Developing Efficient Interpreters Based on Formal Language Specifications

Arnd Poetzsch–Heffter [1]
Institut für Informatik
Technische Universität
D–80290 München
poetzsch@informatik.tu-muenchen.de

Abstract

The paper reports on extensions to the MAX system enabling the generation and refinement of interpreters based on formal language specifications. In these specifications, static semantics is defined by an attribution mechanism that allows to enrich syntax trees by control flow graphs. The dynamic semantics is defined by evolving algebras, a framework that has been successfully used to specify realistic programming languages.

We apply the combined framework to a non–trivial example language and show how the resulting language specification can be refined in order to improve the efficiency of the generated interpreters. The framework provides enough modularity and flexibility so that such refinements can be carried out by local changes within the framework. Finally, we explain the implementation of the extensions to MAX.

1 Introduction

Motivation Certainly, many agree with Tofte that "... a realistic language definition is a very delicate object" ([Tof90], p. 109). Nevertheless, different tasks and levels in language design and implementation are til now solved and supported based on different, often unrelated frameworks; consequently, for each task a specification has to be produced from scratch. E.g. there are frameworks for providing very high–level, readable, and formal language specifications (e.g. [Mos92]); and frameworks to specify compilation tasks focussing on efficiency: attribute grammars to express identification, typing, and context checking; control flow based specifications for data flow analyses to support optimizations; and pattern driven code generators.

Our ideal picture to relate the different specifications is a tree the root of which is the high–level language specification; a step to a child represents refinements or enrichments eventually leading to implementations of specific language–dependent tools. And a child should use as much as possible from the parent specification. The framework we present in this paper should be understood as a step towards this picture, even though we focus here on a specific example, the development of interpreters.

Approach In [PH93b], we showed how first–order recursive functions defined on occurrence structures can be used like attributes in an attribute grammar specification. In fact, recursive functions on occurrence structures provide a more expressive framework than attribute grammars in that they allow to formally specify an enrichment of the syntax tree by non–local edges. (On the other hand, attribute grammars allow in general for more efficient implementations; cf. relation to other work.) Here, we use a slightly extended version of occurrence structures to specify control flow and use evolving algebras developed by Gurevich to specify the dynamic semantics of programming languages.

Figure 1 shows the control flow graph for the expression $7 + f(a, 0)$ as syntax tree enrichment. The evolving algebra specifies how to interprete these graphs; in particular it

[1]Supported by DFG grant Po 432/2-1.

specifies the semantics of the graph nodes — called *tasks*. These tasks may have parameters (omitted in the figure); e.g. the BinOp–task has the operator Add, the argument expressions "Int" and "CallExp" and the result expression "BinExp" as parameters (cf. section 3). The graph nodes denoted by "∘" are called *program points*; they are an auxiliary device enabling to specify the task graph locally for each tree production.

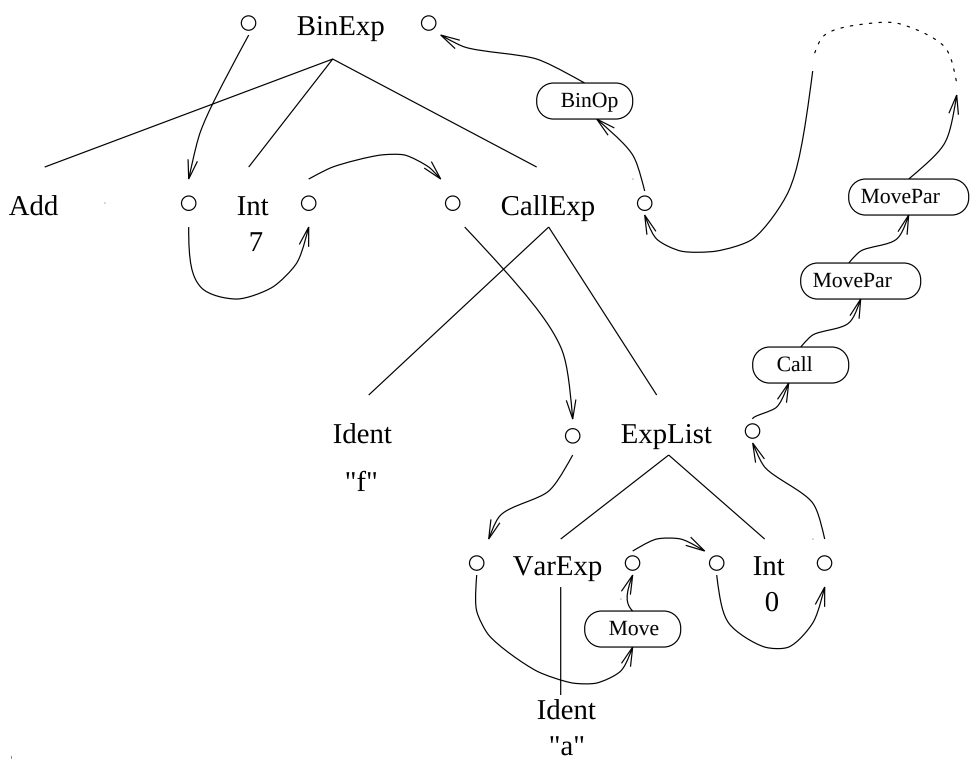

Figure 1: Control flow for expression $7 + f(a, 0)$

After providing the formal background, we illustrate the framework by specifying a non–trivial example language. In section 4, we refine this language specification to an interpreter specification. Section 5 explains how the MAX system ([PH93b]) was extended.

Relation to other Work Our work should be understood as a step towards filling the gap between very high–level language specification frameworks that are optimized versus readability and formality and specification frameworks that are designed to generate tools that can compete in efficiency with hand–writen tools.

To illustrate one of the differences between language specification frameworks like denotational, natural, or action semantics and our framework, let us consider a standard specification of an imperative language with loops and local blocks. If we interprete the shown program fragment according to such a specification, the environment has to be changed each time the loop is entered or exited (the binding for "a" has to be changed). In an interpreter one would like to avoid this overhead. As our specifications can be based on the occurrences of the syntax tree, we could refine such a specification by using a static function from used identifier occurrences to their declaration occurrences and a global environment mapping variable declarations to values. I.e. we seperate a static aspect (namely identifier binding) from

```
int a = 2;
int b = 10
  ...
while( b > 9 ){
    int a;
      ...
      a = a+1 ;
}
```

the dynamic semantics and globalize the environment. In a similar way, we can specify a static function yielding for each jump statement the corresponding target statement, thereby refining control flow.

The goal of the presented work is not to substitute higher–level frameworks that are more convenient to use, trimmed versus readability, or have specific theoretical properties (all this is true e.g. for action semantics; see [Mos92]). The goal is to provide a system supported formal framework that allows to express stepwise refinement of specifications; in particular it should be flexible enough to express globalizing transformations (cf. [Sch85]), seperation of static from dynamic aspects, improvements on the control flow, and refinements based on data types.

Compared to attribute grammar based system, our system supports the specification of dynamic semantics and is more expressive as far as the formal part of specifications is concerned. On the other hand, if e.g. the flow graphs are encapsulated in semantic actions, attribute grammar specifications can be understood as refining our specifications towards efficiency. In addition to that, our work is related to systems enabling the implementation of data flow analyses ([Wil81]) and allows to bring data flow analysis and language semantics together (see [PH93a]; for a denotational approach cf. [Ven89]).

Another very important relation is to works using programming language specifications to define the semantics of programming logics. In particular, it is related to positional semantics as defined in [CO78]. In deed, our framework can be understood as a generalization of positional semantics in that it provides very flexible formal methods to define positions and the transition relation. Thereby it makes the logic approach of [CO78] applicable to realistic programming languages.

2 Formal Background

This section introduces the formal concepts our specifications are based on. First, we define occurrence algebras, an extension of term algebras. Occurrence algebras provide occurrence sorts (e.g. the sort of the variable declaration occurrences in a syntax tree). Having the occurrences as elements in the formal framework allows to specify graphs and thereby enables very rich program representations. Then, we give a short introduction to evolving algebras that are used to specify the dynamic semantics. Both concepts use the notion of "algebras": An *algebra* is given by a set U, called the universe of the algebra, and a set of functions that take their arguments in U^n and yield values in U.

2.1 Occurrence Algebras

Occurrence algebras extend order–sorted term algebras by providing sorts for the occurrences of terms. They simplify and generalize the concept of occurrence structures as presented in [PH93b]. Before the formal definition, we give an informal introduction. Consider the abstract syntax for the expressions of our example language:

```
Exp        =  Int  |  VarExp  |  CallExp  |  BinExp
VarExp     (  Ident  )
CallExp    (  Ident  ExpList  )
ExpList    *  Exp
BinExp     (  Operator  Exp  Exp  )
Operator   =  Add  |  ...
Add        ()
```

i.e. an expression is in integer constant, a variable occurrence, a function call, or a binary expression; and e.g. a call has two components: the identifier of the called function and the actual parameter list. To define control flow graphs as shown in the introduction, we must be able to refer to a "father" of a term. But in term algebras there is no "father" of a term. That is why we consider occurrences in terms. Occurrences are represented as usual by a pair (t, l), where t is a term and and l a list of positive integers $i_1, \ldots, i_n$ describing the path from the root of t to the subterm occurrence. We use the notation $@(t; i_1, \ldots, i_n)$ and write $@(t; \epsilon)$ for the root occurrence of t. E.g. @(CallExp("f",ExpList(VarExp($\underline{\text{“x”}}$),VarExp("x"))); 2,1,1) represents the first occurrence of identifier x in the abstract syntax term for $f(x, x)$. The father of this occurrence is the first occurrence of the corresponding variable expression: @(CallExp("f",ExpList($\underline{\text{VarExp(“x”)}}$,VarExp("x"))); 2,1).

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 Ident@ and VarExp@ respectively. Using occurrence sorts, we can define new sorts. For example the program points before and after expression occurrences as shown in figure 1 can be defined by the productions:

```
ExpPoint   =  Before  |  After
Before     (  Exp@  )
After      (  Exp@  )
```

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

Definition 2.1 Let PRODSORTS and PRIMSORTS be two disjunct sets of symbols called production and primitive[2] sort symbols. A *sort symbol* is a production or primitive sort symbol followed by a possibly empty sequence of @-symbols, i.e. has the form S@*, where $\mathrm{S} \in \mathrm{PRODSORTS} \cup \mathrm{PRIMSORTS}$. The set of sort symbols is denoted by SORTS.

- A *production* has one of the following forms: $\mathrm{S}\,(\mathrm{S}_1 \ldots \mathrm{S}_m)$ (called *tuple production*), or $\mathrm{S} * \mathrm{T}$ (called *list production*), or $\mathrm{S} = \mathrm{S}_0 | \ldots | \mathrm{S}_n$ (called *class production*), where $\mathrm{S} \in \mathrm{PRODSORTS}$, $\mathrm{S}_i, \mathrm{T} \in \mathrm{SORTS}$, and m, n are natural numbers; S is called the *left hand side* of the respective production.

- Let Π be a finite set of productions such that for each $\mathrm{S} \in \mathrm{PRODSORTS}$ 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:

  - $\mathrm{S}(t_1, \ldots, t_n)$ is an element of sort S if $\mathrm{S}\,(\mathrm{S}_1 \ldots \mathrm{S}_n) \in \Pi$ and each t_i is of sort S_i ;
  - $\mathrm{S}(t_1, \ldots, t_n)$ is of sort S if $\mathrm{S} * \mathrm{T} \in \Pi$ and each t_i is of sort T;
  - t is of sort S if $\mathrm{S} = \mathrm{S}_0 | \ldots | \mathrm{S}_n \in \Pi$ and t is of sort S_i for one i, $0 \le i \le n$;
  - $@(t; \epsilon)$ is of sort S@ if t is of sort S;
  - $@(t; i_1, \ldots, i_m)$ is of sort S@ if $t \equiv \mathrm{T}(t_1, .., t_{i_1}, .., t_n)$ is of sort T and $@(t_{i_1}; i_2, \ldots, i_m)$ is of sort S@.

  Elements generated by the first two rules are called *terms*; elements generated by the last two rules are called *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:

[2]Throughout this paper, we assume the primitive sorts Ident, Int, and Bool

- functions defined on occurrences: $fath(x)$, $lbroth(x)$, $rbroth(x)$, $son(i,x)$, the father, left/right brother, i-th son of occurrence x; $term(x)$, the subterm that corresponds to occurrence x.
- functions defined on terms: $subterm(i,t)$, the i-th subterm of t; $append(e,l)$, $first(l)$, $rest(l)$ with the usual meaning on list terms; for each list production S $*$ T the empty list constructor $S()$; and for each tuple production S $*$ T the constructor $S(t_1, \ldots, t_n)$. (For convenience, we use the same name for the sort (roman font) and the corresponding constructor (italic).)
- for each sort S a boolean function $isS(x)$ testing whether x is of sort S.
- the functions defined on the primitive sorts.

Whenever a function is applied to elements where the meaning as described[3] above is not defined, it yields the extra element *nil*.

□

2.2 Evolving Algebras

Evolving algebras are a powerful framework for specifying programming language semantics. They have been used to specify the semantics of many different programming languages including C ([GH92]), PROLOG ([BR92]), and Occam ([GM89]). Here, we summarize the definition of evolving algebras given in [Gur91].

In an evolving algebra specification, the computation states are described by algebras. The evolving algebra specifies how the states are related to their successor states; i.e. it specifies a successor relation on algebras: 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 $\langle variable \rangle := \langle variable \rangle$. We assume the list functions *isempty*, *first*, and *rest*, and the selector functions *lhs*, *rhs* yielding the left/right hand sides of assignments. The state information that changes during execution is represented by the unary function VAL mapping variables to values and the "0-ary function" AL holding the rest of the assignment list. In the following, we call functions that may change during execution *dynamic*; and dynamic 0-ary functions are called *dynamic variables.* Here is an evolving algebra specifying the semantics of assignment lists:

```
IF  ¬isempty(AL)  THEN   AL :=  rest(AL)                              FI
IF  ¬isempty(AL)  THEN   VAL(lhs(first(AL))) :=  VAL(rhs(first(AL)))  FI
```

The evolving algebra has two rules. Rules are executed in parallel: Given an algebra A, evaluate the guards, the right hand side expressions of the updates, and the arguments on the left hand sides of the updates (here `lhs(first(AL))`). Then, perform those updates with a true guard. The following definition makes this more precise:

Definition 2.2 An *evolving algebra* EA is given by a finite set of rules of the form **IF** *guard* **THEN** $f(arg_1, \ldots, arg_n) := rhsexp$ **FI**, where *guard*, arg_i, and *rhsexp* are variable free expressions and n is the arity of f.

[3]A technical report spelling this out in more rigorous terms by algebraic laws is in preparation.

- Let A be an algebra containing the boolean values and a function for all function symbols occurring in EA. An algebra A' is an *EA–successor* of A if updating the functions of A according to the following procedure yields A': Let *UPDATES* be the set of updates of those rules the guard of which evaluate to true in A. Evaluate all arguments and right hand side expressions occurring in *UPDATES* and replace them by their values. The resulting set may contain subsets of contradicting updates, e.g. $g(b_1, \ldots, b_n) := r$ and $g(b_1, \ldots, b_n) := q$ with $r \neq q$. From these subsets of contradicting updates eliminate non–deterministically all updates but one, resulting in a set of non–contradicting updates. Change the functions of A according to this set of updates, i.e. change the value 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 $(\mathrm{A}_i)_{i \in \mathbb{N}}$ of algebras such that A_{i+1} is an EA–successor of A_i for all i is called a *computation* of EA with initial algebra A_0.

□

For practical purposes, the restricted syntax of evolving algebras as defined above is rather inconvenient. Assuming that the considered algebras always contain a binary function $\wedge$ yielding *true* exactly if both arguments are *true* and a unary function $\neg$ yielding true for the argument *false*, we can introduce the following four syntactical extensions — the associated rules show how to transform the extended syntax into the restricted syntax above:

- unguarded assignments as rules: $f(\vec{a}) :=$ *rhsexp*
  $\Rightarrow$ **IF** *true* **THEN** $f(\vec{a}) :=$ *rhsexp* **FI**,

- nested if-rules: **IF** *guard1* **THEN IF** *guard2* **THEN** *body* **FI FI**
  $\Rightarrow$ **IF** *guard1* $\wedge$ *guard2* **THEN** *body* **FI**

- rule lists in the body of if-rules: **IF** *guard* **THEN** $R_1 \ldots R_n$ **FI**
  $\Rightarrow$ **IF** *guard* **THEN** R_1 **FI** ... **IF** *guard* **THEN** R_n **FI**

- if-then-else-rules: **IF** *guard* **THEN** *body1* **ELSE** *body2* **FI**
  $\Rightarrow$ **IF** *guard* **THEN** *body1* **FI** **IF** $\neg$ *guard* **THEN** *body2* **FI**

It goes without saying that any combination of these extensions is allowed.

3 Specifying Control Flow and Dynamic Semantics

In this section, we apply occurrence algebras and evolving algebras to specify the semantics of an imperative example language called TOY in the following. Even though necessarily small, TOY contains some features that are difficult to handle in many other language specification frameworks, namely return statements (showing how to handle jumps or exceptions) and recursive function procedures with side effects (showing how to handle such procedure calls in expressions). The TOY specification demonstrates how to enrich syntax trees by control flow information and how to use this information to specify dynamic semantics. This modularization of the dynamic semantics specification into two parts may seem complicated at first sight. However, its advantages become clear when applying it to realistic size languages (cf. the discussion in section 5) or to the implementation of efficient interpreters (cf. section 4). In addition to this, the flow information or similar enrichments of the syntax tree can be used for data flow analysis and code generation in compilers.

The semantics specification in the following two subsections proceeds as follows: First, we specify the state space of TOY programs. Then, we introduce a set of atomic actions —

so-called *tasks* — and give an evolving algebra semantics for them. Finally, we specify the control flow of TOY in terms of program points and tasks. But before that, we have to give the missing productions of the abstract syntax of TOY (the expression syntax was given in section 2):

```
Program        (  DeclList  )
DeclList       *  Decl
Decl           =  Var  |  Proc
Var            (  Ident  )
Proc           (  Ident  ParamList.params  Body  )
Body           (  StatList  )
ParamList      *  Param
Param          (  Ident  )

StatList       *  Stat
Stat           =  WhileStat | IfStat | AssignStat | ReturnStat
WhileStat      (  Exp  StatList  )
IfStat         (  Exp  StatList  StatList  )
AssignStat     (  Ident  Exp  )
ReturnStat     (  Exp  )
```

As shown in the production for `Proc`, we allow to define selectors for tuple components; i.e. if P is of sort `Proc`, the expressions "P.params" and "subterm(2,P)" are equivalent.

3.1 The Dynamic Semantics of Tasks

This section has three parts: The first defines objects and locations; the second provides the sort definitions for tasks; and the third gives the evolving algebra for TOY.

Objects and Locations TOY has global objects and objects that are local to a procedure incarnation. Global objects need exactly one location to store the corresponding value; local objects need a location for each procedure incarnation. The global objects of TOY are the variables and integer constants[4]. Parameters are local objects. Are there other local objects? As we have to deal with procedure calls in expressions, we must keep track of those parts of expressions that are evaluated before a call. The easiest way to do this is to consider expression occurrences as local objects (this is called a store semantics in [MS76] and reflects the notion of temporary objects in compiler construction). Beside this, it is convenient to consider procedures as local objects for passing their return values. These considerations are formally expressed by the following productions:

```
Object         =  GlobalObject  |  LocalObject
GlobalObject   =  Var@    | Int@
LocalObject    =  VarExp@ | CallExp@ | BinExp@ |  Param@ |  Proc@
Location       =  GlobalObject |  Automatic
Automatic      (  LocalObject  Incar  )
Incar          =  Nat
```

The relation between locations and values is captured by the function VAL,

```
DYN  VAL( Location L ) Int:  IF  Int@ L:  term(L)   ELSE   ??
```

where the keyword `DYN` indicates that VAL is a dynamic function. The body of a dynamic function defines its values in the initial state: VAL yields for an integer node the corresponding value and an arbitrary value of sort Int (the result sort) for all other locations.

[4]Treating constants as objects simplifies the specification.

Tasks and Control Information The atomic actions the TOY semantics is based on are the tasks defined by the following productions. A branch task has three components: the expression node of the corresponding condition and the two possible successors. A return task has the returning procedure as component. A call task the expression where it is called. A move task its source and destination etc. The precise usage of the tasks is specified in section 3.2 — in particular the successors of tasks.

```
Task          =  Branch | Return | SinglSucc | Start | End | Point
SinglSucc     =  Move | MovePar | BinOp | Call
Branch        (  Exp@.cond   Task.ttsucc   Task.ffsucc )
Return        (  Proc@.rproc  )
Call          (  CallExp@.cexp     Task.succ )
Move          (  Object@.src       Object@.dst       Task.succ  )
MovePar       (  LocalObject@.src  LocalObject@.dst  Task.succ  )
BinOp         (  Operator.op  Exp@.left  Exp@.right  Exp@.dst Task.succ )
Start         ()
End           ()
TaskStack     *  Task
IncarStack    *  Incar
```

The control information in a state consists of the current task CT, the control stack CTR_S recording the return points of the active procedures, the current incarnation CI, the stack of active incarnations INCAR_S, and an incarnation counter NEXTINCAR:

```
DYN  CT       () Task:         Start()
DYN  CTR_S    () TaskStack:    push( End(), TaskStack() )
DYN  CI       () Incar:        0
DYN  INCAR_S  () IncarStack:   IncarStack()
DYN  NEXTINCAR() Incar:        1
```

As explained above, the bodies of these dynamic variables specify their initial values.

Evolving Algebra for TOY The evolving algebra for TOY essentially contains for each task sort one rule. The first rule specifies the semantics of a branch:

```
IF  isBranch(CT)  THEN
  IF VAL(loc(CT.cond,CI)) = 0  THEN  CT := CT.ffsucc  ELSE  CT := CT.ttsucc  FI
FI
```

As in C, the value *false* is represented by 0 in TOY. The function *loc* yields for each object its current location:

```
FCT  loc( Object@ OBJ, Incar IC ) Location:
  IF  LocalObject@ OBJ  :  Automatic( OBJ, IC )  ELSE  OBJ
```

The start task initializes CT to the point before the main procedure and the parameter of the main procedure to the input value (the specification of `mproc` and the declaration of the dynamic variables PROGRAM and INPUT are given in the appendix):

```
IF  isStart(CT)   THEN
  CT    :=  Before(mproc(PROGRAM))
  VAL( loc( son(1,mproc(PROGRAM).params), CI ) ) :=  INPUT
FI
```

The rules for the tasks with a single successor are rather straightforward. The function *eval* evaluates a binary operator on two arguments (cf. appendix). In order to understand the rule for the return task below, one should notice here that a call task pushes the point after the call expression on the control stack:

```
IF  isSinglSucc(CT)  THEN
  CT :=  CT.succ
  IF isMove(CT)    THEN  VAL(loc(CT.dst,CI)) := VAL( loc(CT.src,CI) )          FI
  IF isMovePar(CT) THEN  VAL(loc(CT.dst,CI)) := VAL( loc(CT.src,top(INCAR_S))) FI
  IF isBinOp(CT)   THEN  VAL(loc(CT.dst,CI)) :=
                       eval( CT.op, VAL(loc(CT.left,CI)), VAL(loc(CT.right,CI)) )
  FI
  IF  isCall(CT)    THEN
    CTR_S     :=  push( After(CT.cexp), CTR_S)
    INCAR_S   :=  push( CI, INCAR_S )
    CI        :=  NEXTINCAR
    NEXTINCAR :=  NEXTINCAR + 1
  FI
FI
```

On return of a procedure: If the end task is on top of the control stack, then the return value of the procedure is copied to the dynamic variable OUTPUT. Otherwise the return value is copied to the expression node corresponding to the continuation point of the returning procedure:

```
IF  isReturn(CT)  THEN
  CT    :=  top(CTR_S)
  CTR_S :=  pop(CTR_S)
  IF  top(CTR_S) = End()
    THEN  OUTPUT  :=  VAL( loc(CT.rproc,CI) )
    ELSE  VAL( loc( top(CTR_S).node, top(INCAR_S)) ) :=   VAL( loc(CT.rproc,CI) )
          CI      :=  top(INCAR_S)
          INCAR_S :=  pop(INCAR_S)
  FI
FI
IF  isEnd(CT)    THEN   skip             FI
IF  isPoint(CT)  THEN   CT := succ(CT)   FI
```

The specification of the successor function for program points (used in the last line) is the topic of the next section.

3.2 Control Flow

As illustrated in the introduction, control flow can be understood as an enrichment of the syntax trees. Such enrichments are specified in two steps: First, we specify the set of program points; then, we specify by an attribution process how the syntax trees are enriched. To describe control flow for TOY, we introduce program points before and after statements, statement lists, expressions, expression lists, and procedures:

```
ExecNode       =  Exp@ | ExpList@ | Stat@ | StatList@ | Proc@
Before         (  ExecNode.node  )
After          (  ExecNode.node  )
Point          =  Before | After
```

We specify the control flow as a function *succ* that returns for each program point the successor task. For reasons that become clear when we discuss implementation aspects (cf. section 5), we consider *succ* to be an attribute of the points, i.e. a function the values of which are computed once for a given program and stored for use during program execution. From a formal point of view, an attribute is just a unary function. The successor of a point depends on the context of the node to which the point belongs. We use the pattern notation of the MAX system to refer to the different contexts in which such a node may occur. E.g. let P be the point after a node N and N be the node representing the condition of a while statement; then the successor of P is the branch task that has N as first component, the point before the body of the loop as *tt*–successor, and the point after the loop as *ff*–successor; this is expressed as follows (for conditional expressions in the specification language, we use a colon to seperate conditions from then–branches in order to distinguish it from EA-rules):

```
  IF  WhileStat@<N,BD>  W :  Branch( N, Before(BD), After(W) )
```

The following specification shows the other cases for points after nodes (the cases for points before nodes are trivial and given in the appendix). The function *decl* maps every identifier to its declaration and the function *proc* maps each element of sort ExecNode to the enclosing procedure (cf. appendix). The function *next* yields the next program point in a left to right tree traversal (cf. section 5). The most interesting case is a call with parameters, because

```
  ATT  succ( Point P ) Task:
    LET  N == P.node :
    IF  P = After(N) :
      IF  Proc@ N             :  Return( N )
       |  WhileStat@<N,BD>  W :  Branch( N, Before(BD), After(W) )
       |  WhileStat@<E,N>     :  Before( E )
       |  IfStat@<N,TB,EB>    :  Branch( N, Before(TB), Before(EB) )
       |  IfStat@<_,N,_> IS   :  After( IS )
       |  AssignStat@<ID,N> A :  Move( N, decl(ID), After(A) )
       |  ReturnStat@<E> N    :  Move( E, proc(N), After(proc(N)) )
       |  BinExp@<O,L,N> E    :  BinOp( term(O), L, N, E, After(E))
       |  CallExp@<ID,<>N> C  :  Call( C, Before(decl(ID)) )
       |  CallExp@<ID,<PAR,*> N> C:
            Call( C, parcopy( PAR, son(1,decl(ID).params), Before(decl(ID)) ) )
      ELSE  next(P)
     |  P = Before(N) :
        ....  // see  appendix
    ELSE  nil()
```

it demonstrates how to construct task structures that are not directly related to the syntax tree (cf. the picture in the introduction): In the case here, we build up the sequence of parameter moves using the function *parcopy*:

```
  FCT  parcopy( Exp@ APAR, Param@ FPAR, Task SUCC ) Task:
    IF  rbroth(APAR) = nil() :  MovePar(APAR,FPAR,SUCC)
    ELSE  MovePar( APAR, FPAR, parcopy(rbroth(APAR),rbroth(FPAR),SUCC) )
```

4 Towards Specification of Efficient Interpreters

If we add a concrete syntax to the TOY specification given in the previous section and the appendix ([PHE93] shows how to do that), we can execute TOY programs using the prototype system described in section 5. However, the stepping from program point to program point would make these executions slow and the storage consumption would make them almost unfeasible, because we would need storage proportional to the total number of procedure incarnations occurring in an execution (and not proportional to the maximal number of simultaneously active incarnations). In this section, we refine the language specification of TOY to a specification of a reasonably efficient interpreter by eliminating unnecessary stepping and by switching to a stack organisation for the automatic storage. The point we want to make here is that both refinements can be achieved by small and local modifications of the TOY specification and that these modifications can be carried out within the presented framework.

Avoiding Unnecessary Stepping The stepping from program point to program point without executing a "real" task can be avoided by using the attribute *tasksucc* instead of *succ* in the evolving algebra rule for points: *tasksucc* yields for each point the next task to be

```
ATT  tasksucc( Point P ) Task:
  IF  isPoint( succ(P) ) :  tasksucc(succ(P))  ELSE  succ(P)   ,
```

executed. We could even do a little better by avoiding all program points except those that break up loops; in TOY, these are the points before conditions of while statements and before procedures (because of recursion). To do this, we essentially have to make *tasksucc* and *succ* mutually recursive.

Stacks for Automatic Storage This paragraph shows how the TOY specification can be refined to implement automatic storage by a standard stack organization (cf. e.g. [AU77]). Locations will be simply natural numbers (i.e. the dynamic function VAL can be considered as an array). We assume the two attributes *size* and *addr* with the following signature:

```
ATT  size( Proc@   )  Nat
ATT  addr( Object@ )  Nat
```

size(P) is the number of locations needed for an incarnation of procedure P; *addr* yields for a global object its location, for a local object its relative address, i.e. the offset from the first location in a procedure incarnation. (It should be clear that the well–known algorithms to compute *size* and *addr* can be expressed in our framework.)

Using the dynamic variable CI as first location in a procedure incarnation (i.e. as "stack pointer"), we need the following changes to the TOY specification:

- replace the definition for Location by "Location = Nat" and the definition for Incar by "Incar = Location";
- change the body of the function *loc* to

  ```
  IF  LocalObject@ OBJ :  IC + addr(OBJ)  ELSE  addr(OBJ)
  ```

- replace the updates of CI in the rules for the start and call task by:

  ```
  CI  :=   numberofvars(PROGRAM)            //  for the start task
  CI  :=   size( decl( son(1,CT.cexp) ) )   //  for the call  task
  ```

  where the function *numberofvars* yields the number of variable declarations.

After these changes all occurrences of NEXTINCAR can be removed. (And a renaming of Incar, etc. is advisable for mnemonic reasons.)

Further Improvements Of course, further improvements are possible — approximating more and more what a compiler does. To mention only one[5], the costly calls to *loc* can be eliminated by splitting the move task into three tasks reflecting the "addressing modes": MoveGlobToLoc, MoveLocToLoc, and MoveLocToGlob. After a replacement of all occurrences of the move task, the calls to *loc* can be replaced either by the global address or by the relative address plus CI.

The focus of this section was to demonstrate how systematic refinements can be carried out in the presented formal framework. It goes without saying that the illustrated method apply to compiler construction as well.

[5]According to our performance analysis, the most profitable one.

5 Implementation Aspects and Experiences

We implemented a prototype on top of the MAX system that generates interpreters from specifications based on occurrence algebras, attribute and function specifications, and evolving algebras. This section summarizes the implementation aspects of this prototype and discusses the experiences we made so far.

5.1 Implementation Aspects

This section sketches the extensions to the MAX system and the prototype implementation used for executing evolving algebras. For the following it is helpful to consider the run of a generated interpreter as a three step process:

1. Reading: includes parsing and encoding of the nodes; see below.
2. Attribution: includes attribute evaluation (and context checking).
3. Interpretation: execute the input program according to the evolving algebra.

Extensions to MAX The efficiency of our occurrence algebra implementation stems from the fact that during the attribution and interpretation phase for a program PROG we only have to deal with occurrences of one term or a very small number of terms, namely the syntax tree of PROG and possibly some intermediate representations. This allows us to work with efficient encodings[6] for these occurrences (instead of using a direct implementation of the term–natlist–pairs). Let us call this set of occurrences the *nodes*[7] of a program. Encodings of nodes were already explointed in the MAX implementation (see [PH93b] p. 144). As program points showed up to be helpful for many purposes (cf. section 5.2), we implemented program points as well by encodings. Essentially, the system provides a predefined sort PredefPoint and functions *before(n)*, *after(n)* yielding the point before and after a node n as well as the functions *node(p)*, *next(p)* yielding the node corresponding to a point p and the next point according to a left to right tree traversal. The encoding allows us to implement *before*, *after*, and *node* by an array access and *next* by an addition. User–defined program points are then treated as subsorts of sort PredefPoint.

In the specifications, we distinguished functions and attributes (e.g. *succ* was specified as an attribute). As mentioned earlier, the semantics of attributes and unary functions are the same. But the implementation is different. The relevant values of the attributes, i.e. the attribute values for the nodes and points, are computed once and cached for later use. One of the main advantages of encodings is that caching becomes simple: As the encodings of all elements of one sort form an integer interval, the attribute values can be stored in an array having this interval as range. In particular, we can handle attributes on points in an efficient way, in particular control flow information as given by *succ*.

Executing Evolving Algebras The implementation of evolving algebras has two aspects: the representation of the dynamic functions and the stepwise execution of the rules. In our prototype, we only allow dynamic variables and unary[8] dynamic functions. Dynamic functions are implemented by arrays. If the parameter sort has an encoding, this is done the same simple way as for the attributes. If the parameter sort is Int or Nat, the needed parts of the array are dynamically allocated in blocks; the mapping from indices to blocks is recorded by a binary tree. For all other parameter sorts, we hash the term representation of

[6]Our implementation uses a four byte integer for all codings mentioned in the following.

[7]This definition is consistent with the informal use of "node" in the previous sections.

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

the parameters. Of course, this can drastically slow down reads and updates (and by that the interpretation): E.g. an update to VAL in the version of section 3 can be more than 50 times slower than an update to VAL in the version of section 4 with parameters of sort Location = Nat.

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. As a small optimization, we use case statements to implement branching according to sort tests.

5.2 Generation of Language–Specific Programming Tools

The experiences we made so far are very promissing. We generated some tools for small languages like TOY and miniML, and developed a specification for a C subset that includes the entire expression and statement syntax. The control flow graphs based on tasks are not only a good technique for closing the gap between formal language specifications and interpreter specifications, but provide as well a flexible basis for the development of other language specific programming tools. E.g. it showed to be fairly simple to enrich interpreter specifications to get source level debuggers; and for them program points are a very natural way to specify the stepping. In another application we built a (hand–coded) data flow analysis on top of control flow graphs; and again we could reuse almost the whole static part of the language/interpreter specification. Last but not least, some aspects of code generation become simpler in our approach; in particular the generation of jumps and instruction labels is simplified.

6 Conclusions

We proposed a combination of occurrence algebras, recursive first–order functions, and evolving algebras as a programming specification framework. Even though its formal foundations are comparably simple, the framework is sufficiently expressive to specify all kinds of programming languages including parallel ones (cf. [GM89]). Its flexibility makes it attractive for stepwise refinements of specifications. We attempted to demonstrate this by refining a non–trivial language specification. Finally, we described a prototype implementation that generates interpreters or other language–dependent programming tools from specifications.

As described in the introduction, one of the motivations for this research was to fill the gap between high–level language specifications and efficiency oriented specification techniques and to learn how to refine the former towards the latter. Whereas we consider the relation between high–level specifications and our framework a topic for future research, the refinements needed to acheive more efficient language implementations are much better understood. E.g. to develop a production quality compiler from the refined TOY specification of section 4, add optimization techniques, refine the set of tasks w.r.t. a suitable intermediate representation for code generators and then adapt this refined specification to more specialized and more efficient compiler construction tools. The refinement approach not only helps to systematize compiler development and allows for early prototyping, but should eventually provide the techniques to prove realistic compilers correct.

The development of the MAX system is still in progress. Currently, we improve the efficiency of the attribute evaluation ([Mer93]). The next envisaged extension is a fixpoint evaluator to prototype high–level data flow analyses based on control flow graphs.

References

[AU77] A. V. Aho and J. D. Ullman. *Principles of Compiler Design.* Addison–Wesley, 1977.

[BR92] E. Börger and D. Rosenzweig. A simple mathematical model for full Prolog. Technical Report TR–33/92, Dipartimento di Informatica, Universita di Pisa, 1992.

[CO78] R. L. Constable and M. J. O'Donnell. *A Programming Logic: With an Introduction to the PL/CV Verifier.* Winthrop Publishers, 1978.

[GH92] Y. Gurevich and J. Huggins. The semantics of the C programming language. In E. Börger et al., editor, *Computer Science Logic*, pages 274–308, 1992. LNCS 702.

[GM89] Y. Gurevich and L. Moss. Algebraic operational semantics and Occam. In E. Börger et al., editor, *Computer Science Logic*, pages 176–192, 1989. LNCS 440.

[Gur91] Y. Gurevich. *Evolving Algebras*, volume 43, pages 264–284. EATCS Bulletin, 1991.

[Mer93] R. Merk. Generierung von MAX–Attributauswertern. Master's thesis, Technische Universität München, 1993.

[Mos92] P. Mosses. *Action Semantics.* Cambridge University Press, 1992. "Tracts in Theoretical Computer Science".

[MS76] R. Milner and C. Stratchey. *A Theory of Programming Language Semantics.* Chapman and Hall, 1976.

[PH93a] A. Poetzsch-Heffter. Interprocedural data flow analysis based on temporal specifications. Technical Report 93-1397, Cornell University, 1993.

[PH93b] A. Poetzsch-Heffter. Programming language specification and prototyping using the MAX system. In J. Penjam M. Bruynooghe, editor, *Programming Language Implementation and Logic Programming*, 1993. LNCS 714.

[PHE93] A. Poetzsch-Heffter and T. Eisenbarth. The MAX system: A tutorial introduction. Technical Report TUM-I9307, TU, April 1993.

[Sch85] D. A. Schmidt. Detecting global variables in denotational specifications. *Transactions on Programming Languages and Systems*, 7:299–310, 1985.

[Tof90] M. Tofte. *Compiler Generators.* Springer–Verlag, 1990.

[Ven89] G. A. Venkatesh. A framework for construction and evaluation of high–level specification of program analysis techniques. *ACM Conference on Progamming Language Design and Implementation*, 1989.

[Wil81] R. Wilhelm. Global flow analysis and optimization in the MUG2 compiler generating system. In S. Muchnick and N. D. Jones, editors, *Program Flow Analysis: Theory and Applications*, pages 132–159. Prentice–Hall, 1981.

A Completing the TOY Specification

The following parts of the TOY specification describe the identification and complete the specification given in section 3:

```
DeclOrParam      =  Decl | Param
EnvSite          =  DeclOrParam | Body
DeclOrParamList  *  DeclOrParam@


FCT  idstr( DeclOrParam@ D ) String:  idtos( term(son(1,D)) )

ATT  env( EnvSite@ ES ) DeclOrParamList :
   IF  Program@< <ES,*> >      :  append( ES, DeclOrParamList() )
    |  Proc@<_,<ES,*>,_> FD    :  append( ES, env(FD) )
    |  Proc@<_,<>,ES> FD       :  env(FD)
    |  Proc@<_,<*,PD>,ES>      :  env(PD)
    |  DeclList@<*,DC,ES,*>    :  append( ES, env(DC) )
   ELSE                          nil()

FCT  corr_env( ExecNode N ) DeclOrParamList:
   IF  Body@<N> :  env(N)  ELSE  corr_env( fath(N) )

FCT  lookup( String ID, DeclOrParamList ENV ) DeclOrParam@:
   IF  ENV = DeclOrParamList() :   nil()
    |  ID  = idstr(first(ENV)) :   first(ENV)
                          ELSE   lookup( ID, rest(ENV) )

ATT  decl( Ident@ IDN ) DeclOrParam@:
   IF  DeclOrParam@<IDN,*> D   :   D
   ELSE   lookup( idtos(term(IDN)), corr_env(fath(IDN)) )


ATT  proc( ExecNode N  ) Proc@ :   IF  Body@<N> B:  B  ELSE  proc( fath(N) )
FCT  mproc( Program@ P ) Proc@ :   lookup( "main", env(son(-1,son(1,P))) )

ATT  succ( Point P ) Task:
  LET  N == P.node :
  IF  P = After(N) :
     //  see section 3
  |  P = Before(N) :
    IF  Proc@<_,_,<SL>> N   :  Before(SL)
     |  AssignStat@<ID,E> N :  Before(E)
     |  VarExp@<ID> N       :  Move( decl(ID), N, After(N) )
     |  CallExp@<ID,EL> N   :  Before(EL)
     |  BinExp@<_,LO,_> N   :  Before(LO)
    ELSE  next(P)
  ELSE  nil()

FCT  eval( Operator OP, Int L, Int R ) Int:
  IF  OP = Add()   :  L + R
   |  ...
  ELSE  nil()

DYN  INPUT  () Int :       ??
DYN  OUTPUT () Int :       ??
DYN  PROGRAM() Program@:   ??
```