2G-8

表示的意味論によるC言語の意味記述と処理系作成への応用

濱田 和雄[1] 島崎 眞昭[2] 津田 孝夫[1]
京都大学工学部[1] 九州大学大型計算機センター[2]

1. はじめに

表示的意味論は記述能力の点で評価が高く、プログラム言語処理系作成の自動化への応用も早くから研究されてきた。しかし、初期のものは実行時にラムダ式の翻訳実行を行うものが多く、作成された処理系の実用性の点で問題があった。A. Appel[1][2]はラムダ式のコンビネータリダクションの際、副作用としてレジスタ転送コードを生成する方法を示し、Pascalに適用して、表示的意味論を用いた実用的処理系の自動作成について展望を切り開いた。A. AppelはC言語への適用可能性についても言及しているが、必ずしも自明なことではない。我々は表示的意味論による処理系作成の自動化の実用性を調査するため、C言語の意味記述と処理系作成への応用について、検討を進めておりその経過を報告する。

2. C言語の表示的意味記述における問題点

C言語の表示的意味記述を行う際に問題となる点を挙げ、その解決法を示す。

(1)代入式と条件式

一般に式expが値としてのみ評価される簡単な言語に対する表示的意味記述では、式expを表示する意味関数Eは、次のように定義される。

$$E : Exp \rightarrow Env \rightarrow S \rightarrow V$$

ただし、Expは構文単位exp、Ideは識別子、Locはロケーション、Vは値を表す意味領域とし、Env、Sは、

Env (環境) : Ide → Loc
S (状態) : Loc → V

の領域とする。

ところが、C言語では、式の1つに代入式 (exp = exp) があり、式は、値の他に状態更新の意味をもつ。また、条件式 (exp ? exp : exp) のように、内部に制御の流れを変える意味を持つ式もある。これらの問題は、式接続Ec (expression continuation) [3]を用いて意味関数Eを次のように定義することによって解決する。

$$E : Exp \rightarrow Env \rightarrow Ec \rightarrow C$$

ただし、C (接続) : S → A (answer)
Ec (式接続) : V → C とする。

(2)関数の暗黙の宣言

C言語では、関数呼び出し式において未定義の識別子IDが関数名として用いられた場合、この識別子は暗黙のうちに整数を返す関数として次のように宣言されたことになる。

```
extern int ID();
```

従って、式をコンパイルする際にシンボルテーブルが更新されることになる。一般に、表示的意味論において、宣言文以外の式や文では、シンボルテーブルに当たる環境は、参照されるだけのものとして意味領域を決定するが、そのような意味領域では、暗黙の宣言の意味を記述することはできない。よって、式の意味関数Eは次のように変更する必要がある。

$$E : Exp \rightarrow Env \rightarrow (Env \times (Ec \rightarrow C))$$

この定義により、文の意味関数Sも通常とは異なる次のような領域にする必要がある。

$$S : Stm \rightarrow Env \rightarrow (Env \times (C \rightarrow C))$$

ただし、Stmは、構文単位stmを表す意味領域とする。

(3)静的なタイプチェックの必要性

一般にコンパイラは、式のタイプチェックをコンパイル時に静的に行う。しかし、表示的意味論で意味を記述する場合には、一般に、式の型は、評価結果の値がどの型の領域に属するのかによって判断される場合が多い。

C言語の条件式の意味は、「第1式を評価して、それが0でなければ、結果は第2式の値となり、そうでなければ、第3の式の値となる」である。このとき評価されるのは、第2および第3の被演算数のいずれかのみであるが、結果の型は、第2および第3両式の型に依存する。つまり、いずれか一方の式は、評価はせずにその型だけを知る必要がある。同じことは、sizeof演算子の被演算数となる式に対しても言える。よって、この意味を記述するためには、評価実行にあたる領域Ec→Cと型の領域Typeとを分離独立させる必要がある。そこで、意味関数Eを

$$E : Exp \rightarrow Env \rightarrow (Env \times (Type \times (Ec \rightarrow C)))$$

とする。

また、静的タイプチェックのための意味関数Etを別に用意するという方法も考えられる。

$$Et : Exp \rightarrow Env \rightarrow Type$$

(4)左辺値としての式

C言語の代入式において、代入演算子の左辺におかれる式は、左辺値でなければならないが、左辺値をとる式を構文単位として分離すると、曖昧さのある文法となってしまう。よって、C言語の式は、左辺値としての意味と右辺値としての意味を持ち、コンパイラは、意味解析部でその使い分けを行うことになる。

表示的意味記述を行う場合にも、この使い分けは必要である。よって、式の意味を表示する意味関数は、左辺値としての意味を表示するElと右辺値としての意味を表示するErの2つが必要となる。それぞれの意味関数の領域は、次のようになる。

$$El : Exp \rightarrow Env \rightarrow ((Env \times (Type \times (Lc \rightarrow C))) \mid undefined)$$

$$Er : Exp \rightarrow Env \rightarrow (Env \times (Type \times (Rc \rightarrow C)))$$

Denotational Description of the Language C and its Apprication to Compiler Construction

Kazuo HAMADA[1], Masaaki SHIMASAKI[2], Takao TSUDA[1]
[1]Kyoto University, [2]Kyushu University

ただし、Lvを左辺値として取り得る値の領域、Rvを右辺値として取り得る値の領域とした場合、Lc、Rcは、

Lc : Lv → C

Rc : Rv → C

という領域であるものとする。

特に、式が左辺値とならない場合は、Elは、その式を未定義の領域undefinedに評価する。

(5)switch文

C言語のswitch文、case文、default文の構文は、次のようになる。

stm → switch (exp) stm

stm → case const_exp : stm

stm → default : stm

ここで、注意すべき点は、switch文とcase文およびdefault文の間に、構文上の特別なつながりがないことである。よって、コンパイラは、各switch文に対して、そのサブ文中に現れるcase文、default文を調べるという処理を意味解析部で行い、式の値に応じていずれかに制御を渡すコードを生成することになる。

これらの文の意味記述を考える場合も、各switch文に対して、サブ文中に含まれるcase文、default文それぞれに対して、case定数（V）と、その文への接続またはそれを表すラベル（Label）の組（V×Label）を作り、それらをリスト（(V×Label)List）にして、そのリストをもとにswitch文の意味を記述することになる。switch文は、入れ子構造が許されるので、このリストも入れ子構造の領域（Se:((V×Label)List)Tuple）にする必要がある。この領域は、各文で参照及び定義される。

ただし、aList : (a → b → b) → b → b

aTuple : (a × aTuple) | undefined

とする。

3．処理系作成への応用

表示的意味記述は、ソースプログラムと機械コードの間の中間言語としてとらえられる。ソースプログラムから表示的意味記述への変換は、比較的簡単である。問題となるのは、表示的意味記述から機械コードへの変換である。この変換方法をA.Appelが考えている。

3．1 Appelの方式[1][2]

Appelの方式は、まず、λ式のテキスト表現で表されたプログラムの表示的意味を解析してグラフ表現を構成し、次に、構成されたグラフをリダクションルール（reduction rule）というノードの置き換え規則に従って簡約して、その際、副作用としてコードを生成するというものである。リダクションルールの形式は、

pattern → substitution {生成するコード}

であり、左辺のpatternに一致するノード構成を右辺のノード構成に置き換える際に、中括弧内のコードを生成することを意味している。ここで生成されるコードは、レジスタ転送コードであり、生成されたコード列は、J.Davidsonのtarget-code generator[4]によって最適化された後に実際の機械コードに変換される。

3．2 sample-C[5]の処理系

Appelのコード生成方式に従って作成したC言語のサブセットであるsample-Cの処理系を例にシステム構成を説明する。

図1にシステムの全体構成を示す。

意味仕様ファイルは、プリプロセッサによってlex及びyaccのソースファイルに落とされ、lex、yaccによって字句解析部、構文解析部となる。構文解析の結果としての出力は、ソースプログラムの意味に対応するλ式のテキスト表現となる。このテキストは、ノードジェネレータによってグラフ表現に直される。最後にリデューサが、リダクションルールに従って、ノードの置き換えを行い、同時にコードを生成する。

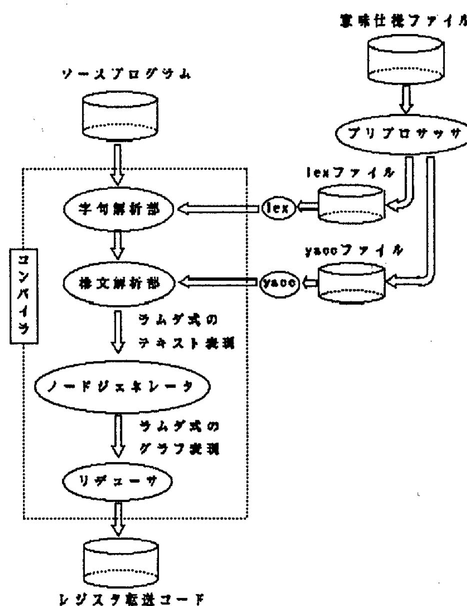

図1 表示的意味記述を用いたコンパイラシステムの全体構成

4．おわりに

本稿では、C言語の表示的意味記述における問題点について検討し、表示的意味記述の処理系への応用方法をC言語のサブセットであるsample-Cの処理系を例として説明した。この方法によるC言語の処理系を開発中である。C言語の処理に適したコンビネータおよびリダクションルールを導入する予定である。

参考文献

[1] Andrew W. Appel : Compile-time Evaluation and Code Generation for Semantics-directed Compilers (CMU-CS-1985-147)

[2] Andrew W. Appel : Semantics-Directed Code Generation, Conference Record of the 12th Annual ACM Symposium on Principles of Programming Languages, (Jan. 1985), 315-324

[3] Michael J. C. Gordon : The Denotational Description of Programming Language (Springer-Verlag, New York, 1979)

[4] Jack W. Davidson : Simplifying Code Generation through Peephole Optimization, Ph.D. dissertation, The University of Arizona, 1981

[5] Axel T. Schreiner and H. George Friedman, Jr : Introduction to Compiler Construction with UNIX (Prentice-Hall, Inc. Englewood Cliffs, NJ 07632) 15-19