0].nsatt = h1( N[1].nsatt, N[2].nsatt )}
Q: N -> T    { N.nsatt    = g1( N.niatt, g2(T) )       }
```

Each construct of the AG corresponds to one construct in the MAX specification: The context–free grammar corresponds to the sorted productions in MAX. The attribute declarations correspond to the headers of the MAX attributes. Each attribute equation corresponds to one case in the MAX attribute definitions:

```
                    ATT  msatt( M@ m ) MsattType:
                      IF  R@< n > m    THEN   h2( n.nsatt )
                       |  ...

                    ATT  nsatt( N@ n ) NsattType:
                      IF  P@<n1,n2> n  THEN
                          h1( n1.nsatt, n2.nsatt )
M  =  R  | ...         |  Q@< t > n    THEN
R  ( N )                  g1( n.niatt, g2(t) )
N  =  P  |  Q          ELSE  nil
P  ( N  N )
Q  ( T )            ATT  niatt( N@ n ) NiattType:
                      IF  R@< n >      THEN   init()
                       |  P@<n,_> n0   THEN   f1( n0.niatt )
                       |  P@<_,n> n0   THEN   f2( n0.niatt )
                      ELSE  nil
```