分割統治法を用いた変換系の検証手法について

2K−1

栗野俊一　深澤良彰

早稲田大学理工学部

1　はじめに

計算機の応用範囲が広がり、今まで利用されていなかったような様々な分野にも、計算機の進出が目立つようになった。これに伴い、ソフトウェアの信頼性の欠如は、社会的な問題や倫理的な問題を引き起こす要因になってきている。その半面、信頼性の根拠となる、ソフトウェアの正当性の検証は、なかなか完全には実行されていない作業の一つである。これは、検証の自動化が難しく、しかも、その困難度がソフトウェアの規模に対して、急速に増大するためである。

このように、人間の介在が不可避であり、しかも大規模な問題を扱うためには、問題の抽象化と分割による規模の縮小化を行う必要がある。問題の抽象化は、それを仕様記述言語、高級言語、あるいは、その分野専用の言語を用いて記述することによって行える。しかし、問題の分割に関しては、一般的な方法はあまりないように思われる。

本発表では、中規模のソフトウェアである変換系プログラムの検証を対象とし、検証作業をどのように分割すれば良いかについて議論する。この分割法は、自己記述可能であることと、その処理対象が自由文脈文法で表現されるという、変換系に共通な、2つの性質に着目している。

2　準備

定義　文法と言語

本論文で扱う文法は曖昧でない文脈自由文法G_1であり、言語はその文法から生成される文脈自由言語Lとする。

定義　変換系

変換系$T^{i}{}_{d}$は、言語L_dで記述された言語L_iから言語L_oへの変換系(記述)を表わす。また、記述言語が問題にならない場合には、単にT^{i}と記述する。

定義　正当性

変換系T^{i}が言語L_i，L_oの意味関数M_i，M_oに対して正当であるとは、次の式が成立することである。

$$\forall x \in L_i.\ M_o(T^{i}(x)) = M_i(x)$$

定義　自己記述可能

$T^{i}{}_{j}$が存在するとき、言語L_iはL_jに対して自己記述可能であるといい、言語L_iはL_jに対する自己記述可能言語であるという。

定義　下位文法と下位言語

文法$G_i = <C_i, V_i, P_i, S_i>$が$G_j = <C_j, V_j, P_j, S_j>$の下位文法であるとは、次の関係が成立する時である。

$$C_i \subset C_j,\ V_i \subset V_j,\ P_i \subset P_j,\ S_i = S_j$$

また、このとき言語L_iはL_jの下位言語であるという。

この定義から、言語L_iがL_jの下位言語であるならば、$L_i \subset L_j$となるが、その逆は成立しない。

定義　極小自己記述言語

言語L_o，L_*に対して、L_+が極小自己記述言語であるとは、L_+がL_oの下位言語で、かつL_*に対して自己記述可能言語であり、しかもL_+の自己記述可能な下位言語が存在しない場合である。

問題　本論文の目的は、与えられた入力言語L_oから出力言語L_*への変換系$T^{0}{}_{*}{}^{*}{}_{*}$を作成することである。ただし、言語L_o，L_*、変換系$T^{0}{}_{*}{}^{*}{}_{*}$は、以下の性質を持っている。

・言語L_o，L_*はいずれも意味関数M_o，M_*が与えられている。

・言語L_o，L_*はいずれも万能である。

・言語L_*は実行可能である。

・$T^{0}{}_{*}{}^{*}{}_{*}$は正当である。

また、これは、厳密なものではないが次の性質も仮定する。

・言語L_oはL_*に較べ、高級である。したがって、L_*より、L_oで記述した方が検証しやすい。

3　変換系の検証手順

我々の行った変換系の作成及びに、検証手順は次の通りである。基本的な方針は、大きな言語の変換系をより小さな言語で記述し、最終的には、小さな言語の変換系のみを、出力言語で記述することである。これによって、分割が行えると同時に、多くの部分が、より高級な言語で記述した形で検証することができる。

1．変換系$T^{0}{}_{o}{}^{*}{}_{o}$の作成
2．変換系$T^{0}{}_{*}{}^{*}{}_{1}$から$T^{+}{}_{*}{}^{*}{}_{+}$の作成
3．変換系$T^{+}{}_{*}{}^{*}{}_{*}$から$T^{0}{}_{*}{}^{*}{}_{*}$の作成
4．自己記述可能な言語の変換系の検証
5．自己記述可能でない言語の変換系の検証
6．変換系$T^{0}{}_{*}{}^{*}{}_{*}$の作成と検証

3．1　変換系$T^{0}{}_{o}{}^{*}{}_{o}$の作成

まず、変換系$T^{0}{}_{o}{}^{*}{}_{o}$を作成する。言語L_oは万能であるので、$T^{0}{}_{o}{}^{*}{}_{o}$は作成できる。ここで、$T^{0}{}_{o}{}^{*}{}_{o}$を作成する際に気を付けることが二つある。これらはいずれも、言語L_oがG_oを持つことを利用したものである。一つ目は、L_oが入力言語である事に着目することである。これは、複合設計などを用い、入力の構造にしたがって$T^{0}{}_{o}{}^{*}{}_{o}$を作成する。L_oの構造はG_oそのものであるので、$T^{0}{}_{o}{}^{*}{}_{o}$のモジュール構成は、G_oに含まれる構文規則に対応することになる。これによって、後で述べるように、$T^{0}{}_{o}{}^{*}{}_{o}$の正当性の検証が容易になる。二つ目は、L_oが$T^{0}{}_{o}{}^{*}{}_{o}$の記述言語である事に着目する。$T^{0}{}_{o}{}^{*}{}_{o}$の構文木を作成すると、その節点に対して、G_oに含まれる構文規則を対応させることができる。この時に、G_oに含まれる構文規則のうち、$T^{0}{}_{o}{}^{*}{}_{o}$の構文木の節点とは対応しないようなものを少なくても一つはあるように$T^{0}{}_{o}{}^{*}{}_{o}$を作成する。これが可能であれば、G_oから、その利用されなかった構文規則を取り除いたG_1を作ることによって、言語L_oの下位言語で、しかも$T^{0}{}_{o}{}^{*}{}_{o}(=T^{0}{}_{o}{}^{*}{}_{1})$が記述可能な言語$L_1$が構成できることになる。

3．2　変換系$T^{0}{}_{o}{}^{*}{}_{o}$から$T^{+}{}_{*}{}^{*}{}_{+}$の作成

次の手順により、L_+と変換系$T^{+}{}_{*}{}^{*}{}_{+}$の作成を行う。

An Application of Divide and Conqure Technique for Verification of a Translator
Syun'ich Kurino, Yoshiaki Fukazawa,
Waseda University

1．i：＝0
2．G_iの構文規則で、$T^{i*}{}_i$に利用されていない構文規則pを一つ探す。なければ8へ。
3．G_iの構文規則の集合P_iからpを取り除き、下位文法G_{i+1}を作成する。また、その結果、利用されないくなった他の要素も取り除く。
4．$T^{i*}{}_i$から、L_{i+1}に無関係な部分を取り除き、これを、$T^{i+1*}{}_{i+1}$とする。
5．i：＝i＋1
6．可能ならG_iのより少ない構文規則で$T^{i*}{}_i$を書き直す。ただし、入力に対応した構造は崩さないことにする。
7．2へ。
8．L_+：＝L_i；$T^{+*}{}_+$：＝$T^{i*}{}_i$

この手順から作成された$T^{i*}{}_i$は、$T^{+*}{}_+$の近似でしかない。なぜなら、手順6で、最少の構文規則数で$T^{i*}{}_i$を書き直すことは困難だからである。手順6は、作成者の仕事であり、完全性を要求することは、逆に変換系の作成並びに検証作業に負担をかけることになる。しかも、L_+は、厳密に極小である必要はないので、充分に近似されていれば、これを検証の中間点にしても良い。

3．3　変換系$T^{+*}{}_+$から$T^{0*}{}_*$の作成

変換系$T^{+*}{}_+$は、これまでと異なり、構文規則を減少させることによって、規模を縮小させることができない。そこで、次のように、言語L_*の埋込みを行うことによって、この問題を回避する。埋込みとは、ある言語L_iにL_*の要素を構文要素の一部として埋込む形で行う。この埋込まれたL_*の要素は、特別な括弧【，】で囲むことにより、L_iの要素と、区別することができる。これを利用することによって、さらに、L_{+1}，L_{+2}，…，L_{+n}＝｛｝といった、下位言語列を作成することができる。

定義　言語L_*の埋込み
言語L_i'がL_iにL_*を埋込んで作られる言語であるとは、L_i'が、L_iの文法G_i＝＜C_i，V_i，P_i，S_i＞と、L_*の文法G＝＜C_*，V_*，P_*，S_*＞と、新しい二つの終端記号｛【，】｝及びに、非終端記号S_i'を用いて作成した文法G_i'＝＜C_i'，V_i'，P_i'，S_i'＞から生成される言語の時である。ただし、各要素は、以下の通りである。

C_i'＝C_i∪C_*∪｛【，】｝
V_i'＝V_i∪V_*∪｛S_i'｝
P_i'＝P_i∪P_*∪｛S_i'：：＝S_i｜【S_*】｝

今、L_iをL_+の下位言語とする。つまり、$T^{i*}{}_+$は存在するが、$T^{i*}{}_i$は存在しない。そこで、埋込みを用いて、L_i'を作成する。すると、L_i'は、L_*に対して自己記述可能である。なぜならば、次のように、$T^{i*}{}_+$を用いて$T^{i'*}{}_{i'}$が構成できるからである。

・$T^{i*}{}_+$のL_+-L_iの部分$\Delta^i{}_+$をL_*で書き直した$\Delta^i{}_*$を作り、‘【’と‘】’で挟む。これは$T^{i*}{}_{i'}$である。
・$T^{i*}{}_{i'}$に‘【’が現れたら‘】’まで出力に単に複写する部分を付け加える。これが、$T^{i'*}{}_{i'}$である。

この埋込みによって、自己記述並びに、より高級な言語の変換系が記述可能になる。

3．4　自己記述された変換系の検証

変換系$T^{0*}{}_0$と$T^{+*}{}_+$の近似は、言語L_0あるいはL_+で記述されている。したがって、L_*で記述されている場合より検証が容易である。さらに、これを記述する際に、変換系のモジュール構造が入力の構造に対応するように作成している。このような形で、変換系を構成した理由は、文脈自由言語は文脈自由文法を与えることによって特徴付けることが可能であり、文脈自由文法は非常にモジュラリティの高い構文規則に分割されているからである。この結果、変換系の構造を、その文法に合せて作成することによって、変換系自身も独立性の高いモジュール分解が可能になる。

このような形で、分解された変換系モジュールの検証は、次のように行える。

1．各々の構文規則　v_0::＝$v_1v_2\cdots v_n$　に対して、v_0の要素を変換するT^{v0}が正しいことを、関数T^{v1}，T^{v2}，…，T^{vn}が存在し、これらが$v_1v_2\cdots v_n$の各要素を正しく変換することを仮定して示す。
2．変換系の入力の長さに関する帰納法を用いることによって、全ての長さの入力に対して、言語L_iの変換系T^{i*}が正しく変換することを示す。

3．5　L_+の下位言語の変換系の検証

言語L_iがL_+の下位言語の場合、$T^{i*}{}_i$が存在しないため、埋込みを行いてL_iを拡張し、$T^{i'*}{}_{i'}$を作成する。したがって、検証の対象はL_iで記述された部分と、L_*で記述された部分の二つに分けられる。前者に関しては、前節と同様に検証するが、後者は異なる方法で検証しなければならない。その方法は、L^*の意味関数を用いて正当性を直接検査するのではなく、L^*の部分と等しい機能を持ち、かつ、L^*より高級な他の言語で記述されたものとを比較することである。

一般に、$T^{i-1*}{}_{i'}$と$T^{i-1*}{}_{i-1'}$を比較すると、この二つは、同じ機能を持つが、L_{i-1}の方がL_iに較べ、言語が拡大しているため、‘【’と‘】’で囲まれたL^*部分は減少している。したがって、その差分を取り出すと、L_{i-1}で記述されたものとL^*で記述された同じ機能を持つ二つの記述を得ることができる。このような二つの記述が同じ意味を持つことを示せば、L^*での記述を直接検査する必要はない。

これらのL^*で記述された差分は、最終的には、$T^{+*}{}_*$の一部分となる。したがって、それぞれの差分を検証することと、$T^{+*}{}_*$の部分部分を検証することは同等である。しかし、$T^{+*}{}_*$全体の検証を一度に、しかも直接検証するよりも、一部分毎に、L_+の記述との比較の形で間接的に検証を行った方が容易である。

3．6　変換系$T^{0*}{}_*$の作成と検証

これまでの手順により、言語列

L_{+n}＝｛｝，…，L_+，…，L_0

と、変換系列

$T^{+n-1*}{}_*$＝$T^{+n-1*}{}_{+n}$，$T^{+n-1*}{}_{+n-1}$，…，$T^{0*}{}_1$

が作成された。後は、これを順に適用し、ブートストラップによって、$T^{0*}{}_*$が作成できる。一方、それらの、変換系記述並びに変換系が全て、正当ならば、作成された$T^{0*}{}_*$が正当であることも、同様に帰納法によって示すことができる。

4　おわりに

変換系の検証を容易にするために、検証を分割して行う手法について述べた。構文規則による分割と、ブートストラップ、特にその前段である、自己記述できない言語の拡張を工夫することにより、段階を増やし、充分な細分化が行えた。次の目標としては、実際に、その検証法を実践することである。現在、具体的な言語を固定し、自己記述できない言語の作成段階までの作業が進んでおり、残る段階に関しても、順に進めて行く予定である。

この手法の問題としては、対象が文脈自由言語だけであることが挙げられる。一般の言語、文脈依存的な処理を行うためには、記号表を持ち込むか、あるいは、属性文法での下降型の属性を考慮する必要がある。これらに関しては、今後の課題と考えている。