ソフトウェア工学　78－4
(1991. 3. 8)

代数的言語による関数型言語のコンパイラの作成

上田 英邦　　東野 輝夫　　谷口 健一
大阪大学基礎工学部情報工学科

あらまし

我々の研究グループでは代数的言語ＡＳＬを定義し，ＡＳＬを用いたプログラムの設計開発支援システムとしてＡＳＬシステムの構築を行なってきた．ＡＳＬシステムには，ＡＳＬの部分クラスである関数型言語ＡＳＬ／Ｆのコンパイラを含む．ＡＳＬシステムを用いたプログラム開発の有用性等を調べるために，今回，そのＡＳＬ／Ｆコンパイラのコード生成部をＡＳＬ自身を用いて順序機械型プログラム（ＡＳＬ／ＡＳＭプログラム）として作成し，ＡＳＬシステム中のＣ言語で作成した既存のコード生成部の代わりに置き換えてみた．それによって支障は生じないこと，例えば実行時間も既存のＣプログラムのものと比較して同程度であることが確かめられた．

A functional language compiler written in algebraic language ASL

Hidekuni UEDA　　Teruo HIGASHINO　　Ken'ichi TANIGUCHI
Department of Information and Computer Sciences,
Faculty of Engineering Science, Osaka University

Abstract

We have defined ASL as an algebraic specification language and constructed a program development support system called ASL system. ASL system has a compiler for functional language ASL/F which is a subclass of ASL. In order to examine how ASL system is useful for the practical program developments, we have implemented a code generation program of ASL/F compiler by ASL, and compared it with the same program implemented by C. Some examples show that it takes almost the same time for them to generate object codes.

1 はじめに

代数的手法を用いたプログラム開発は，プログラム記述の正しさや具体化の検証が比較的容易かつ形式的に行なえるなどの特徴を持つことから近年その研究が盛んに行なわれている [1, 2].

筆者らのグループでは，代数的言語ＡＳＬを定め，ＡＳＬを用いたプログラムの設計・開発環境として，記述支援系，検証支援系，実行系からなるＡＳＬシステムを構築し，代数的手法の有用性について調べている [3].

これまでにＡＳＬシステムを用いて，クイックソートの仕様記述，詳細化 [3]，スクリーンエディタの仕様記述とプログラムへの詳細化 [4]，在庫管理プログラムの記述とその正しさの証明 [5] などが行なわれている．これらの例では，ＡＳＬシステムを用いてプログラム開発，検証を行なうことにより，高信頼性プログラムを作成している．本稿ではより実用規模のプログラム開発例として，関数型言語のコンパイラ (コード生成部のみ) をＡＳＬで作成する．

一般に，実用上大規模なプログラム開発を行なう場合，必ずしもプログラムの全てをＡＳＬのような形式的言語を用いて記述する必要はない．例えば，入出力処理や OS よりの記述が必要な場合はＣ言語で記述する方が簡便である．

プログラムのある部分の信頼性を高めたい場合，その部分はＡＳＬで記述，検証し，問題の本質とはあまり関係のない部分はＡＳＬ以外のプログラミング言語で記述すれば良い．またいわゆる形式的手法で作成されたプログラムと，既に存在するプログラムを結合 (例えばＣ言語のライブラリをリンク) できれば望ましい．

本研究では，ＡＳＬシステムの実行系の一つとしてＣ言語で実現されている関数型コンパイラ (ＡＳＬ／Ｆコンパイラ) のコード生成部をＡＳＬの部分言語である抽象的順序機械型 (ＡＳＬ／ＡＳＭ) で作成し，ＡＳＬシステム中の既存のＡＳＬ／Ｆコンパイラのコード生成部の代わりに置き換えてもシステムとして支障なくしようできることを示す．またＣ言語で作成されたコード生成部とＡＳＬにより作成されたコード生成部を比較し，ほぼ同程度の時間で目的プログラムが生成できることを示す．さらに，ＡＳＬシステムによるプログラム開発の有用性，問題点についても述べる．

2章では代数的言語ＡＳＬ及びＡＳＬを用いたプログラム開発支援システムについての概略，3章では関数型言語ＡＳＬ／Ｆとそのコンパイル法，4章ではＡＳＬ／Ｆコンパイラのコード生成部のＡＳＬによる実現，5章では4章で作成したコード生成部とＣ言語で作成したコード生成部との比較について述べる．

2 代数的言語ＡＳＬとＡＳＬシステム

本章では，代数的言語ＡＳＬと，ＡＳＬを用いたプログラム開発支援環境であるＡＳＬシステムについての概略を述べる．

2.1 代数的言語ＡＳＬ

代数的手法を用いたプログラム記述あるいは仕様記述においては，まず表現式 (文，項と呼ばれる場合もある) の構文を定義し，次にそれらの構文に従った表現式の合同関係を指定することによって，プログラムあるいは仕様の意味定義を行なう．

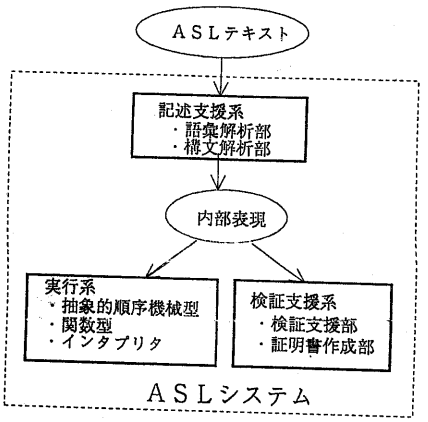

図 1: ＡＳＬシステムの概略図

代数的言語ＡＳＬでは，一つのプログラムあるいは仕様の記述 (ＡＳＬテキストと呼ぶ)t は，文脈自由文法 G(構文指定) と条件つきの公理集合 AX(意味指定) の対 $t = (G, AX)$ として表される．表現式の集合 E_G は G によって定め，E_G 上の合同関係は AX によって指定する．構文指定 G はユーザが自由に指定できることから比較的自由な書き方が許される言語である．

2.2 ＡＳＬシステムの概略

代数的言語ＡＳＬを用いたプログラム設計開発の支援環境として我々はＡＳＬシステムを構築している．システムは図 1で示されるように，記述支援系，検証支援系，実行系に大別することができる．ユーザが記述したＡＳＬテキストはまず記述支援系によって内部表現に変換される．記述支援系はさらにＡＳＬテキストを簡便に記述するためのマクロ命令及び省略記法の展開を行なう語彙解析部と，ＡＳＬテキスト $t = (G, AX)$ の構文指定 G に従って公理集合 AX の構文チェックを行なう構文解析部に分けられる．

記述支援系によって生成された内部表現を前提として，検証支援系，実行系を利用することができる．検証支援系は，検証支援部と証明書作成部に分けられる．検証支援部では，与えられた記述上で別に指定された性質が成り立っているかどうかの検証や，段階的詳細化を行なった場合のより下位レベルの記述が上位レベルの記述の正しい詳細化になっているかどうかの検証を対話的に行なえる．証明書作成部では，ユーザが行なった検証履歴をもとに証明書を作成する．

実行系には，抽象的順序機械型 (ＡＳＬ／ＡＳＭ) 及び関数型 (ＡＳＬ／Ｆ) の記述をそれぞれＣ言語に変換するコンパイラと，項書換え系の記述に対するインタプリタが提供されている．インタプリタは主にデバッグや検証などに用いられ，ＡＳＬ／ＡＳＭコンパイラ及びＡＳＬ／Ｆコンパイラは，与えられた表現式の値を効率良く求めるために用いられる．

3 関数型言語ＡＳＬ／Ｆとそのコンパイルの概略

以下ではＡＳＬ／Ｆプログラムとそのコンパイルの方法を簡単に説明する．

3.1 ＡＳＬ／Ｆプログラム

代数的言語ＡＳＬの部分クラスとして関数型言語ＡＳＬ／Ｆを次のように定義する．すなわち，ＡＳＬテキスト $t=(G,AX)$ の G と AX が以下に示す条件を満たす時，そのテキスト t と，P の計算指定項 u の対 (t,u) をＡＳＬ／Ｆプログラム P と呼ぶ．

1. 文脈自由文法 G は，基本関数（ＡＳＬではプリミティブとしてとり扱う関数），場合分け関数（if関数），定義関数（ユーザが指定する関数）の構文指定，並びにプリミティブなデータタイプの値の生成規則から成る．

2. 公理集合 AX は以下から成る．

   (a) 基本関数については

   $$0+0==1,\ 0+1==1,\ \ldots$$

   のように，引数の各値に対する関数値を指定する一般に無限個の公理から成る（テキストには書かれない）．

   (b) 場合分け関数（if 関数）については，それぞれのデータタイプについて，

   $$\text{if TRUE then } a \text{ else } b == a$$
   $$\text{if FALSE then } a \text{ else } b == b$$

   なる公理から成る（テキストには書かれない）．

   (c) 各定義関数に h ついて次のような形の公理が高々一つ存在する．

   $$h(x_1,\ldots,x_n) == \alpha[x_1,\ldots,x_n]$$

   ただし，h は定義関数名，$x_1,\ldots,x_n$ は異なる変数，$\alpha[x_1,\ldots,x_n]$ は変数が $x_1,\ldots,x_n$ のみ含む表現式である．ここでは定義文の左辺は便宜的に $h(x_1,\ldots,x_n)$ のようなプレフィックス表現を用いているが，G の構文規則に合致していればプレフィックス表現以外の記述をしても良い．

3.2 ＡＳＬ／Ｆプログラムのコンパイルの概略

$P=(t,u)$ をＡＳＬ／Ｆプログラムとする．ＡＳＬ／Ｆの公理の形から計算指定項 u の値を（もし存在するならば）失敗なく求めるための一つの方法は，ＡＳＬ／Ｆの公理を左辺から右辺への書き換え規則とみなし，次のような方法で求めることである [6].

- 基本関数の全引数と if 関数の第一引数は先に書き換える．
- 定義関数は引数を書き換える前に，その定義文自身を先に書き換える．

これらはそれぞれ引数の先行評価（値呼び），遅延評価（名前呼び）に対応する．

しかし，遅延評価法は 3.3で述べる環境の切替えによるオーバヘッドが大きいため効率が良くない．そこで、定義関数の値を求めるために必ず必要となる引数（必須引数という）については先行評価を行ない（必須引数先評価 [7]），その他の引数については遅延評価するようなコード生成を行なう．

3.3 目的プログラムの概略

ＡＳＬ／Ｆコンパイラが生成するＣ言語の目的プログラムでは，公理の項の形から書き換えにより直接値を求めるのではなく，このような書き換えに対応した手続きの呼びだしを繰り返し行なうことにより，目的の表現式の値を求める [6].

ＡＳＬ／Ｆプログラムにおける基本関数はその基本演算を行なう関数，定義関数はその定義関数の値を求める手続きとして実現されているＣ言語の関数（親手続きと呼ぶ）をそれぞれ呼び出すことで値を求める．また，定義関数に渡す引数の遅延評価を行なうために，遅延評価すべき引数の値が必要になった時点でその値を求めるための関数（子手続きと呼ぶ）を呼び出す．さらに，定義関数右辺の同形な部分項（共通部分項と呼ぶ）については，その値を求めるための手続き（サブルーチン手続きと呼ぶ）を呼び出す．

目的プログラムのすべての計算はスタックを用いて行なわれる．このスタック上には，各親手続きの呼出し時に動的に割り当てられるフレームという領域と，計算を行なうための一時的な作業領域がとられる．現在処理中の親手続きが持つフレーム（アクティブフレームと呼ぶ）の開始番地（いわゆるフレームポインタ）は fp という大域変数に記憶される．また大域変数 sp には，現在のスタックトップの位置（いわゆるスタックポインタ）が記憶される．

定義関数 f の親手続き m_f が持つフレームは次のものから構成される．

1. 親手続き m_f を呼び出した手続き（m_f の呼びだし手続きという）が使用していたフレームの開始番地を記憶する領域（以下 fp 領域と呼ぶ）．

2. f へ渡される引数の内容の記憶領域（以下引数領域と呼ぶ）．引数が先行評価される場合は値が記憶され，遅延評価される場合はその値を計算するための子手続きの名前（ラベルと呼ぶ）が記憶される．

3. f の定義文右辺の共通部分項の値を記憶する領域（以下ダグ領域と呼ぶ）．

1. の領域の内容は，実行中の親手続き m_f から m_f を呼び出した手続きに復帰する時のアクティブフレームの切替え（いわゆる環境の切替え）に用いられる．3. のダグ領域は，関数定義文右辺の共通部分項の重複計算の回避のために用いられる．すなわち，共通部分項の値が必要となった場合には，その共通部分項に対応するダグ領域に既に値が保存されていればその値を用い，そうでない時はその部分項の計算を行ない求まった値をダグ領域に保存する．

スタックには値及びラベルを記憶するための領域のほか，スタックの各番地の内容が値であるかラベルであるかを示

top (n)	スタックトップからn番めの内容を返す.
bottom (n)	アクティブフレームの底からn番めの内容を返す.
push (C)	スタックにCを積み, spの内容を1だけ増やす.
pop (n)	spの内容をnだけ減らす.

表 1: スタック操作関数

```
text ex;
  project primitives;
  int   -> g (int,int);
        -> h (int);
  int   a,b;
  g (a ,b) == h (b - 1) + a;
  h (a) == 2 * a;
end;
```

図 2: 例テキスト

すタグが付けられている. ある定義関数 fのある引数の値が必要になった場合, fに渡されている引数の記憶場所のタグ情報を見ることにより, 値であればその値を用い, ラベルであればそのラベルで示される子手続きを呼び出すことで, その引数の値を得ることができる. また共通部分項の重複計算回避のためにも使用される.

全ての基本関数の演算はスタックの内容に対して施される. 基本関数以外にも, スタックに対して特別な処理を行なうスタック操作関数がある (表 1参照).

例として図2のＡＳＬテキストに対する目的プログラムの動きを示す. 計算指定項に $g(1,4)$ を指定した場合 (値は 7 になる) について説明する. 計算過程のスタックの状態を図 3 に示す. スタックには値あるいはラベルが記憶される. 記憶されている値が fp を指すものは他と区別するために< 1 >のように表示している. 定義関数 gの引数は a が名前呼び, b が値呼びで, h の引数は名前呼びで行なうことにする.

まず gの引数 a, b の値 1,4 が入力として与えられ, それぞれスタックに積まれ, このとき fp,sp はそれぞれ 1,3 である (図 3-(a)). 次に gに対する親手続き m_gを呼ぶ. gではまず a の値を求めるための手続き $eval(1)$ を呼び出す. eval は名前呼びの引数に対してその値が必要になった時に用い, 次のように実現されている.

```
eval (n)
{
   if (fp == 1) push (bottom (n));
   else  {
         func = bottom (n);
         push (fp);
         fp = bottom (0);
         func ();   }
}
```

つまり, fp が 1 のときにはアクティブフレームの引数領域には入力された値が入っているのでその値を取り出すだけで良い. fp が 1 でない時には, 引数領域には子手続きのラベルが入っているので, m_fの呼びだし手続きのフレームに環境を切替え子手続きを呼び出す. この場合 fp が 1 であるので bottom (1) の値, すなわち 1 が積まれる.

次に定義関数 h を呼び出すために, まず fp の値を積み, 続いて h に渡す引数の値を求める子手続きのラベル $label1$ をスタックに積む (引数の名前渡しにあたる)(図 3-(b)). そして h の親手続き m_hを呼び出すが, この時に fp が親手続き m_hの持つフレームを指すように変更される (図 3-(c)).

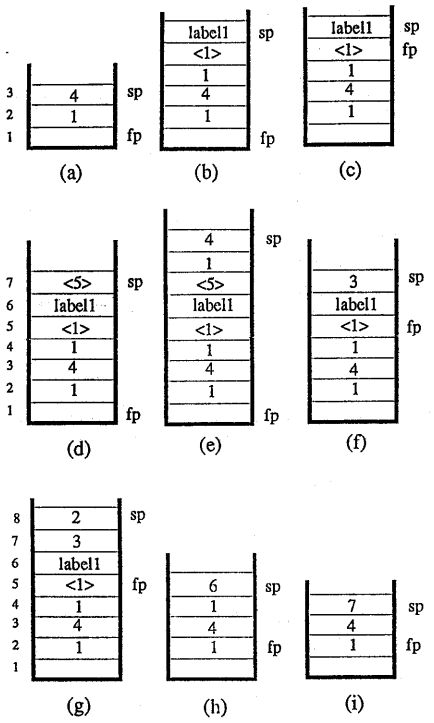

図 3: スタックの状態の変化

親手続き m_hではまず, a の値を必要とするため $eval(1)$ を行なう. この時子手続き $label1$ を呼び出すために, fp の値を push し, さらに fp の値を bottom (0) の値に変更することでアクティブフレームの切替を行なう (図 3-(d)).

子手続き $label1$ では, $b-1$ を計算するための手続きを記述している. まず $push(1)$ を行ない, 次に b の値が必要になるが, b は値呼びで gに渡されるため, $bottom(2)$ で引数領域から値を直接取り出しスタックに積む (図 3-(e)). 最後にスタックの内容に対して減算を行なう命令を実行し, さらに親手続き m_hのフレームが再びアクティブフレームに切替えられる (図 3-(f)).

m_hでは次に $push(2)$ を行ない, さらにスタックの内容に対して乗算を施した結果を m_hの使用していたフレーム領域を解放した後スタックに積む (図 3-(g)〜(h)).

この後, 親手続き m_gに戻りスタックに対して加算を行なう命令を実行し, 親手続きを終了する (図 3-(i)).

4 ＡＳＬ／Ｆコンパイラのコード生成部のＡＳＬ／ＡＳＭによる作成

前節で述べたような目的プログラムを生成するためにＡＳＬ／Ｆコンパイラでは前処理として各定義関数の公理をダグ化木表現 (4.1.1で後述) にし, 共通部分項の検出を行なっ

ている．また，各定義関数の引数が必須引数か否かを判定している (詳細は．4.2.1を参照)．これらの処理をＡＳＬ／Ｆコンパイラの前処理部と呼び，前処理部の情報を用いて実際にコード生成を行なう部分をコード生成部と呼んでいる．

以下では，ＡＳＬ／Ｆコンパイラのコード生成部を抽象的順序機械型 (以下ＡＳＬ／ＡＳＭ型, あるいはＡＳＭ型と呼ぶ) というスタイルで記述する．ＡＳＭ型の記述スタイルについての詳細は文献 [9] を参照されたい．記述スタイルとしてＡＳＭ型を選んだのは，コード生成の過程におけるデータ操作などを自然に記述できる，ＡＳＭ型プログラムに対するコンパイラ (ＡＳＬ/ ＡＳＭコンパイラ) を用いてＣ言語に変換した場合のプログラムの実行効率が比較的優れているという理由による．

コード生成の方法としては文献 [8] の方法を参考にし，中間機械と目的機械という仮想的な機械を設定する．まず中間機械のプログラム (中間プログラムと呼ぶ) を生成し，それから目的機械のプログラム (目的プログラムと呼ぶ) に変換するという二段階の変換方法をとる．すなわち，コード生成部は中間プログラム生成部と目的プログラム生成部から成る．ＡＳＬ／ＡＳＭで作成されたコード生成部は，Ｃ言語で作成された既存のＡＳＬ／Ｆコンパイラの前処理部と結合され，最終的なコンパイラを構成する．今回は目的機械がＣ言語の命令系列から成ることや変換の効率を考えた中間機械を設定している．

4.1 中間機械と目的機械の設定

4.1.1 項のダグ化木表現

まず，定義関数の定義文右辺の項に対して定義される木表現について説明する。項の木表現は，次のように再帰的に構成する。

1. 項が定数あるいは変数の時は，その定数名あるいは変数名をラベルとして持つ頂点である。
2. 項の最も外側に関数がかかっている場合は，根にその関数のラベル名を持つ頂点，その i 番めの子供に第 i 引数の木表現を持つ木である。

このように構成される木表現を用いて，項のダグ化木表現というものを以下のように構成する．

項のダグ化木表現
項の木表現中の頂点のうち以下のような同形な部分木をなす頂点同士は一つにまとめる．

1. 同一の定数あるいは変数名をラベルに持つ頂点は同形である．
2. 同一の基本関数名，定義関数名をラベルに持つ頂点は，その子供の頂点が全て同形である時のみ同形である．

項のダグ化木表現の頂点のうち親頂点を２個以上持つ頂点を共通頂点という．ダグ化木表現の各頂点には，その頂点に付けられているラベルや，そのラベルの種類 (定数, 変数, 基本関数, if関数, 定義関数)，頂点が共通頂点である場合はその共通頂点の番号，また，頂点が定義関数の場合にその関数呼び出しが末端再帰であるかどうかなどの情報が保持されている．

4.1.2 中間機械の概略とその命令系列

3.3で示したＡＳＬ／Ｆプログラムの目的プログラムの動きを抽象化した中間機械というものを設定する．中間機械の抽象レベルでは記憶領域の具体的なデータ構造は定義しない．3.3中の手続きを抽象化した概念としてプロセスというものを考える．プロセスは手続き呼び出しの際に発生し，その手続き終了時に消滅する．そのため，同じ手続き p の呼び出しが複数あったとしても，それらの呼出し時に生成されるプロセスは異なる．3.3で述べた親手続きの呼び出しに対応するプロセスにはフレームという領域が割り当てられる．これらのフレームには3.3で示したフレームの内容に加えて，一時的な作業領域を含むことにし，その領域へのアクセスはフレームの最上部 (フレームトップと呼ぶ) から行なう．ここで頂点が4.1.1で示した定義関数の定義文右辺のダグ化木表現の各頂点を指すとする．プロセスが持つ領域は頂点によって参照される．ただし，参照される領域は頂点とプロセスによって一意に決まるため，同じ頂点であってもプロセスが異なれば参照される領域は異なる．

中間機械の命令には領域の内容の転送命令, 条件分岐命令, 手続き呼びだし命令, フレーム領域の保存命令, 基本演算を行なう命令, ラベルの転送命令, 手続きからの復帰命令を設定した．中間機械の命令のいくつかを以下に示す．

[**save** *node_num*]
頂点 *node_num* に対応する領域の内容をフレームトップの内容と等しくする．

[**prm** *opname*]
opname に対応する基本演算をフレームトップに施す．

[**if** *seq_of_instr1* **else** *seq_of_instr2*]
フレームトップの内容が bool 値として真の時命令系列 *seq_of_instr1* を実行し，偽の時は *seq_of_instr2* を実行する．

[**mcall** *lbl num_of_arg size_of_frame*]
ラベル *lbl* の親手続きを呼び出す．*num_of_arg* は親手続き *lbl* のとる引数の数, *size_of_frame* は親手続き *lbl* 呼出し時のフレームのサイズである．

[**mcall_t** *lbl*]
親手続き *lbl* を呼び出すが，末端再帰を除去する [10] 手続きを行なう．

[**m_return** *offset*] , [**s_return**], [**c_return**]
それぞれ，親手続き, 子手続き, サブルーチン手続きからの復帰命令である．*offset* は親手続きからの復帰時に消去されるべきフレーム領域の大きさである．

4.1.3 目的機械とその命令系列

目的機械では，スタックの具体的構造を定義しスタック操作のためのスタック操作関数を用意する．実際には，目的言語がＣ言語であるのでスタック, 及びタグ領域はＣ言語のデータ構造として実現し，スタック操作関数もそのデータ構造に対して処理を行なうＣ言語の関数として実現する．

関数呼び出し, 条件判定や, 手続きからの復帰命令などはもともとC言語に備わっている機能を利用する. 基本演算を行なう関数は, スタックの内容に対して演算を行なう関数を呼び出す.

目的機械での具体的なデータ構造を次に示す. ただしSTACKSIZE はスタック領域の大きさである.

```
スタック領域          int   stack[STACKSIZE];
タグ領域              int   flag[STACKSIZE];
スタックポインタ      int   sp;
フレームポインタ      int   fp;
```

目的プログラムの命令系列は, これらのデータ構造を用いたC言語のプログラムで, if文, 表1で示したスタック操作関数, 関数呼び出し, return 文, 代入文, 及び中間機械の各手続きに対応する関数の定義文などの系列からなる. その詳細はここでは省略する.

4.1.4 中間機械の命令から目的機械の命令への詳細化

中間プログラムを目的プログラムへ変換するための変換方法として, ここでは中間機械の命令から目的機械の命令への詳細化を行なう. 中間機械の命令を目的のC言語に近いものに設定したため, 中間機械の命令から目的機械への命令への変換は容易である. 以下に詳細化の例を 2,3 示す. ただし TRUE,FALSE はそれぞれ真, 偽をとる値として既に定義されているものとする.

- **[save** *N* **]**
  頂点 *N* に対応するダグ領域の内容をスタックトップの内容と等しくし, また対応するタグの内容を真とする.

```
stack[fp+N] = top (0);
flag[fp+N] = TRUE;
```

- **[if** *S1* **else** *S2* **]**
  スタックトップの内容が真の時は命令系列 *S1* を実行し, 偽の時は命令系列 *S2* を実行する.

```
if (top (0))  {
        pop (1);
        S1
} else {
        pop (1);
        S2
}
```

- **[mcall** *LABEL ARG FRM* **]**
  親手続き *LABEL* を呼び出す. *ARG, FRM* はそれぞれ親手続き *LABEL* がとる引数の数, 親手続き *LABEL* 呼出し時のフレームのサイズである.

```
fp = sp-ARG;
for (i=1; i<=FRM; i++)
   flag[fp+i] = FALSE;
sp += FRM-ARG;
LABEL ();
```

親手続き呼出し直前に行なっていることは, まずフレームポインタを新しいフレームの開始番地に設定し, 次に親手続き呼び出し時に使用されるフレーム領域のタグを FALSE に設定する. そして, スタックポインタを新しいフレーム領域の最上部に設定して親手続きを呼び出す.

4.2 中間プログラム生成部

4.2.1 前処理部が生成するデータ構造

2.2で述べたように, ＡＳＬテキストはＡＳＬシステムの記述支援系によって構文解析され, その結果は既存のＡＳＬ／Ｆコンパイラの前処理部に利用しやすいような形のデータ構造に変換して取り込まれる.

既存のＡＳＬ／Ｆコンパイラの前処理部が生成する内部表現としてのデータ構造は次の通りである.

1. 定義関数の定義文右辺の項のダグ化木表現(詳細は 4.1.1 参照)
2. 定義関数名のリスト
3. 定義関数の引数情報リスト
   各定義関数の引数には前から順番に引数番号が付けられる. 引数情報リストには次のような情報が保持されている.
   (a) 各定義関数の引数の数
   (b) 各定義関数のある引数番号の変数が項のダグ化木表現のどの頂点に現れるか.
   (c) 各定義関数のある引数番号が必須であるかどうか.

4.2.2 中間プログラム生成部が前提とする関数

中間プログラム生成部が 4.2.1での内部表現のデータ構造を利用するために, C言語あるいはＡＳＬの関数型の記述で補助関数を実現している. 例えば次のような補助関数がある.

- **root**　定義関数名が与えられた時, その定義関数の定義文右辺の項のダグ化木表現の根の頂点を返す.

その他の補助関数については必要に応じて説明する.

ＡＳＬ／Ｆコンパイラの前処理部で生成される内部表現のデータ構造はC言語で次のように定義されている.

```
typedef struct  internal_struct {
   int   num_of_func;              /* 定義関数の数 */
   char  **defined_func_name;  /* 定義関数名リスト */
   TREE  **root;        /* 項のダグ化木表現のリスト */
     ... 以下略
} internal_rep;
```

前提とする補助関数の中で上記のデータ構造に直接アクセスする必要があるものはC言語で実現し, その関数を表す表現式の構文のみをＡＳＬのテキストの中で指定する. 例えば, 上述の補助関数 root はＡＳＬテキスト中でその構文指定を次のように行なう.

```
t_node        -> root (string, internal_rep);
```

ただし，t_node は internal_rep と同じようにコンパイラの前処理部で定義されている頂点を表す構造体を指すソート，string は文字列を指すソートである．このように構文指定をしておけば，この関数の本体の定義は単独に行なうか，ＡＳＭコンパイラの組み込み関数追加機能を利用してＡＳＬテキスト中で行なえば良い [9]．ここでは，後者の方法で次のように定義している．

```
(* define root (func, IntRep){
        register int    i;
        for (i=1; strcmp (func,
                IntRep->defined_func_name[i]); i++) ;
        return ((int) IntRep->root[i]);
}*)
```

このように定義される補助関数以外にもリスト操作等を行なう関数を基本関数として実現し前提として用いている．

4.2.3 中間プログラム生成部

中間プログラム生成部を抽象的順序機械型のスタイルで記述する．すなわち，コード生成の流れを一種の状態遷移と捉え，その状態遷移によってあるデータあるいは変数(状態成分という)の値がどのように変化していくかを記述する．中間プログラム生成部で用いている状態成分には以下のものがありそれぞれ抽象状態表す state 型の引数にとる．

```
Generated_Code (state)  現在までに生成された中間コード
Int_Rep (state)         内部表現
C_Last_LBL (state)      最も新しい使用済み命令系列ラベル
Master_List (state)     親手続きリスト
Temp_List (state)
Slave_List (state)      子手続きリスト
Subroutine_List (state) サブルーチンリスト
Instr_Seq (state)       現在生成中の手続きの中間コード
Arg (state)             Main 関数の引数情報
```

これらは，それぞれ initial という初期化関数によって初期状態の値が次の公理によって定義される．ただし nil は空リストを表す．

```
(*      状態成分の初期化        *)
Generated_Code (initial (InRep, Main)) == nil;
Int_Rep (initial (InRep, Main)) == InRep;
C_Last_LBL (initial(InRep,Main)) == last_label(InRep);
Master_List (initial (InRep, Main)) ==
                        defined_function (InRep);
Temp_List (initial (InRep, Main)) == nil;
Slave_List (initial (InRep, Main)) == nil;
Subroutine_List (initial (InRep, Main)) == nil;
Instr_Seq (initial (InRep, Main)) == nil;
Arg (initial (InRep, Main)) == nil;
```

InRep,Main は中間プログラム生成部に渡される入力データで，それぞれ内部表現のデータ，ＡＳＬ／Ｆの計算指定で与えられる Main 関数名である．これらは後述する目的関数の引数として与えられる．また，defined_function, last_label はいずれも内部表現データにアクセスするための補助関数で，前者は定義関数名のリスト，後者は内部表現中で使われたラベルの中で最も新しいものを返す．

状態成分の初期値を上記のように設定し，前処理をいくつか行なった後，実際のコード生成に入る．コード生成の状態遷移を制御するためのループ関数 final を次のように定義している．ただし，define 文は省略記法のためのマクロ命令であり，S は状態を表す変数である．

```
(*      コード生成制御のためのループ関数１      *)
  state         -> final (state);
  define  'IR' := 'Int_Rep (S)';
  define  'gen_node_subroutine' :=
     'gen_node (S ,head (Subroutine_List (S))) ';
  define  'gen_node_slave' := 'gen_ifnyv (S,
     SL_parent (Slave_List (S)), SL_son (Slave_List (S)))';
  define  'gen_node_master' :=
     'gen_node (S, root (head (Master_List(S)), IR))';
  final (S) ==
    if not_empty (Subroutine_List (S)) then
       final (update_lst(subroutine (gen_node_subroutine)))
    else if not_empty (Slave_List (S)) then
       final (update_lst (slave (gen_node_slave)))
    else if not_empty (Master_List (S)) then
       final (master (gen_node_master))
    else  S;
```

ループ関数 final は，もしサブルーチンリスト Subroutine_List が空でないならサブルーチン手続きをコード生成する状態遷移を行ない，そうでない場合，もし子手続きリスト Slave_List が空でないならば子手続きをコード生成する状態遷移を行なう．さらにそうでない場合は，もし親手続きリスト Master_List が空でない場合は親手続きをコード生成する状態遷移を行ない，最後に全てのリストが空になった時ループ関数 final は終了する．各手続きのコード生成を行なう状態遷移が終了した場合は，各リストの更新操作のための状態遷移を行なった後再びループ関数 final を呼ぶ．一般に，子手続きやサブルーチンの中で，さらに子手続き，サブルーチンの呼び出しがあり得るため，これらの命令系列生成中に子手続きリスト，サブルーチンリストは更新される可能性がある．

状態遷移 master は親手続きリスト Master_List の更新を行なう．master の状態遷移によって状態成分の値がどのように変化するかを公理を用いて次のように定義する．ただし，ＡＳＭ型では状態遷移によってその値が変化しない状態成分については定義する必要はない．

```
(*      親手続きの状態遷移 master       *)
  define  'Nd1' := 'root(head(Master_List(S)),IR)';
  state         -> master (state);
  Generated_Code (master (S)) ==
    assign (Generated_Code (S), get_label (Nd1,IR),
      { Instr_Seq (S) +
        [ m_return dug_size (get_label (Nd1, IR), IR) ]});
  Master_List (master (S)) ==
      if is_empty (Slave_List (S)) and
         is_empty (Subroutine_List (S)) then
            tail (Master_List (S))
      else Master_List (S);
  Instr_Seq (master (S)) == nil;
```

root は与えられた定義関数名の定義文右辺のダグ化木表現の根の頂点を返す，get_label は与えられた頂点に付けられている命令系列ラベルを返す補助関数である．また，head はリストの先頭の要素を返す，tail はリストの最初の要素を取り除いたリストを返す組み込み関数である．master の状態遷移後の生成された命令系列のリスト Generated_Code (master (S)) は，master の状態遷移を行なう前のリスト Generated_Code (S) に，新しくコード生成された親手続きの命令系列を表す状態成分 Instr_Seq (S) に親手続きからの復帰命令 **m_return** を加えた命令系列を加えたものになる．assign はある要素にラベルを付けて，リストに加える組み込み関数である．ここでは，ある手続きの命令系列にその手続き名を付けて，中間コード(命令系列からなるリ

スト) Generated_Code に加えるために用いられている。また親手続きリスト Master_List は，もし子手続きリストとサブルーチンリストが空ならばその最初の要素を取り除いたリストになり，そうでない場合は変化しない．Slave_List, Subroutine_List についても同様の定義がされるが詳細は省略する．

次に実際に命令系列の生成を行なう手続きについての説明をする．命令の生成は木表現の頂点に付けられているラベルの種類で決まる状態遷移によって行なわれる．それらの状態遷移を制御するループ関数 gen_node を示しておく．ただし N は頂点を表す．

```
(*      コード生成のためのループ関数 : gen_node      *)
 state          -> gen_node (state);
 gen_node (S,N) ==
   if is_const (N,IR) or is_var (N,IR) then
     gen_leaf (S,N)
   else if is_if (N,IR) then
     gen_if (add (add (gen_ifnyv (S,N,1),N,2),N,3),N)
   else if is_prm (N,IR) then
     gen_prm (gen_prm_sons (S,N,1), N)
   else if is_def (N,IR) then
     gen_def (gen_def_sons(gen_save_frame(S,N),N,1),N)
   else error (S);
```

gen_node では，与えられた頂点の種類によって適当な状態遷移を行なう．例えばある頂点が基本関数かどうかを判定する補助関数 is_prm が真であれば，その基本関数の全ての引数のコード生成をする gen_prm_sons を行ない，さらに基本関数本体を生成する gen_prm の状態遷移を行なう．

命令の生成を行なう状態遷移の例として，gen_prm の状態遷移によって変化する状態成分の定義を示す．ただし，node_label は頂点に付けられているラベルを返す補助関数である．

```
(*      基本関数本体の命令生成 : gen_prm          *)
  Instr_Seq (gen_prm (S, N)) ==
    { Instr_Seq (S) + [ prm  node_label (N, IR) ] };
```

状態遷移 gen_prm を行なった後の中間コードの命令系列 Instr_Seq (gen_prm (S,N)) は，状態遷移 gen_prm を行なう直前の状態における命令系列 Instr_Seq (S) に，基本関数の命令 [**prm** < 基本関数名 >] を加えた命令系列に変化する．

他の命令系列についても状態遷移を用いた同様な方法でコード生成するが，詳細は紙面の都合で省略する．

最後にＡＳＬプログラムでは，ＡＳＬテキスト中の公理のどんな値を計算するかを項で指定する必要がある (計算指定)．抽象的順序機械型ではこの指定を行なうものとして目的関数というものを定義する必要がある．

```
(*      目的関数の定義     *)
  string         Main;
  internal_rep  InRep;
  intermediate_code    ->
     Generate IMcode of string using internal_rep;
  Generate IMcode of Main using InRep ==
     Generated_Code (final (
        main(prepair(initial(InRep,Main)),Main)));
```

Generate IMcode of string using internal_rep はその構文指定で，中間コードを表すソート intermediate_code を返し，引数に文字列を表すソート string と内部表現を表すソート internal_rep をとることを定義している．Main と InRep はそれぞれソート string, internal_rep の変数である．目的関数 Generate IMcode of ... using ... InRep は，その引数として Main と InRep をとり，initial, prepair, main, final の状態遷移を順に行なった状態における状態成分 Generated_Code の値を返すことを定義している．つまり，目的関数はプログラムの入力と出力を定義するものであり，この場合入力として Main と InRep をとり，計算結果として Generated_Code の値を出力することを表す．

4.3 目的プログラム生成部

目的プログラム生成部は，入力として 4.1.2で定義した中間機械の命令系列とそれらに付けられたラベルから成る中間コードをとり，Ｃ言語の目的プログラムをファイルに生成する。中間機械の各命令からＣ言語への変換は 4.1.4で述べたように，中間機械の命令からＣ言語の命令系列への対応関係の定義にしたがって単純に行なえる．

まず目的プログラム生成部で用いている状態成分は以下の通りである．ただし fset,file はそれぞれファイル集合，ファイルを表す変数でシステムで用意されている．また，seq_of_instr は中間機械の命令系列を表すソートである．

```
(*    状態成分の定義  *)
fset          -> Fileset (state);  (* ファイル集合 *)
file          -> OutFile (state);  (* ファイル *)
intermediate_code
              -> ImCode (state); (* 中間コードの命令系列 *)
seq_of_instr -> Instr_Seq (state);  (* 命令系列 *)
int           -> StackPoint (state); (* スタックポインタ *)
arrayint      -> Stack (state);      (* スタック *)
```

Fileset はその状態における UNIX 上の全ファイルの集合を概念的に表すものであり，OutFile はコード出力のためにオープンしたファイルの状態を表す．ImCode は中間プログラム生成部の出力として渡される中間コードを表す．目的プログラムの生成は手続き単位に行なう必要があるため，ImCode から中間コードのある手続きを取り出し，その命令系列が Instr_Seq に記憶される．個々の命令の生成は，Instr_Seq から中間機械の命令を一つずつ取り出し，その命令の種類に応じたＣコードを生成する．StackPoint と Stack は，再帰的なコード生成を実現するための状態成分の退避に用い，それぞれスタックポインタとスタックを表す．これについては 5.1の 3で後述する．

上記の状態成分は次のように初期化される．

```
(*       状態成分の初期化          *)
  Fileset (init (IM, IR, FileName)) ==
                openset (finit, FileName, 'OUT');
  OutFile (init (IM, IR, FileName)) ==
                open (finit, FileName, 'OUT');
  ImCode (init (IM, IR, FileName)) == IM;
  Instr_Seq (init (IM, IR, FileName)) == inil;
  Stack (init (IM, IR, FileName)) ==
                element= 0 size= Stack_Size;
  StackPoint (init (IM, IR, FileName)) == 0;
```

openset, open はファイルをオープンした後のそれぞれファイル集合の状態，ファイルの状態を返す．目的プログラム生成の状態遷移の制御は以下のように行なう．

```
(*       目的プログラム生成の制御          *)
  gen_object (S) ==
     if is_empty (ImCode (S)) then S
    else gen_object (gen_body (
        gen_header (S, get_lst_name (ImCode (S)))));
```

```
(*      関数本体の生成の制御              *)
  gen_body (S) ==
    if is_empty (Instr_Seq (S)) then next_seq (S)
    else gen_body (next_instr (
                translate (S, head (Instr_Seq (S)))));
```

gen_object は親手続きなど各手続きの生成時に用いられるループ関数である．もしImCodeが空でない時，すなわち生成されていない手続きが存在する時には，まずgen_headerで関数のヘッダ部を生成し，続いてgen_bodyで関数の本体を生成する．これをImCodeが空になるまで繰り返す．

gen_bodyは関数本体を生成するためのループ関数である．gen_bodyが呼び出される時には，Instr_Seqの値は生成すべき手続きの中間機械の命令系列に変化している．Instr_Seqが空でない時は，Instr_Seqから命令を一つ取り出してその命令に対するコード生成を行なうtranslateの状態遷移を行ない，続いてInstr_Seqから命令をひとつだけ取り除くためにnext_instrを行なう．これをInstr_Seqが空になるまで繰り返す．

中間機械の命令からC言語への変換を行なう状態遷移translateは次のような形で定義されている．

```
(*      中間コードの各命令の変換のための制御     *)
translate (S, IS) ==
if pr_1 (IS) = SAVE       then Ogen_save (S, IS)
else if pr_1 (IS) = LOAD  then Ogen_load (S, IS)
else if pr_1 (IS) = PRM   then Ogen_prm (S, IS)
else if pr_1 (IS) = IF    then Ogen_if (S, IS)
                  ....中  略....
else gen_error (S, IS);

                  ....
OutFile (Ogen_prm (S, IS)) == Prt_prm (OF, pr_2 (IS));
                  ....
(* define Prt_prm (F, Lab){
      fprintf (F,"\t%s ();\n",(char * ) Lab);
      return (F);
}*)
```

中間機械の各命令ISは，1つのオプコードと複数のオペランドからなるタップル(組)として表現されている．pr_1はタップルの第一要素を取り出す組み込み関数で，これによりISの命令の種類を取り出すことができる．例えば，pr_1 (IS) がPRMすなわち基本関数ならば，Ogen_prmの状態遷移を行なう．Ogen_prmの状態遷移によって内容が変化する状態成分はOutFileつまり出力ファイルであり，その内容はPrt_prmの補助関数を呼び出した結果に変化する．Prt_prmではC言語のコードを直接ファイルに出力するプログラムをC言語で記述している．この部分はＡＳＬ／ＡＳＭの組み込み関数を使っても記述できるが，多少繁雑になるためC言語で記述した(C言語で記述してもこの関数の意味定義に本質的な影響は与えないと判断した)．

その他の命令についても同様な記述法でコード生成を実現している．

4.4 既存のコンパイラの前処理部との結合

4.2,4.3の中間プログラム生成部と目的プログラム生成部(以降ＡＳＬによるコード生成部と呼ぶ)を，C言語で記述されている既存のＡＳＬ／Ｆコンパイラの前処理部と結合する．図4で示されるようにＡＳＬによるコード生成部は，既存のＡＳＬ／Ｆコンパイラのコード生成部(以降Cによるコード生成部と呼ぶ)の代わりにそのまま用いることができる．Cによるコード生成部に渡されている内部表現などのデータ(a)と同じもの(b-1)を中間プログラム生成部に渡し，目的プログラム生成部には中間コード(b-2)が渡される．

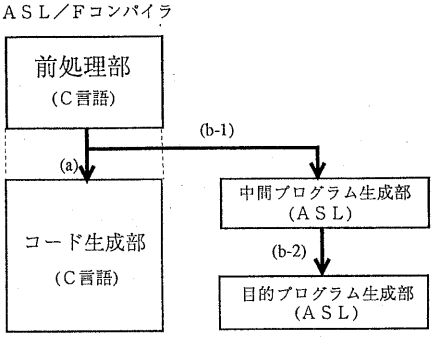

図4: ＡＳＬ／Ｆコンパイラの前処理部との結合

5 C言語記述のプログラムとの比較

Cおよびａｓｌによるコード生成部のプログラムについて以下の点について比較を行なった．

5.1 プログラムの読解性及び記述のしやすさ

1. 読解性

   今回記述に用いたＡＳＭ型は，状態遷移によって状態成分の値がどのように変化していくかを記述するスタイルである．すなわちコンパイラの処理の流れによってデータ(変数)の内容がいかに変化するかを明示的に記述している．このためテキスト中からデータ処理の制御・流れを容易に把握できることがC言語より優れている点である．例えば，ある処理の中である変数の内容を変更したい時，C言語ではその代入文はどの処理の中で何の目的で変更を行なうのかが一見しただけでは判断しにくい場合が多いが，ＡＳＭ型の場合はその代入文に状態遷移関数を明示する必要があるために判断しやすい場合が多い．

2. 記述のスタイル

   ＡＳＭ型の記述スタイルの性格上，いくつかのデータ(変数)がある時にそれらの値を刻々と変化させていくような問題に対しては優れた記述能力を持っている．しかし，状態成分(変数)の値を変化させるためには必ず状態遷移を必要とするため，状態成分の値を一つだけ変更するために状態遷移関数を用意しなければならないのはC言語に比べて記述が繁雑になった．また入出力処理については組み込み関数の充実度の差から，ＡＳＭ型の記述では繁雑になりがちであったが，この問題は組み込み関数を充実させることで解決するであろう．

実行時間の単位は秒，実行時間の括弧中は User Time

テストプログラム	ASL プログラム	C プログラム
最大公約数 (22 行)	0.08 (0.07)	0.17 (0.05)
クイックソート (50 行)	0.28 (0.25)	0.20 (0.10)
ヒープソート (60 行)	0.22 (0.17)	0.20 (0.07)

表 2: コード生成部の実行時間

3. 再帰処理

もう一つ問題となったのは，再帰処理に関する記述能力である．C言語では関数にいわゆるスコープという概念があり局所変数が持てるため再帰処理を自由に記述できる．しかし，ＡＳＭ型ではC言語の変数に当たる状態成分の数はプログラムの実行時において一定であり，いわゆる局所変数の概念はないため，再帰的な処理を行ないにくい．再帰的な処理には補助関数というものを用いて関数型で再帰的に記述することで実現する方法もあるが，再帰処理の過程で状態成分の値を逐次変化させることはできないという制限がある．そこでコード生成部の記述中のif文のthen節及びelse節に再帰的な処理を行なう必要がある部分では，再帰呼びだしに伴う局所的な変数を実現するために，スタックを用いて状態成分の退避を行なった．

上記3で述べた問題については，ＡＳＭコンパイラが状態成分の退避を実現できるような機能を提供するなどの解決法が必要であると考えられる．

5.2 コード生成部の実行時間及び記述量

ＡＳＬによるコード生成部は二段階の変換を行なっているのに対し，Ｃによるコード生成部は直接C言語の目的プログラムを生成するため，単純には比較できないが，目安として実行時間及び記述量の比較を行なった．

5.2.1 実行時間の比較

ＡＳＬ及びＣによるコード生成部の実行時間を表2に示す．実行時間は実際にコード生成プログラムがCPUを占有した時間(user time)とそのプログラムを実行させるためにCPUが要したその他の時間(cpu time)の合計で表される．Ｃによるコード生成部の実行時間中のcpu timeが占める割合が高いのはプログラムがシステムコールをいくつか用いていることによると推測できる．参考までにuser timeについて両者の比較をすると，ＡＳＬはＣの約1.4〜2.5倍の実行時間を要している．この結果は単純比較はできないが，両者の間に極端に開きはないと言える．尚，生成される目的プログラムはほぼ同一であるので，目的プログラムの実行効率は変わらない．

5.2.2 記述量の比較

Ｃによるコード生成部は約1,200行，ＡＳＬによるコード生成部は中間プログラム生成部が約1,000行，目的プログラム生成部が約700行であった．ＡＳＬのプログラム中(略記法を用いている)の文法定義文の数，公理の数については，中間プログラム生成部はそれぞれ120個，80個，また目的プログラム生成部はそれぞれ103個,69個であった．また，ＡＳＭコンパイラによって生成されたCプログラムはそれぞれ約1,100行,900行であった．この他にＡＳＬ／Ｆコンパイラの前処理部とのインターフェイスとなる補助関数(Ｃプログラム)が10個(約150行程度)であった．

6 まとめ

本報告では，大規模プログラムの開発においてその一部をＡＳＬを用いて代数的に実現することが可能であることを，コンパイラのコード生成部を実際に代数的手法で実現した例で示した．このことは，ＡＳＬを用いて代数的に実現したプログラムをＣ言語のプログラムから利用できるという意味も含んでいる．同じコード生成部を実現したＣ言語のプログラムとの比較の結果，プログラム記述の読解性はＡＳＬが，実行時間はＣ言語が有利であることが分かった．

代数的手法を用いたプログラム開発では，プログラムの信頼性を重視したい部分ではＡＳＬを用いて必要なら検証支援系を用いてフォーマルな議論を行ない，余り本質的でない部分やOSに依存するような部分，あるいは少しでも実行効率を上げたい部分の記述にはＣ言語などの言語を用いるといった方法が有効であると考えられる．

参考文献

[1] 酒井, 坂部, 稲垣: “コンパイラの代数的仕様記述と自動生成”, 信学論 (D-I),J73-D-I,12,pp.979-989(1990-12).

[2] K.Futatsugi, T.A.Goguen, J.-P.Jouannaud and J.Meseguer : “Principles of OBJ2”, Proc. 12th ACM symp. on POPL,pp.52-66 (1985)

[3] 東野, 関, 谷口: “代数的仕様から関数型プログラムの導出とその実行”, 情報処理,29, 8,pp.881-896(1988-08).

[4] 大蘆, 谷口: “スクリーンエディタの仕様記述と抽象的順序機械型プログラム”, 信学技法,SS89-24,pp.55-64(1989-11).

[5] 岡野, 北道, 東野, 谷口: “代数的言語ＡＳＬで記述した在庫管理プログラムとその正しさの証明”, 信学技法,SS90-29,pp.89-98(1990-12).

[6] 井上, 関, 谷口, 嵩: “関数型言語ＡＳＬ／Ｆとその最適化コンパイラ”, 信学論 (D),J67-D,4,pp.458-465(1984-04).

[7] 小野: “関数型プログラムのストリクト性関連解析と最適化技術”, 情報処理,29, 8,pp.862-871 (1988-08).

[8] 八木, 関, 谷口, 嵩: “関数型言語 ASL/F コンパイラの代数的記述とその詳細化” , 信学技報, AL85-67 (1985).

[9] 大蘆, 杉山, 谷口: “代数的言語ＡＳＬにおける抽象的順序機械型プログラムとその処理系”, 信学論 (D-I), J73-D-I,12,pp.971-978 (1990-12).

[10] Aho, A.V., Hopcroft,J.E. and Ullman, J.D.: “Data Structuures and Algolithms”, Addison-Wesley, pp.66-67 (1983).