

属性文法の系統的デバッグ法におけるバグ絞り込みの効率化

池添 洋平 佐々木 晃 脇田 建 佐々 政孝

本論文では属性文法に対する新しいデバッグ手法を提案する。これまでに、デバッガがユーザに対して属性の値が正しいかどうかを問い合わせることでバグの位置を特定する、属性文法の系統的デバッグ手法がいくつか提案されてきた。また、系統的デバッグ手法を理論的に一般化し、これを定式化することにより、複数のデバッグ法を統一的に扱うことができるようになった。しかしこれまで、デバッグの効率に関する考察はあまり行われていない。これまでのデバッグ法では、デバッガの質問の選び方によっては、質問の回数が増えるなど、デバッグの効率が悪くなってしまう場合が多かった。特に、実際に属性文法記述をデバッグする場合には、属性の依存関係が複雑になり、かつ、属性の数も多くなるため、デバッグの効率は重要な問題となる。本論文ではこのような問題点を解決するために、系統的デバッグ法の一般化の理論を拡張する。この拡張は、属性の依存関係の逆支配関係を利用し、属性の誤りの影響範囲を詳しく調べることによって、より厳密なバグの絞り込みを可能にするものである。さらに、この拡張された理論を利用したデバッグのアルゴリズムを提案する。このアルゴリズムを利用

することによって、デバッガのユーザに対する質問の回数を減らすことができ、ユーザへの負担を軽減し、デバッグの効率を向上させることができる。また、このデバッグ法を取り入れたデバッガを実装し、このデバッグ法の有用性を確かめた。

1 はじめに

属性文法は記述の形式性と簡潔さ、生成系の利用が容易であることなどの理由から、コンパイラの定式化など様々な分野で用いられている。しかし、属性文法記述をデバッグする場合、構文の再帰性や属性の複雑な依存関係など、属性文法特有の困難さをともなう。

これに対して、アルゴリズムック・デバッグング[9]や、スライス分割を用いた手法[10]などの系統的デバッグ法が属性文法のデバッグ法として適用されてきた[3][8]。属性文法に対する系統的デバッグ法^{†1}とは、複数の属性値の間の関係が正しいかどうかをユーザに問い合わせることで、プログラム中のバグの位置を特定する方法である。このような手法を利用することでユーザは、デバッガの質問に答えることによってバグを特定できるため、複雑な依存関係に煩わされることなくデバッグすることができる。さらに、上で述べたいくつかの方法を相補的に利用したり、ユーザにとって答えやすい形に変形して質問できるよう拡張するために、著者らのグループでは、系統的デバッグ法におけるバグの絞り込みのプロセスを一般化した[14]。この一般化は系統的デバッグ法におけるユー

Improving the Efficiency of Bug Localization in Systematic Debugging Method for Attribute Grammars
Yohei Ikezoe, ソニー(株), SONY Co.

Akira Sasaki, Ken Wakita, Masataka Sassa, 東京工業大学 情報理工学研究所 数理・計算科学専攻, Tokyo Institute of Technology

コンピュータソフトウェア, Vol.20, No.2(2003), pp.??-??.

[論文]2002年4月24日受付。

^{†1} 以下属性文法に対する系統的デバッグ法のことを単に系統的デバッグ法と記す場合がある。

ザへの質問の選び方や、バグの絞り込みの方法に関して抽象的な枠組みを定義したものである。

しかし、これまでの系統的デバッグ法は、デバッグの効率についてはあまり考慮されていない。これまでの方法では、質問の選び方によってはユーザへの質問の回数が増えたり、質問がユーザにとって答えにくくなるなど、デバッグが非効率になってしまう場合が多かった。特に、プログラミング言語の意味などを属性文法で記述する場合、属性の依存関係が複雑なものとなるため、デバッグはさらに困難な作業となる。

本論文では、これまでのデバッグ法の問題点について考察し、その解決法として系統的デバッグ法の一般化の枠組み[14]を拡張する。本論文で行う拡張は、属性の誤りの影響範囲をより詳しく調べることによって、これまでよりも厳密にバグの絞り込みを行えるようにするものである。これによって、これまでの手法よりも少ない質問の回数によってバグの位置を特定することができる。このような厳密な絞り込みを実現するために、属性の依存関係の逆支配関係を利用する。

また、この拡張された枠組みを利用したデバッグのアルゴリズムを提案する。このアルゴリズムは、ユーザの答えによってどの程度バグが絞り込めるかを、全ての質問の候補に対してあらかじめ見積もっておき、少ない質問の回数でバグを特定できるよう質問の位置を選ぶというものである。また、これらのデバッグ法を取り入れたデバッガを実装し、この方法が実際の属性文法記述をデバッグする時に有用であることを確かめる。

これまでに提案された属性文法の系統的デバッグ法と本論文で提案するデバッグ法との関係を図1に示す。灰色の四角が系統的デバッグのアルゴリズムを表し、白色の四角がそれらを一般化するデバッグの枠組みを表している。系統的デバッグのアルゴリズムとそれらの一般化の枠組みの関係を矢印で示す。点線で囲まれた部分が本論文で新しく提案する部分である。本論文では、より少ない質問の回数でバグの位置を特定するためにこれまでの枠組みを拡張し、この枠組みを利用した新しいデバッグのアルゴリズムを提案する。

以下、本論文の構成は次のとおりである。まず、2節で属性文法について概説し、3節において、これま

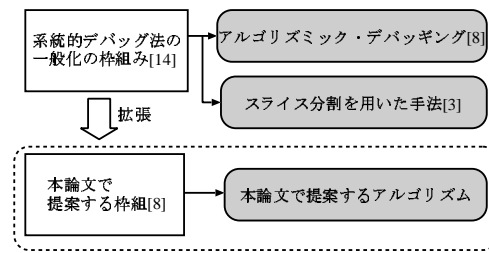


図1 本論文で提案する手法の位置付け

で提案された属性文法における系統的デバッグ法である、アルゴリズムック・デバッグやスライス分割を用いた方法および、これらの一般化の枠組みについて説明する。次に、4節でこれまでのデバッグ法の問題点について考察し、この問題の解決法として一般化の枠組みの拡張とそれを利用したデバッグ・アルゴリズムを5節において提案する。6節で、これらの系統的デバッグ法を取り入れたデバッガAkiを紹介する。さらに7節において、デバッグの効率に関する実験とその結果に関する議論を行い、8節で本方式についての考察を行う。

2 属性文法

属性文法は、言語の構文と意味を統一的に扱う記述法である。本節では、簡単な例を用いて属性文法について概説する。

図2は2進小数表記からその値を計算する属性文法記述の例である。図の「::=」を含む行と、「|」で始まる行は文脈自由文法の生成規則である。それぞれの生成規則の下にある「{」と「}」で囲まれた部分は、その生成規則に付随する意味規則である。この意味規則に従って構文の意味が定義される。

与えられた文字列に対する構文解析と、その意味の計算は次のように行われる。まず、文字列が入力として与えられると、生成規則に従って解析木を生成する。その後、意味規則に従って属性の値が計算される。属性は、解析木の各ノードに付随し、その値が解析木の意味を決定する。属性の値を計算することを属性評価という。

ここでは例として、文字列「.011」を上記属性文法記述に従って属性評価を行った場合について説明す

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

図 2 属性文法の例 (2 進小数の値の計算)

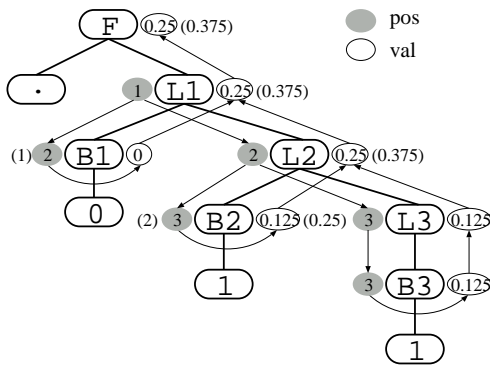


図 3 属性付き構文解析木

る．この文字列に対して構文解析を行い，属性評価を行うと図 3 のようになる．解析木のノードのラベルは「L」のような生成規則の非終端記号であるが，ここではそれぞれのノードを区別するために「L1」などのように番号の付いた非終端記号をラベルとする．この例にはバグを含む (正しい意味規則では，図 2 の (bug) で示された行の「+ 1」は必要ない) ため，属性の値が正しく評価されない．正しい属性の値は図 3 の「(」と「)」で囲まれた値で示す．解析木のノードが持つ属性 *pos* と *val* は小数点以下の桁数と，計算された小数の値をそれぞれ表し，これらの値は，それぞれのノードの導出のために適用された生成規則に

付随する意味規則に従って計算される．この例では，属性 *F.val* (ノード *F* に付随する属性 *val*) が最終的に求めたい属性の値である．この値を計算するためには *L1.val* が必要であることが *F.val* を計算する意味規則 (a) からわかる．さらに，*L1.val* を求めるためには *B1.val* と *L2.val* が必要であることが *L1.val* を計算する意味規則からわかる．このように，それぞれの属性の値は依存関係を持ち，その依存関係に従い属性の値が順番に計算される．図 3 において属性間の依存関係が矢印で示されている．

この例における属性 *pos* のように解析木の親のノード又は，兄弟のノードの属性に依存する属性を継承属性といい，*val* のように子のノードの属性に依存する属性を合成属性という．

3 属性文法の系統的デバッグ法

本節では，これまでに提案された属性文法の系統的デバッグ法について説明する．バグとは，属性評価時に誤った値の計算を行う意味規則のことである．このようなバグによってユーザの意図とは異なった属性値が計算され，この誤りが伝播され，最終的に誤った計算結果がバグの徴候としてユーザの目に触れる．ユーザがこのようなバグの徴候から誤りの原因となる部分，つまり，バグの位置を特定することをデバッグという．系統的デバッグ法においても，最終的な計算結果が誤っている場合にデバッグを行うことを仮定する．

系統的デバッグ法では実行時の情報を利用して，部分的な計算が正しいかどうかをユーザに問い合わせることによってバグを含む部分を狭めてゆく．このようなデバッグ法を利用することによって，ユーザは複雑な属性の依存関係を意識せずに，デバッグの質問に答えることによってバグの位置を特定することができる．本節ではこのような系統的デバッグ法として，アルゴリズムック・デバッグとスライス分割を用いた方法について説明し，さらにこれらの系統的デバッグ法の一般化の枠組みについて説明する．

3.1 アルゴリズムック・デバッキング

アルゴリズムック・デバッキング[9]は、論理型言語のデバッグ手法として提案され、関数型言語や手続き型言語にも応用されている[6][2]。アルゴリズムック・デバッキングでは、計算の過程を計算木と呼ばれる木構造で表現し、その上でバグの範囲を絞り込むことによってデバッグを行う。計算木とは、論理型言語における証明木や、関数型言語における呼び出し木のように、プログラムの再帰的な実行をトレースしたものである。デバッガはバグの範囲を狭めるために、述語の結果、あるいは、関数の計算結果が合っているかどうかをユーザに問い合わせ、その答えからバグを含む範囲を狭め、最終的にバグを含む述語もしくは、関数の位置を特定する。

アルゴリズムック・デバッキングを属性文法に適用するためには、属性評価が持つ再帰的な構造を利用して、計算木を作らなければならない。そこで、佐々らは、Synth 関数[5]と呼ばれる関数をアルゴリズムック・デバッキングに利用した[8]。Synth 関数は、継承属性と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。Synth 関数を再帰的に呼出すことによって、全ての属性の計算を表現できるため、この関数の呼び出し関係を計算木の構築に利用することができる。2節の属性評価の例に対して計算木を作ると、図4のようになる。この例では、属性 $F.val$ の値を計算するために Synth 関数 $f_{F.val}$ を呼び出す。このとき、引数として図3の解析木のノード F を根とする部分木 $tree_F$ をこの関数に与える。この関数の中では、 $F.val$ の値を計算するために $L1.val$ の値が必要になるので、この属性値を計算する Synth 関数 $f_{L1.val}$ を呼び出す。このようにして、属性の依存関係に従って相互再帰的に合成属性を計算する Synth 関数が呼び出され、最終的に $F.val$ の値を計算することができる。このような Synth 関数の呼び出し関係から計算木を作ることができる。

アルゴリズムック・デバッキングを開始すると、デバッガ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質問する。例えば、図4のノード(a)が質問の対象として選ばれた場合、(a)に対応する Synth 関数の振舞い、

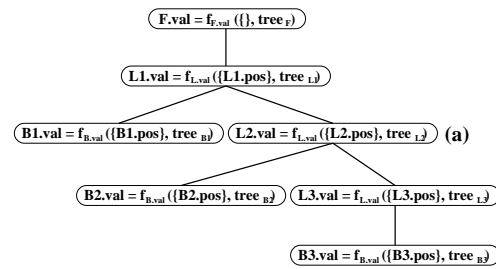


図4 計算木の例

つまり、 $L2.pos$ の値 2 と解析木ノード $L2$ 以下の部分木から計算された結果が $L2.val$ の値 0.25 であることが正しいかどうか（ユーザの意図と一致しているかどうか）をユーザに問い合わせる。このように、ユーザがデバッガの質問に答える際には、属性の値の他に、質問の対象となっている部分に関係する解析木も考慮に入れなければならない。これは、属性評価が解析木の形に依存するためであり、アルゴリズムック・デバッキングに限らず、系統的デバッグ法においてデバッガの質問に答える際には、質問の対象となっている属性評価の一部分において、その計算が行われた解析木を考慮しなければならない。

このような質問に対してユーザが正しいと判断した場合、質問を行なったノードより下にある Synth 関数の呼び出し以外の部分にバグがある^{†2}として、そのノード以下の部分木をバグの候補から除外する。ユーザが誤っていると判断した場合、そのノードより下の Synth 関数にバグがあるとして、それに対応する部分木をバグの候補としてデバッグを進める。デバッガはこれを繰り返すことによって、バグの候補を一つのノードに絞り込むことができ、そのノードを表す Synth 関数に対応する意味規則にバグがあることを指摘することができる。

なお、記述に複数個の誤りを含んでいる場合、アルゴリズムック・デバッキングを含む系統的デバッグ法を利用することによって、少なくともそのうちの1つを特定することができる。この場合、見つかったバ

^{†2} この場合、Synth 関数の呼び出し部分にもバグを含む可能性もあるが、アルゴリズムック・デバッキングでは Synth 関数の呼び出し以外の部分から少なくとも1つのバグを特定することができる。以降の系統的デバッグ法についても同様のことがいえる。

グを修正しても計算結果が誤っている可能性があるが、系統的デバッグとバグの修正を繰り返し行うことによって、最終的に全てのバグを同定することができる。

3.2 スライス分割を用いたデバッグ法

プログラム・スライシング[11][13]とは、プログラム中の要素に影響を及ぼす計算を抽出する技術である。プログラム・スライシングはソフトウェア工学など様々な分野に応用されており、デバッグの分野でも、誤った値に影響を及ぼした計算を抽出するために利用することができ、有効であることが示されている[1][12]。我々はこれまでに、スライス分割を利用したデバッグ法[10]を属性文法に応用した[3]。本節では属性文法上での、スライス分割を利用したデバッグ法について説明する。

あるプログラムの要素(変数や文など)に対するスライス^{†3}とは、その要素からプログラム中に存在する依存関係を逆向きにたどった時に得られるプログラムの要素である。文献[10]では、変数の誤りに影響を与える可能性のある全ての要素について依存関係を持つものとし、そのような依存関係を辿ることによってクリティカルスライスというものをつくり、その上でデバッグを行う。属性文法で値の誤りを伝播する可能性のあるものは、属性間のデータ依存関係のみである。したがって、属性文法上でのクリティカルスライス^{†4}は属性の依存関係を逆向きにたどったものと考えることができる。図5は2節の例において、*F.val*に対して計算されたスライスである。*F.val*は全ての属性に直接または、間接的に依存するため、全ての属性が抽出される。この図は、図3において属性と、その依存関係を抜き出したものと一致する。

この例に対してデバッグを行う手順は次のとおりである。まず、ユーザが *F.val* がおかしいことに気付くことによってデバッグを開始する。デバッガは、スライスを適当な二つの部分に分割し(例えば、図5の点

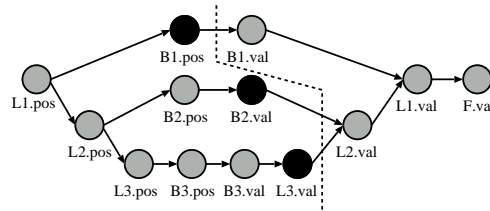


図5 属性文法上におけるスライスの例

線)、分割の手前で定義され、分割の後で使用されている全ての属性(図5の *B1.pos*, *B2.val*, *L3.val*)が正しいかどうかをユーザに問い合わせる。ユーザはその値を計算するまでに関係した解析木を見ることによって、属性の値が正しいかどうかを判断する。全ての属性が正しい場合、分割より後の属性にバグを含むとして分割以降の属性の依存関係に対してデバッグを行う。誤っている属性があった場合、その属性に対するスライスの中にバグを含むとして、そのスライスに対してデバッグを行う。これを繰り返すことによって、バグの位置を絞り込むことができる。

3.3 系統的デバッグ法の一般化

前節までに説明した、アルゴリズムック・デバッグやスライス分割を用いた手法は共に、属性評価の一部分に注目し、その部分が正しいかどうかをユーザに対して問い合わせるというものであった。文献[14]では、二つの方法を一つの枠組みの中で扱えるよう、分割と質問を繰り返すデバッグ法を一般化し、定式化を行った。この定式化は、属性評価の任意の一部分を取り出し、その部分での属性評価が正しいかどうかをユーザに問い合わせ、その答えからバグの位置を絞り込むというものである。これによって、前節まで述べたデバッグ法は、この手法において質問の選び方を制限した特殊なケースと見なすことができ、例えば、アルゴリズムック・デバッグの途中からスライス分割を用いた方法へ移行するというような、二つの方法を協調させてデバッグを行うということが可能になる。また、計算木などの特殊なデータ構造に依存せず、より自由な位置でユーザへの質問を行うことができる。

本節では、系統的デバッグ法の定式化について概説

†3 プログラム・スライシングには、前向き、後向きや、静的、動的の違いがあるが、本論文では動的後向きスライス[4]のみを扱う。

†4 以後ではクリティカルスライスのことを単にスライスと呼ぶ。

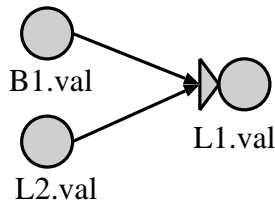


図 6 属性計算の例

する．詳しくは文献[14]を参照されたい．まず，属性の計算に関する基本的な定義について述べる．

定義 1 ある属性 o が，直接依存している属性の集合を $depend(o)$ と記述する．

定義 2 属性 o を，直接依存している属性の集合 $depend(o)$ を使って計算する動作を属性計算といい， $\langle depend(o) \rightarrow o \rangle$ と表記する．

$\langle depend(o) \rightarrow o \rangle$ は省略して，属性 o の計算ともいう．

例 1 2 節の例で， $L1.val$ において，

$$depend(L1.val) = \{B1.val, L2.val\}$$

となる．また， $L1.val$ の計算を

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

と記述する．以後，属性の値を計算する様子を図示する場合，図 6 のように表す．図中の丸が属性を，三角が属性の計算を，矢印が依存関係をそれぞれ表す．

次に，複数の属性を依存関係に従って計算をするという動作について定義する．

定義 3 評価された全ての属性の集合を AS とする．属性の集合 $S \subset AS$ に対して計算の入力 I と出力 O を次のように定める．

$$I = \bigcup_{a \in S} depend(a) - S$$

$$O = \bigcup_{a \in AS - S} depend(a) \cap S$$

I から O に至るまでの属性計算をまとめたものを S 上の属性計算合成といい， $\langle I \Rightarrow O \rangle_S$ と表記する^{†5}．

^{†5} この定義では，2 節の例の $F.val$ のような，属性評

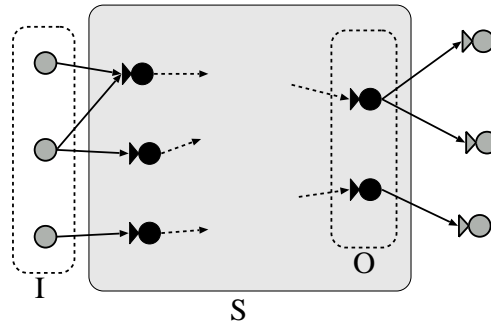


図 7 属性計算合成

このように，入力と出力を決めることによって，複数の属性を計算するステップを，入力 I を利用して出力 O を計算する一つの計算としてまとめることができる．

例 2 図 7 のような属性の依存関係において，実線で囲まれた属性が S に含まれているとする．入力 I と出力 O はそれぞれ図中の点線で囲まれた部分になる．このように，入力 I の要素は S には含まれず， S に含まれている属性の計算に使用されている．また，出力 O の要素は S に含まれており， S に含まれない属性の計算に使用されている．

属性の集合 S 上の属性計算合成 $\langle I \Rightarrow O \rangle_S$ において， S 以外の属性から依存関係を前向きに辿って S の属性に到達するには I の属性の一つを通過しなければならない．また， S の属性から S 以外の属性に到達するには O に含まれる属性の一つを通過しなければならない．この性質を $\langle I \Rightarrow O \rangle_S$ は閉じているといい，定義 3 によって構成される属性計算合成は常にこの性質が成り立つ．この性質は， S に含まれる属性の値が入力を介さずに他の属性の値から影響を受けず，かつ出力を介さずに他の属性の値に影響を与えないことを保証する．この性質によって，デバッガはユーザから得られる属性計算合成の正誤の情報を利用して，属性計算合成の内外どちらにバグを含むか

価において最後に計算される属性の値を O に含めることはできない．そこで属性計算合成を計算する場合に，属性評価の終りを表す補助的な属性が存在するものとし，この補助的な属性は最後に計算される属性に依存するものと仮定する．

を判断することができる。

次に、属性計算合成の正しさについて定義する。

定義 4 属性計算合成 $\langle I \Rightarrow O \rangle_S$ において、属性の集合 I の値を前提として O に含まれる全ての属性の値がユーザが予期した値と一致するとき、 $\langle I \Rightarrow O \rangle_S$ の挙動が正しいといい、一つでも一致しない属性があるとき $\langle I \Rightarrow O \rangle_S$ の挙動は誤っているという。挙動の正誤をそれぞれ

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

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

と記述する。

このときユーザは、属性計算合成 $\langle I \Rightarrow O \rangle_S$ の挙動が正しいかどうかを判断するために、解析木の部分木を考慮に入れなければならない。 $\langle I \Rightarrow O \rangle_S$ が表す計算はこの部分木の上で行われたものであり、 S に含まれる全ての属性は部分木のいずれかのノードに付随するものである。

さらに、属性計算合成の挙動の正誤からバグの位置について次のことがいえる。

定理 1 S と S' はともに属性の集合で $S' \subset S$ を満たすとする。 S, S' 上の属性計算合成をそれぞれ $\langle I \Rightarrow O \rangle_S, \langle I' \Rightarrow O' \rangle_{S'}$ とする。 $\text{Query}(\langle I \Rightarrow O \rangle_S) = \text{incorrect}$ であるとき次のことがいえる。

- $\text{Query}(\langle I' \Rightarrow O' \rangle_{S'}) = \text{correct}$ であるとき、 $S - S'$ に含まれる属性の計算にバグを含む
- $\text{Query}(\langle I' \Rightarrow O' \rangle_{S'}) = \text{incorrect}$ であるとき、 S' に含まれる属性の計算にバグを含む

証明は文献[14]を参照されたい。

以上の性質から、任意の属性の集合を取り出し、その集合上の属性計算合成の入力と出力の関係が正しいかどうかをユーザに問い合わせることによって、バグの範囲を狭めることができる。

上で述べた属性文法の系統的デバッグ法の一般化の枠組みの観点から次のようなことが言える。

1. アルゴリズムック・デバッグは、属性計算合成の入力と出力が解析木と同じノードの属性になるように属性計算合成を構成する特殊なケースである。
2. スライス分割を利用する方法は、入力が必ず空

であるように属性計算合成を構成する特殊なケースである。

例 3 アルゴリズムック・デバッグを本節で述べた一般化の枠組みを利用して説明する。

2 節の 2 進小数の例において、

$$S = \{B2.pos, B2.val, L3.pos, B3.pos, B3.val, L3.val, L2.val\}$$

とすると、入力 I と出力 O はそれぞれ、 $I = \{L2.pos\}$ 、 $O = \{L2.val\}$ となり、 S 上の属性計算合成 $\langle I \Rightarrow O \rangle_S$ を構成することができる。図 8 において黒色で示された属性が S に含まれ、点線で囲まれた属性がそれぞれ I と O に含まれる。この属性計算合成は、アルゴリズムック・デバッグにおける、 $L2.val$ を計算する Synth 関数と同等の働きをする。

デバッガは I と O の値の関係がユーザが予期していた値と一致するかどうかを問い合わせる。一致する場合、 $\text{Query}(\langle I \Rightarrow O \rangle_S) = \text{correct}$ であるから定理 1 より、 S 以外の属性にバグを含む。したがって、次の質問のための属性計算合成を構成する属性の集合は、 S の要素を含まないように選ぶか (例えば、図 8 の (b))、 S の要素を全て含めるように選ぶ (図 8 の (a))。ユーザの意図した値と一致しない場合は、 $\text{Query}(\langle I \Rightarrow O \rangle_S) = \text{incorrect}$ であるから、 S にバグを含む。したがって次の質問で利用する属性計算合成を構成する属性は、 S の部分集合から作る (例えば、図 8 の (c))。これは文献[8]で述べられているアルゴリズムと一致する。

本節で述べた一般化の枠組みを利用することによって、従来の方法に比べてより自由な位置でユーザに質問することができる。しかし、質問の選び方によっては、デバッグの作業が非効率になってしまう可能性があるため、デバッガはユーザに対する質問を注意深く選ぶ必要がある。

4 系統的デバッグ法の問題点

3 節で紹介した系統的デバッグ法によって、これまではプログラマが考えなければならなかった、属性の依存関係などをデバッガが自動的に解析するため、デバッグ時のユーザの負担が大きく軽減される。また、

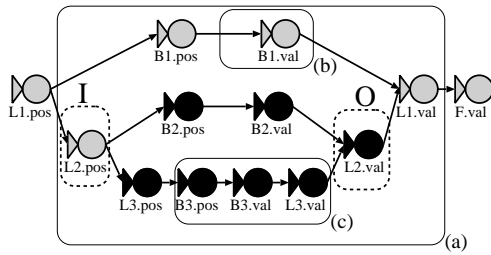


図 8 属性計算合成の例

デバッグ法の一般化の枠組みによってデバッガは自由な位置でユーザに質問を行うことができるようになった。しかしこれまで、デバッグ時の質問の選び方に起因する、デバッグの効率についてはあまり検討されなかった。

デバッグの効率は、バグを特定するまでに行われる質問の選び方によって変化する場合が多い。これまでに述べた系統的デバッグ法のアルゴリズムでは、質問の場所はデバッガが自動的に決めるため、質問の数が増えたり、質問の内容がユーザにとって難しいものとなったりするため、デバッグの作業の効率が悪くなってしまう場合がある。特に属性文法記述において、属性の数が多くなり依存関係が複雑になる場合、デバッグの作業はさらに困難なものとなる。このため系統的デバッグ法において、デバッガのユーザに対する質問の選び方を工夫することが重要になる。

本節では質問の選び方に起因する問題点を二つ明らかにし、解決するための方針を述べる。また次節において、系統的デバッグ法の一般化の枠組みの拡張を行うことによって、デバッグの効率の向上を試みる。

4.1 質問の難しさ

一つめの問題点は、質問の難しさに関するものである。デバッグの効率を考える場合、なるべくユーザにとって簡単な質問によってバグの位置を特定することができれば、デバッグの効率はよいといえる。デバッガのユーザへの質問が難しいかどうかは、ユーザの主観に左右されるものなので、難しさを判断することは簡単ではない。しかし一般的に、質問の対象となる属性計算合成に含まれる属性の数が多いとき、質問が難しくなる。

例えば、図 9 のような依存関係において、スライス分割を用いた方法でデバッグを行う場合について考える。あるスライスの分割に対応する属性計算合成は図の (a) のようになる。つまり、一般化の枠組みの観点では、分割より手前にある全ての属性によって属性計算合成が構成される。スライス分割を用いた方法では、(a) に対応する質問が正しい場合、この分割より後ろの属性の依存関係に対して依存関係を分割し質問を行う。3.3 節で述べたように、スライス分割を用いたデバッグ法は、前提を持たないようにユーザへ質問を行うデバッグ法であった。つまり、デバッガは入力为空であるような属性計算合成を選ばなければならない。そのため次の質問のための属性計算合成は、例えば図 9 の (b) のように、(a) を含む形で選ばなければならない。このような質問は、すでに正しいと判断された属性計算合成 (a) を含んでおり、(b) の正しさを考えるときに、(a) に含まれる属性計算も (b) の属性計算合成の一部として考慮に入れなければならない。

また、例 3 でも述べたように、アルゴリズムック・デバッグでも、ある属性計算合成の挙動が正しいと分かった場合、その属性計算合成を構成する属性を包含するような属性計算合成を用いて、ユーザに質問を行う場合がある。

図 9 の (b) のような「すでにバグの候補から除外されている属性」を含めた広い範囲での質問は、正しいかどうかを判断するときに考慮すべき範囲が広くなり、ユーザにとっては難しい質問となる。このような質問は、大きな解析木上でデバッグを行う場合、特に難しいものとなる。したがってデバッグの効率を考える場合、デバッガは局所的な範囲で質問を行うことによってバグが特定できるよう、質問の選び方を工夫する必要がある。

4.2 質問の回数

二つめの問題点は、バグを特定するまでにユーザに問い合わせる質問の回数に関するものである。系統的デバッグ法では、属性計算合成の入力に対して出力が正しいかどうかをユーザに問い合わせることによってデバッグを進める。このとき、出力に含まれる属性が複数ある場合、出力の一つ一つについて正しいかど

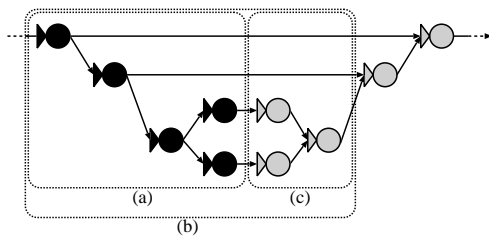


図9 質問が大きくなる例

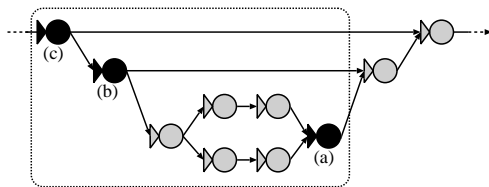


図10 質問の回数が多くなる例

うかを問い合わせなければならない。出力の数の多い属性計算合成を質問の対象として選ぶと、質問の回数が多くなり、デバッグの効率が悪くなる。

例えば、図10のような依存関係においてスライス分割を用いたデバッグ法を行う場合、図中の属性(a)の直後に注目して分割すると、この分割に対応する属性計算合成は図の点線で囲まれた部分になる。このときの出力は(a)の他に(b)と(c)を含み、それぞれに対して正しいかどうか質問しなければならない。このように、質問の対象となる属性計算合成の選び方によっては質問の回数が増えてしまい、デバッグが非効率になってしまう場合がある。実際に系統的デバッグを行う際にもこのような質問は多く、そのために質問の回数が増えてしまう場合が多い。

4.3 解決策の方針

4.1節で述べたように、質問の範囲が大きくなると、ユーザへの質問は難しいものとなる。このような難しさを軽減するためには、なるべく広い範囲を含む質問を行わずに、バグの位置を特定するのが良い。例えば、「この解析木のノードより下がおかしいようだ」というようなユーザの指摘を受け入れることによって、より小さい範囲でのデバッグが可能になる。このようなユーザの指摘によって、より狭い範囲で系統的デバッグのアルゴリズムを適用することによって、質

問の難しさを軽減することができる。

また、図9の(b)のような質問を簡単にするため、図の(c)のように、属性計算合成の入力の位置を動かすことによって、質問の範囲を狭めることができる。(c)のような質問は、(a)の出力を入力として持つことによって、(a)の質問の結果を(c)の前提として考えることができるため、(b)では含まれていた(a)の計算過程の部分を考えずに質問に答えることができる。

このような質問の難しさに関する問題の解決法は、3.3節で述べた系統的デバッグ法の一般化の枠組みを利用し、バグを含む範囲を効果的に管理することによって実現することができる。また、アルゴリズムック・デバッグにおいて、質問の範囲が大きいことに起因する質問の難しさを軽減する方法が文献[14]で述べられている。これらの方法は本研究で実装したデ

バッグにも適用されており、実際に利用可能である。4.2節で述べた質問の回数に関する問題は、属性計算合成の複数の出力に対して質問を行わなければ、バグを絞り込むことができないということに起因する問題であった。この問題点を解決するために、単一の属性に対する質問によってバグを絞り込むことのできるデバッグ手法を、次節で提案する。このデバッグ法を利用することによって、より少ない質問の回数でバグを特定することができる。

5 デバッグ法の改良

系統的デバッグ法において、より少ない質問の回数によってバグを特定することができれば、デバッグの効率は良いといえる。しかし前節でも述べたように、これまでの方法では質問の回数が多くなってしまった場合が多かった。本節では、ユーザへの質問の回数を最小限に抑えるために、系統的デバッグ法の一般化の枠組みを拡張する。これまでの枠組みでは、属性計算合成 $(I \Rightarrow O)_S$ において、 O に含まれる全ての属性に対して正しいかどうかを考えなければならなかったため、複数回質問を行わなければバグの位置を絞り込むことができない場合があった。これに対して、本節で述べる枠組みでは O に含まれる属性のうち一つの属性にのみ注目し、この値の正誤からバグの範囲を絞り込むことができるため、少ない質問の回数でバグの特

定を行うことができる。これは、属性の依存関係の逆支配関係を用いて属性の依存関係をより正確に解析し、属性値の誤りの影響範囲を調べることによって実現することができる。

また、この拡張された枠組みを利用したデバッグのアルゴリズムを提案する。このアルゴリズムは、逆支配している属性の数から、ユーザの答えによってどの程度バグの範囲を絞り込めるかを見積もり、質問の位置を決める方法である。このアルゴリズムは、質問の内容はスライス分割を用いた方法とほぼ同じであり、少ない質問の回数でバグの位置を特定することができる。また、この方法はこれまでのスライス分割を用いた方法とは異なり、前提 I を持つことを許す。これによって、質問の対象となる範囲を狭くすることができるため、質問の難しさを軽減することができる。

5.1 属性の逆支配関係を用いたデバッグ法

本節では、挙動が誤っている属性計算合成内において、属性値の正誤に関する情報と、属性計算合成の挙動の誤りに影響を及ぼす可能性のある属性との関係について述べる。本説ではこのような関係を、属性の依存関係とその逆支配関係を用いることによって説明する。これを利用することによって、3.3 節で述べた一般化の枠組みよりも厳密にバグの範囲を絞り込むことが可能となる。

まず、ある一つの属性に注目した時に、その属性の計算と関わりのある属性の集合について定義する。

定義 5 ある属性計算合成 $\langle I \Rightarrow O \rangle_S$ に対して、ある属性 $a \in S$ から I まで依存関係を後向きに辿ったときに得られる属性の集合から、 I に含まれる属性を除いたものを $DEPEND_{\langle I \Rightarrow O \rangle_S}(a)$ と表す。

例 4 図 11 のような属性の依存関係において、実線で囲まれた属性は S に含まれるものとし、 S 上の属性計算合成 $\langle I \Rightarrow O \rangle_S$ を定める。 $DEPEND_{\langle I \Rightarrow O \rangle_S}(a)$ に含まれる属性を黒色で示す。

属性計算合成 $\langle I \Rightarrow O \rangle_S$ は閉じている。そのため、属性の依存関係を逆向きに辿って S に含まれない属性に到達するためには、 I に含まれる属性を通過しなければならない。したがって、 $DEPEND_{\langle I \Rightarrow O \rangle_S}(a) \subset S$

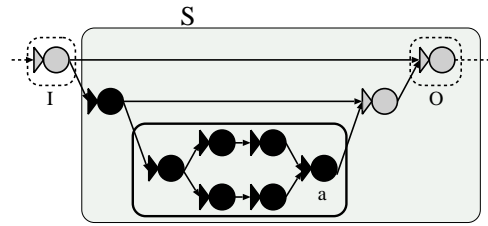


図 11 PDOM と DEPEND の例

が成り立つ。

定義 6 属性計算合成 $\langle I \Rightarrow O \rangle_S$ において、属性 $a \in S$ に対して $S' = DEPEND_{\langle I \Rightarrow O \rangle_S}(a)$ としたとき、 I を利用して a の値を計算する動作のことを $[I \Rightarrow a]_{S'}$ と表記し、 a に関する属性計算合成という。 $[I \Rightarrow a]_{S'}$ の正誤については定義 3 と同様に定義する。

S' に含まれる属性は依存関係を前向きに辿ることによって、 a 以外の属性を通して S' に含まれない属性に到達する可能性があるため、 $[I \Rightarrow a]_{S'}$ は出力に関して閉じていない可能性がある。 a に関する属性計算合成 $[I \Rightarrow a]_{S'}$ は、一般の属性計算合成の性質を必ずしも満たさないが、属性 a の計算結果にのみ注目した属性計算合成といえる。

定義 7 属性計算合成 $\langle I \Rightarrow O \rangle_S$ において、属性 $b \in S$ から属性の依存関係を前向きに辿り、 O に含まれる属性に到達するまでの全てのパスが必ず属性 a を含むとき、 a は b を逆支配するという。 a が逆支配する属性の集合を $PDOM_{\langle I \Rightarrow O \rangle_S}(a)$ と表記する。

例 5 図 11 において、太線で囲まれた属性が $PDOM_{\langle I \Rightarrow O \rangle_S}(a)$ に含まれる。

属性 a が直接または間接に依存している属性は、言い替えるとその属性を前向きに辿って O に到達するまでの全てのパスのうち、 a を含むパスがすくなくとも一つ存在する。したがって、 $PDOM_{\langle I \Rightarrow O \rangle_S}(a) \subset DEPEND_{\langle I \Rightarrow O \rangle_S}(a)$ となる。

このような単一の属性に関する、必ずしも閉じていない属性計算合成の挙動が正しいかどうかを知ることによって、バグの位置について次のことがいえる。
定理 2 属性計算合成 $\langle I \Rightarrow O \rangle_S$ は挙動が誤っている

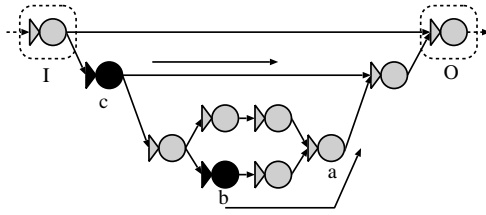


図 12 逆支配関係と属性値の流れ

ものとし, $a \in S$, $S' = \text{DEPEND}_{\langle I \Rightarrow O \rangle_S}(a)$ とする. a に関する属性計算合成 $[I \Rightarrow a]_{S'}$ の挙動の正誤から次のことが言える.

- $\text{Query}([I \Rightarrow a]_{S'}) = \text{correct}$ のとき, $S - \text{PDOM}_{\langle I \Rightarrow O \rangle_S}(a)$ にバグを含む.
- $\text{Query}([I \Rightarrow a]_{S'}) = \text{incorrect}$ のとき, S' にバグを含む.

[証明] (略証) $[I \Rightarrow a]_{S'}$ の挙動が正しい場合, 属性 a は $\langle I \Rightarrow O \rangle_S$ の挙動の誤りに影響を与えていない. a に逆支配されている属性 (例えば, 図 12 の b) はその値が何であろうと a を通さずに O に含まれる属性の値に影響を与えないので, O の誤りに影響を与えることはない. 逆に, a に逆支配されていない属性 (図 12 の c) は, O の値に a を介さずに影響を与える可能性がある. したがって, 属性 a に逆支配されていない属性の集合は O の誤りに影響を与えるバグを少なくとも一つ含む.

$[I \Rightarrow a]_{S'}$ の挙動が誤っている場合, I から a の計算の中に少なくとも一つのバグを含むのは明らかである (証明終)

3.3 節で述べた枠組みでは, 閉じた属性計算合成の全ての出力について入力との関係が正しいかどうかを問い合わせなければならなかった. これに対して本節で述べた枠組みは, 単一の属性に関する, 必ずしも閉じていない属性計算合成において, 一つの出力について入力との関係の正誤を確かめることで, バグの範囲を絞り込むことができるように拡張したものである.

5.2 デバッグ・アルゴリズム

前節で述べた枠組みを利用した, デバッグ・アルゴリズムを提案する. このアルゴリズムでは, デバッグ時の一連の質問において, 入力に常に一定の属性の集合となるように属性計算合成を選び, 出力となる属性を質問ごとに変えながら, バグを特定する. つまり, 質問の前提を一定に固定しておき, その属性とある属性の値の関係が正しいかどうかを問い合わせていくことによって, バグを特定する方法である. この方法において, 質問の前提を空集合に固定すると, 前提なしで属性の値が正しいかどうかを問い合わせることによってバグを特定する, スライス分割を用いた方法と同じ形式の質問になる. また本説で提案するアルゴリズムでは, 絞り込むことのできる属性数をあらかじめ見積っておき, これをもとに質問の位置を決めることによって, 質問の回数を少なく抑える. したがってこのアルゴリズムは, 分割を横切る全ての属性に対して質問を行わなければバグの位置を絞り込むことができなかつた, スライス分割を用いた方法を, 一つの質問によってバグを絞り込むことができるよう拡張したものである.

アルゴリズム 1

1. バグを含む属性の候補を集合 S とし, S 上の属性計算合成を $\langle I \Rightarrow O \rangle_S$ とする.
2. S の任意の要素 a を一つ選ぶ.
3. ユーザに, 属性の集合 I と選ばれた属性 a の関係が正しいかどうかを問い合わせ,
 - ユーザが正しいと答えた場合, $S = S - \text{PDOM}_{\langle I \Rightarrow O \rangle_S}(a)$ とする
 - ユーザが誤っていると答えた場合, $S = \text{DEPEND}_{\langle I \Rightarrow O \rangle_S}(a)$ とする
4. S の要素数が十分小さくなるまで 1 から 3 を繰り返す.

このアルゴリズムにおいて質問の対象となる属性 a を選ぶ際 (ステップ 2) に, ユーザの答えによってバグの候補となる属性の数を半分に減らすことができれば, 質問の回数を最小限に抑えることができる.

ユーザが属性 a に対する質問に「yes」と答えた場合, $\text{PDOM}_{\langle I \Rightarrow O \rangle_S}(a)$ に含まれない属性がバグ

の候補として残り、ユーザが「no」と答えた場合、 $DEPEND_{(I \Rightarrow O)_S}(a)$ に含まれる属性がバグの候補として残る。そこで、 $PDOM_{(I \Rightarrow O)_S}(a)$ に含まれる属性数が S に含まれる属性数の半数に近くなるような属性 a を質問の対象として選ぶと、ユーザが「yes」と答えた場合には、バグの候補を半分に絞り込むことができる。しかし、 $PDOM_{(I \Rightarrow O)_S}(a) \subset DEPEND_{(I \Rightarrow O)_S}(a)$ であるから、 $DEPEND_{(I \Rightarrow O)_S}(a)$ に含まれる属性数は、 $PDOM_{(I \Rightarrow O)_S}(a)$ に含まれる属性数に比べて多いため、「no」と答えた場合に多くの属性がバグの候補として残る可能性がある。しかし、一般に $PDOM_{(I \Rightarrow O)_S}(a)$ に含まれる属性数が総属性数の半数に近い場合、 $DEPEND_{(I \Rightarrow O)_S}(a)$ に含まれる属性数と $PDOM_{(I \Rightarrow O)_S}(a)$ に含まれる属性数が大きく異なることは少ない。

そこで、各属性が逆支配している属性の数を重みとして持たせ、その重みが現在の総属性数の半数に近く、かつ、半数以下であるものを質問の対象として選ぶようにする。

スライス分割を用いた方法では、出力に含まれる全ての属性が正しいことがわかった時のみバグを含む範囲を絞り込むことができたのに対して、このアルゴリズムではある一つの属性が正しいことが分かった場合にもバグの範囲を絞り込むことができる。また、各属性が逆支配している属性の数を重みとして持たせることで、ユーザの答えによってどの程度バグの範囲を絞り込めるかを見積っておき、質問の回数を少なく抑えることができる。

5.3 デバッグの例

2 節の 2 進小数の例に対してアルゴリズム 1 を用いてデバッグを行う。ユーザは計算結果の $F.val$ が間違えていることに気づき、デバッグを開始する。この場合、全ての属性がバグの候補となるので、

$$S = \{ F.val, L1.val, B1.val, B1.pos, L2.val, B2.val, B2.pos, L3.val, B3.val, B3.pos, L3.pos, L2.pos, L1.pos \}$$

となり、属性計算合成の入出力はそれぞれ $I = \{ \}$ 、 $O = \{ F.val \}$ となる。

まずデバッグは、属性の依存関係 (図 3 参照) から

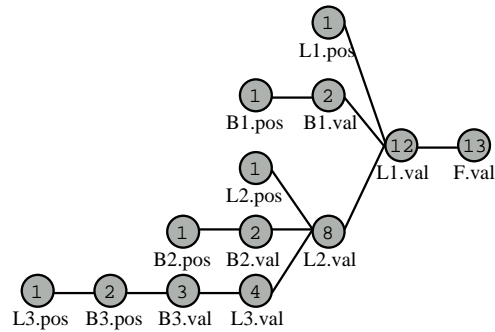


図 13 逆支配木

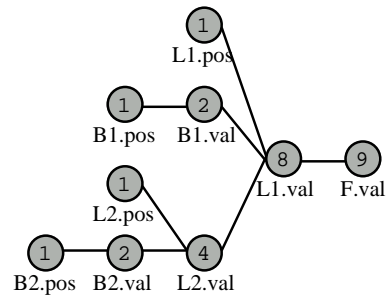


図 14 逆支配木 (バグ絞り込み後)

逆支配関係を計算し、逆支配木を作る (図 13)。このとき、各ノードに部分木が含むノードの数を重みとして付加しておく。図 13 中のノード上のラベルは重みを表している。 S に含まれる属性の個数は 13 であるから、逆支配木のノードの中から、13 の半数に近く、かつ、半数以下である、重みが 4 の属性 $L3.val$ が質問の対象として選ばれる。ここでは、 $I = \{ \}$ であるから単に、属性 $L3.val$ の値が正しいかどうかをユーザに対して質問する。 $L3.val$ の値は正しいのでユーザは yes と答える。このことから、 $L3.val$ が逆支配している属性以外にバグを含むことがわかるので、 $PDOM_{(I \Rightarrow O)_S}(L3.val)$ を S から除き、逆支配木の重みを計算しなおす (図 14)。

次に、重みが 4 である $L2.val$ が選ばれ、正しいかどうかをユーザに問い合わせる。 $L2.val$ の値は誤っているのでユーザは no と答える。これによって、新しい S は $DEPEND_{(I \Rightarrow O)_S}(L2.val) = \{ L2.val, B2.val, B2.pos, L2.pos, L1.pos \}$ となる。

このようにしてデバッグはユーザへの質問と、その

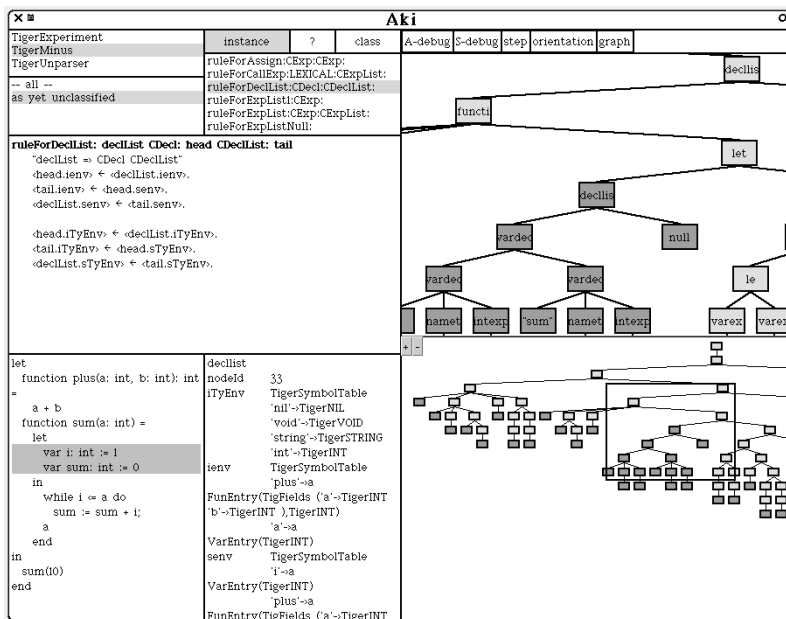


図 15 デバッガAki

答えによるバグの絞り込みを繰り返す．そして，最終的に $S = \{B2.pos\}$ となり， S に含まれる唯一の属性 $B2.pos$ の計算 $\langle \{L2.pos\} \rightarrow B2.pos \rangle$ にバグが含まれていることがわかる．デバッガはこの属性計算を行う際に適用された意味規則「 $B.pos = L0.pos + 1$ 」が誤っていることをユーザに示し，ユーザはこれを見て誤った意味規則を修正することができる．

6 デバッガAki

著者は新しいデバッグ法が有効であることを確かめるために，これまで述べたデバッグ法を取り入れたデバッガ，Aki [3] を実装した．Aki は，属性文法によるコンパイラ開発環境 Jun [7] に追加する形で実装されている．Jun はコンパイラの意味解析や最適化フェーズの動作を表した属性文法記述から，属性評価器を自動生成する生成系である．Aki を利用するために，属性評価時の属性の値をトレースとして残すように Jun を拡張した．Aki は実行時のトレース，属性評価が行われた解析木と，Jun が評価器を生成する際に作成した，静的な属性の依存関係を管理するテーブルを受け取り，属性の依存関係を解析しデバッグを開始する．

Aki の実行画面は図 15 のようになる．Aki では各サブウィンドウに属性文法記述，解析木，属性の値と，構文解析への入力として与えられた文字列が同時に表示され，注目しているノードに関する情報を容易に確かめることができる．Aki では系統的デバッグ法として，アルゴリズムック・デバッグング，スライス分割を用いた方法と，本論文で提案した逆支配関係を利用したデバッグ法を行うことができる．また，系統的デバッグ法の一般化手法を用いて，アルゴリズムック・デバッグングからスライス分割を用いた手法へ移行する方法や，アルゴリズムック・デバッグングの質問の難しさを軽減する方法 [14] も利用することができる．

以下では，2 節 で説明した 2 進小数を計算する属性文法記述に対して，Aki を用いてデバッグを行う様子を例に説明する．

まず，ユーザが解析木内のあるノードにおいて，属性同士の値の関係がおかしいと気付いた場合，そのノード上でマウスを右クリックし，ポップアップメニューからデバッグ法を選択することによって，デバッグを開始することができる．このように，Aki では 4.3 節 で述べたような，ユーザが部分木を指定することによる，より小さい範囲でデバッグが可能であ

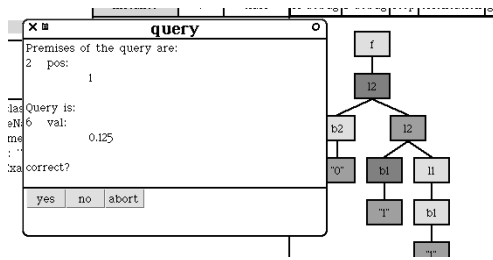


図 16 ユーザへの質問の例

る。デバッグを開始すると、デバッガは逆支配木を作り、ユーザへの質問を行う。図 16 は、Aki の質問の例である。この質問では、番号 2 のノードに付随する属性 *pos* (2 節の例の *L1.pos* に当たる) の値 1 に対して、番号 6 のノードの属性 *val* (*B2.val*) の値が 0.125 で正しいかをユーザに問い合わせている。このとき、質問の対象になっているノードは色分けされて表示され、ユーザは質問の対象となっている属性の値を詳しく調べることができる。このように、Aki ではユーザが質問に答えやすいように質問を行うよう工夫されている。デバッガは、ユーザから与えられた回答から、バグの範囲を絞り込むことができる。

7 実験

本論文で提案したデバッグ法の有用性を確かめるために、質問の回数に関する実験を行った。本節では、実験によって得られた結果をもとに、本論文で提案したデバッグ法の有用性について論じる。

実験の方法は次のとおりである。系統的デバッグ法のアルゴリズムは、一連の「yes」または「no」の答えから一つまたは複数の属性をバグとして絞り込むというものである。例えば、スライス分割を用いた方法において、デバッガの質問に対して全て「yes」と答えることによって、最終的にデバッガは一つの属性をバグとして特定する。そこで、本論文で提案した方法とスライス分割を用いた方法、およびアルゴリズムック・デバッグングについて、属性を一つに絞り込むことができる答え（一連の「yes」と「no」の列）の組み合わせを全て求め、それぞれ属性を特定するまでに行った質問の回数を数え、比較を行う。

このような、属性を一つに絞り込むことのできる

「yes」と「no」の列は、あらゆるバグの可能性に対する、デバッグ時のユーザの答えかたに対応している。本節では、このような全ての場合に対する質問の回数に関する比較を行い、デバッグ法の効率について質問の回数の側面から議論を行う。また、絞り込まれたバグの位置とその時に要した質問の回数についての比較を行い、バグの位置と質問の回数の関係を明らかにする。

7.1 実験結果

表 1 は、いくつかの属性文法記述の例に対して実験を行った結果をまとめたものである。表中の実験の欄の「1」と「2」は 2 進小数の例であり、「3」と「4」はプログラムの意味解析を行う属性文法記述の例である。これらの例が含む属性インスタンスの数を表に示す。それぞれの例に対して、本論文で提案した手法とスライス分割を用いた方法、およびアルゴリズムック・デバッグングを用いて、属性を一つに絞り込むことのできる答えの組み合わせを全て求め、それらの中で質問の回数が最小、最大となるものと、全ての質問の回数の平均を表に示した。表中の「Ours」の項が本論文で提案する手法を、「Slice」の項がスライス分割を用いた方法を、「AD」の項がアルゴリズムック・デバッグングを示している。なお、スライス分割を用いたデバッグ法における依存グラフの分割は、属性を依存関係に従ってトポロジカルソートしたものを途中で二つに分割し、それを依存グラフの分割に用いるというものである。

全ての例において、本論文で提案した手法を用いた方が、スライス分割を用いた場合に比べて、平均的に少ない質問の回数で属性を一つに絞り込んでいることがわかる。5.2 節で述べたように、本手法とスライス分割を用いた方法は同じ形式の質問によってバグを特定する方法であり、平均的に少ない質問の回数でバグを特定することができる本手法を用いた方が有効である場合が多い。特に、表中の最大の項が示すように、スライス分割に比べて、本論文で提案した手法の方が、最大の質問回数が大幅に少ないことがわかる。スライス分割の例において、質問の回数が増えるのは、答えに「yes」が多くなる場合である。このよう

表 1 質問の回数

実験	属性数		最小	最大	平均	本手法が Slice より有利になる属性数
1	13	Ours	3	5	4.13	11
		Slice	2	6	4.75	
		AD	2	5	3.29	
2	25	Ours	4	6	5.45	20
		Slice	3	9	6.53	
		AD	2	7	4.31	
3	90	Ours	6	9	7.55	78
		Slice	2	16	11.06	
		AD	4	9	6.09	
4	229	Ours	7	11	8.90	210
		Slice	4	31	13.38	
		AD	4	11	7.64	

な場合、スライス分割を用いた方法では、属性計算合成の出力に含まれる全ての属性に対して、質問に答えなければバグを絞り込むことができないため質問の回数が極端に多くなってしまふ。

アルゴリズムック・デバッグは本手法と比較して、表 1 ではバグを特定するまでに要する質問の回数が少ないように見える。しかし、アルゴリズムック・デバッグはバグとして特定する属性文法記述の範囲が広がってしまい、バグとして特定される意味規則が複数個になってしまう場合がある[8]。例えば図 2 の属性文法記述の例に対して、アルゴリズムック・デバッグを用いた場合、バグとして特定されるのは図中の (bug) で示された行を含む 3 つの意味規則である。

これに対して本手法では、最後まで質問に答えることによって必ず一つの意味規則が特定される。この他にも質問の形式が異なるなどの違いもあるため、ここでは細かい議論は省略する。詳しくは文献[15]を参照されたい。

最後に文献[14]で提案されているデバッグ法との関係を述べる。文献[14]の方法は、デバッグの質問に対して「質問の前提がおかしい」というようなユーザの意図を反映させたり、実行時エラーを扱う、不完全な部分木を扱う、などによってデバッグの効率を向上

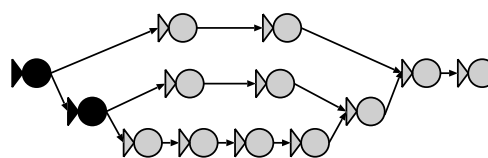


図 17 質問の回数とバグの位置の関係

させるものであった。これらは属性文法の系統的デバッグ法の基本的な枠組みを応用することによって実現していた。これに対して本論文で提案したデバッグ法は、なるべく少ない質問の回数によってバグを特定することが出来るよう、系統的デバッグ法の基本的な枠組みを拡張したものである。このデバッグ法は、「ある属性の値の間関係がユーザの意図と一致するかどうか」という系統的デバッグ法の基本的な質問のみで進められる。このように本研究での実験は、文献[14]を基本にしつつ、バグの絞り込みの効率向上の部分の効果を評価したものであり、文献[14]での評価とは別の方向性を持ったものである。

7.2 バグの位置と質問の回数の関係

次に、バグを特定するまでに行う質問の回数と、特定されたバグの位置の関係について述べる。

表 1 の「1」の実験は、2 節で用いた 2 進小数の例である。図 17 において黒色で示された属性は、この

例において、その属性がバグとして絞り込まれたときに、本論文で提案した手法よりも、スライス分割を用いた方法の方が少ない質問の回数で特定できたものである。このように、属性評価の最初の方で計算される属性が誤っている場合、スライス分割を利用した方が、少ない質問の回数でバグを特定していることがわかる。他の例においても、同様の傾向があった。これは、スライス分割を用いた方法において、問い合わせの対象となる属性計算合成の出力の一つが誤っており、たまたまこの属性に対する質問のみでバグを絞り込むことができる場合である。

表1の最後の項は、本手法の方がスライス分割を用いた手法に比べて少ない質問の回数によって、バグとして特定することのできる属性の数を表している。このように、ほとんどの属性において、本手法を用いた方が、少ない質問の回数によってバグを特定できることがわかる。また、スライス分割を用いた方法の方が少ない質問の回数でバグを特定できる場合において、本手法を用いてもそれほど質問の回数は多くならない。

8 考察

3節で述べた属性文法の系統的デバッグ法を利用することによって、ユーザは属性の複雑な依存関係を意識せず、属性値の関係が正しいかどうかを答えることによってバグを特定することが出来るようになった。また、5節で提案された新しいデバッグ法は、ユーザへの質問の回数を最小限に抑えるため、さらに効率よくデバッグを行うことができる。しかし、デバッグの効率はバグの種類やユーザの主観にも左右されるため、効率の善し悪しを簡単に決めることはできない。本節では、デバッグの様々な側面から系統的デバッグ法の効率について論じる。

質問の回数

デバッグの効率について考える場合、なるべく少ない質問の回数によって、バグを特定することができれば効率がよいと言える。これまでのデバッグ法では、属性計算合成の出力に含まれる全ての属性について、入力との関係が正しいかどうかを確かめなければ、バグの範囲を絞り込むことができず、質問の回数が増え

てしまうことがしばしばあった。これに対して、5.1節で述べたデバッグ法の枠組みでは、ある一つの属性に対してのみ、入力との関係を確認することでバグの範囲を絞り込むことができる。さらに、この枠組みを利用したアルゴリズムによって、逆支配している属性の数を重みとして持たせ、質問の後にバグの候補となる属性の数を見積もることによって、質問の数を少なく抑えることができる。また、これまでの方法に比べて本手法を用いた方が、多くの場合において少ない質問の回数でバグを特定できることを、7節の実験によって確かめることができた。

質問の難しさ

属性文法の系統的デバッグ法では、デバッグはユーザに対して属性値間の関係が正しいかどうかを質問することによってバグの位置を特定する。ユーザにとって簡単な質問によってバグを特定することができれば、デバッグの効率は良いといえる。3.3節で述べた系統的デバッグ法の一般化の枠組みを利用することによって、デバッグは自由な位置でユーザに質問できるようになった。しかし、不用意な質問の選び方はユーザにとって答えづらいものとなり、デバッグの効率が悪くなる可能性がある。例えば、解析木上の全く異なった部分木のノード上にある属性間関係はユーザにとって、正しいかどうか判断しづらいものとなる場合が多い。したがって系統的デバッグを行う場合、ユーザにとって答えやすい質問を選びながらバグを特定するような工夫が必要となる。

これに対してアルゴリズムック・デバッグングは、同じ解析木のノードに付随する属性同士関係をユーザに質問する方法であり、スライス分割を用いた方法は、入力が空集合で出力の値が正しいかどうかを、ユーザに質問する方法である。これらの方法は、質問の選び方に制限があるものの、ユーザにとっては比較的答えやすい質問のしかたといえる。また、一般化の枠組みを利用することによって、このような系統的デバッグ法を組合せて利用することができ、ユーザは答えやすい質問を選択しながらデバッグを進めることができる。

本論文で提案したデバッグ法での質問は、属性計算合成の質問の前提が常に同じであり、その前提に対し

である属性値が正しいかどうかを問い合わせるというものである。この前提を空集合とした場合、スライス分割を用いた方法と同じ質問の形式になる。しかし4節で述べたように、スライス分割を用いた手法では、前提が空であるために質問の属性計算合成の範囲が広がる可能性があった。これに対して本手法では、前提を持たせることによって質問の範囲を狭め、質問を簡単に行うことができる。例えば、4.3節で述べたような、ある部分木に注目してデバッグを行う場合、その部分木における属性計算への入力として、部分木の根のノードに付随している属性が与えられ、これらの属性に対して、質問の対象となっている部分木内の属性が正しいかどうかを問い合わせることができる。このような、部分木の根に付随する属性を部分木内の属性計算への入力とし、これらの属性を前提として、部分木に含まれる属性が正しいかどうかを問い合わせることで、その部分木に集中して系統的デバッグを行うことができる。このような質問の難しさを軽減する機能はAkiでも提供されている。

しかし、質問の難しさはユーザの主観に依存するものであり、さらに効率のよいデバッグ法の考案が期待される。

ユーザ・インターフェース

デバッグがユーザの理解を補助することによって、質問の難しさをさらに軽減することができる。ユーザは、デバッグが示した質問の入力に対する出力の値と、ユーザの意図していた出力の値を比較し、一致しているかどうかを確かめなければならない。このときの入力と出力の関係は、これらのうちの変化した部分を見ることで確かめることができる場合が多い。例えば、プログラムの意味解析を行う属性文法記述において、変数名とその型を管理するテーブルを属性値として持ち、その属性がデバッグ時に質問の対象となる場合、入力と出力の関係は追加された変数の情報による差の部分のみを見ることで確かめることができる。そこでAkiでは系統的デバッグ法の質問時に、質問の属性計算合成の入力と出力の違いが強調されてユーザに提示される。ユーザは強調された部分のみを見ることで、属性計算合成の挙動が正しいかどうかを容易に確かめることができる。

その他に、現在注目している解析木の部分木や、対応するソースプログラム等を表示することによって、ユーザの理解を補助する。

大きな例に対するデバッグ

属性評価器への入力として長大な文字列が与えられた場合、解析木のノードや属性の数が多くなる。このような例に対して系統的デバッグを行う場合、質問に含まれる属性数が多くなり、4.1節で述べたような質問の難しさに関する問題が生じる。このような場合Akiでは、質問の難しさの項で述べたような、ユーザが誤りを含む部分木を指摘し、その部分でデバッグを行うことができる。このような部分木を早く見つけるためには、デバッグによる補助が不可欠である。Akiには、解析木全体を見るためのサブウィンドウがあり、これによってユーザが注目しているノードの解析木全体における位置を確かめることができる。しかし、このような機能だけでは不十分であり、より効果的に解析木のナビゲーションを行う機能が必要となる。

また、大きな例では属性の値そのものが複雑になり、デバッグの質問がユーザにとって難しいものとなる場合がある。ユーザ・インターフェースの項でも述べたように、Akiからユーザへの質問では、入力と出力の差の部分が強調されるため、質問の難しさが軽減される場合もあるが、このような属性の値の表示に関してもさらなる工夫が必要である。

エラーによる途中終了

ユーザがデバッグを開始する局面には、最終的に計算された属性の値の異常に気付く場合の他に、属性の値の異常により属性評価が途中で終了してしまう場合がある。途中で終了してしまった局面で、アルゴリズムック・デバッグを実行する場合、Synth関数の一部の入力や出力が計算されていない可能性があるため、ユーザは質問に答えられないことがある[14]。これに対してスライス分割を用いた方法や本論文で提案したデバッグ法では、プログラムの終了の原因となった属性を計算するまでに計算された属性に対してデバッグを行うことができる。

他のデバッグ法との関係

5節で提案したデバッグ法と3.3節で説明した一般化の枠組みはともに、バグの候補となる属性の集合から、ユーザへの質問によってバグの候補を絞り込むという方法である。したがって、本手法を用いて絞り込んだバグの候補となる属性の集合から、属性計算合成を作り、一般化手法を利用することが可能である。3.3節で説明した一般化の枠組みと比べると本手法は、閉じていない属性計算合成の正誤の情報によってバグの位置を絞り込むことができるよう拡張したものであり、これによって少ない質問の回数でバグを特定できる方法であるといえる。

文献[14]では、一般化の枠組みの応用として、アルゴリズムック・デバッグからスライス分割を用いた方法へ移行する方法を紹介しており、Akiでもこれを踏襲している。これによってユーザは、より答えやすい形の質問を選びながら、バグの特定を行うことができる。5.2節で述べたデバッグ・アルゴリズムは、スライス分割を用いた方法を拡張したものであるから、アルゴリズムック・デバッグから、5.2節のアルゴリズムへ移行することも可能である。

9 まとめ

本論文では、属性文法記述をデバッグする際の問題点を明らかにし、より効率的なデバッグ法を提案した。これまで提案された系統的デバッグ法では、質問の回数や質問の難しさなど、質問の選び方に起因した、デバッグの効率を悪化させる問題点があった。

これに対して、系統的デバッグ法の一般化の理論を拡張し、より少ない質問の回数でバグの位置を絞り込むことができる枠組みを提案した。この方法は、属性の依存関係の逆支配関係を利用することによって、属性の値の影響範囲を詳しく調べ、より厳密にバグを絞り込むことにより、少ない質問の回数によってバグの位置を特定するというものである。さらに、この枠組みを利用したデバッグ・アルゴリズムを提案した。このアルゴリズムでは、ユーザの答えからどの程度バグを絞り込むことができるかを、属性ごとにあらかじめ見積っておき、それをもとに質問の位置を選ぶため、少ない質問の回数でバグの位置を特定することが

できる。

また、このデバッグ法を属性文法デバッガAki上に実装した。Akiでは本論文で述べた、デバッグの効率を悪化させる種々の問題点を軽減する方法が実装されており、ユーザはAkiを使うことによって、これまでよりも効率的にデバッグを行うことができる。また、Akiを用いて質問の回数に関する実験を行い、多くの場合において、効率的にデバッグが行えることを確かめた。

今後は、系統的デバッグの理論を利用したより効率的なデバッグ法の研究や、ユーザが属性値の正誤をより容易に理解するためのユーザ・インターフェースの開発など、デバッグの理論とユーザ・インターフェースの両面から、より効率のよいデバッグ環境の実現を目指したい。

参考文献

- [1] Agrawal, H., DeMillo, R. A. and Spafford, E. H.: Debugging with Dynamic Slicing and Backtracking, *Software Practice and Experience*, Vol. 23(1993), pp. 589-616.
- [2] Fritzson, P., Shahmehri, N., Kamkar, M. and Gyimothy, T.: Generalized Algorithmic Debugging and Testing, *ACM Lett. Prog. Lang. Syst.*, Vol. 1, No. 4(1992), pp. 303-322.
- [3] Ikezoe, Y., Sasaki, Y., Ohshima, Y., Wakita, K. and Sassa, M.: Systematic debugging of Attribute Grammars, *Proceedings of the Fourth International Workshop on Automated Debugging AADEBUG 2000*, 2000, pp. 235-240.
- [4] Korel, B. and Laski, J.: Dynamic Slicing of Computer Programs, *J. System Software*, Vol. 13(1990), pp. 187-195.
- [5] Mayoh, B. H.: Attribute Grammars and Mathematical Semantics, *SIAM Journal on Computing*, Vol. 10, No. 3(1981), pp. 503-518.
- [6] Nilsson, H.: *Declarative Debugging for Lazy Functional Languages*, PhD Thesis, Linkoping University, 1998.
- [7] 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, 1991, pp. 56-70.
- [8] Sassa, M. and Ookubo, T.: Systematic debugging method for attribute grammar description, *Information Processing Letters*, Vol. 62(1997), pp. 305-313.
- [9] Shapiro, E. Y.: *Algorithmic Program Debugging*, The MIT Press, 1983.
- [10] Shimomura, T.: Critical Slice-Based Fault Lo-

- calization for Any Type of Error, *IEICE Transactions on Information and Systems*, Vol. E76-D, No. 6(1993), pp. 656–667.
- [11] Tip, F.: A survey of program slicing techniques, *Journal of programming languages*, Vol. 3(1995), pp. 121–189.
- [12] Weiser, M.: Programmers Use Slicing When Debugging, *Comm. of the ACM*, Vol. 25, No. 7(1982), pp. 446–452.
- [13] Weiser, M.: Program slicing, *IEEE Transaction on Software Engineering*, Vol. 10, No. 4(1984), pp. 352–357.
- [14] 佐々木晃, 池添洋平, 佐々政孝: 属性文法の系統的デバッグ法, *情報処理学会論文誌: プログラミング*, Vol. 43, No. SIG 3(PRO14)(2002), pp. 1–16.
- [15] 池添洋平: 属性文法の系統的デバッグ手法とその環境, 修士論文, 東京工業大学, 2002.