数理解析研究所講究録
第586巻 1986年 91-111

Practical Attribute Grammar Forms Allowing Continuations

Hiroyuki MATSUDA 松田裕幸

NEC Scientific Information System Development, Ltd.

ABSTRACT

We propose a new attribute grammar form that separates attributes into syntactic attributes and semantic attributes and may describe continuation semantics. We call the form "semantic attribute grammar form" since it has semantic attributes, which play the role of denotations.

1. Introduction

On designing programming languages, a designer often wishes to run the program written in the current designing language to check the validity of the syntactic and semantic specification of the language. However, the implementor and the designer of the language are different in general, so the designer should complete the specification, then pass it to the implementor, and then waits for the compiler to make up.

As a solution of this dilemma, we have studied the executable definitions of programming languages. Executable definitions are specifications and at the same time are programs: that is, the definition itself is written in another language and is translated into the target language's compiler by the meta-compiler, or interpreted by the meta-interpreter. We have designed and developed an executable definition language LEAG[5], which is based on extended attribute grammars[1][2]. LEAG is a Lisp-like language and its grammatical symbols are functions whose input values are inherited attributes and whose output values are synthesized attributes. The specification of the language L written in LEAG is translated into Lisp functions and then the functions works for the programs written in L as a compiler or an interpreter.

While developing and using LEAG, one question arises: how can we describe the formal semantics of code generation parts? Historically attribute grammars have dealt with the syntactic and context-sensitive properties of a language, then as a result have many advantages in constructing syntactic analyzers formally but few advantages in constructing code generators from the formal semantics. To overcome this situation, at first we have reached Paulson' semantic grammars[5]. Semantic grammars are attribute grammars whose attributes are as a denotation from denotational semantics. The specification of a programming language L described by the semantic grammar is translated into a compiler and then the compiler compile the programs written in L. Two defects of semantic grammars are: (1) no dealing with continuation semantics; (2) no discriminating between static type check-

ing and dynamic type checking.

It is well known that continuation semantics may be described by non-absolute circular attribute grammars having only synthesized attributes[2], or by Wand's special-purpose combinators[4], or by Sethi's cutting-points which are analogous to UNIX's pipes and do function as a continuation[8]. Unfortunately the above attribute grammars need the reformulations of themselves, and Wand's combinators are not a primitive so that new combinators have to be created whenever new denotational entities are added. On the other hand Sethi's approach is a more realistic one since its intended specification of semantics are described by YACC-like notations and are translated into the inputs of YACC system[12].

Our primary concern is to make up an executable definition system that supports designing new languages, or prototyping the compilers of experimental languages, or describing the semantics by executing the definitions. We have already designed and carried out the prototype of an executable definition system LEAG. Next step is to fill up the semantics system of executable definitions. Then we propose a new attribute grammar form that separates attributes into syntactic attributes and semantic attributes and may describe continuation semantics. We call the form "semantic attribute grammar form" since it has semantic attributes, which play the role of denotations.

2. Semantics-Running Approaches To Programming Language Designs

In this chapter we offer a short overview of "semantics-running" approaches to programming language designs.

2.1 Programming Denotational Semantics

Allison[3] has programmed in Pascal the denotational semantics of a small language called 'contlang'. Contlang is not a very useful programming language but it contains some of the more difficult language features -- "goto", "valof", "resultis".

Denotations are represented as an uncurried function and passed to another denotations as a function argument. Information for continuations is hold by the function-call stacking mechanisms. Denotational semantics has no concrete syntax, therefore Allison implemented the lexical analyzer and syntactical analyzer, and then contlang's programs are converted into intermediate tree forms, which are interpreted by the programmed semantics(meta-interpreter).

2.2 Attribute Grammar Based Language LEAG

LEAG(a Language based on Extended Attribute Grammars)[5] is a programming language that may specify the syntax and semantics of programming languages. LEAG has been implemented in Lisp and presented Lisp-like notations.

LEAG's advantages are:

(i) definitions described by LEAG are executable under Lisp environment.

(ii) LEAG's programs almost have one-to-one correspondences to intended grammars' rules.

(iii) Attribute grammar rules are expressed as a function. As a result, to identify inherited or synthesized attributes is easily done since inherited attributes are functions' inputs and synthesized ones are functions' outputs. (note. LEAG supports multi-values) And the relations of inherited and synthesized attributes are also understood clearly because of the same fact.

(iv) LEAG supports pattern-directed syntax analysis and attribute (constraint) checking or composing mechanism. With these facilities, designers can easily describe grammars' specifications without annoying detail aspects.

On the other hand LEAG's disadvantages are:

(i) LEAG is a programming language not a declarative specification language. Hence, more powerful facilities, which

ordinal attribute-grammar-based translator systems have, can't be provided; for example, static attribute type checking, circularity checking between attributes, so on. (ii) LEAG supports only code generation mechanisms and no rigid semantics writing system. Hence, designers might write down incorrect or ambiguous semantics, so they should check the validity of the semantics by himself.

2.3 Semantic Grammar: A Combination Of Denotational Semantics And Attribute Grammars

Paulson's semantic grammars[6] are extended attribute grammars[1][2] whose attributes are the denotations from denotational semantics. The strong features of Paulson's system are:

(i) attributes are defined on the mathematical domains. Hence, formal treatment of semantics might be possible.

(ii) with DAG(directed acyclic graph) evaluator[2], the system can deal with every non-circular attribute grammar classes (note. the current version only allows LALR(1) input grammars) and enforce the several optimization to the DAGs.

Two defects of semantic grammars are:

(i) no dealing with continuation semantics.

(ii) no discriminating between static type checking and dynamic type checking.

3. Semantic Attribute Grammar Forms

Attribute grammars are powerful and practical specification languages. Therefore it is natural that attribute grammars will be extended to covering the formal semantics of programming languages. Paulson's semantic grammars is one attempt. The system, however, has the two defects as described in the previous chapter. Then we propose a new attribute grammar form called "semantic attribute grammar form". This form is similar to Paulson's grammars, but provides more clear concepts and lucid notations of

semantically-extended attribute grammars. To explain the concrete examples of semantic attribute grammar forms, we will write down the specifications of a mini language SMALL from Gordon[9] by it (see Appendix1).

3.1 Type Checking By Syntactic Attributes And Semantic Attributes

(1) syntactic/semantic attributes

Attribute grammars ordinally deal with the syntactic and context-sensitive properties, therefore all attributes appeared here hold syntactic information. We will call them "syntactic attributes" from now. On the other hand considering attributes as a denotation, we call them "semantic attributes". Paulson's grammars do not distinguish the properties of the two kinds of attributes clearly. If strong typed programming languages are considered, no syntactic attributes are needed to check the types of program entities in run time. If static bindings are possible, syntactic attributes may play the role of holding the information of certain allocations.

In semantic attribute grammar forms, syntactic attributes and semantic attributes are discriminated by the symbols "< ... >" and "[...]". The examples are:

```
expression<Tenv : Type>[Env,Kont,Store : Cont]
@identifier<: Name>[: Ide]
```

Figure 3.1 Attribute declarations

In Figure 3.1, attribute types are declared. Inherited attributes and synthesized attributes are separated by the symbol ":". For example, the rule symbol "expression" has two syntactic attributes and four semantic attributes, and the type of the inherited attribute of the expression is Tenv. The identifier has no inherited syntactic attributes, which is suggested by "<: Name>".

```
expression<TENV,TENV[NAME]>[env, kont, s1, cont] =
  @identifier<NAME>[ide]

  with cont = (env ide = 'unbound') -> err, kont(env ide);
```

Figure 3.2 Rule program

In Figure 3.2, the attribute variable TENV of the expression is an inherited syntactic attribute and whose type is Tenv. The identifier has one synthesized attribute NAME, which is not explicitly expressed by the description but expressed implicitly by the declaration of the identifier (see Figure 3.1).

(2) semantics of types

Type checking is inherently a syntactic process. One idea to describe it denotationally is to introduce the two types of meanings: "static meaning" and "dynamic meaning" [9] [11]. The static meanings of entities are types and the dynamic meanings are denotations of the types, that is set of values. For this purpose two denotations might be introduced: type environment 'Tenv' and type 'T' [9]:

```
Tenv = Ide -> [Type + 'unbound']

T = Type -> set of type value.
```

Figure 3.3 Type values

And domain equations must be extended for the compile-time type checking to be performed.

Our approach is exactly different. Attribute grammars may completely decide the types of the entities of programs in compile time if the compiled language is syntactically well typed. Also even if the language is not syntactically well typed and has a feature of dynamic bindings, syntactic

process must be separated from semantic process. Some of the type checking examples described by semantic attribute grammar forms are:

```
expression<TENV,TYPE>[env, kont, s1, cont] =
  "if" expression1<TENV,bool>
        [env, lam b.cond(cont1,cont2)(Bool? b), s1, cont]
  "then"  expression2<TENV,TYPE>[env, kont, s1, cont1]
  "else"  expression3<TENV,TYPE>[env, kont, s1, cont2];
```

Figure 3.4 Type checking

It is supposed that the type of expression1 is boolean and the types of expression2 and expression3 are the same. The constraint to be boolean is illustrated by the type constant "bool", and the sameness of the value of expression2 and that of expression3 is guaranteed by the same attribute variable name "TYPE". If the condition's value type is known in compile time, run-time checking suggested by "(Bool? b)" is unnecessary.

(3) constraints of attributes

As illustrated in the previous examples (Figure 3.4), semantic attribute grammar forms support the two type of constraints: to be identical and to be sameness. This idea and notation are borrowed from extended attribute grammars[1].

(4) attribute-directed parsing

On parsing syntactically ambiguous expressions, it is necessary to solve the conflict which expressions should be processed. Attribute-directed parsing method[10] solves the conflict by knowing the type of the attributes whose types are declared by attribute domains. For example, when we encounter the identifier conflict case which is a function name or an array name, it is solved by the synthesized attributes of expression "func(....)" and "array(...)" where func and array are type tags, which is proved by type

theories (see [10]).

```
expression<TENV,func(...)> =
  @identifier<NAME>
  "(" expression<TENV,TYPE> ")"

expression<TENV,array(...)> =
  @identifier<NAME>
  "(" expression<TENV,TYPE> ")"
```

Figure 3.5 Attribute-directed parsing

3.2 Attribute As a Continuation

Denotational semantics describe denotations as a function. Especially continuations might be represented as a higher order function. On the other hand attribute grammars embeds attributes into grammar symbols, therefore if attributes are expressed as a continuation then passing a continuation to another one must be described by a temporary attribute not by a grammar symbol itself. That is, in denotational semantics grammar symbols themselves have meanings but in attribute grammars the symbols pass meanings by attributes. From these inspects, we know that we must solve the following problems.

(1) allowing circular attribute grammars by DAG evaluator

Let's see the following example:

```
expression<TENV,TYPE>[env, kont, s1, cont] =
  expression1<TENV,TYPE>[env, lam e1.cont1, s1, cont]
  "="
  expression2<TENV,TYPE>
        [env, lam e2.kont equal(^e1,e2), s1, cont1];
```

Figure 3.6 Attributes as a continuation

The attribute "cont1" expressing a command continuation is a expression1's inherited attribute and expression2's synthesized attribute. Hence, this attribute grammar class is not a kind of "compile-time-evaluated" one. In general attribute grammars accept the circularity relations between attributes. DAG evaluator[2] may deal with non-absolutely circular attribute grammars. This evaluator constructs directed acyclic graphs, whose node called DAG is a function computing a value from the attributes associated with this function, and evaluate the graph recursively descent manner.

(2) scope of attributes

In Figure 3.6, the attribute e1 is not bounded within a lambda expression "lam e2.kont equal(^e1,e2)....". To solve this problem we introduce an annotation "^". This annotation suggests that the variable following "^" is evaluated dynamically. Here the variable e1 is evaluated first after the continuation attribute cont1, which include e1 as an argument of "equal", is passed from expression 2. The attribute cont1 has the following form on evaluation:

```
lamb e1.( ... lam e2.kont equal(e1,e2) ....)
```

Figure 3.7 Dynamic evaluation of attributes

Critical conditions related to this dynamic evaluations are released by delayed evaluations: for example "lam e1." or "lam e2.". Nevertheless we should consider another resolutions in the following example; that is, the variable r is not bounded in the function env and r is evaluated before evaluating env.

```
command<TENV>[env, cont, s1, cont1] =
  "begin"
  declaration<TENV1 : TENV2>
            [env, lam r.cont2, s1, cont1] ";"
  command<TENV2>[env, env[^r], s1, cont2]
```

```
"end";
```

Figure 3.8 Troublesome example of dynamic evaluation

3.3 Designing a Language SMALL in Semantic Attribute Grammar Forms

We have described the full syntax and semantics of SMALL[9] (see Appendix1) by semantic attribute grammar forms. SMALL itself is described in denotational semantics and every denotations are defined as a curried function. On the other hand semantic functions ("expression","command" for example) are expressed as an uncurried one and semantic domains ("command continuation","expression continuation" for example) as a curried one in semantic attribute grammar forms. Curried functions are powerful on passing itself to another denotation as a partially evaluated function, but need extra mechanisms to curry out them. They are necessary for designers to design realistic and large languages. Attributed symbols (functions), however, are inherently uncurried. How can we solve this problem? This will be our next theme.

4. Discussions

Semantic attribute grammar forms are only forms not programming languages, hence the programs written in the language L whose specification is described by semantic attribute grammar forms can not be executed. Our first starting point was to design and implement an executable definition system in which the semantics may be treated formally. From these reason, we wish to modify the notation of semantic attribute grammar forms so that they might be executable.

References

[1]
David A. Watt.
An Extended Attribute Grammar for Pascal.
Sigplan Notices 14, 60-74, 1979.

[2]
David A. Watt, Ole Lehrmann Madsen.
Extended Attribute Grammars.
Computer Journal Vol.26, No.2, 142-153, 1983.

[3]
Lloyd Allison.
Programming Denotational Semantics.
Computer Journal Vol.26, No.2, 164-174, 1983.

[4]
Mitchell Wand.
Deriving Target Code as a Representation of Continuation Semantics.
ACM TOPLAS Vol.4, No.3, 496-517, 1982.

[5]
Hiroyuki Matsuda.
A Language Based On Extended Attribute Grammars (LEAG):
Its Theory, Implementation, and Applications.
Master Thesis (in Japanese), The Department of
Information Science, Tokyo Institute of Technology, 1985.

[6]
Lawrence Paulson.
A Semantics-Directed Compiler Generator.
in ACM 8th POPL, pages 224-233, 1982.

[7]
Ole Lehrmann Madsen.
On Defining Semantics by the Means of
Extended Attribute Grammars.
in Semantics-Directed Compiler Generation,
ed. Neil D. Jones, Springer-Verlag, pages 259-299, 1980.

[8]

Ravi Sethi.
Control Flow Aspects of Semantics-Directed Compiling.
ACM TOPLAS Vol.5, No.4, 554-595, 1983.

[9]
Michael J. C. Gordon.
The Denotational Description of Programming Languages.
Springer-Verlag, 1979.

[10]
David A. Watt.
Rule Splitting and Attribute-Directed Parsing.
in Semantics-Directed Compiler Generation,
ed. Neil D. Jones, Springer-Verlag, pages 363-392, 1980.

[11]
James E. Donahue.
Complementary Definition of Programming Language Semantics.
Lecture Notes in Computer Science 42, Springer-Verlag, 1976.

[12]
S. C. Johnson.
Yacc-yet another compiler compiler.
Computer Science Tech. Rep. 32, Bell Laboratories, July 1975.

Appendix1 Syntax And Semantics Of SMALL Written In Semantic Attribute Grammar Forms

---

Meta Symbols:

\# syntactic attributes
<inh1 .. inhn : syn1 ... synn>

\# semantic attributes
[inh1 ... inhn : syn1 ... synn]

Attributes:

program<>[File : Ans]

```
expression<Tenv : Type>[Env,Kont,Store : Cont]
command<Tenv>[Env,Cont,Store : Cont]
declaration<Tenv : Tenv>[Env,Dont,Store : Cont]

@integer<>[: Num]
@true<>[: Bool]
@false<>[: Bool]
@identifier<: Name>[: Ide]

   note. @ is an annotation for primitive tokens.

-------------
Rule Program:

# program

[1]
program<>[in, cont s1] =
  "program" command<[]>[(), lam s.'stop', s1, cont]

  with s1=(in/'input');

# expression

[2]
r-value<TENV,TYPE>[env, kont, s1, cont] =
  expression<TENV,TYPE>[env, deref(Rv? kont), s1, cont];

[3]
expression<TENV,integer>[env, kont, s1, kont int] =
  @identifier<>[int];

[4]
expression<TENV,bool>[env, kont, s1, kont 'true'] =
  @true<>['true'];

[5]
expression<TENV,bool>[env, kont, s1, kont 'false'] =
  @false<>['false'];

[6]
expression<TENV,any>[env, kont, s1, cont] =
```

```
  "read"

   with cont = let in1=s1('input')
               in null(in1) -> 'error',
                               kont hd(in1) s([tl(in1)/'input']);

[7]
expression<TENV,TENV[NAME]>[env, kont, s1, cont] =
  @identifier<NAME>[ide]

  with cont = (env ide = 'unbound') -> err, kont(env ide);

[8]
expression<TENV,func(TYPE1->TYPE2)>[env, kont, s1, cont] =
  expression<TENV,TYPE1]>[env, lam f.cont1, s1, cont]
  "(" expression<TENV,TYPE2>[env, (Fun? ^f)kont, s1, cont1] ")";

[9]
expression<TENV,TYPE>[env, kont, s1, cont] =
  "if" expression<TENV,bool>
                [env, lam b.cond(cont1,cont2)(Bool? b), s1, cont]
  "then"  expression<TENV,TYPE>[env, kont, s1, cont1]
  "else"  expression<TENV,TYPE>[env, kont, s1, cont2];

[10]
expression<TENV,TYPE>[env, kont, s1, cont] =
  expression<TENV,TYPE>[env, lam e1.cont1, s1, cont]
  "+"
  expression<TENV,TYPE>[env, lam e2.kont plus(^e1,e2), s1, cont1];

[11]
expression<TENV,TYPE>[env, kont, s1, cont] =
  expression<TENV,TYPE>[env, lam e1.cont1, s1, cont]
  "="
  expression<TENV,TYPE>[env, lam e2.kont equal(^e1,e2), s1, cont1];

# commands

[12]
command<TENV>[env, cont, s1, cont1] =
  expression<TENV,TYPE>[env, lam l.cont2, s1, cont1]
  ":="
```

```
  r-value<TENV,TYPE>[env, update(Loc? ^l)cont, s1, cont2];

[13]
command<TENV>[env, cont, s1, cont1] =
  "output" expression<TENV,TYPE>
                    [env, lam e s.(env, cont), s1, cont1];

[14]
command<TENV>[env, cont, s1, cont1] =
  expression<TENV,proc(TYPE)>[env, lam p.cont2, s1, cont1]
    "(" expression<TENV,TYPE>[env, (Proc? ^p)cont, s1, cont2] ")";

[15]
command<TENV>[env, cont, s1, cont1] =
  "if" expression<TENV,bool>
                [env, lam b.cond(cont2,cont3)(Bool? b), s1, cont1]
  "then"  expression<TENV,TYPE>[env, kont, s1, cont2]
  "else"  expression<TENV,TYPE>[env, kont, s1, cont3];

[16]
command<TENV>[env, cont, s1, cont1] =
  "while" expression<TENV,bool>
                    [env, lam b.cond(cont2,cont)(Bool? b), s1, cont1]
  "do" command<TENV>[env, cont1, s1, cont2];

[17]
command<TENV1>[env, cont, s1, cont1] =
  "begin"
  declaration<TENV1 : TENV2>[env, lam r.cont2, s1, cont1] ";"
  command<TENV2>[env, env[^r], s1, cont2]
  "end";

[18]
command<TENV>[env, cont, s1, cont1] =
  command<TENV>[env, cont2, s1, cont1]
  ";"
  command<TENV>[env, cont, s1, cont2];

# declarations

[19]
declarations<TENV : TENV[NAME->int]>[env, dont, s1, cont] =
```

```
  "int" @identifier<NAME>[ide]
  ":="
  expression<TENV,int>[env, ref(lam l.dont(l/ide)), sl, cont];

[20]
declarations<TENV : TENV[NAME->bool]>[env, dont, sl, cont] =
  "bool" @identifier<NAME>[ide]
  ":="
  expression<TENV,bool>[env, ref(lam l.dont(l/ide)), sl, cont];

[21]
declarations<TENV : TENV[NAME->const]>[env, dont, sl, cont] =
  "const" @identifier<NAME>[ide]
  ":="
  expression<TENV,any>[env, ref(lam l.dont(l/ide)), sl, cont];

[22]
declarations<TENV : TENV[NAME->proc(any)]>
            [env, dont, sl, cont] =
  "proc" @identifier<NAME>[ide]
  "(" @identifier<NAME>[ide] ")"
  ";"
  command<TENV>[env[^e/idel], ^c, sl, contl]

  with cont = dont((lam c e.contl)/ide);

[23]
declarations<TENV : TENV[NAME->func(any->TYPE)]>
            [env, dont, sl, cont] =
  "func" @identifier<NAME>[ide]
  "(" @identifier<NAME>[ide] ")"
  ";"
  expression<TENV>[env[^e/idel], ^k, sl, contl]

  with cont = dont((lam k e.contl)/ide);

[24]
;no check for double definitions
declarations<TENVl : TENV3>[env, dont, sl, cont] =
  declarations<TENVl : TENV2>[env, lam rl.contl, sl, cont]
  ";"
  declarations<TENV2 : TENV3>[env[^rl], lam r2.dont(^rl[r2]), sl, cont];
```

```
---------
Resolution:

(* omission *)
```

Appendix2 Attribute Domain Declaration, and
Primitive Semantic Functions of SMALL[9]

```
-------------------------
Syntactic Attribute Domains:

# type
Type = int + bool + any + func(Type->Type) + proc(Type)

#type environment
Tenv = [Name -> Type]*

[] : null environment.

# name
Name = string



-------------------------
Semantic Attribute Domains:

# constants
  '...'

# primitive values
Loc : locations: 'input' 'output'
Num : numbers
Bool : boolean: 'true' 'false'
Ide : identifiers

# denotable values
Dv = Loc + Rv + Proc + Fun
```

```
# storable values
Sv = File + Rv

# expressible values
Ev = Dv

# right-values
Rv = Bool + Num

# files
File = Rv*

# function values
Fun = Kont -> Kont

# procedure values
Proc = Cont -> Kont

# final answers
Ans = 'error' + 'stop' + Rv x Ans

# environment
Env = Ide -> [Dv + 'unbound']

() = lam id.'unbound' : null environment

# store
Store = Loc -> [Sv + 'unused']

# command continuations
Cont = Store -> Ans

# expression continuations
Kont = Ev -> Cont

# declaration continuations
Dont = Cont -> Kont
```

Primitive Semantic Functions:

```
# substitution : D -> D
f[d'/d] == lam d1.d1=d -> d', fd.  where d',d,d1 in D

# condition : [D x D] -> Bool -> D
b -> d1,d2 == cond(d1,d2)b.

cond(d1,d2)b == d1 if b='true'
                d2 if b='false'

# domain check : D -> D
D? == lam k e.isD e -> k e, err

isD e = 'true' if e in Di
        'false' otherwise. where D=[D1 + ... + Dn]

# error
err == lam s. 'error'

# sequences
for D* = {(d1,d2,...,dn)| 0<n, di in [D + ()]},

hd(d1,d2,...,dn) = d1,
tl(d1,d2,...,dn) = (d2,...,dn).

# operation : [D x D] -> D
plus(a,b) == lam(a,b).a + b

equal(a,b) == lam(a,b).a = b

# update : Loc -> Cont -> Kont
update == lam l c e s.isSv e -> c(s[e/l]), error

# ref : Kont -> Kont
ref == lam k e s.let n=new(s)
                 in (n='error' -> 'error',
                                  update(n (k n)) e s
 note. function 'new' generates a new location.

# deref : Kont -> Kont
deref == lam k e s.isLoc e
             -> (s e = 'unused' -> 'error', k (s(e))s),
```

k e s

override : [D x D] -> D
a[b] : a and b are merged into the new domain where there
exists no same items.

dynamic evaluation
^ : the variable following this annotation
must be evaluated dynamically