Attribute Grammars as Record Calculus — A Structure-Oriented Denotational Semantics of Attribute Grammars by Using Cardelli's Record Calculus —

K. Gondow[1] & T. Katayama[2]

1: Dept. of Information Science, Japan Advanced Inst. of Science and Technology, 1-1 Asahidai Tatsunokuchi Nomi Ishikawa, 923-1292, Japan
gondow@jaist.ac.jp
2: Dept. of Information Science, Japan Advanced Inst. of Science and Technology, 1-1 Asahidai Tatsunokuchi Nomi Ishikawa, 923-1292, Japan
katayama@jaist.ac.jp

Abstract

In this paper, we present a new denotational semantics of attribute grammars (AGs) by using Cardelli's record calculus. This new denotational semantics is simple, natural and structure-oriented.

AGs have been considered useful in describing interactive programming environments as well as in specifying the semantics of programming languages. Using AGs, interactive programming environments are often described as attributed trees with several AG extensions, e.g., higher-order features, subtree replacement, and remote access. Unfortunately, it was not easy to compare various definitions for these extensions in a formal way. One of the reasons is that previous studies for AG semantics are not structure-oriented, that is, they are based on attribute valuation, not an attributed tree itself. For example, AG semantics based on attribute valuation can not deal directly with program transformation such as $a \times (b + c) \Rightarrow a \times b + a \times c$.

In our new semantics, an attributed tree is represented as a nested record to preserve the structural information of the attributed tree. This enables us to deal directly with attributed trees rather than attribute valuation as AG semantics. We also represent higher-order AGs, recursive AGs and OOAG as record calculus by extending the semantics to show that our semantics can formalize such AG extensions. Both of higher-order AGs and OOAG are computational models to deal with tree transformation depending on attribute values.

Keywords: attribute grammars, Cardelli's record calculus, structure-oriented denotational semantics,

1. Introduction

In this paper, we present a new denotational semantics of attribute grammars (AGs) [Knu68] [Knu71] [DJL88] based on Cardelli's record calculus [CW85][Car88]. This semantics is structure-oriented as well as natural and simple. Unlike previous works, an attributed tree is represented as a nested record to preserve its structural information.

AGs[Knu68][Knu71][DJL88] are a formal system for specifying semantics of programming languages, and many compiler generators are studied and implemented [Far84][KHZ82]. Since the latter half of 1980s, however, syntax-directed editors based on AGs have been considered useful in describing and generating interactive programming environments [RT89]. Declarative structures, separation of semantics and syntax definition, local description resulting in high readability and high maintainability, and clear description caused by functional computation of attributes are the positive characteristics of AGs.

Using AGs, interactive programming environments are often described as attributed trees with several AG extensions, e.g., higher-order AGs (HAGs)[VSK89][TC90], subtree replacement in the Synthesizer Generator[Gra96] and in object-oriented AGs(OOAG)[SK90][GISK93], recursive AGs(RAGs)[Far86], and remote access[Gra96][KW94][Hed99]. Unfortunately, it was not easy to compare various definitions for these extensions in a formal way. One of the reasons is that previous studies(e.g., [TK82][Joh87]) for AG semantics are not structure-oriented, that is, they are based on attribute valuation, not an attributed tree itself. For example, AG semantics based on attribute valuation can not deal directly with program transformation such as $a \times (b+c) \Rightarrow a \times b + a \times c$, since it focuses only on attribute values, not on the structure of an attributed tree.

In [TK82], Takeda and Katayama defined a semantics of AGs as a sequence of all attribute values in an attributed tree. In [Joh87], Johnsson defined it as a collection of functions to compute values of synthesized attributes. These semantics are essentially based on *attribute valuation*, not *an attributed tree* itself. Thus, these formal semantics lack the structural information in AGs, so they do not suit to formalize structure-oriented aspects of OOAG, HAGs, and so on.

Fig.1 shows the overview of the new semantics. A derivation tree is represented as a term like $\mathsf{p}_1(\mathsf{p}_2, \mathsf{p}_3)$, and an attributed tree is represented as a nested record like $\langle \mathrm{a} = 1, \mathrm{X} = \langle \mathrm{a} = 1 \rangle \rangle$ where, for example, a denotes an attribute and X denotes a nonterminal. A production rule p and its semantic rules $R(p)$ are translated into a function $p_{\mathcal{E}}$. In other words, an AG is represented as a set of $p_{\mathcal{E}}$. The semantic function $\mathcal{E}$ is the key of our AG semantics; the semantic function $\mathcal{E}$ corresponds to an attribute evaluator. The definition of $\mathcal{E}$ is defined as simple recursion on tree structures in Def.3.6 as follows.

$$\mathcal{E}[\![\mathsf{p}(t_1, \ldots, t_n)]\!] = \mu \mathrm{self}.p_{\mathcal{E}}(\mathcal{E}[\![t_1]\!], \ldots, \mathcal{E}[\![t_n]\!], \mathrm{self})$$

where $t_1, \ldots, t_n$ are subtrees of the derivation tree, and μ is a fixed-point operator.

We think the semantics is a good theoretical groundwork for modeling AG extensions (especially structure-oriented ones). To show it, we also represent HAGs, RAGs and OOAG as record calculus.

The rest of the paper is organized as follows. In the next section, Section 2, we present a brief background of AGs and Cardelli's record calculus. Section 3 provides the new AG semantics by using record calculus. In Section 4, we represent AG extensions as record calculus. Section 5 gives a simple implementation using SML/NJ[TL96]. Section 6 summarizes this paper and future works are presented in Section 7.

2. Backgrounds

2.1. Definition of Attribute Grammars

This section provides a tuple-style definition of AGs and some terminology.

An AG is defined by a 3-tuple $AG = (G, A, R)$, where G is an underlying context free grammar, A is a finite set of *attributes* and R is a finite set of *semantic rules*. A context free grammar is defined by a 4-tuple $G = (N, T, S, P)$, where N is a finite set of *nonterminals*, T is a finite set of *terminals*, $S \in N$ is a start symbol, and P is a finite set of production rules.

Each nonterminal is associated with two disjoint finite set $Inh(X)$ and $Syn(X)$, where $Inh(X) \cap Syn(X) = \phi, A = \bigcup_{X \in N}(Inh(X) \cup Syn(X))$. An element of $Inh(X)$ is called an *inherited attribute*,

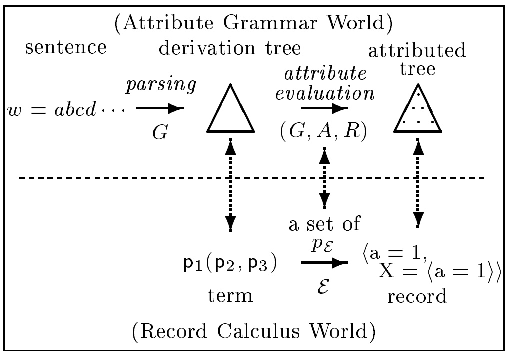

Figure 1: Relations among Attribute Grammars, Attribute Evaluator, Semantic Function $\mathcal{E}$, and Function $p_{\mathcal{E}}$

and that of $Syn(X)$ is called a *synthesized attribute.*

For a production rule $p : X_0 \rightarrow X_1 \cdots X_n$, we call an attribute a of X_i occurring in the semantic rules in $R(p)$ an *attribute occurrence*, which is written as $X_i \cdot a$, where $0 \leq i \leq n$ and $a \in Inh(X_i) \cup Syn(X_i)$[1]. The following $Occur(p)$ is a set of attribute occurrences that may occur in p.

$$Occur(p) = \{X_i \cdot a | (0 \leq i \leq n) \wedge (a \in Inh(X_i) \cup Syn(X_i))\}$$

A set $R(p)$ of semantic rules associated with a production rule $p : X_0 \rightarrow X_1 \cdots X_n$ is defined:

$$\begin{aligned} R(p) = \{X_i \cdot a = e | (i = 0 \wedge a \in Syn(X_0)) \vee \\ (1 \leq i \leq n \wedge a \in Inh(X_i)), e \in Exp(Occur(p))\} \end{aligned}$$

where $R = \bigcup_{p \in P} R(p)$, and $Exp(Occur(p))$ is a set of terms constructed by attribute occurrences $Occur(p)$ and function symbols.

For a given derivation tree on an AG, the denotational semantics of the AG is the attributed tree where all attribute values on the derivation tree satisfy their associated semantics rules. In other words, the semantic function of an AG is a mapping from any derivation tree to the attributed tree where all values of attributes are consistent.

A condition "attribute dependencies on any derivation tree is cycle free" is a sufficient condition (not a necessary one) to be able to compute all attribute values on the derivation tree. AGs that hold this condition are called *non-circular* AGs. This paper does not suppose non-circular AGs. A wider class of recursive AGs introduced by Farrow[Far86] is represented as records in Def.4.2.

2.1.1. An Example AG

Example 2.1 (AG_1: n-radix numerals)

$AG_1 = (G, A, R)$ is defined as follows.

$$\begin{aligned} N &= \{\mathrm{N}, \mathrm{I}, \mathrm{D}\ \} \\ T &= \{0, 1, \ldots, 9\ \} \\ S &= \mathrm{N} \\ P &= \{p_{\mathrm{N}} : \mathrm{N} \rightarrow \mathrm{I},\ p_{\mathrm{I1}} : \mathrm{I} \rightarrow \mathrm{I}\ \mathrm{D},\ p_{\mathrm{I2}} : \mathrm{I} \rightarrow \mathrm{D}, \\ & \quad p_{\mathrm{D0}} : \mathrm{D} \rightarrow 0,\ \ldots,\ p_{\mathrm{D9}} : \mathrm{D} \rightarrow 9\ \} \end{aligned}$$

[1]We use ' $\cdot$ ' to distinguish an attribute occurrence from record selecting operator ' . ' given in Section 2.2.

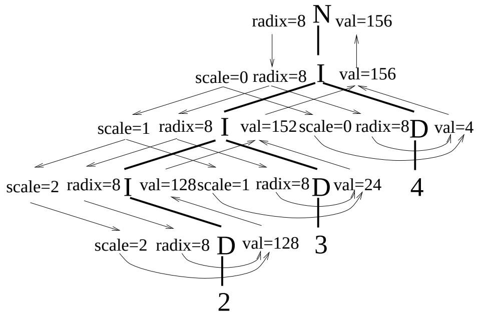

Figure 2: An Attributed Tree where $w = 234, \mathrm{N.radix} = 8$

$$
\begin{array}{rcl}
Inh(\mathrm{N}) & = & \{\mathrm{radix}\} \\
Inh(\mathrm{I}) & = & Inh(\mathrm{D}) = \{\mathrm{scale}, \mathrm{radix}\} \\
Syn(\mathrm{N}) & = & Syn(\mathrm{I}) = Syn(\mathrm{D}) = \{\mathrm{val}\} \\
R(p_{\mathrm{N}}) & = & \{\mathrm{I} \cdot \mathrm{scale} = 0, \\
 & & \mathrm{I} \cdot \mathrm{radix} = \mathrm{N} \cdot \mathrm{radix}, \\
 & & \mathrm{N} \cdot \mathrm{val} = \mathrm{I} \cdot \mathrm{val} \ \ \} \\
R(p_{\mathrm{I1}}) & = & \{\mathrm{I}_2 \cdot \mathrm{scale} = \mathrm{I}_1 \cdot \mathrm{scale} + 1, \\
 & & \mathrm{D} \cdot \mathrm{scale} = \mathrm{I}_1 \cdot \mathrm{scale}, \\
 & & \mathrm{I}_2 \cdot \mathrm{radix} = \mathrm{I}_1 \cdot \mathrm{radix}, \\
 & & \mathrm{D} \cdot \mathrm{radix} = \mathrm{I}_1 \cdot \mathrm{radix}, \\
 & & \mathrm{I}_1 \cdot \mathrm{val} = \mathrm{I}_2 \cdot \mathrm{val} + \mathrm{D} \cdot \mathrm{val} \ \ \} \\
R(p_{\mathrm{I2}}) & = & \{\mathrm{D} \cdot \mathrm{scale} = \mathrm{I} \cdot \mathrm{scale}, \\
 & & \mathrm{D} \cdot \mathrm{radix} = \mathrm{I} \cdot \mathrm{radix}, \\
 & & \mathrm{I} \cdot \mathrm{val} = \mathrm{D} \cdot \mathrm{val} \ \ \} \\
R(p_{\mathrm{D}i}) & = & \{\mathrm{D} \cdot \mathrm{val} = i \times \mathrm{D} \cdot \mathrm{radix} \ \uparrow \ \mathrm{D} \cdot \mathrm{scale} \ \ \}
\end{array}
$$

We assume that the value of inherited attribute $\mathrm{N} \cdot \mathrm{radix}$ is given a priori. Symbols $+$, $\times$, and $\uparrow$ are infix operators for addition, multiplication, and power respectively. To distinguish between different occurrences of the same nonterminal I in a production, indexing is used like I_1 and I_2. $R(p_{\mathrm{D}i}) = \{\cdots\}$ is an abbreviation for each definition of $R(p_{\mathrm{D0}}) \cdots R(p_{\mathrm{D9}})$.

□

Readers can understand AG_1 intuitively as follows. Let the value of $\mathrm{N} \cdot \mathrm{radix}$ be n, then the value of $\mathrm{N} \cdot \mathrm{val}$ indicates the value of $w \in L(G)$ as n-radix numeral. Fig.2 shows attributed trees where $w = 234$ and $\mathrm{N} \cdot \mathrm{radix} = 8$. Arrows in Fig.2 indicate dependencies between attribute instances ($a \rightarrow b$ represents "b depends on a").

Any traditional semantics is enough when you need only the value of $\mathrm{N} \cdot \mathrm{val}$, but not when you want to handle attribute trees as databases (or interactive programming environments). For example, you may want to search leaves whose values are 3 and change them to 0. The traditional semantics can not deal with such cases. Therefore we need a new semantics that handle not only attribute values but also attributed trees.

2.2. Definition of Cardelli's Record Calculus

In this section, we briefly explain a record and its calculus introduced by Cardelli[CW85][Car88]. We do not explain a record type, since it is beyond the scope of this paper to consider the aspects of record types. A Cardelli's record is a finite mapping from labels to values. A record is written[2]:

$$\langle l_1 = v_1, \ldots, l_n = v_n \rangle$$

where $l_1, \ldots, l_n$ are labels, and $v_1, \ldots, v_n$ are associated values, respectively. Each $l_i = v_i$ $(1 \leq i \leq n)$ is called a field. The following is an example record.

$$\langle a = 1, b = \text{false} \rangle$$

Selecting the value associated with a label l of a record r is given by $r.l$. Therefore, the value of the following expression is 1.

$$\langle a = 1, b = \text{false} \rangle.a$$

We assume the selecting operator is left associative, so we abbreviate $(\cdots((r.l_1).l_2)\cdots.l_n)$ to $r.l_1.l_2.\cdots.l_n$, where r is a record and $l_1, l_2, \ldots, l_n$ are labels. For example, the value of following expression is false.

$$\langle a = \langle a = 1, b = \text{false} \rangle, b = \text{true} \rangle.a.b$$

A record calculus used in this paper is λ-calculus with the following reduction rule $(1 \leq i \leq n)$.

$$\langle l_1 = v_1, \ldots, l_n = v_n \rangle.l_i \Rightarrow v_i$$

2.3. Objects and Classes as Records

Many studies have been done to formalize and discuss the various concepts in object-oriented programming such as objects, classes, encapsulation and inheritance by using record calculus (e.g., [Car88][CP89][CCHO89][CHC90]). This section gives a brief review for objects and classes represented as records.

- *Objects* are represented as records whose fields are methods and instance variables. For example, an object *point* has two instance variables x and y and one method *dist*.

  $$point = \langle x = 10, y = 20, dist = sqrt(x^2 + y^2) \rangle$$

  Note that the method *dist* refers to labels x and y in *point*. Thus, the methods of an object may refer to each other. To eliminate references to labels x and y, a binding variable *self* and a fixed-point operator μ are traditionally used; the variable *self* corresponds to pseudo-variable self in object-oriented programming languages.

  $$\begin{aligned} point &= \langle x = 10, y = 20, dist = sqrt(point.x^2 + point.y^2) \rangle \\ &= (\lambda self.\langle x = 10, y = 20, dist = sqrt(self.x^2 + self.y^2) \rangle)(point) \\ &= \mu self.\langle x = 10, y = 20, dist = sqrt(self.x^2 + self.y^2) \rangle \end{aligned}$$

  where μ is a fixed-point operator. A fixed-point of f (that is, x that satisfies $x = f(x)$) is represented as $\mu x.f(x)$. The fixed-point operator has the unrolling rule:

  $$\mu x.f(x) = f(\mu x.f(x))$$

[2]In [CW85][Car88], symbols '{' and '}' are used as record constructors. But, to distinguish between a record and a set explicitly, we use '⟨' and '⟩' for record instead of '{' and '}'.

For readability, we also use a fixed-point combinator $\mathbf{Y}$.

$$\mu x.f(x) = \mathbf{Y}(f)$$

The combinator $\mathbf{Y}$ has the following reduction rule.

$$\mathbf{Y}(f) \Rightarrow f(\mathbf{Y}(f))$$

- *Classes* are parameterized objects (i.e. functions) that return object records.

$$\mathit{pointclass} = \lambda i_x.\lambda i_y.\mu \mathit{self}.\langle x = i_x, y = i_y, \mathit{dist} = \mathit{sqrt}(\mathit{self}.x^2 + \mathit{self}.y^2)\rangle$$

  The methods of an object may refer to any class to create new instances, so a class may be (mutual) recursively defined. In the following example, we use *myclass* and μ to eliminate a recursive occurrence of *pointclass*.

$$\begin{aligned} \mathit{pointclass} &= \lambda i_x.\lambda i_y.\mu \mathit{self}.\langle x = i_x, y = i_y, \mathit{dist} = \mathit{sqrt}(\mathit{self}.x^2 + \mathit{self}.y^2), \\ &\quad\ \mathit{move} = \lambda dx.\lambda dy.\mathit{pointclass}(\mathit{self}.x + dx)(\mathit{self}.y + dy)\rangle \\ &= \mu \mathit{myclass}.\lambda i_x.\lambda i_y.\mu \mathit{self}.\langle x = i_x, y = i_y, \mathit{dist} = \mathit{sqrt}(\mathit{self}.x^2 + \mathit{self}.y^2), \\ &\quad\ \mathit{move} = \lambda dx.\lambda dy.\mathit{myclass}(\mathit{self}.x + dx)(\mathit{self}.y + dy)\rangle \end{aligned}$$

3. Formalizing AGs by Using Record Calculus

In this section, we define a new denotational semantics of AGs by using record calculus.

First, an attributed tree is represented as a nested record (Section 3.1). In Section 3.2, we introduce a function $p_{\mathcal{E}}$, which corresponds to a production rule p and a set of semantic rules $R(p)$. Here we use a term as a linear notation to express a derivation tree. In Section 3.3, we define the semantic function $\mathcal{E}$. We give an example to show a process of computing attributed trees from derivation trees (Section 3.4).

3.1. Record Representation for Attributed trees

In this section, we show how to represent an attributed tree as records.

Each node in an attributed tree has:

- attributes and their values
- subtrees

Therefore, it is natural to regard an attributed tree as a record which has fields for both attributes and subtrees.

Definition 3.1 (Record Representation for an Attributed Tree)

Let the top production rule of an attributed tree be $p : X_0 \to X_1 \cdots X_n$. The record representation for an attributed tree is defined by the record that has the following fields:

1. fields whose labels are X_i $(1 \leq i \leq n)$, and whose values are records that represents attributed subtrees rooted in X_i $(1 \leq i \leq n)$, respectively
2. fields whose labels are attributes $a \in \mathit{Inh}(X_0) \cup \mathit{Syn}(X_0)$, and whose values are values of attribute instances, respectively

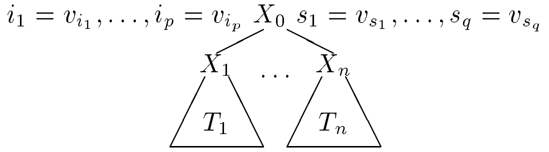

Figure 3: An Attributed Tree: T_0

□

To illustrate Def.3.1, consider an attributed tree in Fig.3. An attributed tree T_0 in Fig.3 is represented as the following record by Def.3.1:

$$\begin{array}{lll} T_0 & = \langle & i_1 = v_{i_1}, \ldots, i_p = v_{i_p}, \\ & & s_1 = v_{s_1}, \ldots, s_q = v_{s_q}, \\ & & X_1 = T_1, \ldots, X_n = T_n \quad \rangle \end{array}$$

where $i_j (1 \leq j \leq p)$ is an instance of an inherited attribute in $Inh(X_0)$, $s_k (1 \leq k \leq q)$ is an instance of a synthesized attribute in $Syn(X_0)$, v_a is the value of an attribute instance a, and $T_l (1 \leq l \leq n)$ is a record which represents an attributed subtree rooted in X_l.

For example, the following record represents an attributed tree N in Fig.2.

$$\begin{array}{l} \mathrm{N} = \langle \mathrm{radix} = 8, \mathrm{val} = 156, \\ \quad \mathrm{I} = \langle \mathrm{scale} = 0, \mathrm{radix} = 8, \mathrm{val} = 156, \\ \quad\quad \mathrm{I}_2 = \langle \mathrm{scale} = 1, \mathrm{radix} = 8, \mathrm{val} = 152, \\ \quad\quad\quad \mathrm{I}_2 = \langle \mathrm{scale} = 2, \mathrm{radix} = 8, \mathrm{val} = 128, \\ \quad\quad\quad\quad \mathrm{D} = \langle \mathrm{scale} = 2, \mathrm{radix} = 8, \mathrm{val} = 128 \rangle \rangle, \\ \quad\quad\quad \mathrm{D} = \langle \mathrm{scale} = 1, \mathrm{radix} = 8, \mathrm{val} = 24 \rangle \rangle, \\ \quad\quad \mathrm{D} = \langle \mathrm{scale} = 0, \mathrm{radix} = 8, \mathrm{val} = 4 \rangle \rangle \rangle \end{array}$$

3.2. Translating from an AG to a Set of Functions $p_{\mathcal{E}}$

In this section, we define a function $p_{\mathcal{E}}$, which we need later in Section 3.3 to define the semantic function $\mathcal{E}$. A function $p_{\mathcal{E}}$ has the following information.

1. a production rule $p : X_0 \rightarrow X_1 \cdots X_n$
2. a set of semantic rules $R(p)$
3. a set of attributes $Syn(X_0) \cup Inh(X_0)$

Roughly speaking, input of $p_{\mathcal{E}}$ is (1) attributed subtrees λ-abstracted with inherited attributes on their roots ('child$_i$' in Def.3.2), and (2) an attributed tree λ-abstracted with inherited attributes on its root (a 'self' in Def.3.2), which is equal to $p_{\mathcal{E}}$'s output. Output (=self) of $p_{\mathcal{E}}$ has the top production rule p on its root. Thus, an attributed tree 'self' is recursively defined by $p_{\mathcal{E}}$, attributed subtrees 'child$_1$',…,'child$_n$', and 'self' itself as follows.

$$\mathrm{self} = p_{\mathcal{E}}(\mathrm{child}_1, \ldots, \mathrm{child}_n, \mathrm{self})$$

(See Def.3.2,3.6 for the formal definitions.)

Definition 3.2 (Function $p_{\mathcal{E}}$)

For a production rule $p : X_0 \to X_1 \cdots X_n$, let $Inh(X_j)$, $Syn(X_j)$ $(0 \le j \le n)$ and $R(p)$ be as follows.

$$\begin{array}{rcl} Inh(X_j) & = & \{i_{j,1}, \ldots, i_{j,p_j}\} \\ Syn(X_j) & = & \{s_{j,1}, \ldots, s_{j,q_j}\} \end{array}$$

$$\begin{array}{rl} R(p) \quad = \{ & X_0 \cdot s_{0,1} = e_{0,1}, \ldots, X_0 \cdot s_{0,q_0} = e_{0,q_0}, \\ & X_1 \cdot i_{1,1} = e_{1,1}, \ldots, X_1 \cdot i_{1,p_1} = e_{1,p_1}, \\ & \vdots \\ & X_n \cdot i_{n,1} = e_{n,1}, \ldots, X_n \cdot i_{n,p_n} = e_{n,p_n} \quad \} \end{array}$$

Then, $p_{\mathcal{E}}$ is defined as follows,

$$\begin{array}{lcl} p_{\mathcal{E}} & = & \lambda \mathrm{child}_1 \cdots \lambda \mathrm{child}_n . \lambda \mathrm{self} . \lambda \mathrm{inh}_1 \cdots \lambda \mathrm{inh}_{p_0} . \langle \\ & & i_{0,1} = \mathrm{inh}_1, \ldots, i_{0,p_0} = \mathrm{inh}_{p_0}, \\ & & s_{0,1} = e'_{0,1}, \ldots, s_{0,q_0} = e'_{0,q_0}, \\ & & X_1 = \mathrm{child}_1(e'_{1,1}, \ldots, e'_{1,p_1}), \\ & & \quad \vdots \\ & & X_n = \mathrm{child}_n(e'_{n,1}, \ldots, e'_{n,p_n}) \ \rangle \end{array}$$

where for any j, k $((j = 0 \wedge 1 \le k \le q_0) \vee (1 \le j \le n \wedge 1 \le k \le p_j))$, $e'_{j,k}$ is a term where all attribute occurrences α are replaced with $\alpha_{\mathcal{E}}$ in $e_{j,k}$(each right-hand side of a production rule p). A term $\alpha_{\mathcal{E}}$ is defined as follows.

$$\alpha_{\mathcal{E}} = \begin{cases} \mathrm{self}(\mathrm{inh}_1, \ldots, \mathrm{inh}_{p_0}).X_i.a & (\textbf{if } \alpha = X_i \cdot a \wedge 1 \le i \le n) \\ \mathrm{self}(\mathrm{inh}_1, \ldots, \mathrm{inh}_{p_0}).a & (\textbf{if } \alpha = X_0 \cdot a) \end{cases}$$

□

In Def.3.2, the reason why 'self' appears recursively in the body of $p_{\mathcal{E}}$ as its argument is:

1. $p_{\mathcal{E}}$ is a constructor i.e. a mapping from attributed subtrees 'child$_i$' to an attributed tree 'self', that is, 'self' depends on 'child$_i$'.
2. Inherited attributes in 'child$_i$' may depend on inherited attributes in 'self', that is, 'child$_i$' depends on 'self'.
3. From (1) and (2), 'self' depends on 'self' itself, so 'self' needs to be defined recursively.

If non-circular AGs are supposed, it is, of course, possible to define AG semantics without μ operator nor other recursive directives like **letrec**, but the definition would be so complicated; using μ operator helps to make the definition simple.

The definition of $\alpha_{\mathcal{E}}$ needs to be divided into two cases in Def.3.2, since "$X_0 \cdot a$ is an attribute *of* 'self', but on the other hand $X_i \cdot a (1 \le i \le n)$ is an attribute *of* 'child$_i$' *of* 'self' ", where *of* corresponds to the record selector '.'.

For function symbols 'self' and 'child$_i$' in Def.3.2, a notation $f(x)$ represents an application of function f to an argument x, and $f(x_1)(x_2)\ldots(x_n)$ is abbreviated to $f(x_1, \ldots, x_n)$. And a nullary function $f()$ will be also abbreviated to f.

3.3. Definition of the Semantic Function $\mathcal{E}$

This section presents the definition of the semantic function $\mathcal{E}$ by using $p_{\mathcal{E}}$ defined in Section 3.2. The semantic function defined here is a mapping from a set of derivation trees $\mathbb{T}$ to a set of attributed trees $\mathbb{R}$. More precisely, $\mathbb{T}$ is a set of terms which represents derivation trees (Def.3.3), and $\mathbb{R}$ is a set of records which represent attributed trees as semantics of the AG.

Before defining $\mathcal{E}$, we provide a term representation for an attributed tree and a definition of $\mathbb{T}$, then we define semantic function $\mathcal{E}$.

Definition 3.3 (Term Representation for an Attributed Tree)

We define a set of sorts N and a set of operators P associated with a set of nonterminals N and a set of production rules P, respectively. For each production rule $p : X_0 \to X_1 \cdots X_n \in P$, we also define its arity and sort as follows.

$$\begin{aligned} \mathrm{arity}(\mathsf{p}) &= \mathsf{X}_1\mathsf{X}_2 \cdots \mathsf{X}_n \\ \mathrm{sort}(\mathsf{p}) &= \mathsf{X}_0 \end{aligned}$$

In other words, we regard a production rule p as a function symbol p typed:

$$\mathsf{p} : \mathsf{X}_1, \ldots, \mathsf{X}_n \mapsto \mathsf{X}_0$$

Let $X_0 \to X_1 \cdots X_n$ be a production rule at the root of a derivation tree T_0, and let $t_i (1 \leq i \leq n)$ be a term which represents a derivation subtree rooted in X_i. Then, a term that represents a derivation tree T_0 is defined by $\mathsf{p}(t_1, \ldots, t_n)$.

□

For example, a term that represents a derivation tree in Fig.2 produced by the AG of Example 2.1 is as follows.

$$\mathsf{p}_{\mathrm{N}}(\mathsf{p}_{\mathrm{I1}}(\mathsf{p}_{\mathrm{I1}}(\mathsf{p}_{\mathrm{I2}}(\mathsf{p}_{\mathrm{D2}}), \mathsf{p}_{\mathrm{D3}}), \mathsf{p}_{\mathrm{D4}}))$$

Definition 3.4 (Term Set of Derivation Trees)

We define recursively $\mathcal{T}_{\mathsf{X}_0}(\Sigma_G, \phi)$ (a set of ground terms of sort X_0) produced by a signature $\Sigma_G = (\mathsf{N}, \mathsf{P})$ associated with a grammar G as follows.

1. If arity(p)=ϕ and sort(p)=X_0, then $\mathsf{p} \in \mathcal{T}_{\mathsf{X}_0}(\Sigma_G, \phi)$
2. If arity(p)=$\mathsf{X}_1 \ldots \mathsf{X}_n$,sort($\mathsf{p}$)=$\mathsf{X}_0$ and $t_i \in \mathcal{T}_{\mathsf{X}_i}(\Sigma_G, \phi)$ where $1 \leq i \leq n$, then $\mathsf{p}(t_1, \ldots, t_n) \in \mathcal{T}_{\mathsf{X}_0}(\Sigma_G, \phi)$

□

$\mathcal{T}_{\mathsf{X}_0}(\Sigma_G, \phi)$ is a set of terms that represents derivation trees whose roots are labeled by X_0. Here ϕ means that $\mathcal{T}_{\mathsf{X}_0}(\Sigma_G, \phi)$ has no variables.

Definition 3.5 ($\mathbb{T}$: Term Set of Grammar G)

A term set $\mathbb{T}$ of derivation trees produced by grammar $G = (N, T, S, P)$ is defined as follows.

$$\mathbb{T} = \bigcup_{\mathsf{X} \in \mathsf{N}} \mathcal{T}_{\mathsf{X}}(\Sigma_G, \phi)$$

□

Note that $\mathbb{T}$ includes not only $\mathcal{T}_{\mathsf{S}}(\Sigma_G, \phi)$ (a term set for derivation trees whose roots are labeled with the start symbol S), but also $\mathcal{T}_{\mathsf{X}}(\Sigma_G, \phi)$ whose roots are labeled with any other nonterminals $X \in N$.

Def.3.3,3.4 and 3.5 were originally defined by Vogt et al. in [VSK89], and we slightly modified them to fit our notations.

Definition 3.6 (A Semantic Function of AGs: $\mathcal{E}$)

A semantic function $\mathcal{E} : \mathbb{T} \to \mathbb{R}$ is defined here, which is a mapping from a set of terms that represents derivation trees to a set of records that represents attributed trees.

For a production rule $p : X_0 \to X_1 \cdots X_n$, and $t_i \in \mathcal{T}_{\mathsf{X}_i}(\Sigma_G, \phi)$ where $1 \le i \le n$, the semantics of a derivation tree $\mathsf{p}(t_1, \ldots, t_n)$ is defined by using the semantic function $\mathcal{E}$ as follows:

$$\mathcal{E}[\![\mathsf{p}(t_1, \ldots, t_n)]\!] = \mu\text{self}.p_{\mathcal{E}}(\mathcal{E}[\![t_1]\!], \ldots, \mathcal{E}[\![t_n]\!], \text{self})$$

□

Here we illustrate the meaning of Def.3.6. Def.3.6 defines that the semantics $\mathcal{E}[\![\mathsf{p}(t_1, \ldots, t_n)]\!]$ of a derivation tree $\mathsf{p}(t_1, \ldots, t_n)$ is the record that represents its attributed tree. This is obtained by applying a function $p_{\mathcal{E}}$ to attributed subtrees $\text{child}_1, \ldots, \text{child}_n$, and self.

$$\mathcal{E}[\![\mathsf{p}(t_1, \ldots, t_n)]\!] = p_{\mathcal{E}}(\text{child}_1, \ldots, \text{child}_n, \text{self})$$

We can derive Def.3.6 from the three facts: (1) a right-hand side of this expression is equal to 'self' itself, (2) $\text{child}_i = \mathcal{E}[\![t_i]\!]$ $(1 \le i \le n)$, and (3) the definition of μ, as follows.

$$\begin{array}{rcl} \text{self} & = & p_{\mathcal{E}}(\text{child}_1, \ldots, \text{child}_n, \text{self}) \\ & = & \mu\text{self}.p_{\mathcal{E}}(\text{child}_1, \ldots, \text{child}_n, \text{self}) \\ & = & \mu\text{self}.p_{\mathcal{E}}(\mathcal{E}[\![t_1]\!], \ldots, \mathcal{E}[\![t_n]\!], \text{self}) \\ \mathcal{E}[\![\mathsf{p}(t_1, \ldots, t_n)]\!] & = & \mu\text{self}.p_{\mathcal{E}}(\mathcal{E}[\![t_1]\!], \ldots, \mathcal{E}[\![t_n]\!], \text{self}) \end{array}$$

Note that if $Inh(S) \neq \phi$, $\mathcal{E}[\![t]\!]$ is a record λ-abstracted with elements of $Inh(S)$ as defined in Def.3.2. For example, let t be the derivation tree in Fig.2, then we need to apply $\mathcal{E}[\![t]\!]$ to 8 (that is, $\mathcal{E}[\![t]\!](8)$) to obtain the attributed tree of Fig.2 with $\text{N} \cdot \text{radix} = 8$.

3.4. An Example of Evaluating an AG as Records

In this section, we show an evaluating process of an AG as record calculus in order to provide intuitive understanding of $\mathcal{E}$ defined in Section 3.3.

First, we translate the example AG of Section 2.1.1 into a set of functions $p_{\mathcal{E}}$ for all $p \in P$. Then, we apply the semantic function $\mathcal{E}$ to $t = \mathsf{p}_{\text{N}}(\mathsf{p}_{\text{I1}}(\mathsf{p}_{\text{I1}}(\mathsf{p}_{\text{I2}}(\mathsf{p}_{\text{D2}}), \mathsf{p}_{\text{D3}}), \mathsf{p}_{\text{D4}}))$, resulting in a record that represents an attributed tree in Fig. 2.

By Def.3.2, we can translate the example AG into the following functions.

$p_{\text{N}\mathcal{E}} = \lambda\text{child}_1.\lambda\text{self}.\lambda\text{inh}_1.\langle\text{radix} = \text{inh}_1,\ \text{val} = \text{self}(\text{inh}_1).\text{I.val}, \text{I} = \text{child}_1(0,\ \text{self}(\text{inh}_1).\text{radix})\rangle$
$p_{\text{I1}\mathcal{E}} = \lambda\text{child}_1.\lambda\text{child}_2.\lambda\text{self}.\lambda\text{inh}_1.\lambda\text{inh}_2.\langle\text{scale} = \text{inh}_1,\ \text{radix} = \text{inh}_2,$
$\quad \text{val} = \text{self}(\ \text{inh}_1,\ \text{inh}_2).\text{I}_2.\text{val}+\ \text{self}(\text{inh}_1,\ \text{inh}_2).\text{D.val},$
$\quad \text{I}_2 = \text{child}_1(\ \text{self}(\text{inh}_1,\ \text{inh}_2).\text{scale}{+}1,\ \text{self}(\text{inh}_1,\ \text{inh}_2).\text{radix}),$
$\quad \text{D} = \text{child}_2(\ \text{self}(\text{inh}_1,\ \text{inh}_2).\text{scale},\ \text{self}(\text{inh}_1,\ \text{inh}_2).\text{radix})\rangle$

$$
\begin{array}{l}
\mathcal{E}[\![\mathsf{p}_{\mathrm{N}}(\mathsf{p}_{\mathrm{I1}}(\mathsf{p}_{\mathrm{I1}}(\mathsf{p}_{\mathrm{I2}}(\mathsf{p}_{\mathrm{D2}}),\mathsf{p}_{\mathrm{D3}}),\mathsf{p}_{\mathrm{D4}}))]\!]\\
= \lambda \mathrm{inh}_1.\langle \mathrm{radix} = \mathrm{inh}_1, \mathrm{val} = ((2\times \mathrm{inh}_1 \uparrow 2) + (3\times \mathrm{inh}_1)) + 4,\\
\quad \mathrm{I} = \langle \mathrm{scale} = 0, \mathrm{radix} = \mathrm{inh}_1, \mathrm{val} = ((2\times \mathrm{inh}_1 \uparrow 2) + (3\times \mathrm{inh}_1)) + 4,\\
\quad\quad \mathrm{I}_2 = \langle \mathrm{scale} = 1, \mathrm{radix} = \mathrm{inh}_1, \mathrm{val} = (2\times \mathrm{inh}_1 \uparrow 2) + (3\times \mathrm{inh}_1),\\
\quad\quad\quad \mathrm{I}_2 = \langle \mathrm{scale} = 2, \mathrm{radix} = \mathrm{inh}_1, \mathrm{val} = 2\times \mathrm{inh}_1 \uparrow 2,\\
\quad\quad\quad\quad \mathrm{D} = \langle \mathrm{scale} = 2, \mathrm{radix} = \mathrm{inh}_1, \mathrm{val} = 2\times \mathrm{inh}_1 \uparrow 2\rangle\rangle\\
\quad\quad\quad \mathrm{D} = \langle \mathrm{scale} = 1, \mathrm{radix} = \mathrm{inh}_1, \mathrm{val} = 3\times \mathrm{inh}_1\rangle\rangle\\
\quad\quad \mathrm{D} = \langle \mathrm{scale} = 0, \mathrm{radix} = \mathrm{inh}_1, \mathrm{val} = 4\rangle\rangle\rangle
\end{array}
$$

Figure 4: Result of Example Record Calculation

$p_{\mathrm{I2}\mathcal{E}} = \lambda$child$_1$.$\lambda$self.$\lambda$inh$_1$.$\lambda$inh$_2$.$\langle$scale $=$ inh$_1$, radix $=$ inh$_2$, val$=$ self(inh$_1$, inh$_2$).D.val,
D $=$ child$_1$(self(inh$_1$, inh$_2$).scale, self(inh$_1$, inh$_2$).radix)$\rangle$
$p_{\mathrm{D}i\mathcal{E}} = \lambda$self.$\lambda$inh$_1$.$\lambda$inh$_2$.$\langle$scale $=$ inh$_1$, radix $=$ inh$_2$,val $= i\times$ self(inh$_1$, inh$_2$).radix$\uparrow$ self(inh$_1$, inh$_2$).scale$\rangle$

Similar to Def.2.1, '$p_{\mathrm{D}i\mathcal{E}} = \ldots$' is an abbreviation for each definition of $p_{\mathrm{D0}\mathcal{E}}, \cdots, p_{\mathrm{D9}\mathcal{E}}$.

Here we apply the semantic function $\mathcal{E}$ to a term $t = \mathsf{p}_{\mathrm{N}}(\mathsf{p}_{\mathrm{I1}}(\mathsf{p}_{\mathrm{I1}}(\mathsf{p}_{\mathrm{I2}}(\mathsf{p}_{\mathrm{D2}}),\mathsf{p}_{\mathrm{D3}}),\mathsf{p}_{\mathrm{D4}}))$ that represents the derivation tree for a word "234". That is, we calculate $\mathcal{E}[\![\mathsf{p}_{\mathrm{N}}(\mathsf{p}_{\mathrm{I1}}(\mathsf{p}_{\mathrm{I1}}(\mathsf{p}_{\mathrm{I2}}(\mathsf{p}_{\mathrm{D2}}),\mathsf{p}_{\mathrm{D3}}),\mathsf{p}_{\mathrm{D4}}))]\!]$ as follows.

$$
\begin{array}{lr}
\mathcal{E}[\![\mathsf{p}_{\mathrm{D2}}]\!] & \\
= \mu\mathrm{self}.p_{\mathrm{D2}\mathcal{E}}(\mathrm{self}) & [\text{By Def.3.6}]\\
= \mathbf{Y}(p_{\mathrm{D2}\mathcal{E}}) \qquad \cdots(1) & [\mu x.f(x) = \mathbf{Y}(f)]\\
= p_{\mathrm{D2}\mathcal{E}}(\mathbf{Y}(p_{\mathrm{D2}\mathcal{E}})) & [\mathbf{Y}(f) = f(\mathbf{Y}(f))]\\
= \lambda \mathrm{inh}_1.\lambda \mathrm{inh}_2.\langle\ \mathrm{scale} = \mathrm{inh}_1, \mathrm{radix} = \mathrm{inh}_2, & [\text{By applying } p_{\mathrm{D2}\mathcal{E}} \text{ to } \mathbf{Y}(p_{\mathrm{D2}\mathcal{E}})]\\
\quad \mathrm{val} = 2\times \mathbf{Y}(p_{\mathrm{D2}\mathcal{E}})(\mathrm{inh}_1,\mathrm{inh}_2).\mathrm{radix} \uparrow \mathbf{Y}(p_{\mathrm{D2}\mathcal{E}})(\mathrm{inh}_1,\mathrm{inh}_2).\mathrm{scale}\rangle \quad \cdots(2) & \\
= \lambda \mathrm{inh}_1.\lambda \mathrm{inh}_2.\langle\ \mathrm{scale} = \mathrm{inh}_1, \mathrm{radix} = \mathrm{inh}_2,\ \mathrm{val} = 2\times \mathrm{inh}_2 \uparrow \mathrm{inh}_1\rangle & [(1){=}(2)]
\end{array}
$$

Similarly, we can show the rest part of the calculation.

$$
\begin{array}{l}
\mathcal{E}[\![\mathsf{p}_{\mathrm{I2}}(\mathsf{p}_{\mathrm{D2}})]\!]\\
= \mu\mathrm{self}.p_{\mathrm{I2}\mathcal{E}}(\mathcal{E}[\![\mathsf{p}_{\mathrm{D2}}]\!], \mathrm{self}) = \mu\mathrm{self}.p_{\mathrm{I2}\mathcal{E}}(\mathcal{E}[\![\mathsf{p}_{\mathrm{D2}}]\!])(\mathrm{self})\\
= \mathbf{Y}(p_{\mathrm{I2}\mathcal{E}}(\mathcal{E}[\![\mathsf{p}_{\mathrm{D2}}]\!])) = p_{\mathrm{I2}\mathcal{E}}(\mathcal{E}[\![\mathsf{p}_{\mathrm{D2}}]\!])(\mathbf{Y}(p_{\mathrm{I2}\mathcal{E}}(\mathcal{E}[\![\mathsf{p}_{\mathrm{D2}}]\!])))\\
= \lambda \mathrm{inh}_1.\lambda \mathrm{inh}_2.\langle \mathrm{scale} = \mathrm{inh}_1, \mathrm{radix} = \mathrm{inh}_2, \mathrm{val} = \mathbf{Y}(p_{\mathrm{I2}\mathcal{E}}(\mathcal{E}[\![\mathsf{p}_{\mathrm{D2}}]\!]))(\mathrm{inh}_1,\mathrm{inh}_2).\mathrm{D}.\mathrm{val},\\
\quad \mathrm{D} = \mathcal{E}[\![\mathsf{p}_{\mathrm{D2}}]\!](\ \underline{\mathbf{Y}(p_{\mathrm{I2}\mathcal{E}}(\mathcal{E}[\![\mathsf{p}_{\mathrm{D2}}]\!]))(\mathrm{inh}_1,\mathrm{inh}_2).\mathrm{scale},\ \mathbf{Y}(p_{\mathrm{I2}\mathcal{E}}(\mathcal{E}[\![\mathsf{p}_{\mathrm{D2}}]\!]))(\mathrm{inh}_1,\mathrm{inh}_2).\mathrm{radix}}\ \rangle\\
= \lambda \mathrm{inh}_1.\lambda \mathrm{inh}_2.\langle \mathrm{scale} = \mathrm{inh}_1, \mathrm{radix} = \mathrm{inh}_2,\ \mathrm{val} = \underline{\mathbf{Y}(p_{\mathrm{I2}\mathcal{E}}(\mathcal{E}[\![\mathsf{p}_{\mathrm{D2}}]\!]))(\mathrm{inh}_1,\mathrm{inh}_2).\mathrm{D}}.\mathrm{val},\\
\quad \mathrm{D} = \mathcal{E}[\![\mathsf{p}_{\mathrm{D2}}]\!](\mathrm{inh}_1,\mathrm{inh}_2)\ \rangle\\
= \lambda \mathrm{inh}_1.\lambda \mathrm{inh}_2.\langle \mathrm{scale} = \mathrm{inh}_1, \mathrm{radix} = \mathrm{inh}_2,\ \mathrm{val} = \underline{\mathcal{E}[\![\mathsf{p}_{\mathrm{D2}}]\!](\mathrm{inh}_1,\mathrm{inh}_2).\mathrm{val}},\\
\quad \mathrm{D} = \underline{\mathcal{E}[\![\mathsf{p}_{\mathrm{D2}}]\!](\mathrm{inh}_1,\mathrm{inh}_2)}\ \rangle\\
= \lambda \mathrm{inh}_1.\lambda \mathrm{inh}_2.\langle \mathrm{scale} = \mathrm{inh}_1, \mathrm{radix} = \mathrm{inh}_2,\ \mathrm{val} = 2\times \mathrm{inh}_2 \uparrow \mathrm{inh}_1,\\
\quad \mathrm{D} = \langle\ \mathrm{scale} = \mathrm{inh}_1, \mathrm{radix} = \mathrm{inh}_2,\ \mathrm{val} = 2\times \mathrm{inh}_2 \uparrow \mathrm{inh}_1\ \rangle\rangle
\end{array}
$$

$\mathcal{E}[\![\mathsf{p}_{\mathrm{I1}}(\mathsf{p}_{\mathrm{I2}}(\mathsf{p}_{\mathrm{D2}}),\mathsf{p}_{\mathrm{D3}})]\!] = \cdots$ (omitted)

Underlines in the above calculation mean reducing places. Finally, we obtain the result of this example record calculation shown in Fig.4. By applying the result to '8' as the argument 'inh1', we get a record that represents the attributed tree shown in Fig.2.

expression	record calculus	description
X_0	self/self$(old_1, \ldots, old_n)$	syntactic reference (attributed tree rooted in nonterminal X_0)
X_i	child$_i$/child$_i(old_1, \ldots, old_n)$	syntactic reference (attributed tree rooted in nonterminal $X_i (1 \leq i \leq n)$)
$e.a$	$e'.a$	selection (an attribute value a in the root of attributed tree e)
$p(e_1, \ldots, e_n)$	$\mu \text{self}'.p_{\mathcal{E}}(e'_1, \ldots, e'_n, \text{self}')$	construction (an attributed tree constructed by a constructor p and attributed subtrees $e_1, \ldots, e_n$)
$e\{i_1 = new_1, \ldots, i_n = new_n\}$	$e'(new'_1, \ldots, new'_n)$	attribution expressions (an attributed tree e applied to its inherited attribute values)

semantic rule	record calculus	description
$X = e$	$X = e'(new'_1, \ldots, new'_n)$	a semantic rule that defines by e the attributed tree rooted in a nonterminal X

Table 1: Extended Syntax and Semantics of Attributed HAGs

4. Formalizing AG extensions

In this section, we formalize as records the following AG extensions: higher-order AGs, recursive AGs and object-oriented AGs. This demonstrates that our denotational semantics fits well with structure-oriented computational models based on AGs.

4.1. Higher-Order AGs

Higher Order AGs (HAGs)[VSK89][TC90] are a structure-oriented computational model based on AGs. In HAGs, (parts of) attributed trees can be defined by attribute values, and vice versa. As suggested in [TC90], HAGs should handle both of *attributed* trees and *unattributed* ones. But, to simplify this paper, we assume all trees are attributed. Table 1 summarizes attributed HAGs. In Table 1, e' is the same as e except that all attribute occurrences are replaced as defined in Def.3.2. In Table 1, attribute values $old_1, \ldots, old_n$ are inherited attributes in *old* context, while $new_1, \ldots, new_n$ are those in *new* context.

Note that syntactic reference X_0 has the two meanings: self and self$(old_1, \ldots, old_n)$, because

- we want e to be attributed in *old* context when accessing to the attributes of e, so the meaning of e should be 'self$(old_1, \ldots, old_n)$',
- e should be λ-abstracted to be attributed in *new* context when constructing or grafting trees, so the meaning of e should be 'self'.

Evaluation strategies and techniques like incremental attribute re-evaluation are often important in AG systems, but it is not so straightforward to handle them in record calculus. Of course, it is possible to integrate the two meanings into something like the pair (self, $(old_1, \ldots, old_n)$), which is not given here to keep this paper simple.

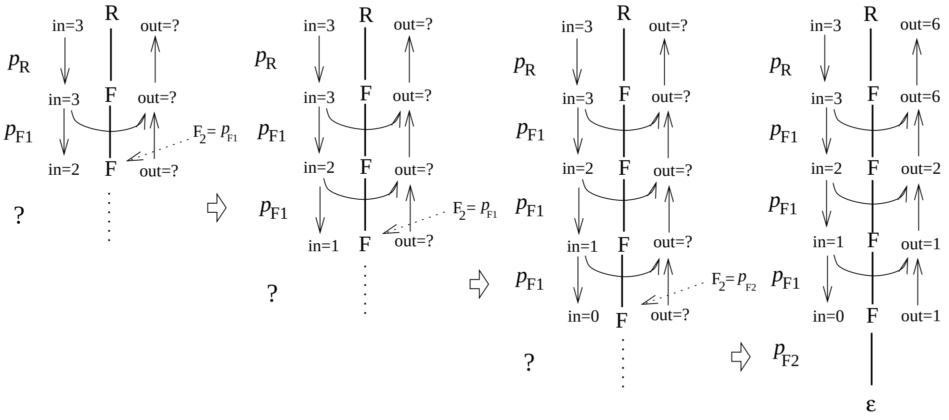

Figure 5: A Calculation of the factorial of 3

4.1.1. An Example HAG

In this section, we show an example HAG to compute factorial numbers, which is almost taken from [VSK89].

Example 4.1 (factorial numbers by HAGs)

$HAG_1 = (G, A, R)$ is defined as follows.

$$\begin{array}{rcl} P & = & \{p_{\mathrm{R}} : \mathrm{R} \to \mathrm{F}, p_{\mathrm{F1}} : \mathrm{F} \to \mathrm{F}, p_{\mathrm{F2}} : \mathrm{F} \to \varepsilon\} \\ R(p_{\mathrm{R}}) & = & \{\mathrm{F} \cdot \mathrm{in} = \mathrm{R} \cdot \mathrm{in}, \mathrm{R} \cdot \mathrm{out} = \mathrm{F} \cdot \mathrm{out} \ \ \} \\ R(p_{\mathrm{F1}}) & = & \{\mathrm{F}_2 = \mathrm{if}(\mathrm{F}_2 \cdot \mathrm{in} = 0, \mathsf{p}_{\mathrm{F2}}, \mathsf{p}_{\mathrm{F1}}), \\ & & \ \mathrm{F}_2 \cdot \mathrm{in} = \mathrm{F}_1 \cdot \mathrm{in} - 1, \\ & & \ \mathrm{F}_1 \cdot \mathrm{out} = \mathrm{F}_1 \cdot \mathrm{in} \times \mathrm{F}_2 \cdot \mathrm{out} \ \ \} \\ R(p_{\mathrm{F2}}) & = & \{\mathrm{F} \cdot \mathrm{out} = 1 \ \ \} \end{array}$$

□

The following semantic rule in $R(p_{\mathrm{F1}})$ is important here.

$$\mathrm{F}_2 = \mathrm{if}(\mathrm{F}_2 \cdot \mathrm{in} = 0, \mathsf{p}_{\mathrm{F2}}, \mathsf{p}_{\mathrm{F1}})$$

It defines the subtree that grows while $\mathrm{F}_2 \cdot \mathrm{in} \neq 0$. Fig.5 shows the process of attribute evaluation with R.in = 3. In Fig.5, the leftmost object shows "the attributed tree for $\mathsf{p}_{\mathrm{R}}(\mathsf{p}_{\mathrm{F2}})$ where all attributes except nonterminal attributes[3] are evaluated" and white right arrows show "the value of the bottom nonterminal attribute F is bound to p_{F1} or p_{F2}".

4.1.2. An Example of Calculating Record Semantics of a HAG

We can obtain the following three functions by applying Def.3.2 and Table 1 to the HAG_1 in Example 4.1.

$p_{R\mathcal{E}} = \lambda \mathrm{child}_1.\lambda \mathrm{self}.\lambda \mathrm{inh}_1.\langle \mathrm{in} = \mathrm{inh}_1, \mathrm{out} = \mathrm{self}(\mathrm{inh}_1).\mathrm{F.out}, \mathrm{F} = \mathrm{child}_1(\mathrm{self}(\mathrm{inh}_1).\mathrm{in})\rangle$
$p_{F1\mathcal{E}} = \lambda \mathrm{self}.\lambda \mathrm{inh}_1.\langle \mathrm{in} = \mathrm{inh}_1, \mathrm{out} = \mathrm{self}(\mathrm{inh}_1).\mathrm{in} \times \mathrm{self}(\mathrm{inh}_1).\mathrm{F}_2.\mathrm{out},$
$\quad \mathrm{F}_2 = (\mathrm{if}(\mathrm{self}(\mathrm{inh}_1).\mathrm{in} - 1 = 0, \mu \mathrm{self}_2.p_{\mathrm{F2}\mathcal{E}}(\mathrm{self}_2), \mu \mathrm{self}_1.p_{\mathrm{F1}\mathcal{E}}(\mathrm{self}_1)))(\mathrm{self}(\mathrm{inh}_1).\mathrm{in} - 1)\rangle$
$p_{F2\mathcal{E}} = \lambda \mathrm{self}.\lambda \mathrm{inh}_1.\langle \mathrm{in} = \mathrm{inh}_1, \mathrm{out} = 1\rangle$

[3] A nonterminal occurring in the left-hand side of semantic rules is called a *nonterminal attribute*[VSK89].

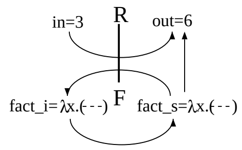

Figure 6: An Attributed Tree of RAG_1 with $\mathrm{R} \cdot \mathrm{in} = 3$

Now we can compute $\mathcal{E}[\![\mathsf{p}_R(\mathsf{p}_{F1})]\!](3)$. This calculation process corresponds to Fig.5. First, we calculate $\mathcal{E}[\![\mathsf{p}_{F1}]\!](3)$.

$$\begin{array}{l}
\mathcal{E}[\![\mathsf{p}_{\mathrm{F1}}]\!](3) = \langle \mathrm{in} = 3, \mathrm{out} = 3 \times (\mu \mathrm{self}_1.p_{\mathrm{F1}\mathcal{E}}(\mathrm{self}_1))(2).\mathrm{out}, \mathrm{F}_2 = (\mu \mathrm{self}_1.p_{\mathrm{F1}\mathcal{E}}(\mathrm{self}_1))(2)\rangle \\
\mu \mathrm{self}_1.p_{\mathrm{F1}\mathcal{E}}(\mathrm{self}_1))(2) = \langle \mathrm{in} = 2, \mathrm{out} = 2 \times (\mu \mathrm{self}_1.p_{\mathrm{F1}\mathcal{E}}(\mathrm{self}_1))(1).\mathrm{out}, \mathrm{F}_2 = (\mu \mathrm{self}_1.p_{\mathrm{F1}\mathcal{E}}(\mathrm{self}_1))(1)\rangle \\
\mu \mathrm{self}_1.p_{\mathrm{F1}\mathcal{E}}(\mathrm{self}_1))(1) = \langle \mathrm{in} = 1, \mathrm{out} = 1 \times (\mu \mathrm{self}_2.p_{\mathrm{F2}\mathcal{E}}(\mathrm{self}_2))(0).\mathrm{out}, \mathrm{F}_2 = (\mu \mathrm{self}_2.p_{\mathrm{F2}\mathcal{E}}(\mathrm{self}_2))(0)\rangle \\
\mu \mathrm{self}_2.p_{\mathrm{F2}\mathcal{E}}(\mathrm{self}_2))(0) = \langle \mathrm{in} = 0, \mathrm{out} = 1\rangle
\end{array}$$

$$\begin{array}{l}
\mathcal{E}[\![\mathsf{p}_{\mathrm{R}}(\mathsf{p}_{\mathrm{F1}})]\!](3) \\
= \langle\ \mathrm{in} = 3, \mathrm{out} = 6, \\
\quad \mathrm{F} = \langle\ \mathrm{in} = 3, \mathrm{out} = 6, \\
\qquad \mathrm{F}_2 = \langle\ \mathrm{in} = 2, \mathrm{out} = 2, \\
\qquad\quad \mathrm{F}_2 = \langle\ \mathrm{in} = 1, \mathrm{out} = 1, \\
\qquad\qquad \mathrm{F}_2 = \langle\ \mathrm{in} = 0, \mathrm{out} = 1\rangle\rangle\rangle\rangle\rangle
\end{array}$$

The result record matches the rightmost attributed tree in Fig.5.

4.2. Recursive AGs

In [Far86], Farrow pointed out that even a circularly (i.e. recursively) defined AG can be evaluated if its recursive definition has the least fixed-point. Such AGs are called recursive AGs (RAGs). Our formal semantics can deal with RAGs as it is. To show this, this section gives an example RAG and translates the RAG into records.

Example 4.2 (RAG_1: factorial numbers by RAGs)
$RAG_1 = (G, A, R)$ is defined as follows.

$$\begin{array}{rcl}
P & = & \{p_{\mathrm{R}} : \mathrm{R} \rightarrow \mathrm{F},\ p_{\mathrm{F}} : \mathrm{F} \rightarrow \epsilon,\ \ \} \\
R(p_{\mathrm{R}}) & = & \{\mathrm{R} \cdot \mathrm{out} = \mathrm{apply}(\mathrm{F} \cdot \mathrm{fact_s}, \mathrm{R} \cdot \mathrm{in}), \mathrm{F} \cdot \mathrm{fact_i} = \mathrm{F} \cdot \mathrm{fact_s}\ \ \} \\
R(p_{\mathrm{F}}) & = & \{\mathrm{F} \cdot \mathrm{fact_s} = \lambda x.\mathrm{if}(x = 0, 1, x \times \mathrm{F} \cdot \mathrm{fact_i}(x-1))\ \ \}
\end{array}$$

□

RAG_1 defines factorial numbers as follows.

$$\begin{array}{l}
\mathrm{R} \cdot \mathrm{out} = FACT(\mathrm{R} \cdot \mathrm{in}) \\
\mathrm{F} \cdot \mathrm{fact_i} = \mathrm{F} \cdot \mathrm{fact_s} = FACT = \lambda x.\mathrm{if}(x = 0, 1, x \times FACT(x-1))
\end{array}$$

Fig.6 shows an attributed tree of RAG_1 with $\mathrm{R} \cdot \mathrm{in} = 3$. As shown Fig.6, fact_i and fact_s are circularly defined. In the following, we represent RAG_1 as record calculus. By Def.3.2, we can translate RAG_1 into the two functions.

$p_{\mathrm{R}\mathcal{E}} = \lambda \mathrm{child}_1.\lambda \mathrm{self}.\lambda \mathrm{inh}_1.\langle$
$\quad \mathrm{in}{=}\mathrm{inh}_1,\ \mathrm{out}{=}\mathrm{apply}(\mathrm{self}(\mathrm{inh}_1).\mathrm{F.fact\_s},\ \mathrm{self}(\mathrm{inh}_1).\mathrm{in}),$
$\quad \mathrm{F}{=}\mathrm{child}_1(\mathrm{self}(\mathrm{inh}_1).\mathrm{F.fact\_s})\rangle$
$p_{\mathrm{F}\mathcal{E}} = \lambda \mathrm{self}.\lambda \mathrm{inh}_1.\langle$
$\quad \mathrm{fact\_i}{=}\mathrm{inh}_1,$
$\quad \mathrm{fact\_s}{=}\lambda x.\mathrm{if}(x = 0{,}1{,}x{\times}(\mathrm{self}(\mathrm{inh}_1).\mathrm{fact\_i})(x-1))\rangle$

The following is the record semantics of RAG_1 (calculating steps are omitted).

$\mathcal{E}[\![\mathsf{p}_{\mathrm{R}}(\mathsf{p}_{\mathrm{F}})]\!]$
$= \lambda \mathrm{inh}_1.\langle \mathrm{in} = \mathrm{inh}_1, \mathrm{out} = FACT(\mathrm{inh}_1),$
$\quad \mathrm{F} = \langle \mathrm{fact\_i} = FACT, \mathrm{fact\_s} = FACT\rangle\rangle$

where

$FACT$
$\equiv \mathbf{Y}(p_{\mathrm{R}\mathcal{E}}(\mathcal{E}[\![\mathsf{p}_{\mathrm{F}}]\!]))(\mathrm{inh1}).\mathrm{F.fact\_}s$
$= \lambda x.\mathrm{if}(x = 0, 1, x \times FACT(x-1))$

4.3. OOAG

We have introduced a computational model OOAG (Object-Oriented AGs)[SK90][GISK93], which is an extension of standard AGs by importing message passing and assignment to instance variables (i.e. multiple subtree replacement). This extension makes it easier to describe dynamic aspects of systems such as:

- *retrieval*: getting attribute values or attributed (sub)trees by sending retrieval messages to attributed trees
- *updating*: changing tree structures depending on their attribute values or by sending updating messages to attributed trees

In OOAG, an attributed tree is regarded as an aggregated object in object-oriented programming, and attributed subtrees and attributes are regarded as instance variables. Message passing may cause subtree replacements. As a result of the subtree replacements, attribute values in the grafting point generally become inconsistent with each other, that is, attribute values do not satisfy their semantic rules. To recover the situation, the evaluation process of attributes will be executed for the entire attributed tree in consistent with the semantic rules. Thus, in OOAG, message passing and attribute evaluation are alternately repeated.

OOAG is similar in nonterminal occurrences to HAGs, but OOAG is definitely different from HAGs. OOAG can update an attributed subtree with another one, while HAGs can not. To show this, here is given an example OOAG where a subtree swapping method is implemented. The method just returns a new object where subtrees are swapped. So, strictly speaking, this implementation given here does not implement mutable objects.

Example 4.3 (swapping subtrees by OOAG)

$$\begin{array}{rcl} P & = & \{p_{\mathrm{X}} : \mathrm{X} \to \mathrm{Y\ Y},\ p_{\mathrm{Y1}} : \mathrm{Y} \to \varepsilon,\ p_{\mathrm{Y2}} : \mathrm{Y} \to \varepsilon\} \\ R(p_{\mathrm{X}}) & = & \{\mathrm{X} \cdot \mathrm{swap} = f_{p_{\mathrm{X}}}(\mathrm{Y}_2, \mathrm{Y}_1)\} \\ R(p_{\mathrm{Y1}}) & = & \{\mathrm{Y} \cdot \mathrm{id} = 1\} \\ R(p_{\mathrm{Y2}}) & = & \{\mathrm{Y} \cdot \mathrm{id} = 2\} \end{array}$$

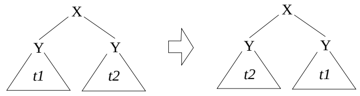

Figure 7: Swapping Subtrees

□

By Def.3.2, we obtain the following functions from Example 4.3.

$p_{X\mathcal{E}}$=μmyclass.λchild$_1$.λchild$_2$.λself.$\langle$
swap = μself$_2$.myclass(child$_2$, child$_1$, self$_2$),
Y$_1$=child$_1$, Y$_2$=child$_2\rangle$
$p_{Y1\mathcal{E}}$ = λself.$\langle$ id=1 $\rangle$
$p_{Y2\mathcal{E}}$ = λself.$\langle$ id=2 $\rangle$

First, we have the following attributed tree t as a result of evaluating $\mathcal{E}[\![\mathsf{p}_X(\mathsf{p}_{Y1}, \mathsf{p}_{Y2})]\!]$.

$$\begin{aligned}
\mathrm{t} &= \mathcal{E}[\![\mathsf{p}_{\mathrm{X}}(\mathsf{p}_{\mathrm{Y1}}, \mathsf{p}_{\mathrm{Y2}})]\!] \\
&= \langle\ \mathrm{swap} = \mu\mathrm{self}_2.p_{\mathrm{X}\mathcal{E}}(\langle \mathrm{id} = 2\rangle, \langle \mathrm{id} = 1\rangle, \mathrm{self}_2), \\
&\quad \mathrm{Y}_1 = \langle \mathrm{id} = 1\rangle, \mathrm{Y}_2 = \langle \mathrm{id} = 2\rangle\rangle
\end{aligned}$$

Next, we can obtain the result of t.swap as follows.

$$\begin{aligned}
\mathrm{t.swap} &= \mathcal{E}[\![\mathsf{p}_{\mathrm{X}}(\mathsf{p}_{\mathrm{Y1}}, \mathsf{p}_{\mathrm{Y2}})]\!].\mathrm{swap} \\
&= \langle\ \mathrm{swap} = \mu\mathrm{self}_2.p_{\mathrm{X}\mathcal{E}}(\langle \mathrm{id} = 1\rangle, \langle \mathrm{id} = 2\rangle, \mathrm{self}_2), \\
&\quad \mathrm{Y}_1 = \langle \mathrm{id} = 2\rangle, \mathrm{Y}_2 = \langle \mathrm{id} = 1\rangle\rangle
\end{aligned}$$

This record represents an attributed tree where Y_1 and Y_2 are swapped.

5. A Simple Implementation using SML/NJ

Our new formalization introduced in Section 3 and 4 can be straightforwardly implemented in functional programming languages with record types, especially lazy ones. To show this, we provide a simple implementation using Standard ML of New Jersey (SML/NJ for short)[TL96]. Using this implementation method, readers can easily experiment new ideas of AG extentions as running codes.

Fig. 8 shows a SML code implementing the record semantics of the HAG described in Section 4.1.2[4]. If you load and execute the SML code, you can see the following result, which shows $\mathcal{E}[\![\mathsf{p}_R(\mathsf{p}_{F1})]\!](5) = 120$.

```
% sml
- use "hag.sml";
 (omitted)
val tree = p_R1_ p_F1_ : R_
val atree = p_R1 {F=fn, in1=fn, out=fn} : R
val out = 120 : int
val it = () : unit
-
```

[4] We implemented all examples in this paper in the same way, but we do not give them for lack of space.

Note that a closure technique of delaying evaluation with functions [Rea89] is used in the SML code in Fig. 8, since SML/NJ uses call-by-value evaluation rather than lazy evaluation. Without this technique, the computation may not terminate on call-by-value evaluators including SML/NJ. The technique is to write:

- "`fn () =>` *expr*", to delay the evaluation of *expr*.

  This represents an anonymous function whose argument `()` is a dummy, unused empty tuple. The type of `()` is `unit`. A function body *expr* is not evaluated until the function is applied.

- "*delayed_expr* `()`", where *delayed_expr* is an expression that evaluated the form "`fn () =>` *expr*", to force the evaluation of the function body *expr*.

  This represents the application of *delayed_expr* to an empty tuple. The body *expr* of *delayed_expr* is evaluated now.

From our simple implementation, we found SML/NJ is powerful enough to describe our AG formalization straightforwardly except the following points.

- We have to write many `case` pattern matchings even for records that have the same fields, since SML/NJ has no support for subtyping or inclusion polymorphism.
- We have to define the semantic function $\mathcal{E}$ as several functions (e.g., `eval_R` and `eval_F` in Fig. 8) depending on a given AG. This is because in SML/NJ there is no way to simply describe a function to operate on different user-defined types that can not be parameterized.

6. Conclusion

In this paper, we first presented a new denotational semantics of attribute grammars (AGs) based on Cardelli's record calculus. This semantics is structure-oriented as well as natural and simple. Unlike previous works, an attributed tree is represented as a nested record to preserve the structural information.

Our AG semantics is simple and natural because:

- Our record representation for AGs preserves structures of attributed trees as well as values of attributes. There are no extra fields in records; all fields represent only attributes or attributed subtrees (Def.3.1).
- A function $p_{\mathcal{E}}$ is easily obtained by rewriting p and $R(p)$ (Def.3.2).
- The definition of $\mathcal{E}$ is defined as simple recursion on tree structures as follows (Def.3.6).

$$\mathcal{E}[\![\mathsf{p}(t_1, \ldots, t_n)]\!] = \mu \mathrm{self}.p_{\mathcal{E}}(\mathcal{E}[\![t_1]\!], \ldots, \mathcal{E}[\![t_n]\!], \mathrm{self})$$

- Underlying record calculus is simple.

We think the semantics is a good theoretical groundwork for modeling AG extensions (especially structure-oriented ones). To show this, we represented HAGs, RAGs and OOAG as record calculus in Section 4. We also showed the semantics can be implemented straightforwardly and simply in a functional language SML/NJ.

```
infix 8 >>;
fun record >> label = label record (); (* >>: field selector *)
fun fix f inhs = f (fix f) inhs;       (* fix: fixed-point operator *)

(* type definitions of abstract syntax trees (production rules) *)
datatype R_ = p_R1_ of F_
and      F_ = p_F1_ | p_F2_
  ;
(* type definitions of attributed trees *)
datatype R = p_R1 of {F:unit->F, in1:unit->int, out:unit->int}
and      F = p_F1 of {F2:unit->F, in1:unit->int, out:unit->int}
           | p_F2 of {in1:unit->int, out:unit->int}
  ;
(* semantic rules *)
(* p_RE *)
val rec p_R_e = fn child1 => fn self => fn inh1 =>
  p_R1 {
          in1 = inh1, (* 'in1' is used instead of 'in' since 'in' is a keyword in SML/NJ *)
          out = fn () => (case (self inh1) of
                             p_R1 r => (case r >> #F of p_F1 r2 => r2 >> #out)),
                             (* self(inh1).F.out *)
          F   = fn () => child1 (fn () => case (self inh1) of p_R1 r => r >> #in1)
                             (* child1(self(inh1).in) *)
       }
(* p_F1E *)
and p_F1_e = fn self => fn inh1 =>
  p_F1 {
          in1 = inh1,
          out = fn () => (case (self inh1) of p_F1 r => r >> #in1) *
              (case (self inh1) of p_F1 r =>
                 (case r >> #F2 of p_F1 r2 => r2 >> #out | p_F2 r2 => r2 >> #out)),
                (* self(inh1).in×self(inh1).F2.out *)
          F2  = fn () => (if (case (self inh1) of p_F1 r => r >> #in1)-1=0
                          then fix p_F2_e else fix p_F1_e)
                         (fn () => (case (self inh1) of p_F1 r => r >> #in1)-1)
                (* (if(self(inh1).in − 1 = 0, μself2.p_F2E(self2), μself1.p_F1E(self1))) (self(inh1).in − 1) *)
       }
(* p_F2E *)
and p_F2_e = fn self => fn inh1 =>
  p_F2 {
          in1 = inh1,
          out = fn () => 1
       }
  ;
(* functions that implements the semantic function E *)
fun eval_R t = case t of p_R1_ t1 => fix (p_R_e (eval_F t1))
and eval_F t = case t of p_F1_ => fix p_F1_e | p_F2_ => fix p_F2_e
  ;
(* for sample execution *)
val tree  = p_R1_ p_F1_;
val atree = eval_R tree (fn () => 5);
val out   = case atree of p_R1 r => r >> #out;
```

Figure 8: A Simple Implementation in SML/NJ of the HAG Example 4.1

7. Future Works

The paper emphasizes that Cardelli's record calculus makes it easy to formalize AGs, HAGs, RAGs and OOAG with a structure-oriented view. In [CHC90], however, Cardelli's record calculus is used to formalize inheritance. We will provide some formalization of AG inheritance extending our AG semantics, and compare it with previous works on AG inheritance, e.g., [Hed89][Paa95][MLAv99].

Another issue is the generality of our AG semantics. We plan to apply our AG semantics to many other AG extensions, e.g., modularity concepts[KW94], remote access[Gra96][Hed99], and so on.

Acknowledgment

The authors would like to thank our OOAG project members, especially Takashi Hagiwara, for their efforts on developing MAGE2 system and their useful discussions. We would also like to thank Kikuchi, Yutaka for reading the draft and making a number of helpful suggestions.

Bibliography

[Car88] L. Cardelli. A semantics of multiple inheritance. In *Information and Computation*, pages 138–164, 1988.

[CCHO89] Peter Canning, William Cook, Walter Hill, and Walter Olthoff. F-bounded polymorphism for object-oriented programming. In *FPCA '89 Conf. Proc.*, pages 273–281, 1989.

[CHC90] William R. Cook, Walter L. Hill, and Peter S. Canning. Inheritance is not subtyping. In *Proc. 17th ACM Sympo. on Principles of Programming Languages*, pages 125–135. ACM, 1990.

[CP89] W. Cook and J. Palsberg. Denotational semantics of inheritance and its correctness. In *Proc. 4th ACM Conf. on Object-Oriented Programming Systems, Languages, and Applications*, pages 433–443, 1989.

[CW85] L. Cardelli and P. Wegner. On understanding types, data abstraction, and polymorphism. In *ACM Computing Surveys, Vol.17, No.4*, pages 471–522. ACM, 1985.

[DJL88] Pierre Deransart, Martin Jourdan, and Bernard Lorho. *Attribute Grammars: Definitions, Systems, and Bibliography*, volume 323 of *Lec. Notes in Comp. Sci.* Springer-Verlag, 1988.

[Far84] R. Farrow. Generating a Production Compiler from an Attribute Grammar. *IEEE Software*, 1(4):77–93, 1984.

[Far86] Rodney Farrow. Automatic generation of fixed-point-finding evaluators for circular, but well-defined, attribute grammars. In *Proc. the ACM SIGPLAN '86 Sympo. on Compiler Construction*, pages 85–98, Palo Alto, Calif., 1986. ACM.

[GISK93] Katsuhiko Gondow, Takashi Imaizumi, Yoichi Shinoda, and Takuya Katayama. Change management and consistency maintenance in software development environments using object oriented attribute grammars. In *Object Technologies for Advanced Software (Proc. 1st JSSST Int. Sympo.)*, volume 742 of *Lec. Notes in Comp. Sci.*, pages 77–94. Springer Verlag, 1993.

[Gra96] GrammaTech, Inc., Ithaca, NY. *The Synthesizer Generator Reference Manual*, fifth edition, 1996.

[Hed89] Görel Hedin. An object-oriented notation for attribute grammars. In S. Cook, editor, *Proc. ECOOP '89*, pages 329–345, Nottingham, July 1989. Cambridge University Press.

[Hed99] Görel Hedin. Reference Attributed Grammars. In *Proc. Second Workshop on Attribute Grammars and their Applications (WAGA99)*, pages 153–172, 1999.

[Joh87] Thomas Johnsson. *Attribute Grammars as a Functional Programming Paradigm*, volume 274 of *Lec. Notes in Comp. Sci.*, pages 154–173. Springer Verlag, 1987.

[KHZ82] U. Kastens, B Hutt, and E Zimmermann. *GAG: A Practical Compiler Generator*, volume 141 of *Lec. Notes in Comp. Sci.* Springer Verlag, 1982.

[Knu68] D.E. Knuth. Semantics of context-free languages. *Mathematical Systems Theory*, 2(2):127–145, 1968.

[Knu71] D.E. Knuth. Semantics of context-free languages: Correction. *Mathematical Systems Theory*, 5(1):95–96, 1971.

[KW94] Uwe Kastens and William M. Waite. Modularity and Reusability in Attribute Grammars. *Acta Informatica*, 31:601–627, 1994.

[MLAv99] Marjan Mernik, Mitja Lenič, Enis Avdičaušević, and Viljem Žumer. Multiple Attribute Grammar Inheritance. In *Proc. Second Workshop on Attribute Grammars and their Applications (WAGA99)*, pages 57–76, 1999.

[Paa95] Jukka Paakki. Attribute grammar paradigms—a high-level methodology in language implementation. In *ACM Computing Surveys, Vol.27, No.2*, pages 196–255. ACM, 1995.

[Rea89] Chris Reade. *Elements of Functional Programming.* Addison Wesley, 1989.

[RT89] Thomas W. Reps and Tim Teitelbaum. *The Synthesizer Generator.* Texts and Monographs in Computer Science. Springer-Verlag, 1989.

[SK90] Yoichi Shinoda and Takuya Katayama. Object Oriented Extension of Attribute Grammars and Its Implementation Using Distributed Attribute Evaluation Algorithm. In *Proc. Int. Workshop on Attribute Grammars and their Applications,* Lec. Note in Comp. Sci. *Vol. 461*, pages 177–191. Springer-Verlag, 1990.

[TC90] Tim Teitelbaum and R. Chapman. Higher-order attribute grammars and editing environments. In *Proc. ACM SIGPLAN '90 Conf. on Programming Language Design and Implementation, Vol. 25, No. 6*, pages 197–208, White Plains, NY, 1990.

[TK82] Masayuki Takeda and Takuya Katayama. On defining denotational semantics for attribute grammars. *Journal of Information Processing*, 5(1):21–29, 1982.

[TL96] Lucent Technologies and Bell Laboratories. SML/NJ: Standard ML of New Jersey. http://cm.bell-labs.com/cm/cs/what/smlnj/index.html., 1996.

[VSK89] H.H̃. Vogt, S.D̃. Swierstra, and M.F̃. Kuiper. Higher order attribute grammars. In *Proc. ACM SIGPLAN'89 Conf. on Programming Language Design and Implementation, Vol. 25, No. 6*, pages 131–145, 1989.