

属性文法の系統的デバッグ法

佐々木 晃[†] 池添洋平[†] 佐々 政孝[†]

属性文法プログラムのデバッグ作業は、構文の再帰性や属性依存関係等による独特の困難さを伴う。本論文では、ユーザがこれらの複雑な因果関係を意識せずに、属性文法のデバッグを行うことができるデバッグ方法を提案する。従来、属性文法のデバッグ手法として、アルゴリズムック・デバッグングを用いたもの、およびプログラム・スライシングを利用した手法が提案されている。これらはともに、対話的な手法であり、デバッガがプログラムの動作の正しさに関する質問をユーザに問い合わせることで、バグの範囲を減らしていく方法である。しかし、これらの方法には、質問に答えることが困難な問い合わせが行われる場合がある等の問題点があった。また、それぞれの手法を統合して扱えず、独立して用いる必要がある点も問題であった。本論文で提案する手法は、アルゴリズムック・デバッグングを属性文法に応用したデバッグ法であるが、従来の属性文法のアルゴリズムック・デバッグングを包含する一般的なものである。この一般化による大きな利点は、従来のアルゴリズムック・デバッグング、およびプログラム・スライシングを用いたデバッグ手法の両者を同じ枠組みの中で扱えることである。これにより、従来の2種類の手法を融合した新たなデバッグ法を導くことができた。本研究ではまた、この融合したデバッグ法を、属性文法デバッガ Aki を拡張することで実装し、評価実験を行った。これを通して、提案するデバッグ法の有用性を確かめた。

Systematic Debugging Method for Attribute Grammars

AKIRA SASAKI,[†] YOHEI IKEZOE[†] and MASATAKA SASSA[†]

Although attribute grammars (AGs) are easy to understand and write, debugging AGs is not simple. The major part of the problem arises from AG's specific difficulty, such as recursive grammar structure and attribute dependency. Our goal is to let programmers easily debug AGs but freeing them from such complicated dependency.

Formerly, two debugging methods of AGs have been proposed, one of which based on algorithmic debugging, and the other based on program slicing. Although these methods achieve a semi-automatic way of debugging AG descriptions, several problems remain in either methods. Furthermore, they cannot work simultaneously because of the distinction of their underlying theory.

In this paper, a new systematic debugging method of AGs is proposed. Our approach is, in principle, based on the former algorithmic debugging for AGs but can treat the two previous methods by more general form. In fact, the two previous methods can be explained by our new method in a unified way. This leads to a new debugging method that enables the integration of the two methods.

This integrated method have been implemented in Aki, a debugger for AG description. We evaluate our new approach by the experience of using Aki, which shows the usability of our debugging method.

1. はじめに

属性文法プログラムのデバッグ作業は、構文の再帰性や属性依存関係等による独特の困難さを伴う。本論文では、ユーザがこれらの複雑な因果関係を意識せずに、属性文法のデバッグを行うことができるデバッグ方法を提案する。

本方式は、アルゴリズムック・デバッグング¹³⁾を属

性文法に応用したデバッグ法である。アルゴリズムック・デバッグングは、もともと論理型言語のデバッグ法として提案された。この方法は、プログラム実行時の、述語の計算結果を局所的にプログラマに提示し、この結果がプログラマの意図に合致するものかどうかを確認する、という作業を繰り返し行うことにより、バグを含んだ述語を特定するものである。

佐々らによる属性文法のデバッグ手法¹¹⁾は、同じくアルゴリズムック・デバッグングを属性文法に応用したデバッグ法である。属性文法のプログラムには、述語にあたるものは存在しないが、その計算過程には、

[†] 東京工業大学 情報理工学研究所

Department of Mathematical and Computing Sciences,
Tokyo Institute of Technology

述語の計算と同じ振る舞いをする Synth 関数⁸⁾が存在する。文献 11) では、この性質を用いてアルゴリズム・デバッグを属性文法に適用した。

また、著者らのグループはこの手法と相補的に用いる目的で、プログラム・スライシングを用いたデバッグ法を属性文法に応用した⁶⁾。この手法も、デバッグが質問を提示しその結果によってバグの範囲を狭める方法をとる。

文献 11) の手法では、デバッグが行う問い合わせは、ある部分木に対する属性間の値の関係の正しさを問うもので、大きな木に感わされず質問に集中できるという利点がある。一方、スライスを用いた手法は、属性値そのものの正しさを問い合わせるため、ユーザが属性値の正しさを直観的に答えられる場合などは、属性間の値の関係を答えるより、簡単に質問に答えることができる。

しかし、これらを単独で用いてデバッグを行うには限界があった。たとえば、バグが存在する場所によっては、ユーザは問い合わせに答えるために、大きな部分木の内容を確認しなければならない、という問題点が挙げられる。また、ユーザはデバッグ中に、値などの異変に気づいたとしても、デバッグの問い合わせに答える以外の形では、デバッグのための情報を伝えられないという問題もあった。最大の問題点は、両者の手法は融合されておらず、片方の手法でデバッグを始めた後は、もう一方の手法に移ることは、必ずしもできないという点である。

本論文で提案する属性文法のデバッグ法は、これらの問題点を解決する手法である。基礎となるのはアルゴリズム・デバッグであるが、計算の振る舞いを Synth 関数だけに限らないように拡張した。したがって、文献 11) の手法を一般化したものと言えるが、スライスを用いた方法も、この一般化によって統一的に説明できる。本デバッグ法は、この 2 手法を含め、様々な問い合わせのやり方を同じ枠組みで扱えるという大きな利点がある。これにより、単独の手法でのデバッグ作業における問題点が解決され、効率よいデバッグ作業が可能となる。

本研究ではさらに、この一般化したデバッグ手法の一適用として、従来の 2 手法を融合した新たなデバッグ法を実装し、予備的評価実験を行った。これを通して、本デバッグ法の有用性を示す。

以下、本論文は次のように構成されている。2 節では、準備として、属性文法および従来提案された属性文法のデバッグ法について述べる。3 節、4 節において、従来手法の問題点と、一般化したアルゴリズム

ク・デバッグを説明する。5 節では、属性文法のデバッグ Aki について述べ、6 節で、本方式についての評価と議論を行う。7 節では関連する研究について述べる。

2. 準備

2.1 属性文法

属性文法は、言語の文法と意味とを統一して扱うことができる記述法である。プログラミング言語の意味記述や、言語処理系での解析や変換の記述として広く用いられている。

属性文法の簡単な例として、文法 G_1 を挙げる(図 1)。これは、2 進表記の小数の値を求める記述である。なお、後の説明のために、図中 (bug) の部分にバグを意図的に埋め込んである。

$$\begin{aligned}
 F & ::= \cdot L \\
 & \quad \{ L.\text{pos} = 1; \quad (a) \\
 & \quad \quad F.\text{val} = L.\text{val} \} \\
 L_0 & ::= B L_1 \\
 & \quad \{ L_1.\text{pos} = L_0.\text{pos} + 1; \\
 & \quad \quad B.\text{pos} = L_0.\text{pos} + 1; \quad (\text{bug}) \\
 & \quad \quad L_0.\text{val} = B.\text{val} + L_1.\text{val} \} \\
 & \quad | B \\
 & \quad \{ B.\text{pos} = L_0.\text{pos}; \\
 & \quad \quad L_0.\text{val} = B.\text{val} \} \\
 B & ::= 1 \\
 & \quad \{ B.\text{val} = 2^{-B.\text{pos}} \} \\
 & \quad | 0 \\
 & \quad \{ B.\text{val} = 0 \}
 \end{aligned}$$

図 1 属性文法 G_1

Fig.1 Attribute Grammar G_1

属性文法は、文脈自由文法によって言語の文法を定義し、文脈自由文法の生成規則に付随する意味規則によって、意味を定義する。この意味規則の記述は、属性と属性評価規則を用いて行う。属性とは、文脈自由文法の各文法記号に付随する、意味を表す変数のことである。属性評価規則は、各属性の値を計算するための規則である。

属性文法にしたがって意味を計算することを属性評価という。これは、与えられた入力から得られる構文解析木の上で、ノードの属性値を計算することにあたる。一般に、属性の値は他の属性の値に依存しているので、適切な順序で計算しなくてはならない。 G_1 の生成規則 " $L_0 ::= B L_1$ " の属性 $L_0.\text{val}$ の計算には、

B.val, $L_1.val$ の値がともに必要である。このように、属性文法の計算においては、意味規則に現れる属性の依存関係にしたがって属性値の計算を行う。

例 1 G_1 に対して “.101” の評価は次のように行われる。

- (1) 構文解析によって “.101” の解析木を作る。
- (2) 属性の依存関係にしたがって、各属性の値を計算して求める。

このようにして、属性付き解析木 (図 2) が作られる。この場合、 $F.val$ が求めたい結果である。(正しい値は $5/8$ であるが、バグが含まれているので間違った値となっている。)

なお、この属性付き解析木のノードのラベルは、“L” のような非終端記号であるが、それぞれのノードを区別するために、“ L_1, L_2 ” のようにラベルを付けている。

実際に解析木のノードに付随する属性を、属性文法記述の属性と区別して、属性インスタンスと呼ぶ。例えば、属性インスタンス $L_1.pos$ は、ノード L_1 に付随する属性 pos のことである。これは、文法規則に出てくる $L_1.pos$ とは、区別する必要があることに注意されたい。

ここで、 pos のように親および兄弟のノードから値が定まる属性を継承属性、 val のように子供のノードから値が定まる属性を合成属性という。

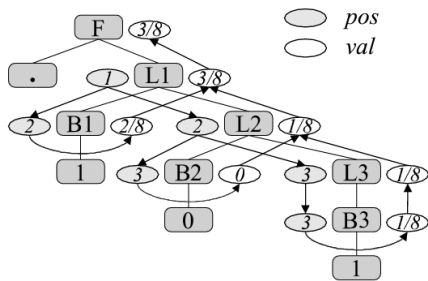


図 2 属性付き解析木
Fig.2 Attributed parse tree

2.2 属性文法のアルゴリズムック・デバッグ

アルゴリズムック・デバッグは、論理型言語の系統的デバッグ手法として提案され、その後関数型言語にも適用された。アルゴリズムック・デバッグは計算木によって定式化される。計算木とは、論理型では証明木、関数型では実行時の関数呼び出し木にあたるものである。デバッグは、バグの範囲を狭めるために、ユーザに問い合わせを行う。問い合わせとは、論理式の述語の結果、あるいは、関数の挙動がユーザ

の意図と合っているかどうかを尋ねるものである。デバッグは、問い合わせを繰り返し行うことで、バグの範囲を狭めていき、最終的にバグが発生した述語、あるいは関数を特定する。

さて、アルゴリズムック・デバッグを属性文法に適用するためには、属性評価の再帰的な構造を抽出して、計算木を構築しなければならない。そこで佐々ら¹¹⁾は、Synth 関数⁸⁾と呼ばれる関数構造をアルゴリズムック・デバッグに利用した。Synth 関数とは、属性の計算を表現するために用いられる仮想的な関数で、継承属性と構文解析木の部分木を引数として、合成属性の値を計算する関数である。この Synth 関数を再帰的に呼び出すことで、すべての属性の計算を表現できるため、この呼び出し関係を用いて、計算木を構築することができる。

例 1 の属性評価によって作られる計算木は図 3 のようになる。以下、この計算木上でのアルゴリズムック・デバッグの例を示す。

ここでの計算木のノードは関数名、引数、計算結果の三つ組で表現されている。アルゴリズムック・デバッグを開始すると、デバッグは計算木のノードの一つを選び、そのノードに対応する計算が正しいかどうかをユーザに対して問い合わせる。ここでは、ノード (a) が選ばれたものとする。(a) は、 $L_2.pos$ の値 2 と構文解析木のノード L_2 以下の部分木に対して、 $L_2.val$ の計算結果が $1/8$ であることを示しており、これが正しいかどうかをユーザは答える。この場合 L_2 以下の部分木は入力 “.101” の部分にあたり、この部分の小数点以下の桁数を表している $L_2.pos$ の値は 2 であるから、計算結果は $1/8$ で正しいといえる。ユーザが正しいと答えた場合、これより下の Synth 関数の呼び出し以外の部分に誤りが含まれるもののみなし、(a) 以下の部分木を計算木から除外し、問い合わせを繰り返す。

次にデバッグは残った計算木のノードから、(b) を選んだものとする。(b) は “.101” に対応する部分木と小数点以下の桁数 $L_1.pos = 1$ から計算結果が $5/8$ であることが期待されるが実際の結果は $3/8$ であり、この計算結果は正しくないことがわかる。ユーザが正しくないと答えた場合、計算木のこれ以下の部分に誤りがあるとして、(b) より下以外の部分を計算木から除外する。さらに、(c) についても同様の質問を行うとした場合、この計算結果は正しいことがわかる。この結果、残ったノードは (b) のみとなり、Synth 関数 (b) に対応する属性評価規則 (この場合は “ $L_0 ::= B L_1$ ” に含まれる 3 つの属性評価規則) に間違いがあることを指摘することができる。

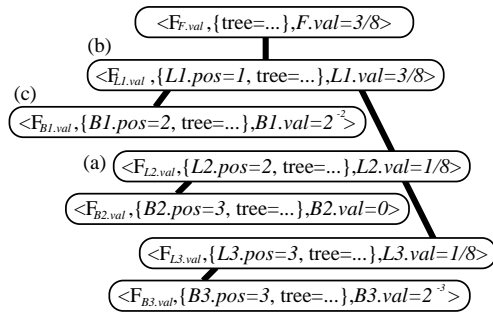


図3 計算木
Fig. 3 Computation tree

2.3 スライスを用いた属性文法のデバッグ法

プログラム・スライシング¹⁵⁾⁷⁾は、プログラムの中から、ある計算に影響をおよぼす全ての計算（スライス）を抜き出す技術である。この技術は、ソフトウェア工学など様々な分野への応用があり、デバッグの手段としても有効である。著者らのグループは、下村によって提案されたスライスの分割を用いたデバッグ法¹⁴⁾を、属性文法に適用した⁶⁾。

属性評価では、「属性評価規則を適用しその結果を属性インスタンスへ代入する計算」を計算の単位と考えることができる。このような計算を属性計算と呼ぶ。したがって、属性文法におけるスライスは、「ある属性計算に影響を与える属性計算の列」である。

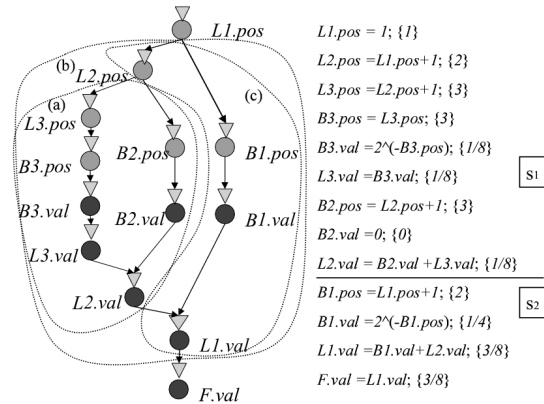


図4 属性計算と属性評価

Fig. 4 Attribute computation and attribute evaluation

例1における属性評価の様子を図4に示す。図中、左側のグラフでは、円が属性インスタンス、下向き三角形が属性計算、辺が値の流れをそれぞれ表す。また、右側の各文が属性計算であり、属性計算の列によって

属性評価の過程を表している*。各文の後ろの括弧内は、その属性計算によって求められた属性値である。（グラフ中の記号、点線等は、後の説明で使用する）。

スライスを用いた属性文法のデバッグ法の原理は、次の通りである。F.valの値が誤っているとす。まず、デバッガは、図4のように、属性計算の実行をある境界でs1とs2の二つに分割する。デバッガは、この境界を通る属性の値（この例では、L1.posとL2.valの値）が正しいかどうかをユーザに問い合わせる。すべて正しい場合は、s2にバグが一つ以上含まれる。どれかが誤っているときは、その属性値を計算した属性計算のスライスにバグが含まれる。このような分割と問い合わせを繰り返すことで一つの属性計算が残り、バグが特定される。

3. 従来手法の問題点および新手法の概要

本節では、まず、属性文法に対するアルゴリズムック・デバッグの従来手法の問題点を指摘し、本提案の概要を述べる。

3.1 属性文法のアルゴリズムック・デバッグの問題点

属性文法に対するアルゴリズムック・デバッグの特長には次の二つがある。

- 問い合わせの回数が計算木のノードの数 n に対して $\log_2 n$ のオーダーであるため、計算木が大きくなっても質問の回数はほとんど増大しない。
- 質問に答えるときは、対象となる部分木にのみ注目すればよく、大きな木構造に煩わされることがない。

その一方で、Synth関数を用いて、アルゴリズムック・デバッグを属性文法へ適用するナイーブな方法では、次のような問題点が挙げられる。

- 問い合わせの場所が計算木の根に近いほど、質問に答えることが困難になる。
- 提示された Synth関数の挙動の質問に対する答え以外の情報を、ユーザがデバッガに反映させることができない。

第一の問題は、次のようなことである。例1のデバッグを行う場合、図3の計算木ノード(a)と(b)での問い合わせを比べると、根に近い(b)の方が、大きな部分木を質問の対象とするので、正しいかどうかを調べるのがより難しいと言える(図5)。Synth関数は、部分木を引数として、再帰呼び出しによって、子供の部分木を訪れながら意味を計算していく関数と

* 様々な評価順が考えられるが、そのうちの一つを示した

してとらえられるので、計算木の根に近い部分では、大きな部分木を引数にとる可能性が高い。特に、根に近い部分の属性の計算に、バグの原因があるときには、問い合わせる Synth 関数のノードも根に近い部分におよぶため、大きい部分木を対象とする問い合わせが起りやすい。

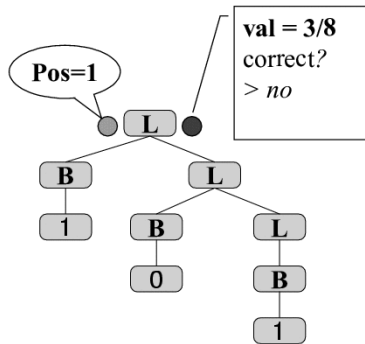


図 5 大きい部分木を対象とする質問
Fig. 5 Query with a large sub-tree

第二の問題は、例えば、ユーザが、提示された Synth 関数の挙動の質問の前提となる継承属性の値が、木全体に対しておかしいと気づくことがある。このような、バグの徴候をもつ属性をユーザが指摘できる場合は、Synth 関数の挙動に対する答えを考えるのではなく、属性文法のプログラム・スライシングを用いて、値がおかしいと分かっている属性に影響を与えている計算を追跡することが考えられる。しかし、スライス分割によるデバッグをはじめた後は、アルゴリズムック・デバッグの手法を用いること（あるいはその逆）は、一般にはできない。たとえば、スライス分割によってバグを含む可能性のある属性計算の集合をある程度特定し、その後にアルゴリズムック・デバッグを行うおうとしたとする。しかし、前のデバッグによって得られた情報が Synth 関数の構造に対応するものでなければ、アルゴリズムック・デバッグではその情報を利用することはできない。

この他の問題点として、従来のアルゴリズムック・デバッグは、実行時エラーの際の適用が難しい、最終的にバグの候補として複数の属性評価規則が残ってしまう等の点が挙げられる。したがって、この手法を単独で用いてデバッグを行うには限界がある。

3.2 提案するデバッグ法の概要

上述した問題点は、問い合わせのパターンを特定の方法に限っていることが原因である。Synth 関数だけ

を用いる場合は、必ず（完全な）部分木を質問の対象としなければならない。しかし、部分木および属性間の値の関係を問い合わせる仕方は、Synth 関数の形に限る必要はなく、図 6 のような質問も考えられる。

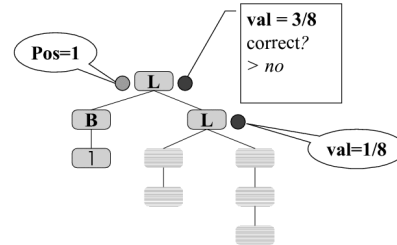


図 6 完全でない部分木を対象とする質問
Fig. 6 Query with an incomplete sub-tree

Synth 関数では、引数となる木は完全な部分木でなければならないが、この例は、部分木の一部が欠けているものを対象としている。この質問では、欠けている部分木の属性 *val* の値 $1/8$ が与えられているとして、残りの部分で行われるはずの計算による影響のみを考えれば良い。この質問は、直観的には図 3 の計算木ノード (b) の質問の代わりに、「(b) のうち (a) に関する部分を除いた計算」に関する質問と考えることができる。

以下で提案する属性文法のデバッグ法は、このように一般的な質問の形を扱うことのできる、拡張されたアルゴリズムック・デバッグである。このデバッグ法により、Synth 関数では大きくなってしまいう質問を分割することが可能となる。また、このデバッグ法は、Synth 関数に基づいた質問のみならず、属性値の誤りの指摘（従来、スライスを用いたデバッグ手法に移行しなければならなかった）を、特殊な場合として含む統合的な新手法である。したがって、場面に応じて（可能であれば）、Synth 関数の聞き方を行ったり、スライスを利用することができる。

このように Synth 関数とは異なる問い合わせが行えるのは、属性文法におけるアルゴリズムック・デバッグの計算木のノードとなる関数構造は Synth 関数に限らなくてもよい、という点に着目したことによる。例 1 において、 $L2.val$ の Synth 関数は、 $L2$ を根とする部分木と $L2.pos$ の値を入力として $L2.val$ の値を計算する。この計算を属性評価のグラフ（図 4）に示すと、(a) の境界で囲まれた部分の計算に対応する。同様に、 $L1.val$ の Synth 関数は、図 4 の (b) に対応する。一方 (c) の計算は、Synth 関数ではないが、こ

れも関数の計算と見なすことができる。この (c) の関数の挙動に対する質問が、図 6 の質問に対応する。^{*} 4 節では、このような関数の作り方、およびそれらを用いたアルゴリズムック・デバッグの手法を詳しく論ずる。

4. 属性文法の一般化アルゴリズムック・デバッグ

4.1 属性計算

属性評価をモデル化すると、

- (1) 構文解析を行い、解析木を作る。
- (2) 属性の依存関係に従った順番で各属性インスタンスの値を求める。

となる。実際にある属性インスタンスの値を計算することを属性計算と呼ぶことは先に述べたが、もう一度その定義を述べる。

定義 1 [属性計算] ある属性インスタンス o の値を計算する、という計算動作を o の属性計算 (Attribute Computation) と呼ぶ。また、 o が直接依存する属性インスタンスの列を I として、 $\langle I \rightarrow o \rangle$ と記述する。なお、 o がどの属性インスタンスにも依存しない場合、 $\langle \rightarrow o \rangle$ と書く。

例: 例 1 において、属性計算 $L1.val = B1.val + L2.val$ と属性計算 $L1.pos = 1$ は、それぞれ

$$\langle (B1.val, L2.val) \rightarrow L1.val \rangle, \langle \rightarrow L1.pos \rangle$$

と表される。

以下、本節で例を示す場合、例 1 の属性評価 (図 2 あるいは、図 4) の例を用いる。

定義 2 [属性計算の挙動] 属性インスタンスの列 $I = (i_1, \dots, i_n)$ の値が $X = (x_1, \dots, x_n)$ の時、属性計算 $\langle I \rightarrow o \rangle$ が o に値 y を代入する計算を $\langle \langle I \rightarrow o \rangle, X, y \rangle$ と表し、これを属性計算 o の挙動と呼ぶ。

例: 属性計算 $\langle (B1.val, L2.val) \rightarrow L1.val \rangle$ の挙動は、以下ようになる。

$$\langle \langle (B1.val, L2.val) \rightarrow L1.val \rangle, (1/4, 1/8), 3/8 \rangle$$

4.2 属性計算合成

従来のアルゴリズムック・デバッグにおける Synth 関数を、属性計算の観点から考えると、「ノード N の継承属性の列 $N.I$ から合成属性 $N.s$ までの属性計算を合成してできる関数」と考えることができる。このように、属性計算をまとめることで作られる関数

を属性計算合成 (Attribute Computation Composition) と呼ぶ。属性計算合成は、Synth 関数のような同じノードの継承属性と合成属性の間に限らず、任意の (直接的あるいは間接的に) 依存する属性間で作ることができる。

例えば、例 1 において、 $L1.pos$ から $B3.pos$ までの属性計算合成は、

$$\langle L1.pos \rightarrow L2.pos \rangle$$

$$\langle L2.pos \rightarrow L3.pos \rangle$$

$$\langle L3.pos \rightarrow B3.pos \rangle$$

^{**} という属性計算の列を合成した $B3.pos = L1.pos + 2$ である。このような属性計算をまとめたものの動作を、ユーザが抽象的にとらえることができれば、バグの範囲を効率よく狭めることが可能となる。

定義 3 [属性計算合成] 属性インスタンスの列 $I = (i_1, \dots, i_n)$ および $O = (o_1, \dots, o_m)$ が以下を満たすとする。

- 任意の $i \in I$ に対し、 $\exists o \in O : o$ は直接的あるいは間接的に i に依存する。

このとき、 I の値がすべて与えられた時に、属性インスタンスの列 O の値を決める計算を属性計算合成といい $\langle I \Rightarrow O \rangle$ と表す。特に $I = \{\}$ の場合は $\langle \Rightarrow O \rangle$ と表す。また、 $\langle I \Rightarrow O \rangle$ の I および O をそれぞれ、この属性計算合成に対する入力および出力と呼ぶ。

また、 I から O を計算する際に、計算が行われる属性計算の集合を $Set(\langle I \Rightarrow O \rangle)$ で表し、次のように定義する。

- (1) 任意の $o \in O$ に対して

$$\langle X \rightarrow o \rangle \in Set(\langle I \Rightarrow O \rangle)$$

- (2) $\langle X \rightarrow y \rangle \in Set(\langle I \Rightarrow O \rangle)$

$$\Rightarrow x \in X \wedge x \notin I \text{ なる任意の } x \text{ に対して}$$

$$\langle X' \rightarrow x \rangle \in Set(\langle I \Rightarrow O \rangle)$$

これは直観的には、各 $o \in O$ から依存辺を逆に $i \in I$ までたどる際に、通過した属性計算を集めたものである。ただし、後述の例 A の $B1.val$ のように、依存辺を逆にたどった結果、どの $i \in I$ にもたどり着かず、 $\langle \rightarrow o \rangle$ なる計算に行きつくこともある。

この $Set(\langle I \Rightarrow O \rangle)$ を $\langle I \Rightarrow O \rangle$ に含まれる属性計算集合と呼ぶこともある。

例 A: 例 1 において、

$$Set(\langle L2.pos \Rightarrow (L3.pos, B1.val) \rangle)$$

$$= \{ \langle L2.pos \rightarrow L3.pos \rangle, \langle \rightarrow L1.pos \rangle, \langle L1.pos \rightarrow B1.pos \rangle, \langle B1.pos \rightarrow B1.val \rangle \}$$

^{*} この関数の出力は境界線を外にまたぐ $L2.pos$ と $L1.val$ である。したがって、(c) の関数の完全な問い合わせとしては、図 6 の質問に加え、 $L2.pos$ に関する質問も必要となる。

^{**} 例を示す際、列の長さが 1 の場合は、括弧を省略する。

なお、属性評価の入力となる解析木が T 、 T の根の合成属性の列を (s_1, \dots, s_n) とすると、この属性評価全体は、属性計算合成 $\langle \Rightarrow(T.s_1, \dots, T.s_n) \rangle$ で表すことが出来る。

定義 4 [属性計算合成の挙動] 属性計算合成 $\langle I \Rightarrow O \rangle$ の挙動は、属性計算の挙動と同様に表記する。すなわち、 I 、 O の属性インスタンスの値がそれぞれ X 、 Y の時、 $\langle I \Rightarrow O \rangle$ の挙動は、 $\langle \langle I \Rightarrow O \rangle, X, Y \rangle$ と記す。

例: 例 1 において $\langle L1.pos \Rightarrow B3.pos \rangle$ の挙動は、 $\langle \langle L1.pos \Rightarrow B3.pos \rangle, 1, 3 \rangle$

なお、定義 3 および定義 4 では、属性計算合成の出力となる値は、入力となる属性値のみから決まるものとしている。本来、属性インスタンスの値は部分解析木にも依存するので、部分解析木も属性計算合成の入力に含める定義が考えられる。しかしここでは、解析木全体が定数としてあらかじめ与えられているものとするので、部分解析木の形は属性の値には影響を与えないと考える。よって、定義 3 および定義 4 では、部分解析木は属性計算合成の入力に含めていない。

4.3 属性計算合成の性質

属性計算合成の性質について述べる。

包含関係

$Set(\langle I_1 \Rightarrow O_1 \rangle) \supset Set(\langle I_2 \Rightarrow O_2 \rangle)$ とは、 $\langle I_2 \Rightarrow O_2 \rangle$ が $\langle I_1 \Rightarrow O_1 \rangle$ の内側にネストされた計算であることを示す。

例 a: 例 1 において (図 4) ,
 $Set(\langle L1.pos \Rightarrow L1.val \rangle)$
 $\supset Set(\langle L3.pos \Rightarrow L3.val \rangle)$

アルゴリズムック・デバッグでは、計算木の $L3.val$ に関する Synth 関数は、 $L1.val$ に関する Synth 関数の子孫となっている。このように、Synth 関数の子孫関係は、属性計算合成の包含関係と対応する。

例 b: 例 1 において (図 4 の (b) および (c)) ,
 $Set(\langle L1.pos \Rightarrow L1.val \rangle)$
 $\supset Set(\langle \langle L1.pos, L2.val \rangle \Rightarrow \langle L1.val, L2.pos \rangle \rangle)$

この例のように、Synth 関数に対応しない属性計算合成に対しても、包含関係を考えることができる。

排反関係

$Set(\langle I_1 \Rightarrow O_1 \rangle) \cap Set(\langle I_2 \Rightarrow O_2 \rangle) = \phi$ すなわち、両集合が排反であるときは、二つの属性計算合成 $\langle I_1 \Rightarrow O_1 \rangle$ 、 $\langle I_2 \Rightarrow O_2 \rangle$ の計算が互いに影響を与えあわないことを意味する。

例 a: 例 1 において、

$$Set(\langle L2.pos \Rightarrow L2.val \rangle) \\ \cap Set(\langle B1.pos \Rightarrow B1.val \rangle) = \phi$$

アルゴリズムック・デバッグの計算木において、二つの Synth 関数が子孫の関係ではない場合、対応する属性計算合成同士の間には排反関係が成り立つ。

例 b: 例 1 において (図 4 の (a) および (c)) ,
 $Set(\langle L2.pos \Rightarrow L2.val \rangle)$
 $\cap Set(\langle \langle L1.pos, L2.val \rangle \Rightarrow \langle L1.val, L2.pos \rangle \rangle)$
 $= \phi$

この例のように、Synth 関数に対応しない属性計算合成に対しても、排反関係を考えることができる。

4.4 閉じた属性計算合成

属性計算合成 $\langle I \Rightarrow O \rangle$ において、その挙動の入力の値と出力の値の関係がおかしいとユーザが判断できる場合、 $Set(\langle I \Rightarrow O \rangle)$ にバグを含むことが言える。一方、 $\langle I \Rightarrow O \rangle$ が正しいと判断できたとしても、 $Set(\langle I \Rightarrow O \rangle)$ にバグを含まないとは一般には言えない。このことは 4.5 節で示す。閉じた属性計算合成とは、 $\langle I \Rightarrow O \rangle$ が正しいとユーザが判断したならば、 $Set(\langle I \Rightarrow O \rangle)$ の中にはバグを含まないことを保証できるような属性計算合成 $\langle I \Rightarrow O \rangle$ である。*

定義 5 [真に閉じた属性計算合成] $\langle I \Rightarrow O \rangle$ に対し、 $AS = Set(\langle I \Rightarrow O \rangle) - \{ \langle X \rightarrow o \rangle \mid o \in O \}$ とおく。属性計算合成 $\langle I \Rightarrow O \rangle$ が真に閉じているとは、AS に含まれる全ての計算が、 $\{ \langle X \rightarrow o \rangle \mid o \in O \}$ を経ない限り、 $Set(\langle I \Rightarrow O \rangle)$ の外の計算、すなわち $Set(AE) - Set(\langle I \Rightarrow O \rangle)$ に到達できないことを言う。ただし AE は、属性評価全体を表す属性計算合成を示す。

図 7 によって、この定義を説明する。今、 $\langle I_2 \Rightarrow O_2 \rangle$ は真に閉じていない。なぜなら、 $AS_2 = Set(\langle I_2 \Rightarrow O_2 \rangle)$ に含まれる属性計算 (a) および (b) は、 O_2 の計算を経ずに、 $Set(\langle I_2 \Rightarrow O_2 \rangle)$ の外側の計算 (c) および (d) に影響を与えるからである。

例 a: 例 1 において、 $\langle \Rightarrow B1.pos \rangle$ は真に閉じていない (図 8 の (a))。今、

$$Set(\langle \Rightarrow B1.pos \rangle) \\ = \{ \langle \rightarrow L1.pos \rangle, \langle L1.pos \rightarrow B1.pos \rangle \}$$

である。上記の定義の AS に対応する集合は、 $\{ \langle \rightarrow L1.pos \rangle \}$ である。ところが、 $\langle \rightarrow L1.pos \rangle$ は、 $B1.pos$ の計算を経ずに外の計算、例えば $L2.pos$ の計算に到達する。よって、 $\langle \Rightarrow B1.pos \rangle$ は真に閉じてい

* 正確には、 $\langle I \Rightarrow O \rangle$ を内側の計算として含む属性計算合成の挙動が誤っているとき、その原因となるバグが $Set(\langle I \Rightarrow O \rangle)$ に含まれないことを保証する。

の計算を通じて、 $L1.pos$ の計算から影響を受けるからである。

4.5 アルゴリズムックデバッグ

定義 7 [挙動が正しい/誤っている] 属性計算合成の挙動 $\langle\langle I \Rightarrow O \rangle\rangle, X, Y$ において、ユーザの意図する $\langle I \Rightarrow O \rangle$ に照らし合わせ、 X と Y の関係が正しい場合、 $\langle I \Rightarrow O \rangle$ の挙動が正しいといい、逆に関係が誤っている場合、 $\langle I \Rightarrow O \rangle$ の挙動が誤っていると言う。以下、これらをそれぞれ、

$$\text{Query}(\langle I \Rightarrow O \rangle) = \text{correct}$$

$$\text{Query}(\langle I \Rightarrow O \rangle) = \text{incorrect}$$

と記述することもある。 ■

問い合わせの対象となる木

ある属性計算合成の挙動が正しいかどうかをユーザが判断する際、その判断の基準となるのは、与えられた木全体である。しかし、実際には部分的な木があればよい。

解析木が与えられているとすると、属性計算合成 $\langle I \Rightarrow O \rangle$ は、 $\text{Set}(\langle I \Rightarrow O \rangle)$ から一意に定まる。よって、 $\langle I \Rightarrow O \rangle$ に対応する部分木は $\text{Set}(\langle I \Rightarrow O \rangle)$ に含まれる各属性計算を定義している生成規則を、適切に貼り合わせた形の木となる。

例えば、例 1 の $\langle \Rightarrow B1.val \rangle$ の挙動をユーザが知るために必要な木は次のように決まる。今、 $\text{Set}(\langle \Rightarrow B1.val \rangle)$ に含まれる各属性計算を定義している生成規則は、次のようになる。

属性計算	対応する生成規則
$\langle \rightarrow L1.pos \rangle$	$F ::= . L$
$\langle L1.pos \rightarrow B1.pos \rangle$	$L_0 ::= B L_1$
$\langle B1.pos \rightarrow B1.val \rangle$	$B ::= 1$

したがって、対応する部分木は、生成規則 $\{F ::= . L, L_0 ::= B L_1, B ::= 1\}$ を貼りあわせたもので、具体的には図 2 において、木全体のうち $L2$ の子供の部分木を刈取った形のものである。

定義 8 [バグ] 属性計算合成 $\langle I \Rightarrow o \rangle$ に対し、 $\text{Set}(\langle I \Rightarrow o \rangle)$ が、ただ一つの要素 $\langle I \rightarrow o \rangle$ から成るとする。このとき、 $\langle I \Rightarrow o \rangle$ の挙動が誤っているならば、 $\langle I \rightarrow o \rangle$ をバグと呼ぶ。* ■

定理 1 $\text{Query}(\langle I \Rightarrow O \rangle) = \text{incorrect}$ ならば、 $\text{Set}(\langle I \Rightarrow O \rangle)$ 内にバグが一つ以上含まれる。 ■

証明は、ほとんど明らかであるので省略する。

定理 2 $\text{Query}(\langle I_1 \Rightarrow O_1 \rangle) = \text{incorrect}$ とする。こ

* 「 o に対する属性評価規則にバグが含まれる」という言い方が自然であるが、以下の議論では「バグ」を属性計算に対して用いる。

のとき、 $\langle I_1 \Rightarrow O_1 \rangle$ に関して閉じた任意の $\langle I_2 \Rightarrow O_2 \rangle$ に対して、 $\text{Query}(\langle I_2 \Rightarrow O_2 \rangle) = \text{correct}$ とユーザが判断できる場合、 $\text{Set}(\langle I_1 \Rightarrow O_1 \rangle) - \text{Set}(\langle I_2 \Rightarrow O_2 \rangle)$ 内にバグが一つ以上含まれる。 ■

[証明] (略証) $\text{Set}(\langle I_1 \Rightarrow O_1 \rangle) - \text{Set}(\langle I_2 \Rightarrow O_2 \rangle)$ にバグが含まれないと仮定する。すると、 $\text{Set}(\langle I_2 \Rightarrow O_2 \rangle)$ に誤りがあることになる。そこで、その誤った属性計算を $\langle I' \rightarrow o' \rangle$ とする。 $\text{Query}(\langle I_2 \Rightarrow O_2 \rangle) = \text{correct}$ であった。 $\text{Query}(\langle I_1 \Rightarrow O_1 \rangle) = \text{incorrect}$ となるためには、誤った計算 $\langle I' \rightarrow o' \rangle$ が O_2 の計算を経ずに、 O_1 に影響を与えなければならない。これは、前提 $\text{Closed}(\langle I_2 \Rightarrow O_2 \rangle, \langle I_1 \Rightarrow O_1 \rangle)$ に反する。(証明終り)

定理 2 において、 $\langle I_2 \Rightarrow O_2 \rangle$ が $\langle I_1 \Rightarrow O_1 \rangle$ に関して閉じているという条件がないと、バグを発見できない場合を示す。仮に、文法 $G1$ にもう一つバグがあり、図 1 の (a) の評価規則が「 $L.pos = 0$ 」であったとする。このとき結果は $F.val = 3/4$ と誤った値となる。今、真には閉じていない $\langle \Rightarrow B1.pos \rangle$ に注目する (図 8)。このとき、 $\langle \langle \Rightarrow B1.pos \rangle, \{ \}, 1 \rangle$ であるから、この挙動は正しい。ここで仮に、 $\text{Set}(\langle \Rightarrow B1.pos \rangle)$ にバグがないと結論づけるとする。 $\text{Set}(\langle \Rightarrow B1.pos \rangle)$ 以外の属性計算にも誤りがないので、属性評価全体の中にバグは含まれないことになり矛盾する。これは、 $\langle \Rightarrow B1.pos \rangle$ が誤った二つの規則によって、偶然正しい挙動となったが、閉じていないため誤っている計算 $\langle \rightarrow L1.pos \rangle$ が $B1.pos$ の計算を経ずに、つまり $L2.pos$ の計算を通じて、 $\text{Set}(\langle \Rightarrow B1.pos \rangle)$ 以外の属性計算に影響を与えていることによる。

系 1 $\text{Query}(\langle I \Rightarrow O \rangle) = \text{incorrect}$ とする。属性計算合成の集合 $\{\text{ACC}_k\} (1 \leq k \leq n)$ が以下を満たすとき (図 9) ,

- $\text{Closed}(\text{ACC}_k, \langle I \Rightarrow O \rangle)$
- $\text{Query}(\text{ACC}_k) = \text{correct}$
- $\text{Set}(\text{ACC}_i), \text{Set}(\text{ACC}_j) (i \neq j, 1 \leq i \leq n, 1 \leq j \leq n)$ が排反

$$\text{Set}(\langle I \Rightarrow O \rangle) - \bigcup_{k=1}^n \text{Set}(\text{ACC}_k)$$

にバグが一つ以上含まれる。 ■

なお、系 1 における各 ACC_k が排反でない場合には、必ずしもこの系が成立しないことを示す。今、 ACC_1 と ACC_2 は、系 1 の条件を満たすが互いに排反でない属性計算合成とする。このとき、

- $\text{Closed}(\text{ACC}', \text{ACC})$
- $\text{Set}(\text{ACC}') = \text{Set}(\text{ACC}_1) \cup \text{Set}(\text{ACC}_2)$

を満たす、属性計算合成 ACC' が存在する。仮定より、

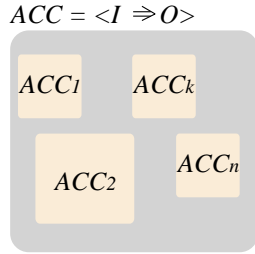


図 9 系 1 の説明

Fig. 9 Illustration for Corollary 1

ACC_1 と ACC_2 の挙動は正しい。しかしこのことは、次の例に示すように、 ACC' の挙動が正しいことを必ずしも保証しない。したがって、 $Set(ACC) - Set(ACC')$ にバグを含むとは言えない。

例: ある解析木の属性評価において、整数値をとる 4 つの属性インスタンス a, b, c, d があり、また、 b, c, d に対して適用する評価規則が、 $f(x) = -x$ とおいて、それぞれ $b = f(a), c = f(b), d = f(c)$ であったとする。しかし、この f が誤りで、実際は $f(x) = x$ がユーザの意図した評価規則であったとする。ここで、真に閉じた属性計算合成 $\langle a \Rightarrow c \rangle \equiv ACC_1$, $\langle b \Rightarrow d \rangle \equiv ACC_2$, $\langle a \Rightarrow d \rangle \equiv ACC'$ を考える。すると、 $Set(ACC_1) \cup Set(ACC_2) = Set(ACC')$ である。このとき、 $Query(\langle a \Rightarrow c \rangle) = correct$, $Query(\langle b \Rightarrow d \rangle) = correct$ 。しかし $\langle a \Rightarrow d \rangle$ の挙動は必ずしも正しいとは言えない。なぜなら、 a の属性値 x が非零ならばユーザが意図する挙動は $\langle \langle a \Rightarrow d \rangle, x, x \rangle$ であるが、実際は $\langle \langle a \Rightarrow d \rangle, x, -x \rangle$ と、誤った挙動となるからである。■

系 2 $ACC = \langle I \Rightarrow O \rangle$, $Query(ACC) = incorrect$ とする。今、属性計算合成の集合 $\{ACC_k\} (1 \leq k \leq n)$ が、系 1 の仮定の $\{ACC_k\}$ と同じ条件を満たすものとする。このとき、以下を満たす ACC' を考える (図 10)。

- $Closed(ACC', ACC)$
- $Set(ACC_k) \subset Set(ACC') \quad (1 \leq k \leq m)$
- $Set(ACC_k)$ と $Set(ACC')$ は排反 ($m < k \leq n$)

このとき、 $Query(ACC') = correct$ の場合、

$$Set(ACC) - (Set(ACC') \cup \bigcup_{k=m+1}^n Set(ACC_k))$$

★ にバグが一つ以上含まれる。また、 $Query(ACC') =$

★ この式は次の式とも同値

$$Set(ACC) - (Set(ACC') \cup \bigcup_{k=1}^n Set(ACC_k))$$

incorrect の場合、

$$Set(ACC') - \bigcup_{k=1}^m ACC_k$$

にバグが一つ以上含まれる。 ■

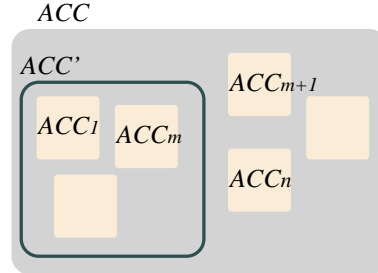


図 10 系 2 の説明

Fig. 10 Illustration for Corollary 2

4.6 デバッグングアルゴリズム

前節で述べた定理から、属性文法の一般化アルゴリズム・デバッグングのアルゴリズム GAD (図 11) が次のような簡潔な形で得られる。

このアルゴリズムは、系 2 を再帰的に適用することで、バグの範囲を狭めていくものである。 GAD_{init} は、初期条件を与える関数であり、 GAD_{main} が実際に系 2 の再帰的な適用を行う関数である。 GAD_{main} の引数に与えられる ACC は、 $Query(ACC) = incorrect$ すなわち挙動が誤っている属性計算合成である。 $\{ACC_1, \dots, ACC_n\}$ は、この ACC に関して系 1 の条件をみたすような属性計算合成の集合である。

$getNextACC$ は、系 1 の条件が成り立っている、 ACC と、 $\{ACC_1, \dots, ACC_n\}$ を引数とし、次の質問の対象とする属性計算合成として、系 2 を満たす ACC' および m を一つ選ぶ関数である。

なお、このアルゴリズムは、 $getNextACC$ が正しく実装されていれば、有限回の問い合わせで、バグを含む属性計算の候補を ϵ 個以内に絞りこむことができる。このことは、(1) GAD_{main} の呼び出しでは、 $ACC, \{ACC_1, \dots, ACC_n\}$ は系 1 を満たし、かつ (2) 再帰的に GAD_{main} を呼び出す際、必ずバグの探索範囲が減る、という 2 点から言える。

このアルゴリズムは、従来法を包含する一般的な形になっている。 $getNextACC$ 関数がユーザに与える質問の形を決めるので、この関数の実現の仕方によってさまざまなデバッグ法を扱うことができる。

従来の $Synth$ 関数によるアルゴリズム・デバッグングは、 $getNextACC$ 関数を、 $Synth$ 関数に対応

```

GADinit(ACC){
  return GADmain(ACC, {});
}

GADmain(ACC, {ACC1, ..., ACCn}){
  /* 系 1 */
  bugACs = Set(ACC) -  $\bigcup_{1 \leq k \leq n} Set(ACC_k)$ ;
  if size(bugACs) ≤ ε
    /* 要素数が少なければバグとして報告 */
    return bugACs;
  /* 系 2 の条件を満たす ACC' および m を選ぶ */
  ACC' = getNextACC(ACC, {ACC1, ..., ACCn});
  if Query(ACC') == correct
    /* 挙動が正しいなら ACC' を
       挙動の正しい ACC 集合に加える */
    return
      GADmain(ACC, {ACC', ACCm+1, ..., ACCn});
  else
    /* 挙動が誤っているなら探索域を ACC' に狭める */
    return GADmain(ACC', {ACC1, ..., ACCm});
}

```

図 11 アルゴリズム GAD
Fig. 11 Algorithm GAD

する属性計算合成のうち、系 2 の ACC' を満たすものを選択する関数とすることで実現できる。また、スライス分割によるデバッグ法も、入力が空である属性計算合成 ($\Rightarrow O$) のうち、系 2 の ACC' を満たすものを選ぶ関数とすることで実現できる。これらの手法では 2 分探索を用いて、効率的にバグを含む範囲を減らせるように ACC' を選ぶ。具体的には、Query(ACC') の結果が correct, incorrect のどちらとなっても、次に問い合わせる属性計算合成の候補の数が、ほぼ等しくなるような ACC' を選択する。

4.7 新しいデバッグ法

属性文法の一般化アルゴリズム・デバッグの一適用として、新しいデバッグ法を導入した。この方法は、従来の Synth 関数によるアルゴリズム・デバッグの方法を基本として、これにスライス分割による方法を融合した形になっている。これは、アルゴリズム GAD の getNextACC 関数において ACC' として選ぶ属性計算合成を、Synth 関数の形および、スライス分割の形の 2 通りを扱えるようにし、また、従来法のように 2 分法の戦略を用いて質問の回数を抑えるように ACC' を選ぶように実現される。このデ

バッグ法は、以下のような特徴をもっている。

一つ目の特徴は、Synth 関数の挙動の問い合わせ時に、Synth 関数の前提となる継承属性 i がおかしいとユーザが指摘できるようになっていることである。もし、 i がおかしいとユーザが指摘した場合は、その継承属性を出力に含む属性計算合成の挙動に関する質問を行う。この質問は、属性計算合成の入力となるある属性を与え、それに対する継承属性 i の値が正しいかどうかを尋ねるものである。これは、Synth 関数の形をしておらず、従来の拡張になっている。

この後のデバッグのやり方は、スライスを分割する手法と同様のやり方で、属性計算合成を分割しながらバグを含む範囲を狭めていく。この時、質問は Synth 関数の形とは限らないが、適切な場所で Synth 関数の挙動の問い合わせに戻るようになっている。

二つ目の特徴は、実行時エラーが起った場合にも、適用可能であるということである。通常の場合、すなわち、属性評価が完全に行われた場合は、デバッグは Synth 関数によるアルゴリズム・デバッグの方法から開始する。一方、計算 ($I \rightarrow o$) で実行時エラーが起き、属性評価が完了しなかった場合は、属性 o の値が誤っている指摘があったとして、スライスの分割によるデバッグから始める。これにより、Synth 関数の素朴な適用における「出力がエラーだがこの挙動は正しいか」という難しい質問を出題しないようにできる。

三つ目の特徴は、完全でない部分木、すなわち部分木のうちいくつかのノードが欠けたような形を扱える点である（例えば、図 6）。特に、Synth 関数の問い合わせで、対象となる部分木が大きくなる時は、図 5 のような完全な部分木では答えにくい場合がある。この場合、ユーザが違う質問を要求することで、図 6 のような完全でない部分木に対する質問に答えればよいようにした。

四つ目の特徴は、精度の高いバグの特定である。従来のアルゴリズム・デバッグでは、バグの候補として最終的に特定できるのは、一般に複数の属性評価規則である。これは、従来のアルゴリズム・デバッグが、属性評価における計算の単位を、Synth 関数としているからである。一方、本手法では、計算の単位を属性計算としているため、一つの属性評価規則をバグとして発見することができる。

これらの特徴は、様々な形の問い合わせを必要とする。しかし、それぞれの形の問い合わせを、属性計算合成の挙動に対する質問として考えることにより、前節の一般化アルゴリズム・デバッグの中で統一的に扱うことができる。

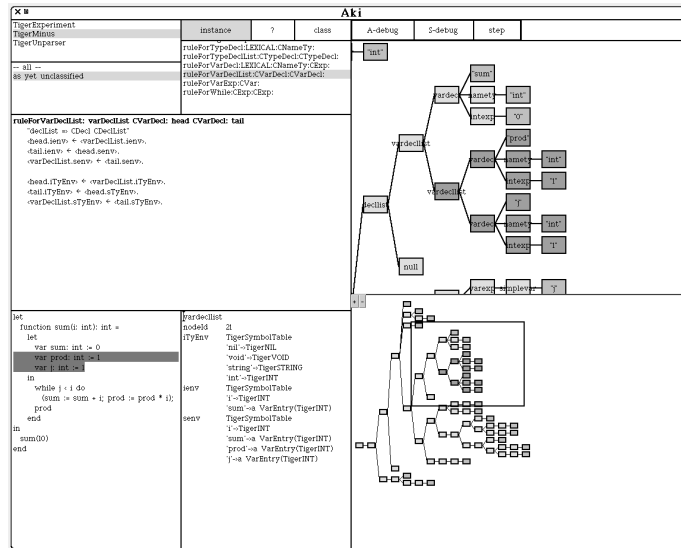


図 12 デバッガ Aki
Fig. 12 Debugger Aki

5. デバッガ Aki

デバッガ Aki は、属性文法によるコンパイラ開発環境¹⁰⁾に追加する形で実装された。この開発環境では、コンパイラの各フェーズを属性文法によって記述する。評価器生成系 Rie および Jun は、これらの属性文法記述から、それぞれフロントエンドおよびバックエンドを自動生成する。Aki は、これらのコンパイラの仕様記述のデバッグを支援するシステムである。

Aki は、デバッグ手法として、2 節で述べたアルゴリズムック・デバッグ、スライス分割によるデバッグ法を提供している。さらに本研究において、4.7 節で提案したデバッグ方式を、新たに実装した。

Aki は、原理的にはどのような属性文法システムにも適用できるが、現在の実装では Aki は評価器生成系 Jun とともに用いることを前提とする。Aki によって属性文法記述をデバッグする場合には、Jun が生成する、デバッグ用に拡張された評価器を用いる。この拡張された評価器は、実行時（コンパイル時）に属性評価の過程で計算される属性インスタンスの値をトレースとして記録することができる。

Aki は、属性文法記述および入力となる解析木から、属性インスタンスの依存関係を計算し、また、評価器のトレースから得られる属性値をもとに、系統的デバッグを行う。

Aki によるコンパイラ仕様記述のデバッグの様子を 図 12 に示した。各ペーンは、属性文法記述、ソースプログラム、属性値、解析木をそれぞれ提示する。問

い合わせは、図 13 に示すダイアログウィンドウによって行われる。

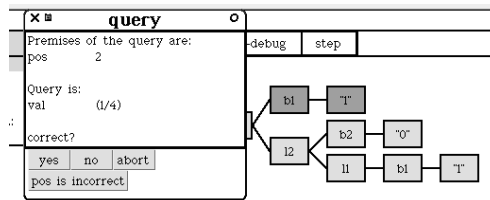


図 13 問い合わせ
Fig. 13 Query

Aki によるデバッグ例を以下に示す。この例は、例 1 を、4.7 節で述べた方法を用いて、デバッグする様子である。まず、最初の質問 (a) は、挙動 $\langle\langle L2.pos \Rightarrow L2.val \rangle\rangle, 2, 1/8$ が正しいかどうかを尋ねる。属性の関係が正しい時は “yes”，誤っているときは “no” ボタンを押す。pos=2 に対し、val=1/8 は合っているためユーザは “yes” を選択する。

次の質問 (b) は、挙動 $\langle\langle L1.pos \Rightarrow L1.val \rangle\rangle, 1, 3/8$ が正しいかどうかであるが、これは誤っているため “no” を選択する。

次の質問 (c) は、挙動 $\langle\langle B1.pos \Rightarrow B1.val \rangle\rangle, 2, 1/4$ が正しいかどうかを尋ねている (図 13)。注目している部分木は、小数点以下 1 桁目であるにも関わらず pos が 2 であることから、前提がおかしいと答えることができる。そこで “pos is incorrect” を選択する。このように誤った属性値を指摘することは、従来のア

ルゴリズムック・デバッグングでは行えなかった。

次の質問 (d) は $\langle(L1.pos \Rightarrow B1.pos), 1, 2\rangle$ が正しいかどうかである。この問い合わせは、異なるノードの継承属性 $L1.pos$, $B1.pos$ の値の関係を尋ねている。このような、異なるノードの属性値の関係を問う質問は、本研究で提案するデバッグ法によってはじめて可能となった。この質問に対しては、 $L1$ と $B1$ の桁位置は同じであるべきであるが、増えているので、ユーザは誤りと判断し“no”を選択する。すると、バグを含む意味規則が一つ特定され、図 14 のように、その意味規則がハイライトされる。

前提が間違っていると答えた問い合わせ (c) では、“yes”と答えることもできたが、その場合はその後の問い合わせは、従来のアルゴリズムック・デバッグングと同じ過程となり、図 14 の 3 つの意味規則がバグを含む候補として特定される。

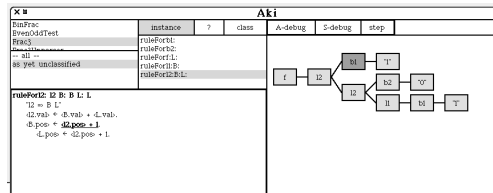


図 14 バグを特定した様子

Fig. 14 An erroneous rule inferred by Aki

6. 評価および議論

本節では、4 節で提案したデバッグ法の有効性について初期の評価を含め論ずる。

6.1 ユーザテスト

前節で述べたデバッガ Aki を用いて、ユーザテストを行なった。ユーザテストは、バグを含んだ属性文法記述と、挙動誤りを起こすソースプログラムを例題として用意し、3 名のユーザに、スライス分割による手法、従来のアルゴリズムック・デバッグング、および本手法のそれぞれを用いて、バグを発見してもらうというものである。本ユーザテストは、それぞれの手法による質問の答えやすさを比較することを主な目的とした。

例題となる属性文法記述はいずれも、コンパイラ構成の教科書¹⁾に例題として用いられている、Tiger 言語の静的意味チェックを行うプログラム (以下、Tiger-Front) である。この属性文法記述は、25 個の生成規則、105 個の意味規則からなる。

表 1 は、バグを発見するまでにユーザが答える質問の回数である。表の中で、属性数の項目は、属性

インスタンスの個数を表すが、実行時エラーが起った場合は、そのうちの値が求められるものの個数である。ノード数は、解析木のノードの個数である。Slice, AD, GAD の各項目はデバッグの方法を表し、それぞれ、スライス分割による方法、従来のアルゴリズムック・デバッグング、4.7 節で提案した新しいデバッグ法を用いた時のものである。括弧内の数字は、最終的に特定されたバグを含む意味規則の候補の数である。なお E および F は、実行時エラーによって、属性評価が途中で中断する例である。ここで、問い合わせには、答えることが難しいもの、簡単に答えられるものがあるため、回数の少ない方がデバッグの効率が良い、とは一概には言えない。このことについては以下の節で論ずる。なお GAD では、継承属性の誤りに気づいた場合に、これを指摘することができるので、この指摘の仕方によってユーザが答えるべき質問の回数は変わる場合がある。A の例では、一名のユーザが他のユーザと異なる場所で継承属性の誤りを指摘したが、この結果は角括弧内に示した通りである。

表 1 質問の回数の比較
Table 1 Result of query count

例	属性数	ノード数	Slice	AD	GAD
A	78	52	7(1)	8(3)	4(1)[6(1)]
B	146	103	6(1)	9(4)	5(1)
C	60	43	4(1)	8(3)	4(1)
D	56	34	9(1)	6(4)	7(1)
E	45	32	5(1)	5(2)	7(2)
F	104	69	7(1)	6(2)	2(2)

6.2 議論

本節では、ユーザテストで得られた結果および著者らがデバッガ Aki を使用した経験に基づき、本手法の特長について考察する。

ユーザが答えやすい質問を選べる Synth 関数による従来のアルゴリズムック・デバッグング (表 1 の AD) やスライスの方法 (同, Slice) を用いたデバッグでは、ユーザは、デバッガの出題する質問に答えることでのみ、デバッガにバグの範囲を狭める手がかりを与える。一方、本手法 (表 1 の GAD) では、継承属性の値がおかしいとユーザが気づいた場合、その情報をデバッガに反映することができる。すると、デバッガは、その異変のある値をもつ属性に関する質問をユーザに提示する。この方法により、従来の方法に比べ質問の回数は増えてしまう場合もあるが、ユーザが関心をもつ属性に対する問い合わせが行われるので、ユーザにとって答えやすい質問となることが多い。した

がって、結果的に効率よくバグを発見できる。

実行時エラーの扱い 従来のアルゴリズム・デバッグでは、実行時エラーが起こる例をデバッグする場合、合成属性の値が求められていない計算木ノードが質問の対象となる場合がある（表1のE, FをADで行う場合）。すなわち「ある前提に対して、合成属性の値が求められないのは正しいかどうか」という問い合わせが行われる。この質問に正確に答えることは困難である。実行時エラーが起こった計算から、スライスを用いてバグの範囲を狭めることは可能である。しかし、その後スライス分割による方法を用いることになるが、入力となる解析木を良く理解していない場合は、木全体を慎重に確かめて質問に答える必要がある。

一方、本手法では実行時エラーの場合でも、合成属性の値が計算出来なかったノードに対する難しい質問を行うことはない。また、スライスによる方法のように木全体ではなく、局所的な木を対象とした問い合わせを行うことができる。これは、従来のアルゴリズム・デバッグとスライス分割による手法とを融合することで可能となった。

大きな部分木に対する質問を減らす 前の項目に関連するが、3節で述べた通り、従来のアルゴリズム・デバッグでは、質問となる対象の木は、あるノード以下の全てのノードを含む完全な部分木である。一方、本手法は、質問の対象とする木として、部分木の中で、あるノード以下が欠けたような不完全な部分木を扱うことができる。

精度の高いバグの特定 従来のアルゴリズム・デバッグの手法でデバッグを行った場合は、最終的に特定できる意味規則は、一般的には複数となる。TigerFrontの場合、最大4つの意味規則がバグを含むものの候補として最終的に残る例があった。一方、本手法では、最後に見つけられるバグをただ一つの意味規則に特定することが可能である。（ただし、現在の実装では、表1のE, FのGADの列の通り、複数残る場合もある）。

以上、本手法によるデバッグの実用上のメリットとなる点を挙げた。上記の中には、従来のアルゴリズム・デバッグをアドホックに拡張して扱うことのできる項目もあるが、本手法では、これら全てを同じ一般化した枠組みの中で扱うことができるという利点がある。

実装上の工夫

問い合わせによって対話的にデバッグを行うシステ

ムでは、ユーザが質問を正しく把握することが大切である。そこで、Akiでは、質問の理解を助けるための次のような工夫が施されている。

まず、大きなデータ構造を値として持つ属性同士に関する質問を行う場合は、その差分を表示することで属性値がどのように変わったかを分かりやすくしている。例えば、TigerFrontでは、シンボルと型の対応を表す記号表を属性として扱うが、この記号表同士の関係を問う質問では、変化している部分を強調して表示する。

また、木の位置や形を把握しやすいように、木のノードと、ソースコードの対応する部分が瞬時に分かるようになっている。

課題

次のような課題も明らかとなった。

- 図6のような完全ではない部分木を扱えることは、質問に答えるためにユーザが確かめるべき、木の範囲を小さくできるというメリットがある。しかし、質問の対象となる各属性が、どのノードに属しているのかが分かりづらいことがある。そのような木の表示の仕方、質問の与え方に工夫が必要である。
- 今回実装したデバッグ法は、Synth関数の前提がおかしいとユーザが判断できた場合のみ、デバッグにそのような情報を反映することができた。この他にも、さらに4節で述べた枠組みの中で行えるデバッグ法を開発する必要がある。

7. 関連研究

Declarative Debugging Scheme⁹⁾は、アルゴリズム・デバッグを一般化した枠組みである。論理型言語以外のプログラミング言語にも適用可能であり、また、色々なクラスのバグを統一的に扱うことができるという特徴がある。したがって、この枠組みは、本論文で提案した属性文法の一般化アルゴリズムも包含する。しかし文献9)では、どのような計算が問い合わせの対象となるか、という指針が与えられていないため、属性文法に適用するには不十分である。一方、本論文では、問い合わせの対象となる属性計算合成の条件を明らかにしたため、新しい属性文法のデバッグ法を容易に導くことができる。

また、アルゴリズム・デバッグを用いる場合、他の技術を補助的に用いる場合が多い。そのような技術として以下が挙げられる。

Interprocedural slicing

通常、スライスは手続き内の文に対して計算するが、Interprocedural Slicing⁵⁾は、手続き呼び出しの依存

関係を考慮にいれ、手続き間にわたってスライスを計算する技術である。GADT¹²⁾は、アルゴリズムック・デバッグングを手続き型言語に適用したシステムである。このシステムでは、ユーザがある手続き呼び出しの挙動誤りを発見したあとに、誤っている出力パラメータに影響を与える手続き呼び出しのみをより精度高く抽出するために、Interprocedural Slicingを用いている。

4.7節のデバッグング・アルゴリズムでは、Synth関数の入力である継承属性の誤りを指摘した場合、その属性に影響を与えたスライスを他のSynth関数内にわたって求めているという点で、この技術を用いていると言える。

assertion

assertionとは、プログラムの持つべきある性質のことであり、実用的にはプログラムの実装のミスを効率よく発見するための手段として用いられる。関数にassertionを与えた場合、assertionを満たしている関数は挙動が正しいと言えるので、問い合わせの回数を減らすことが可能となる。また、アルゴリズムック・デバッグングの問い合わせにユーザが“yes/no”で答えることは、プログラムの性質を部分的に与えることと考えられるので、アルゴリズムック・デバッグングの自然な拡張として、assertionを用いることができる³⁾。

FORMAN²⁾は、assertionを用いてプログラムテスト、デバッグなどを半自動的にを行うためのシステムである。FORMANでは、アルゴリズムック・デバッグングを、assertionの特殊な形式として扱うことができるため、assertionとアルゴリズムック・デバッグングが統一的に扱える。このFORMANを用いて、EGADT⁴⁾では、手続き型言語でのアルゴリズムック・デバッグングにassertionを取り入れた。

このような、アルゴリズムック・デバッグングの拡張としてのassertionは、問い合わせの数を減らすことを目標としている。一方、本手法による属性文法の一般化アルゴリズムック・デバッグングは、属性評価における関数的な挙動を様々な形で考えることで、ユーザが答えやすい質問を行えるようにすることを目標としている。

8. おわりに

本論文では、属性文法の系統的デバッグ法を提案した。

まず、属性文法のアルゴリズムック・デバッグングの一般化を行った。この一般化により、様々な形の質

問を扱えるようになり、従来提案された、Synth関数を用いるアルゴリズムック・デバッグング、およびスライス分割によるデバッグの両手法を、統一的に説明することができた。さらに、この一般化したアルゴリズムック・デバッグングの一適用として、両手法を融合した新しいデバッグ法を提案した。

このデバッグ法を、属性文法デバッガAkiの拡張という形で実装した。また、新たに提案したデバッグ法と従来の手法を、Akiを用いた評価実験によって比較、考察をし、本デバッグ法の有用性を示した。

今後、本デバッグ法をより実用的なものとすることを目指し、本論文で提案した一般化アルゴリズムック・デバッグングを用いたより効率の良い新たなデバッグ手法の開発、assertionなど他の手法との組み合わせによる質問の省略や簡略化、質問の内容を把握しやすくするためのユーザインタフェースの研究、などを行っていく予定である。

謝辞 研究上の議論やユーザテストに参加してくださった東京工業大学、佐々・脇田研究室の皆様、および、有益なコメントをいただいた査読者の方々に感謝いたします。

参考文献

- 1) Appel, A. W.: *Modern Compiler Implementation in Java*, Cambridge University Press (1998).
- 2) Auguston, M.: FORMAN – A Program Formal Annotation Language, *Proceedings of the 5th Israel Conference on Computer Systems and Software Engineering*, IEEE Computer Society Press, pp. 149–154 (1991).
- 3) Drabent, W., Nadjm-Tehrani, S. and Maluszynski, J.: The use of assertions in algorithmic debugging, *Proceedings of the International Conference of fifth generation computer systems*, pp. 573–581 (1988).
- 4) Fritzson, P., Auguston, M. and Shahmehri, N.: Using Assertions in Declarative and Operational Models for Automated Debugging, *Journal of Systems and Software*, No. 25, pp. 223–239 (1994).
- 5) Horwitz, S., Reps, T. and Binkley, D.: Interprocedural slicing using dependence, *ACM Trans. Prog. Lang. Syst.*, Vol. 12, No. 1, pp. 26–60 (1990).
- 6) Ikezoe, Y., Sasaki, A., Ohshima, Y., Wakita, K. and Sassa, M.: Systematic debugging of Attribute Grammars, *Proceedings of the Fourth International Workshop on Automated Debugging AADEBUG2000*, pp. 235–240 (2000).

- 7) Korel, B. and Laski, J.: Dynamic Slicing of Computer Programs, *System Software*, Vol. 13, pp. 187–195 (1990).
- 8) Mayoh, B.H.: Attribute Grammars and Mathematical Semantics, *SIAM Journal on Computing*, Vol. 10, No. 3, pp. 503–518 (1981).
- 9) Naish, L.: A Declarative Debugging Scheme, *Journal of Functional and Logic Programming*, Vol. 1997, No. 3 (1997).
- 10) Sassa, M.: Rie and Jun: Towards the Generation of all Compiler Phases, *Proceedings of the 3rd International Workshop on Compiler Compilers, LNCS 477*, Springer-Verlag, pp. 56–70 (1991).
- 11) Sassa, M. and Ookubo, T.: Systematic debugging method for attribute grammar description, *Information Processing Letters*, Vol. 62, pp. 305–313 (1997).
- 12) Shahmehri, N.: *Generalized algorithmic debugging*, PhD Thesis, Linköping University (1991).
- 13) Shapiro, E. Y.: *Algorithmic Program Debugging*, The MIT Press (1982).
- 14) Shimomura, T.: Critical Slice-Based Fault Localization for Any Type of Error, *IEICE Transactions on Information and Systems*, Vol. E76-D, No. 6, pp. 656–667 (1993).
- 15) Weiser, M.: Program slicing, *IEEE Transaction on Software Engeneering*, Vol. 10, No. 4, pp. 352–357 (1984).

(平成 13 年 7 月 09 日受付)

(平成 13 年 10 月 12 日採録)



佐々木 晃

1972 年生。1994 年東京工業大学理学部情報科学科卒業。1996 年同大学大学院博士前期課程修了。2001 年同大学大学院博士後期課程満期退学。現在同大学大学院研究生。プログラミング言語、コンパイラの実装、コンパイラ生成系などに興味を持つ。日本ソフトウェア科学会会員



池添 洋平

1976 年生。2000 年東京工業大学理学部情報科学科卒業。現在同大学大学院修士課程に在学中。プログラミング言語、プログラミング環境などに興味を持つ



佐々 政孝 (正会員)

1948 年生。1970 年東京大学理学部物理学科卒業。1974 年同大学院博士課程中退、東京工業大学理学部情報科学科助手。1981 年筑波大学電子・情報工学系。1992 年東京工業大学理学部。現在同大学情報理工学研究科数理・計算科学専攻教授。理学博士。プログラミング言語、コンパイラ生成系、属性文法、プログラミング環境に興味を持つ。著書「プログラミング言語処理系」(岩波書店)。日本ソフトウェア科学会、ACM、IEEE 各会員。