

教育用C言語の設計と型検査に関する 研究

平成18年度

三重大学大学院工学研究科
博士前期課程 情報工学専攻

渡辺 和 樹

修士論文

教育用C言語の設計と型検査に関する 研究



平成18年度修了
三重大学大学院 工学研究科
博士前期課程 情報工学専攻

渡辺 和樹

目次

はじめに	1
第1章 研究背景	2
第2章 従来の検査法	3
2.1 ポインタ代数による検査法	3
2.2 問題点	4
第3章 C言語のサブセットの定義	5
3.1 構文規則	5
3.2 演算子の優先順位	6
第4章 型の整合性の検査	7
4.1 後置表記	7
4.1.1 後置表記の読み	7
4.1.2 後置表記への変換	7
4.1.3 後置表記への変換例	8
4.2 整合性の表明	9
4.2.1 公理と推論規則	9
4.2.2 型の整合性	9
4.2.3 公理と推論規則の集合	10
4.3 型の整合性の検査方法	12
4.3.1 推論規則と公理を使った検査例	12
4.4 全体の流れ	12
謝辞	15
参考文献	16

はじめに

C 言語 [3] は広く普及している汎用プログラミング言語である。言語が大規模ゆえに、型の複雑な構文規則を持ち、全てのプログラムの意味を理解することは難しい。特に、ポインタの概念と複雑な型宣言の理解が困難である。したがって、C 言語の初心者が学びやすく、C 言語の特徴を残したサブセットを定義し、分かりやすい構文と意味を提示することは初心者にとって有用と考える。また、コンパイルエラー時に適切な型情報を提示するなどの機能を持つツールを備えたコンパイラを作成することはポインタの理解に特に有用と考える。

本研究では、初心者が学びやすい C 言語のサブセットとして教育用 C 言語の設計およびコンパイラの作成、そして学習を支援するツールの作成を行った。特に、ツールの機能を実現するための手法、型の後置表記および公理と推論規則による型の整合性の検査法を提案する。C 言語では識別子の前後に型情報を記述できるが、複数の読み方を許し、型を正しく判断できない場合がある。型情報を後置表記にすることで一意な型の読み方を提供する。型の整合性の検査に関しては、谷口 [1] がポインタ代数で行っているが、適用できる範囲が狭く、プログラム作成者に提示する型情報が分かりにくいという問題点があった。本論文の提案手法はポインタ代数よりも分かりやすく適用範囲が広いという点で、谷口の手法よりも優れている。

2 章では従来の型の整合性の検査法を、3 章では提案手法で検査できる C 言語のサブセットの定義を、4 章では型の整合性を検査する一連の手法を述べる。

第1章 研究背景

C 言語 [3] は大規模で、できることは数多く存在する。BNF による構文 [2] は約 4 ページに渡る。C 言語の初心者が覚えられることがたくさんあるため、学習する際に混乱を招く。そこで、C 言語を学習する前に、C 言語の特徴的な機能を集めた、学びやすいサブセットで学習することは、いきなり C 言語で学習するよりも有用であると考え、このサブセットを教育用 C 言語 [4] と呼ぶ。

ポインタは C 言語の特徴の一つである。ポインタはプログラム中によく現れるが、ポインタを絡めた複雑な型宣言および式は初心者を混乱させる。さらに、ポインタという概念も理解する必要があり、ポインタの習得しづらさに拍車をかけている。例えば、宣言 `int **pp, *pa[10];` の下で、式 `pp = pa` と式 `pa = pp` はそれぞれ正しい代入式なのか。一見するとどちらも正しいように見えるが、前者は正しく、後者は誤っている。この代入式は、`*` と `[10]` の優先順位を知らないと正しく判断できない。優先順位に依存しない一意な表現があれば誤った代入式の記述が減ると考え、型の後置表記を提案する。

また、従来のコンパイラはプログラムの誤りを教えてくれるが、型をどう修正すればいいか等を教えてくれない。講義等で C 言語を学習する際、演算子に必要とされる型を正確に教わらないのが原因と考える。そこで、公理と推論規則による型の整合性の検査法を提案する。型の整合性が取れていないことが導出された場合には、誤りを修正するために必要な情報を提供する。

第2章 従来の検査法

本章では，ポインタ代数 [1] による手法の概要を紹介し，その手法の問題点を述べる．

2.1 ポインタ代数による検査法

[1] はプログラムからポインタ代数と呼ばれる式に変換し，そのポインタ代数に型に対する書換えルールを与えて，型の整合性を検査する．もともとポインタ代数はポインタの指し先を解析するために考案されたものであり，指し先に関する書換えルールを与えることで指し先を解析できる．プログラムからポインタ代数への変換例を下に示す．

// 宣言 int a[10][20][30]; int *p; // 式 p = &a[5][10][15];	bound(bound(bound(int, 30), 20), 10) →(int) L-value : p R-value : adrs(array(array(array(a, 5), 10), 15))
--	--

左がプログラム，右がポインタ代数の式である．bound は配列の領域確保，→ はポインタ，array は配列の要素を求める演算，adrs はアドレス演算を意味する．この例は，変数 p への配列 a[5][10][15] のアドレスの代入を表している．

型の整合性を検査するには，ポインタ代数の L-value と R-value にルール type を適用する．上記の例に type と書き換えルールを適用し，型の整合性の検査したものを下に示す．書き換えルールについては割愛する．

p = &a[5][10][15];	L-value : type(p) = dec(p) = →(int) R-value : type(adrs(array(array(array(a, 5), 10), 15))) = →(type(array(array(array(a, 5), 10), 15))) = →(deref(type(array(array(a, 5), 10)))) ⋮ = →(deref(deref(deref(dec(a)))))) = →(deref(deref(deref(bound(bound(bound(int, 30), 20), 10)))))) ⋮ = →(int)
--------------------	--

L-value と R-value が共に →(int) なので，型の整合性が取れている．

2.2 問題点

ポインタ代数による検査法は、関数呼び出しや代入式を構成する部分式に対応していない。また、ツールの機能として考えると、ポインタ代数による型の表記は分かりにくい。理由はポインタ代数による型の読み方を新たに覚えなければならないからである。ポインタ代数を使うのはC言語の初心者には向かないと考える。

一つ目の問題に対しては、型の推論規則を各部分式に定義し、それらの組み合わせから型の整合性を検査することで解決する。二つ目に対しては、C言語の型情報の表記を使い、これを後置表記に変換することで解決する。宣言 `int a[10][20][30];` は、ポインタ代数なら `bound(bound(bound(int, 30), 20), 10)` となるが、後置表記なら `a : int [30] [20] [10]` となる。後置表記は左から読むということを知っていれば、あとはC言語の型を知っているだけで型の意味が理解できる。これから提案する具体的な手法は、4章で紹介する。

第3章 C言語のサブセットの定義

本章では、本論文で想定するC言語のサブセットを定義する。4章で扱う手法は、このサブセットから生成可能なプログラムを入力とする。提案手法が適用可能な規模をサブセットとして想定しており、コンパイラで扱う教育用C言語とは別とする。

3.1 構文規則

C言語のサブセットの構文規則をBNFで定義する。BNFを以下に示す。

< 翻訳単位 >	::= < 翻訳単位 > < 外部宣言 > < 翻訳単位 >
< 外部宣言 >	::= < 関数定義 > < 宣言 >
< 関数定義 >	::= < 型指定子 > < ポインタ > 識別子 (< 仮引数並び > _{opt}) < 複合文 > < 型指定子 > < ポインタ > 識別子 (VOID) < 複合文 >
< 宣言 >	::= < 型指定子 > < 宣言子 > ;
< 宣言並び >	::= < 宣言 >
< 型指定子 >	::= INT VOID
< 宣言子 >	::= < ポインタ > _{opt} < 直接宣言子 >
< 直接宣言子 >	::= 識別子 (< 宣言子 >) < 直接宣言子 > [整数] < 直接宣言子 > (< 仮引数並び > _{opt}) < 直接宣言子 > (VOID)
< ポインタ >	::= * < ポインタ > _{opt}
< 引数並び >	::= < 引数並び > , < 引数 > < 引数 >
< 引数 >	::= < 型指定子 > < 宣言子 >
< 文 >	::= < 複合文 > < 式文 > < 選択文 > < 繰返し文 > < 分岐文 >
< 複合文 >	::= { < 宣言並び > _{opt} < 文並び > _{opt} }
< 文並び >	::= < 文並び > ; < 文 > < 文 >
< 式文 >	::= < 式 > _{opt} ;
< 選択文 >	::= IF (< 式 >) < 文 > IF (< 式 >) < 文 > ELSE < 文 >
< 繰返し文 >	::= WHILE (< 式 >) < 文 >
< 分岐文 >	::= BREAK ; RETURN < 式 > _{opt} ;
< 式 >	::= < 式 > < 二項演算子 > < 式 > < 単項演算子 > < 式 > ++ < 式 > -- < 式 > < 式 > ++ < 式 > -- < 式 > [< 式 >] < 式 > (< 実引数式並び > _{opt}) 識別子 整数 (< 式 >)
< 二項演算子 >	::= * / % + - < > <= >= == != && = * = / = % = + = - =
< 単項演算子 >	::= & * + - !
< 実引数式並び >	::= < 実引数式並び > , < 式 > < 式 >

開始規則は < 翻訳単位 > である。添字 *opt* が付加されている非終端記号は省略可能であることを意味する。この規則には演算子の優先順位を反映していない。曖昧な構文木を生成しないようにするために、演算子の優先順位と結合規則を次節で定義する。

3.2 演算子の優先順位

サブセットで使用する演算子の優先順位と結合規則を以下に示す。

演算子	結合規則
() []	左から右
! ++ -- + - * &	右から左
* / %	左から右
+ -	左から右
< > <= >=	左から右
== !=	左から右
= += -= *= /= %=	右から左

優先順位は上のものほど高い。上 2 行目は単項演算子であり、下 5 行目以降は二項演算子である。結合規則より、 $a + b + c$ は $((a + b) + c)$ と等価である。

第4章 型の整合性の検査

本章では、型の整合性を検査する一連の手法について述べる。

4.1 後置表記

後置表記は宣言を理解しやすくするため、および型の整合性を検査するための公理と推論規則に使われる。宣言子は識別子の宣言に付加する型情報 $[]$, $*$, $()$ の3つを意味する。C言語の宣言では、識別子の前後に宣言子をつけることができる。また、宣言子に優先順位が存在し、 $()$ によって宣言子の結合順序を変えることができる。これらの要因がC言語の宣言を理解しにくくしている。

4.1.1 後置表記の読み

提案する後置表記は、“識別子：基本型 宣言子並び”の形で表現される。基本型は int または void である。宣言子並びは宣言の宣言子が結合順序の遅い順に左から並ぶため、優先順位の知識や結合順序を変える $()$ は不要である。後置表記を左から読みさえすれば、一意な正しい宣言の読みを得られる。例えば、宣言 $\text{int } * \text{daytab}[13];$ は後置表記にすると、 $\text{daytab} : \text{int } * [13]$ となる。この後置表記は、 daytab は int へのポインタの配列型、と読む。後置表記でなければ int の配列へのポインタ型と読む可能性がある。

4.1.2 後置表記への変換

後置表記に変換するための入力である識別子の宣言は、 $\langle \text{外部宣言} \rangle$ (3.1節参照) を開始規則にして表現可能なものとして扱う。 $\langle \text{外部宣言} \rangle \rightarrow * t_1$ とし、 t_1 は終端記号の列かつ $t_1 \in T_1$ である。以後、規則に現れる識別子を ident 、整数を N として扱う。終端記号の集合は $\{ \text{INT}, \text{VOID}, \text{識別子}, \text{整数}, (,), [,], \{, \}, \{, \}, ; \}$ である。宣言子の優先順位は $*$ が最も低く、 $[]$ と $()$ が最も高い。

後置表記に変換する前に、 t_1 に以下の処理を行う。

1. t_1 から $;$ と $\{ \}$ 内の記号列を取り除き、宣言子の優先順位に従って t_1 に $()$ を挿入する
2. 取り除かれた記号列内の宣言は別途後置表記に変換する
3. t_1 に $()$ を挿入した終端記号の列を t_2 とする

$()$ を挿入するのは、後置表記に変換する関数を適用する際、曖昧さをなくするためである。 $()$ を挿入する例を挙げると、 $\text{int } * \text{daytab}[13];$ は $\text{int } * ((\text{daytab})[13])$, $\text{int main(void)} \{ \dots \}$ は int (main)(void) になる。 t_2 を後置表記に並び替えた列 t_3 を出力する関数 $rpn : T_2 \rightarrow T_3$ を定義する。また、識別子を含む t_2 の部分集合 t_4 から識別子を返す関数 $\text{name} : T_4 \rightarrow I$ も定義する。 $t_4 \in T_4$, I は識別子の集合。

$rpn(int\ t)$	$= name(t) : int\ rpn(t)$
$rpn(void\ t)$	$= name(t) : void\ rpn(t)$
$rpn(* (t))$	$= * rpn(t)$
$rpn((t)(t'))$	$= (rpn(t'))\ rpn(t)$
$rpn((t)(void))$	$= ()\ rpn(t)$
$rpn((t)[N])$	$= [N]\ rpn(t)$
$rpn((t))$	$= rpn(t)$
$rpn(int\ t,\ t')$	$= name(t) : int\ rpn(t),\ rpn(t')$
$rpn(void\ t,\ t')$	$= name(t) : void\ rpn(t),\ rpn(t')$
$rpn(ident)$	$= \varepsilon$
$rpn(\varepsilon)$	$= \varepsilon$
$name(* (t))$	$= name(t)$
$name((t)(t'))$	$= name(t)$
$name((t)(void))$	$= name(t)$
$name((t)())$	$= name(t)$
$name((t)[N])$	$= name(t)$
$name((t))$	$= name(t)$
$name(ident)$	$= ident$

なお、関数の展開によって複数の関数が現れた場合、任意の順番で関数の展開を行っても同じ結果が得られる。一つの関数の結果が他の関数に影響を及ぼさないからである。

3.1 節で与えられた宣言の規則は、 int の関数を返す関数型といった不正な宣言も記述可能である。不正な宣言を利用した式に推論規則を適用した検査の結果、型の整合性が取れている場合がある。不正な宣言は型に関する意味解析でコンパイルエラーとされるにもかかわらず、型の整合性が取れているという結果と矛盾する。したがって、不正な宣言を入力から除外する必要がある。正しい宣言のみに推論規則を適用するため、 t_3 を入力とし t_3 か空を返す関数 $check : T_3 \rightarrow T_3 \cup \{\varepsilon\}$ を定義する。

$check(ident : void)$	$= \varepsilon$
$check(t\ [N]\ ()\ t')$	$= \varepsilon$
$check(t\ ()\ ()\ t')$	$= \varepsilon$
$check(t\ ()\ [N]\ t')$	$= \varepsilon$
$check(t)$	$= t$

関数 rpn と関数 $check$ の実行で得られた t_3 を仮の公理とする。 t_3 が関数の場合、 t_3 とは別に引数も仮の公理とする。

4.1.3 後置表記への変換例

$()$ を補った宣言 $int\ (*(fp))(int\ x, int\ y)$ を後置表記に変換する。変換の手順を以下に示す。関数 $name$ の詳細な展開は省略する。

$ \begin{aligned} \text{rpn}(\text{int } (*\text{fp})(\text{int } x, \text{int } y)) &= \text{name}((*\text{fp})(\text{int } x, \text{int } y)) : \text{int } \text{rpn}((*\text{fp})(\text{int } x, \text{int } y)) \\ &= \text{fp} : \text{int } \text{rpn}((*\text{fp})(\text{int } x, \text{int } y)) \\ &= \text{fp} : \text{int } (\text{rpn}(\text{int } x, \text{int } y)) \text{rpn}(*\text{fp}) \\ &= \text{fp} : \text{int } (\text{name}(x) \text{ int} : \text{rpn}(x), \text{rpn}(y)) \text{rpn}(*\text{fp}) \\ &= \text{fp} : \text{int } (x : \text{int } \text{rpn}(x), \text{rpn}(\text{int } y)) \text{rpn}(*\text{fp}) \\ &= \text{fp} : \text{int } (x : \text{int}, \text{rpn}(\text{int } y)) \text{rpn}(*\text{fp}) \\ &= \text{fp} : \text{int } (x : \text{int}, \text{name}(y) : \text{int } \text{rpn}(y)) \text{rpn}(*\text{fp}) \\ &= \text{fp} : \text{int } (x : \text{int}, y : \text{int } \text{rpn}(y)) \text{rpn}(*\text{fp}) \\ &= \text{fp} : \text{int } (x : \text{int}, y : \text{int}) \text{rpn}(*\text{fp}) \\ &= \text{fp} : \text{int } (x : \text{int}, y : \text{int}) * \text{rpn}(\text{fp}) \\ &= \text{fp} : \text{int } (x : \text{int}, y : \text{int}) * \end{aligned} $
--

変換の結果, 仮の公理 $\text{fp} : \text{int } (x : \text{int}, y : \text{int}) *$, $x : \text{int}$, $y : \text{int}$ が得られる.

4.2 整合性の表明

型の整合性を検査する体系として, $e : (T, S)$ の三つ組を使う. e は 3.1 節の規則 < 式 > から表現可能な式を, T は 3.1 節の規則 < 宣言 > から表現可能な宣言の後置表記で e の型を, S は e が左辺値か非左辺値を表す. L を左辺値, NL を非左辺値, $S \in \{L, NL\}$ と定義する. 左辺値は代入式の左辺になれて変更可能な値を意味する. 式の中には左辺値を要求するものがあるため, S は体系に必要である. この体系を公理や推論規則の構成要素とする. 公理と推論規則の集合を定義し, 公理と推論規則を組み合わせて型の整合性を検査する.

4.2.1 公理と推論規則

公理は必ず成立する型の規則であり, 結論で表現される. 整数に関する公理と識別子に関する公理が存在する. これらをまとめた公理の集合を A とおく. 4.1.2 節で定義した関数 rpn と関数 check の実行で得られた仮の公理 t_3 は, 型の整合性の検査に必要な左辺値の情報が欠けている. 識別子は左辺値なので, t_3 を $\text{ident} : (T, L)$ の形に拡張し, 公理に追加する. 公理の例を以下に示す.

$$\frac{}{x : (\text{int}, L)} \quad \frac{}{13 : (\text{int}, NL)}$$

推論規則は演算前に必要な被演算子の型と演算後の式の型の規則であり, 前提結論で表現される. 推論規則の集合を R と置く. 本論文で用いる推論規則の一例を以下に示す.

$$\frac{e : (\text{int}, L) \quad e' : (\text{int}, S)}{e = e' : (\text{int}, NL)}$$

この推論規則は, 式 e が int 型で左辺値, かつ式 e' が int 型であるならば, 式 $e = e'$ は int 型で非左辺値, を表している.

4.2.2 型の整合性

式 e の型の整合性が取れているとは, $e : (T, S)$ が公理と推論規則から導出できる型 T と左辺値 S を持つことである. 例えば, 式 $x = y$ の型の整合性が取れているかについては, 公理から x

と y の型を調べることで $=$ の推論規則が適用でき、分かる。 x や y がもっと複雑な式であれば、推論規則を複数適用する。

4.2.3 公理と推論規則の集合

型の整合性を検査する公理の集合 A と推論規則の集合 R を以下に示す。 e, e' は 3.1 節で表現可能な式を、 T, T' は 3.1 節で表現可能な型を表す。 $ident$ は識別子、 num, N は整数を表す。 P は 0 個以上の引数の型、 p は 0 個以上の実引数式並びを表す。

公理

$\frac{}{ident : (T, L)}$ ※ $ident$ は宣言された識別子、 T はその宣言の型
 $\frac{}{0 : (void*, NL)}$ ※ $0 : void* : NL$ を必要とされるところのみ使用
 $\frac{}{num : (int, NL)}$

代入式の推論規則

$\frac{e : (int, L) \ e' : (int, S)}{e \ op \ e' : (int, NL)}$ ※ $op \in \{=, +, -, *, /, \% \}$
 $\frac{e : (T*, L) \ e' : (int, S)}{e \ op \ e' : (T*, NL)}$ ※ $op \in \{+, - \}$
 $\frac{e : (T*, L) \ e' : (T*, S)}{e = e' : (T*, NL)}$ $\frac{e : (T*, L) \ e' : (void*, S)}{e = e' : (T*, NL)}$

OR 演算子

$\frac{e : (int, S) \ e' : (int, S)}{e \ || \ e' : (int, NL)}$ $\frac{e : (T*, S) \ e' : (T'* , S)}{e \ || \ e' : (int, NL)}$
 $\frac{e : (int, S) \ e' : (T*, S)}{e \ || \ e' : (int, NL)}$ $\frac{e : (T*, S) \ e' : (int, S)}{e \ || \ e' : (int, NL)}$

AND 演算子

$\frac{e : (int, S) \ e' : (int, S)}{e \ \&\& \ e' : (int, NL)}$ $\frac{e : (T*, S) \ e' : (T'* , S)}{e \ \&\& \ e' : (int, NL)}$
 $\frac{e : (int, S) \ e' : (T*, S)}{e \ \&\& \ e' : (int, NL)}$ $\frac{e : (T*, S) \ e' : (int, S)}{e \ \&\& \ e' : (int, NL)}$

等値演算子

$\frac{e : (int, S) \ e' : (int, S)}{e \ op \ e' : (int, NL)}$ $\frac{e : (T*, S) \ e' : (T*, S)}{e \ op \ e' : (int, NL)}$
 $\frac{e : (T*, S) \ 0 : (void*, NL)}{e \ op \ 0 : (int, NL)}$ $\frac{0 : (void*, NL) \ e' : (T*, S)}{0 \ op \ e' : (int, NL)}$
 $\frac{e : (T*, S) \ e' : (void*, S)}{e \ op \ e' : (int, NL)}$ $\frac{e : (void*, S) \ e' : (T*, S)}{e \ op \ e' : (int, NL)}$ ※ $op \in \{==, != \}$

関係演算子

$\frac{e : (int, S) \ e' : (int, S)}{e \ op \ e' : (int, NL)}$ $\frac{e : (T*, S) \ e' : (T*, S)}{e \ op \ e' : (int, NL)}$ ※ $op \in \{<, >, <=, >= \}$

加法演算子

$\frac{e : (int, S) \ e' : (int, S)}{e \ op \ e' : (int, NL)}$ $\frac{e : (T*, S) \ e' : (int, S)}{e \ op \ e' : (T*, NL)}$ ※ $op \in \{+, - \}$

$$\frac{e : (int, S) \quad e' : (T^*, S)}{e + e' : (T^*, NL)}$$

$$\frac{e : (T^*, S) \quad e' : (T^*, S)}{e - e' : (int, NL)}$$

乗法演算子

$$\frac{e : (int, S) \quad e' : (int, S)}{e \text{ op } e' : (int, NL)} \quad \text{※ } op \in \{*, /, \%\}$$

単項演算子

$$\frac{e : (T, L)}{\&e : (T^*, NL)}$$

$$\frac{e : (T^*, S)}{*e : (T, L)}$$

$$\frac{e : (int, L) \quad e : (T^*, L)}{op \ e : (int, NL) \quad op \ e : (T^*, NL)} \quad \text{※ } op \in \{++, --\}$$

$$\frac{e : (int, S)}{op \ e : (int, NL)} \quad \text{※ } op \in \{+, -\}$$

$$\frac{e : (int, L) \quad e : (T^*, L)}{!e : (int, NL) \quad !e : (int, NL)}$$

後置演算子

$$\frac{e : (T^*, S) \quad e' : (int, S)}{e[e'] : (T, L)}$$

$$\frac{e : (T(P)^*, S) \quad p : (P, S)}{e(p) : (T, NL)} \quad \text{※ } P \text{ のすべての型と } p \text{ のすべての型が一致しなければならない}$$

$$\frac{e : (int, L) \quad e : (T^*, L)}{e \text{ op } : (int, NL) \quad e \text{ op } : (T^*, NL)} \quad \text{※ } op \in \{++, --\}$$

括弧

$$\frac{e : (T, S)}{(e) : (T, S)}$$

例外的な推論規則

$$\frac{e : (T(), L)}{e : (T()^*, NL)}$$

$$\frac{e : (T[N], L)}{e : (T^*, NL)}$$

推論規則を適用する順番によって結果が変わらないように、推論規則には優先順位がある。上記の集合では推論規則の優先順位の高い順に推論規則を並べている。代入式が最も高く、括弧が最も低い。式に同じ優先順位の二項演算子が2つ以上ある場合は、結合規則にしたがって式を分解する。例えば、 $a + b + c + d$ は $a + b + c$ と d に分解される。例外的な推論規則だけは、適用する時期が他の推論規則と違うため、優先順位を考える必要はない。

型の整合性の検査中に、ある識別子の生存期間を抜ける場合がある。そのときは、ある識別子の公理を削除する。また、同じレベルで識別子を再定義している場合は検査を中止する。

4.3 型の整合性の検査方法

3.1 節で記述可能な式 e 、公理の集合 A 、推論規則の集合 R を入力とし、空かエラーメッセージを出力するアルゴリズムを示す。

1. e に対応する推論規則を優先順位と結合規則にしたがって積み上げる。積み上げる複数の候補がある場合は左側を優先し、あとで右側を積み上げる。すべての式が識別子と整数に分解されるまで繰り返す。(このとき、推論規則の T と S はまだ空欄である。)
2. 公理をすべての識別子と整数に左側優先で適用する。適用できる公理が A に存在しない場合、識別子が宣言されていないというメッセージを出力し終了する。
3. 公理の型をもとに、積み上げた推論規則を適用する。 e に使ったすべての推論規則の型を導出できない場合、4 に進む。そうでなければ終了する。
4. 型を導出できない推論規則に対し、例外的な推論規則を適用する。例外的な推論規則を適用できない場合、型が正しくないというメッセージと適用できなかった式を出力し終了する。適用できた場合、3 に進む。

4.3.1 推論規則と公理を使った検査例

公理 $\frac{p : (int^*, L)}{p : (int^*, L)} \quad \frac{pp : (int^{**}, L)}{pp : (int^{**}, L)} \quad \frac{5 : (int, NL)}{5 : (int, NL)}$ 、推論規則の集合 R 、代入式 $p = *(pp + 5)$ を入力とする。公理と推論規則を適用し、4.3 のアルゴリズムの 1. までを行った結果と 3. までを行った結果を以下に示す。

$$\begin{array}{c}
 \frac{\frac{\frac{pp : (,) \quad 5 : (,)}{pp + 5 : (,)}}{(pp + 5) : (,)}}{*(pp + 5) : (,)}}{p : (,) \quad \frac{(pp + 5) : (,)}{*(pp + 5) : (,)}} \\
 \hline
 p = *(pp + 5) : (,)
 \end{array}$$

$$\begin{array}{c}
 \frac{\frac{\frac{pp : (int^{**}, L) \quad 5 : (int, NL)}{pp + 5 : (int^{**}, NL)}}{(pp + 5) : (int^{**}, NL)}}{*(pp + 5) : (int^*, L)}}{p : (int^*, L) \quad \frac{(pp + 5) : (int^{**}, NL)}{*(pp + 5) : (int^*, L)}} \\
 \hline
 p = *(pp + 5) : (int^*, NL)
 \end{array}$$

結果より、式に使ったすべての推論規則の T と S が導出できたので、型の整合性が取れていることが分かる。

おわりに

本論文では，型の整合性の検査に推論規則を使い，[1] の手法よりも適用範囲を拡大した．また，宣言子を後置表記に変換し，理解しやすい型の表示を提供した．

推論規則は教育用 C 言語 [4] のプログラムには対応しているが，構造体や共用体，不完全型には対応していない．C 言語 [3] まで適用範囲を拡大することが課題である．拡大すればコンパイラのツールから独立して使えるのではないかと考える．

ポインタの学習を支援するためのツールとして，ポインタに関する型の整合性を示せただけでは不十分である．誤ったポインタの扱いを防止するために，ポインタの指し先を示す機能を追加することが課題である．

謝辞

研究を行うにあたり，日頃から親切にご指導いただきました大山口通夫教授，山田俊行講師，ならびに何かとお世話になりました落合美子事務職員に深く感謝いたします。また，熱心に議論していただいた研究室の学生諸氏に感謝いたします。

参考文献

- [1] 谷口 高章, C 言語のポインタの静的解析に関する研究, 三重大学大学院 工学研究科修士論文, 2005.
- [2] B.W.Kernighan, D.M.Ritchie, 石田晴久 訳, プログラミング言語 C 第 2 版, 共立出版株式会社, 1989.
- [3] 財団法人 日本規格協会, 社団法人 情報処理学会, JIS X 3010:2003 プログラム言語 C, 2003.
- [4] 平井 智宏, 教育用 C 言語の設計と言語処理系の構成に関する研究三重大学大学院 工学研究科 修士論文, 2006.