情報処理学会論文誌 プログラミング Vol. 1 No. 3 19–33 (Oct. 2008)

制御フローの合流のための計算系[★1]

上 野 雄 大[†1] 大 堀 淳[†1]

本論文では，制御フローの合流を扱うための計算系を提案し，その計算系から導出される中間表現を構築する．制御フローの合流は，条件式などをコンパイルするうえで必須の機構である．たとえば，3 番地コードを用いた手続き型言語のコンパイラでは，条件式は，各分岐先のコードの実行の後，同一のラベルにジャンプするコードに翻訳される．しかしながら，関数型言語の中間表現として使用される継続渡し形式（CPS）や A-正規形などでは，制御フローの合流の機構が含まれていないため，条件式などを機械的に翻訳すると，分岐の後に続く継続が各分岐に複製されてしまう．この問題は，それら中間表現に対応する計算系に，分岐後の制御フローの合流に相当する規則を追加することで解決できるはずである．本論文では，この洞察に基づき，著者の 1 人によって示されたシーケント計算と A-正規形の対応を洗練し，論理和に対する左規則が唯一の前提（上式）を持つようなシーケント計算を構築する．この結果から，証明論的意味付けに立脚し，かつ制御フローの合流を取り扱うことができるコンパイラの中間表現と，その言語へのコンパイルアルゴリズムを導く．本論文の結果は関数型言語の実用的なコンパイラのより系統的な実現を可能にするものである．Standard ML の拡張言語である SML# のコンパイラの中間表現の 1 つは，この計算系に基づいて設計され，その中間言語へのコンパイルフェーズは本論文で示すコンパイルアルゴリズムを基礎に実装されている．

A Calculus for Merging Control Flows[★1]

Katsuhiro Ueno[†1] and Atsushi Ohori[†1]

This article proposes a new calculus for representing control flow merge, which is necessary for compiling case or conditional expressions, and develops a compiler intermediate language based on this calculus. Existing formalisms such as A-normal forms or CPS do not contain a mechanism for control flow merge. As a consequence, in these formalisms, mechanical application of a compilation algorithm to a conditional expression results in duplication of its continuation. This seems to be due to the lack of proper formalism for representing conditional expressions in the underlying calculi. This article refines the correspondence between the sequent calculus and A-normal forms shown by one of the authors, and develops a proof system where control flow merge is directly represented as a logical rule for disjunction. These results yield a term calculus suitable for compiler intermediate representations and a compilation algorithm that deals with control flow merge. This approach scales up to intermediate languages for practical compilers. The proposed calculus and the compilation algorithm have been successfully used in implementation of SML# – an extension of the Standard ML.

1. 序 論

高水準プログラミング言語のプログラムは，複雑な計算の構造を含む．高水準プログラミング言語を逐次機械で実行可能なコードにコンパイルするためには，この構造を持った式を，逐次的に実行される計算の列に変換する必要がある．

条件分岐を含まないプログラムの場合，この変換は，直感的には，以下のような処理の組合せと理解することができる．

(1) 式を計算実行の単位に対応する部分式に分割する．

(2) 各部分式に対して，その結果を保持する変数を導入し，その部分式の実行結果を変数に代入する文を生成する．

(3) 評価戦略に従い，生成した代入文を逐次的に並べる．

条件分岐を含む一般のプログラムでは，上記に加え，さらに以下の処理が必要となる．

(4) 各分岐部分とそれに続く式の範囲をブロックと同定し，ブロックにラベルを付け，ブロック間を分岐命令でつなぎ合わせる．

このような変換は，手続き型言語においては，3 番地コードへの変換処理として知られた一般的な処理である．たとえば C 言語の条件式

```
(x ? f(x + y) * 2 : x) * 3 + 4
```

は，この処理によって以下のようなコード列に変換される．

```
ifZero x goto L1
$3 = x + y
$4 = f($3)
$2 = $4 * 2
```

†1 東北大学電気通信研究所
Research Institute of Electrical Communication, Tohoku University

★1 本研究の一部は，科学研究費補助金基盤研究（C）（課題番号：19500021）の補助の下に行われたものである．

© 2008 Information Processing Society of Japan

```
     goto L2
L1:  $2 = x
L2:  $4 = $2 * 3
     $5 = $4 + 4
```

高階の関数を含む関数型言語に対しては，条件分岐を必要としない (1) から (3) の処理は，A-正規形への変換[7] として系統的な変換アルゴリズムが与えられ，さらに，その変換は自然演繹システムからある種のシーケント計算への変換に対応することが示され[17]，これらの結果によって，動的意味論と静的意味論（型）を保存する系統的変換の理論が与えられたといえる．たとえばラムダ式

$f(x+y)*2$

は

```
let $1 = x + y in
app f $1 is $2 in
let $3 = $2 * 2 in
$3
```

のような A-正規式に変換される*1．この変換結果から，文献 7) と 17) で示された結果は，条件分岐を含まない式に関しては，手続き型言語の 3 番地コードへの変換とほぼ同等の効果を達成していると確認できる．しかしながら，これら結果には制御ブロックやブロック間の分岐の概念が含まれていないため，分岐を必要とする条件式を定義に従って変換すると，非現実的なコードの複製が発生してしまう．たとえば，

```
(if x then 1 else 2) * 3 + 4
```

のような条件式を含むプログラムは，以下のような式に変換される．

```
case x of
  true =>
    let $3 = 1 in
    let $2 = $3 * 3 in
    let $1 = $2 + 4 in
    $1
| false =>
    let $3 = 2 in
    let $2 = $3 * 3 in
    let $1 = $2 + 4 in
    $1
```

*1 表記法は文献 17) による．`app x y is z in M` は関数 `x` に引数 `y` を適用し，結果を `z` に束縛し `M` に継続する計算を表す．結果の束縛に `let` 項を用いないのは，`app` 項は結果の束縛を含めてシーケント計算の $\supset$ の左規則に対応し，一方 `let` 項は Cut 規則に対応することを明確にするためである．

この例では，条件式に続く `([ ] * 3) + 4` の計算に対応する部分のコードが，`true` と `false` の双方のブランチに複製されてしまっている．この問題は，A-正規形の項 M が「単位計算 e を実行しその結果を変数 x に束縛し，それに続き M 実行する」という計算を表す

`let` x `=` e `in` A

の形の項に限定されていることによる．単位処理 e が組からの取出しや関数の適用などの場合はこの形で十分であるが，e が `case` x `of true =>` A_1`, false =>` A_2 の形の場合分けの場合，それぞれの分岐の実行コード A_i に続けて，それに続く A をそれぞれの分岐先にコピーし

`case` x `of true =>` $A_1; A$`, false =>` $A_2; A$

の形にしなければ，A-正規形の形に変形できない．同一の問題は，ラムダ式の CPS 式への変換においても発生する．

もちろんこの問題は，3 番地コードへの変換で通常実践されているとおり，各分岐コード A_i の実行結果を同一の変数で束縛し，それに続くコード A_0 に合流するように実装すれば解決する．実際，CPS 変換に関する文献 6) では，条件分岐後の継続を束縛するコードを条件分岐式の前に挿入し，各分岐先の末尾でその継続を呼び出すことで合流を表す解決法が示唆されており，また，A-正規形に関する文献 7) では `let` x `=` e `in` A の e の位置に `case` 式を認める拡張によって継続の複製を防ぐ方法が示唆されている．実際のコンパイラにおける CPS 変換や A-正規形への変換においても，そのような方法が多く採用されている．

本研究の目的は，条件式を低レベルコードに変換する際に必要となる条件分岐における制御フローの合流の問題を証明論的に分析し，以上のような既知の解決法を理解するための論理学的な枠組みを確立することである．我々の SML# 言語コンパイラ[22] 開発の経験は，この基礎の確立が，SML# 言語のように種々の先端機能で拡張された関数型言語の堅牢なコンパイラを実現するうえで重要な要素であることを示唆している．

この実現のために，本研究では，文献 17) で示された A-正規形の証明論を基礎に，証明論的分析を通じて制御フローの合流を可能にする型規則を導出し，それら規則を含む計算系を構築する．そのためにまず 2 章で，文献 17) で示された A-正規形の証明論を概観した

© 2008 Information Processing Society of Japan

後，選言命題に対する証明規則を分析する．3 章では，制御フローの合流を許すシーケント計算を与え，そこから，Curry-Howard 同型関係の原理に従い，制御フローの合流を許す計算系を抽出する．4 章では，定義した計算系の操作的意味論を与え，その健全性を示す．5 章では，自然演繹システムからの証明変換と，その同型関係にあるラムダ計算からのコンパイルアルゴリズムを与える．6 章では，以上の結果を基に設計され実装された SML# の中間表現の定義とコンパイルフェーズを報告する．7 章と 8 章で，関連研究およびまとめと今後の課題を述べる．

2. 条件分岐の証明論的分析

前章で論じた問題点の核心は，現在の A-正規形の条件式に対する構成規則が，それらの構文を実装するうえで必要な機構と対応していないことである．この問題は，文献 17) で示された論理学の証明論と A-正規形の対応を基礎とし，シーケント計算を通じて分析することができる．

文献 17) の基本的な洞察は，ある種のシーケント計算はラムダ計算より粒度の小さい計算ステップを表現しており，コンパイラの用いる A-正規形に対応する，というものである．この枠組みでは，計算は，値の束縛に対応する Cut 規則と計算ステップの実行に対応する左規則の組合せで表現される．たとえば論理積に関する左規則

$$(\wedge\text{-}L)\quad \frac{A, B, \Delta \vdash A}{A \wedge B, \Delta \vdash A}$$

は，以下のような組の要素取り出しを行う A-正規式の導出規則に対応する．

$$\frac{\Gamma, x\!:\!A, y\!:\!B \vdash M : A}{\Gamma, z\!:\!A \wedge B \vdash \textbf{proj}\ z\ \textbf{on}\ (x, y)\ \textbf{in}\ M : A}$$

したがって，組 M の左の要素を取り出す計算を表す A-正規式

let x = M **in**

proj x **on** (a,b) **in** a

は以下の証明図で表現される．

$$\frac{\Delta \vdash A \wedge B \quad \dfrac{A, B, \Delta \vdash A}{A \wedge B, \Delta \vdash A}\,(\wedge\text{-}L)}{\Delta \vdash A}\,(Cut)$$

上記の例から理解されるとおり，直積の場合，シーケント計算の左規則は，以下のような性質を持つ低レベルのコードの実行に正確に対応している．

- データ構造ごとに基本操作が用意されている．
- 基本演算は，項に対応する操作ではなく，変数に対して行う．
- 演算の結果は変数に束縛され，継続する計算で使用される．

この対応から，ラムダ計算が自然演繹システムに対応するように，低レベルの中間言語はシーケント計算に対応し，中間言語への変換は自然演繹システムからシーケント計算への証明変換に対応すると期待される．これが，文献 17) の基本となる洞察である．

しかしながら，我々の SML# の実装の経験から，この対応にはいくつかの問題点が明らかになった．最も大きな問題点は，シーケント計算の論理和に関する規則が，現実の条件分岐コードの実行に対応していないことである．シーケント計算の論理和に関する左規則は以下のように与えられる．

$$(\vee\text{-}L)\quad \frac{A, \Delta \vdash C \quad B, \Delta \vdash C}{A \vee B, \Delta \vdash C}$$

この規則は，変数に束縛された直和データの場合分けを行い，その結果に応じて分岐するコードと解釈できる．ラムダ計算と違い，シーケント計算では，この演算の 2 つの分岐先は，この演算に続く継続計算に対応する．その結果，ラムダ計算をこの規則を含むシーケント計算に翻訳すると，条件式の後の継続計算が，それぞれの条件式の後にコピーされてしまう．この問題を回避するためには，分岐して行う計算とそれに続く継続計算を分離し，分岐後に継続計算に合流するように変更する必要がある．そのための 1 つの方法は，この規則を，上で述べた低レベルの中間コードの構造にあった以下の規則に変更することである．

$$(\vee\text{-}L)\quad \frac{C, \Delta \vdash D}{A \vee B, A \supset C, B \supset C, \Delta \vdash D}$$

この規則は，変数に束縛された直和データのタグを分析し，変数で与えられたコードに分岐し，結果を継続計算に返すコードの表現と見なせる．さらにこの規則は，論理和の左導出を $\supset$ の左導出と同時に行う規則と解釈でき，健全であるのはもちろん，証明論的にも妥当なものである．

なお，継続計算への合流を以下のような形の規則によって表すことも考えられる．

$$(\vee\text{-}L)\quad \frac{A, \Delta \vdash C \quad B, \Delta \vdash C \quad C, \Delta \vdash D}{A \vee B, \Delta \vdash D}$$

この規則は，MLKit [2] や MinCaml [23] 等が採用している K-正規形[3] のように，`let` x `=` e `in` A の e の位置に `case` 式を認めることに対応する．これは，継続の複製を避けるために行われている方法の 1 つである，`let` を追加する方法に構文的に一致する．また，この規則は，著者の 1 人によるコードの証明システム[18] の規則の 1 つとしても用いられている．しかし，この規則は暗黙の Cut を含んでおり，論理結合子に関する推論規則であるにもか

© 2008 Information Processing Society of Japan

かわらず subformula property などの有用な性質を満たしていない．

この論理和の規則以外に，我々の SML# の実装の経験から，さらに以下の問題点が明らかとなった．

（1）　前に議論したとおり，コンパイラの中間言語では変数を演算の対象とする．この観点から，$\supset$ の左規則に現れる関数の引数に対応する項を変数に制限する必要がある．

（2）　ネストした値の生成は変数を介して実行され，$(x,(y,z))$ のようなネストした構造を持つ値項は許さない．

左規則にこれらの改良を加え，さらに，現実の中間言語に対応した変数のスコープ規則に対応させるために，基礎となるシーケント計算系として，Gentzen のオリジナルの体系ではなく，文献 17) で採用されている Kleene によるシーケント計算系 G3 [14] [Ch.XV, §80] をとると，現実のコンパイラの中間表現に対応した計算体系が実現できる．

3. 制御フローの合流を許す計算系

以上の洞察に基づき，制御フローの合流を可能にする型規則およびそれを含む計算系を構築する．本章では，まず Kleene によるシーケント計算系 G3 に対して前章で述べた改良を加えた証明システム $\mathcal{GK}^{\mathcal{M}}$ およびそれを型システムとする計算系 λ^M を定義する．次に，その計算系における A-正規形である A^M-正規形と，それに対応する証明システム $\mathcal{GKA}^{\mathcal{M}}$ を与える．

最初に，本論文中で用いる記法を定義する．型は，論理との対応を明確にするため，以下の文法で定義する．

$$\tau ::= b \mid \tau \supset \tau \mid \tau \wedge \tau \mid \tau \vee \tau$$

ここで b は基本型を表す．型環境は Γ と表記し，変数から型への有限写像とする．写像 Γ によって x と対応付けられた要素を $\Gamma(x)$ と書く．写像 Γ の定義域を $dom(\Gamma)$ と書く．写像 Γ について，$dom(\Gamma') = dom(\Gamma) \cup \{x\}$ かつ $\Gamma'(x) = \tau$ となる写像 Γ' を $\Gamma, x{:}\tau$ と書く．本論文中で定義するすべての項において，束縛変数は互いに異なりまた自由変数とも異なると仮定する．ある項 M 中の変数 x のすべての出現を M' に置き換えたものを $[M'/x]M$ と書く．

Kleene によるシーケント計算系 G3 に対して前章で述べた改良を加えた計算系 λ^M を定義する．この計算系における項を以下のように定義する．

$$\begin{aligned} M ::= {} & c^b \mid x \mid \lambda x.M \mid (M,M) \mid \mathbf{in1}(M) \mid \mathbf{in2}(M) \\ & \mid \mathbf{app}\ x\ x\ \mathbf{is}\ x\ \mathbf{in}\ M \\ & \mid \mathbf{proj}\ x\ \mathbf{on}\ (x,x)\ \mathbf{in}\ M \\ & \mid \mathbf{case}\ x\ \mathbf{of}\ x,x\ \mathbf{is}\ x\ \mathbf{in}\ M \\ & \mid \mathbf{let}\ x = M\ \mathbf{in}\ M \end{aligned}$$

(M_1, M_2)，$\mathbf{in1}(M)$，$\mathbf{in2}(M)$ は，それぞれ，M_1 と M_2 の組，M の直和の左への埋め込み，M の直和の右への埋め込みを表す．$\mathbf{app}\ x\ y\ \mathbf{is}\ z\ \mathbf{in}\ M$ は $\supset$ の左規則に対応する項であり，変数 x に束縛された関数を変数 y に束縛された値に適用し，結果を z に束縛し M に継続する計算を表す．$\mathbf{proj}\ x\ \mathbf{on}\ (y,z)\ \mathbf{in}\ M$ は $\wedge$ の左規則に対応する項であり，変数 x に束縛された組型データの各要素を変数 y と変数 z に束縛し M に継続する計算を表す．$\mathbf{case}\ x\ \mathbf{of}\ f,g\ \mathbf{is}\ y\ \mathbf{in}\ M$ は $\vee$ の左規則に対応する項であり，変数 x に束縛された直和データ（タグ付きデータ）のタグを分析し，左への埋め込みであれば変数 f に束縛された関数をその値を値に適用し結果を y に束縛し，右への埋め込みであれば変数 g に束縛された関数をその値に適用し結果を y に束縛し，いずれの場合も M に継続する計算を表す．

この計算系の型付け規則となる証明システム $\mathcal{GK}^{\mathcal{M}}$ を**図 1** に示す．$\mathcal{GK}^{\mathcal{M}}$ は元となったシーケント計算に対して以下の点について異なる．

- すべての左規則は上式を 1 つのみ持つ．これは，各演算は変数のみを演算の対象とし逐次的に並ぶことを表している．
- 論理和の左規則 ($\vee$-L) を，前章で述べた制御フローの合流を表現できる規則で置き換えている．

次に，この計算系から値と演算のネストを排除した正規形である A^M-正規形を定義する．この正規形は，演算だけでなく組などのネストする値の構築についても，値の取り出しを変数からのみに制限し，かつネストする **let** 項や変数の複写を何らかの戦略に基づいてすべて展開したものである．A^M-正規形の項 A を以下のように与える．

$$\begin{aligned} A ::= {} & T \\ & \mid \mathbf{app}\ x\ x\ \mathbf{is}\ x\ \mathbf{in}\ A \\ & \mid \mathbf{proj}\ x\ \mathbf{on}\ (x,x)\ \mathbf{in}\ A \\ & \mid \mathbf{case}\ x\ \mathbf{of}\ x,x\ \mathbf{is}\ x\ \mathbf{in}\ A \\ & \mid \mathbf{let}\ x = V\ \mathbf{in}\ A \\ T ::= {} & x \mid V \\ V ::= {} & c^b \mid \lambda x.M \mid (x,x) \mid \mathbf{in1}(x) \mid \mathbf{in2}(x) \end{aligned}$$

© 2008 Information Processing Society of Japan

$$(\text{axiom})\quad \Gamma \triangleright c^b : b \qquad (\text{taut})\quad \Gamma, x\!:\!\tau \triangleright x : \tau \qquad (\supset\text{-R})\quad \frac{\Gamma, x\!:\!\tau_1 \triangleright M : \tau_2}{\Gamma \triangleright \lambda x.M : \tau_1 \supset \tau_2}$$

$$(\wedge\text{-R})\quad \frac{\Gamma \triangleright M_1 : \tau_1 \qquad \Gamma \triangleright M_2 : \tau_2}{\Gamma \triangleright (M_1, M_2) : \tau_1 \wedge \tau_2} \qquad (\vee\text{-R}i)\quad \frac{\Gamma \triangleright M : \tau_i \qquad (i \in \{1,2\})}{\Gamma \triangleright \mathbf{in}i(M) : \tau_1 \vee \tau_2}$$

$$(\supset\text{-L})\quad \frac{\Gamma, x\!:\!\tau_1 \supset \tau_2, y\!:\!\tau_1, z\!:\!\tau_2 \triangleright M : \tau_3}{\Gamma, x\!:\!\tau_1 \supset \tau_2, y\!:\!\tau_1 \triangleright \mathbf{app}\ x\ y\ \mathbf{is}\ z\ \mathbf{in}\ M : \tau_3}$$

$$(\wedge\text{-L})\quad \frac{\Gamma, x\!:\!\tau_1 \wedge \tau_2, y\!:\!\tau_1, z\!:\!\tau_2 \triangleright M : \tau_3}{\Gamma, x\!:\!\tau_1 \wedge \tau_2 \triangleright \mathbf{proj}\ x\ \mathbf{on}\ (y,z)\ \mathbf{in}\ M : \tau_3}$$

$$(\vee\text{-L})\quad \frac{\Gamma, x\!:\!\tau_1 \vee \tau_2, f\!:\!\tau_1 \supset \tau_3, g\!:\!\tau_2 \supset \tau_3, y\!:\!\tau_3 \triangleright M : \tau_4}{\Gamma, x\!:\!\tau_1 \vee \tau_2, f\!:\!\tau_1 \supset \tau_3, g\!:\!\tau_2 \supset \tau_3 \triangleright \mathbf{case}\ x\ \mathbf{of}\ f, g\ \mathbf{is}\ y\ \mathbf{in}\ M : \tau_4}$$

$$(\text{Cut})\quad \frac{\Gamma \triangleright M_1 : \tau_1 \qquad \Gamma, x\!:\!\tau_1 \triangleright M_2 : \tau_2}{\Gamma \triangleright \mathbf{let}\ x = M_1\ \mathbf{in}\ M_2 : \tau_2}$$

図 1　Proof system $\mathcal{GK}^{\mathcal{M}}$
Fig. 1　Proof system $\mathcal{GK}^{\mathcal{M}}$.

この計算系の型付け規則となる証明システム $\mathcal{GKA}^{\mathcal{M}}$ を**図 2** に示す[*1]．$\mathcal{GKA}^{\mathcal{M}}$ では右規則のネストを許さないため，$\mathcal{GKA}^{\mathcal{M}}$ の右規則は ($\supset$-R) を除き上式をともなわない公理となる．

A^M-正規形の集合は，λ^M 項から λ^M 項への型なしの簡約関係における正規形の集合として特徴付けることができる．簡約関係を定義するために，まず簡約の対象となる項を認識するための文脈 C，N および U を以下のように定義する．

$$\begin{aligned} C ::= & [\,]\ \mid\ \lambda x.C\ \mid\ (C, M)\ \mid\ (M, C)\ \mid\ \mathbf{in}i(C) \\ & \mid\ \mathbf{app}\ x\ x\ \mathbf{is}\ x\ \mathbf{in}\ C \\ & \mid\ \mathbf{proj}\ x\ \mathbf{on}\ (x, x)\ \mathbf{in}\ C \\ & \mid\ \mathbf{case}\ x\ \mathbf{of}\ x, x\ \mathbf{is}\ x\ \mathbf{in}\ C \\ & \mid\ \mathbf{let}\ x = C\ \mathbf{in}\ M \end{aligned}$$

Values

$$(\text{axiom})\quad \Gamma \triangleright c^b : b \qquad (\text{taut})\quad \Gamma, x\!:\!\tau \triangleright x : \tau \qquad (\supset\text{-R})\quad \frac{\Gamma, x\!:\!\tau_1 \triangleright A : \tau_2}{\Gamma \triangleright \lambda x.A : \tau_1 \supset \tau_2}$$

$$(\wedge\text{-R})\quad \Gamma, x\!:\!\tau_1, y\!:\!\tau_2 \triangleright (x, y) : \tau_1 \wedge \tau_2 \qquad (\vee\text{-R}i)\quad \Gamma, x\!:\!\tau_i \triangleright \mathbf{in}i(x) : \tau_1 \vee \tau_2 \quad (i \in \{1,2\})$$

Computation Terms

$$(\supset\text{-L})\quad \frac{\Gamma, x\!:\!\tau_1 \supset \tau_2, y\!:\!\tau_1, z\!:\!\tau_2 \triangleright A : \tau_3}{\Gamma, x\!:\!\tau_1 \supset \tau_2, y\!:\!\tau_1 \triangleright \mathbf{app}\ x\ y\ \mathbf{is}\ z\ \mathbf{in}\ A : \tau_3}$$

$$(\wedge\text{-L})\quad \frac{\Gamma, x\!:\!\tau_1 \wedge \tau_2, y\!:\!\tau_1, z\!:\!\tau_2 \triangleright A : \tau_3}{\Gamma, x\!:\!\tau_1 \wedge \tau_2 \triangleright \mathbf{proj}\ x\ \mathbf{on}\ (y,z)\ \mathbf{in}\ A : \tau_3}$$

$$(\vee\text{-L})\quad \frac{\Gamma, x\!:\!\tau_1 \vee \tau_2, f\!:\!\tau_1 \supset \tau_3, g\!:\!\tau_2 \supset \tau_3, y\!:\!\tau_3 \triangleright A : \tau_4}{\Gamma, x\!:\!\tau_1 \vee \tau_2, f\!:\!\tau_1 \supset \tau_3, g\!:\!\tau_2 \supset \tau_3 \triangleright \mathbf{case}\ x\ \mathbf{of}\ f, g\ \mathbf{is}\ y\ \mathbf{in}\ A : \tau_4}$$

$$(\text{Cut})\quad \frac{\Gamma \triangleright V : \tau_1 \qquad \Gamma, x\!:\!\tau_1 \triangleright A : \tau_2}{\Gamma \triangleright \mathbf{let}\ x = V\ \mathbf{in}\ A : \tau_2}$$

図 2　Proof system $\mathcal{GKA}^{\mathcal{M}}$
Fig. 2　Proof system $\mathcal{GKA}^{\mathcal{M}}$.

$$\begin{aligned} & \mid\ \mathbf{let}\ x = M\ \mathbf{in}\ C \\ U ::= & ([\,], M)\ \mid\ (M, [\,])\ \mid\ \mathbf{in}i([\,]) \\ N ::= & U\ \mid\ \mathbf{let}\ x = [\,]\ \mathbf{in}\ M \end{aligned}$$

ここで $[\,]$ は項が埋め込まれるべき穴を表す．文脈 C に項 M を埋め込んで得られる項を $C[M]$ と書く．直感的には，C は任意の部分項にマッチする文脈，U はネストした値にマッチする文脈，N はネストした演算にマッチする文脈である．これらを用いて，A^M-変形規則 $\Longrightarrow$ を**図 3** のように定義する．この変形規則において，文脈 N に関する規則はネストした式を直列化するための規則であり，文献 7) で定義された A-簡約規則の一部と対応する．文脈 U に関する規則は，値の構築を変数を介して行うようにするために，すべてのネストした値構築子に対してその結果を変数に代入するための **let** 項を挿入する規則である．最後の変形規則は変数間のコピーを除去する規則である．この規則は，簡約前に含まれるコピー

*1 $\mathcal{GKA}^{\mathcal{M}}$ の証明システムの定義では，可能な証明を項を用いて制限している．これは，定義から本質的でない複雑さを排除するための略記法である．項による制限を導出の制限として展開することによって，A^M-正規形の構文の定義と厳密に対応する証明システムの定義全体を得ることができる．

© 2008 Information Processing Society of Japan

$$N[\textbf{app}\ x\ y\ \textbf{is}\ z\ \textbf{in}\ M] \Longrightarrow \textbf{app}\ x\ y\ \textbf{is}\ z\ \textbf{in}\ N[M]$$
$$N[\textbf{proj}\ x\ \textbf{on}\ (y,z)\ \textbf{in}\ M] \Longrightarrow \textbf{proj}\ x\ \textbf{on}\ (y,z)\ \textbf{in}\ N[M]$$
$$N[\textbf{case}\ x\ \textbf{of}\ f,g\ \textbf{is}\ y\ \textbf{in}\ M] \Longrightarrow \textbf{case}\ x\ \textbf{of}\ f,g\ \textbf{is}\ y\ \textbf{in}\ N[M]$$
$$N[\textbf{let}\ x = M_1\ \textbf{in}\ M_2] \Longrightarrow \textbf{let}\ x = M_1\ \textbf{in}\ N[M_2]$$
$$U[c^b] \Longrightarrow U[\textbf{let}\ x = c^b\ \textbf{in}\ x]$$
$$U[\lambda x.M] \Longrightarrow U[\textbf{let}\ y = \lambda x.M\ \textbf{in}\ y]$$
$$U[(M_1,M_2)] \Longrightarrow U[\textbf{let}\ x = (M_1,M_2)\ \textbf{in}\ x]$$
$$U[\textbf{in}i(M)] \Longrightarrow U[\textbf{let}\ x = \textbf{in}i(M)\ \textbf{in}\ x]$$
$$\textbf{let}\ x = y\ \textbf{in}\ M \Longrightarrow [y/x]M$$

図 3　A^M-変形規則
Fig. 3　A^M-transformation rules.

の除去だけでなく，簡約の過程で生じる中間変数のコピーを除去するためにも用いられる．

A^M-簡約 $\longrightarrow$ は，$\Longrightarrow$ を用い，項から項への書き換えとして以下のように定義する．

$$\frac{M \Longrightarrow M'}{C[M] \longrightarrow C[N]}$$

$\longrightarrow$ の反射的推移的閉包を $\stackrel{*}{\longrightarrow}$ と書く．$M \longrightarrow M'$ となる M' が存在しないとき，$M \not\longrightarrow$ と書く．また，この簡約関係の性質を示すために，途中まで正規化された状態を表す文脈である P を以下のように定義する．

$$\begin{aligned} P ::= &\ [\ ] \\ |&\ \textbf{app}\ x\ x\ \textbf{is}\ x\ \textbf{in}\ P \\ |&\ \textbf{proj}\ x\ \textbf{on}\ (x,x)\ \textbf{in}\ P \\ |&\ \textbf{case}\ x\ \textbf{of}\ x,x\ \textbf{is}\ x\ \textbf{in}\ P \\ |&\ \textbf{let}\ x = V\ \textbf{in}\ P \end{aligned}$$

以下の補題は，P の構造に関する帰納法より簡単に示すことができる．

補題 3.1. 任意の N，P，M について，$N[P[M]] \stackrel{*}{\longrightarrow} P[N[M]]$.

補題 3.2. $\mathcal{GK}^{\mathcal{M}} \vdash \Gamma \triangleright P[T] : \tau$ ならば $\mathcal{GKA}^{\mathcal{M}} \vdash \Gamma \triangleright P[T] : \tau$.

簡約関係 $\longrightarrow$ は合流性を持たない．したがって，ある λ^M 項が与えられたとき，その A^M-正規形は文脈 C の取り方によって複数存在しうる．たとえば，

$$(\textbf{app}\ f\ x_1\ \textbf{is}\ y_1\ \textbf{in}\ y_1, \textbf{app}\ g\ x_2\ \textbf{is}\ y_2\ \textbf{in}\ y_2)$$

という λ^M 項を正規化するとき，組の左の要素を先に簡約するならば，

$$\begin{aligned} &(\textbf{app}\ f\ x_1\ \textbf{is}\ y_1\ \textbf{in}\ y_1, \textbf{app}\ g\ x_2\ \textbf{is}\ y_2\ \textbf{in}\ y_2) \\ &\longrightarrow \textbf{app}\ f\ x_1\ \textbf{is}\ y_1\ \textbf{in}\ (y_1, \textbf{app}\ g\ x_2\ \textbf{is}\ y_2\ \textbf{in}\ y_2) \\ &\longrightarrow \textbf{app}\ f\ x_1\ \textbf{is}\ y_1\ \textbf{in}\ \textbf{app}\ g\ x_2\ \textbf{is}\ y_2\ \textbf{in}\ (y_1,y_2) \end{aligned}$$

一方，右の要素を先に簡約するならば，

$$\begin{aligned} &(\textbf{app}\ f\ x_1\ \textbf{is}\ y_1\ \textbf{in}\ y_1, \textbf{app}\ g\ x_2\ \textbf{is}\ y_2\ \textbf{in}\ y_2) \\ &\longrightarrow \textbf{app}\ g\ x_2\ \textbf{is}\ y_2\ \textbf{in}\ (\textbf{app}\ f\ x_1\ \textbf{is}\ y_1\ \textbf{in}\ y_1, y_2) \\ &\longrightarrow \textbf{app}\ g\ x_2\ \textbf{is}\ y_2\ \textbf{in}\ \textbf{app}\ f\ x_1\ \textbf{is}\ y_1\ \textbf{in}\ (y_1,y_2) \end{aligned}$$

となり，簡約の順番によって異なる A^M-正規形を得る．これは，評価戦略が異なるならば A^M-正規形も異なることを意味する．実際のコンパイラでは，A^M-正規形を得るためにはプログラミング言語の仕様で定められるある一定の評価規則に基づいて A^M-簡約を行うべきである．

A^M-簡約関係が型を保存することは，変形規則 $\Longrightarrow$ の定義から明らかである．

補題 3.3. $\mathcal{GK}^{\mathcal{M}} \vdash \Gamma \triangleright M : \tau$ かつ $M \Longrightarrow M'$ ならば $\mathcal{GK}^{\mathcal{M}} \vdash \Gamma \triangleright M' : \tau$.

系 3.4. $\mathcal{GK}^{\mathcal{M}} \vdash \Gamma \triangleright M : \tau$ かつ $M \stackrel{*}{\longrightarrow} M'$ ならば $\mathcal{GK}^{\mathcal{M}} \vdash \Gamma \triangleright M' : \tau$.

A^M-簡約関係は正規化性を持ち，その正規形は A^M-正規形となる．

補題 3.5. 任意の M について，$M \stackrel{*}{\longrightarrow} P[T]$ となる P，T が存在する．

証明. M の構造に関する帰納法によって示す．いくつかの場合のみ示す．

$M = (M_1, M_2)$ の場合．帰納法の仮定より，ある P_1，P_2，T_1，T_2 が存在し，$M_1 \stackrel{*}{\longrightarrow} P_1[T_1]$，$M_2 \stackrel{*}{\longrightarrow} P_2[T_2]$．補題 3.1 より，

$$\begin{aligned} (P_1[T_1], P_2[T_2]) &\stackrel{*}{\longrightarrow} P_1[(T_1, P_2[T_2])] \\ &\stackrel{*}{\longrightarrow} P_1[P_2[(T_1,T_2)]] \end{aligned}$$

ここで，T_1，T_2 について場合分けを行う．

$T_1 = x_1$，$T_2 = x_2$ のとき，$P = P_1[P_2[[\ ]]]$，$T = (x_1, x_2)$ ととればよい．

$T_1 = V_1$，$T_2 = x_2$ のとき，

$$\begin{aligned} P_1[P_2[(V_1,x_2)]] &\longrightarrow P_1[P_2[(\textbf{let}\ x_3 = V_1\ \textbf{in}\ x_3, x_2)]] \\ &\longrightarrow P_1[P_2[\textbf{let}\ x_3 = V_1\ \textbf{in}\ (x_3,x_2)]] \end{aligned}$$

したがって，$P = P_1[P_2[\textbf{let}\ x_3 = V_1\ \textbf{in}\ [\ ]]]$，$T = (x_3, x_2)$ ととればよい．$T_1 = x_1$，$T_2 = V_2$ のときも同様である．

$T_1 = V_1$，$T_2 = V_2$ のとき，

$$P_1[P_2[(V_1,V_2)]]$$

© 2008 Information Processing Society of Japan

$\longrightarrow P_1[P_2[(\mathbf{let}\ x_5 = V_1\ \mathbf{in}\ x_5, V_2)]]$
$\longrightarrow P_1[P_2[(\mathbf{let}\ x_5 = V_1\ \mathbf{in}\ x_5, \mathbf{let}\ x_6 = V_2\ \mathbf{in}\ x_6)]]$
$\longrightarrow P_1[P_2[\mathbf{let}\ x_5 = V_1\ \mathbf{in}\ (x_5, \mathbf{let}\ x_6 = V_2\ \mathbf{in}\ x_6)]]$
$\longrightarrow P_1[P_2[\mathbf{let}\ x_5 = V_1\ \mathbf{in}\ \mathbf{let}\ x_6 = V_2\ \mathbf{in}\ (x_5, x_6)]]$

したがって，$P = P_1[P_2[\mathbf{let}\ x_5 = V_1\ \mathbf{in}\ \mathbf{let}\ x_6 = V_2\ \mathbf{in}\ [\ ]]]$，$T = (x_5, x_6)$ ととればよい．

$M = \mathbf{let}\ x = M_1\ \mathbf{in}\ M_2$ の場合．帰納法の仮定より，ある P_1，P_2，T_1，T_2 が存在し，$M_1 \xrightarrow{*} P_1[T_1]$，$M_2 \xrightarrow{*} P_2[T_2]$．補題 3.1 より

$$\mathbf{let}\ x = M_1\ \mathbf{in}\ M_2 \xrightarrow{*} \mathbf{let}\ x = P_1[T_1]\ \mathbf{in}\ P_2[T_2] \xrightarrow{*} P_1[\mathbf{let}\ x = T_1\ \mathbf{in}\ P_2[T_2]]$$

ここで，T_1 について場合分けを行う．$T_1 = V_1$ のとき，$P = P_1[\mathbf{let}\ x = V_1\ \mathbf{in}\ P_2[\ ]]$，$T = T_2$ ととればよい．$T_1 = x_1$ のとき，$P_1[\mathbf{let}\ x = x_1\ \mathbf{in}\ P_2[T_2]] \longrightarrow P_1[[x_1/x](P_2[T_2])]$．したがって，$P = P_1[[x_1/x](P_2[\ ])]$，$T = [x_1/x]T_2$ ととればよい．□

補題 3.5，3.2 および系 3.4 から以下の定理を得る．

定理 3.6. $\mathcal{GK}^{\mathcal{M}} \vdash \Gamma \triangleright M : \tau$ ならば，$M \xrightarrow{*} A \not\longrightarrow$ となる A が存在し，$\mathcal{GKA}^{\mathcal{M}} \vdash \Gamma \triangleright A : \tau$．

以上の結果より，λ^M から A^M-正規形を求める簡約の系列は，$\mathcal{GK}^{\mathcal{M}}$ から $\mathcal{GKA}^{\mathcal{M}}$ への証明変換と見なすことができる．

4.　A^M-正規形の操作的意味論と健全性

本章では，前章で定義した A^M-正規形に操作的意味論を与え，A^M-正規形の型付け規則である証明システム $\mathcal{GKA}^{\mathcal{M}}$ がその意味論に関して健全であることを示す．さらに，本章で定義する操作的意味論と $\mathcal{GKA}^{\mathcal{M}}$ の証明変換アルゴリズムとの関連，および各論理規則とプログラムの評価との対応についてその概要を述べる．

証明システムとの対比を明確にするために，A^M-正規形の操作的意味論は自然意味論[13] で定義する．評価結果となる値 v を以下のように定義する．

$$v ::= c^b \mid cls(E, \lambda x.M) \mid (v, v) \mid \mathbf{in1}(v) \mid \mathbf{in2}(v) \mid wrong$$

$cls(E, \lambda x.M)$ は値としての関数を表すクロージャ，$wrong$ は実行時エラーを表す．E は実行時環境で，変数から値への有限写像である．実行時環境 E のもとで項 M を評価したとき結果 v を得ることを $E \triangleright M \Downarrow v$ と書く．評価規則を**図 4** に与える．ただし，図 4 のどの規則にもマッチしない場合，または上式のいずれかが $E' \triangleright M' \Downarrow wrong$ となる場合，$E \triangleright M \Downarrow wrong$ とする．

Values

(axiom) $E \triangleright c^b \Downarrow c^b$　　(taut) $E, x{:}v \triangleright x \Downarrow v$

($\supset$-R) $E \triangleright \lambda x.M \Downarrow cls(E, \lambda x.M)$

($\wedge$-R) $E, x{:}v_1, y{:}v_2 \triangleright (x, y) \Downarrow (v_1, v_2)$　　($\vee$-Ri) $E, x{:}v \triangleright \mathbf{in}i(x) \Downarrow \mathbf{in}i(v)$

Computation Rules

($\supset$-L) $$\frac{E', w{:}v_1 \triangleright M' \Downarrow v_2 \quad E, x{:}cls(E', \lambda w.M'), y{:}v_1, z{:}v_2 \triangleright M \Downarrow v_3}{E, x{:}cls(E', \lambda w.M'), y{:}v_1 \triangleright \mathbf{app}\ x\ y\ \mathbf{is}\ z\ \mathbf{in}\ M \Downarrow v_3}$$

($\wedge$-L) $$\frac{E, x{:}(v_1, v_2), y{:}v_1, z{:}v_2 \triangleright M \Downarrow v_3}{E, x{:}(v_1, v_2) \triangleright \mathbf{proj}\ x\ \mathbf{on}\ (y, z)\ \mathbf{in}\ M \Downarrow v_3}$$

($\vee$-L) $$\frac{E_i, y_i{:}v_1 \triangleright M_i \Downarrow v_2 \quad E, x{:}\mathbf{in}i(v_1), f_1{:}cls(E_1, \lambda y_1.M_1), f_2{:}cls(E_2, \lambda y_2.M_2), z{:}v_2 \triangleright M \Downarrow v_3}{E, x{:}\mathbf{in}i(v_1), f_1{:}cls(E_1, \lambda y_1.M_1), f_2{:}cls(E_2, \lambda y_2.M_2) \triangleright \mathbf{case}\ x\ \mathbf{of}\ f_1, f_2\ \mathbf{is}\ z\ \mathbf{in}\ M \Downarrow v_3}$$

(Cut) $$\frac{E \triangleright V \Downarrow v_1 \quad E, x{:}v_1 \triangleright M \Downarrow v_2}{E \triangleright \mathbf{let}\ x = V\ \mathbf{in}\ M \Downarrow v_2}$$

図 4　A^M-正規形の操作的意味論
Fig. 4　Operational semantics for A^M-normal form.

健全性の証明は，プログラムの型と結果の値の型が等しいことを示すことによって行う．値 v が型 τ を持つことを $\models v : \tau$ と書き，値の型をこの式を導出するシステムによって定義する．値に対する型付けシステムを**図 5** に与える．このシステムは，値に型を与える判定 $\models v : \tau$ と，クロージャに型を与えるための補助的な判定 $\Gamma \triangleright cls(E, M) : \tau$ を導出する．$dom(E) = dom(\Gamma)$ かつ任意の $x \in dom(E)$ について $\models E(x) : \Gamma(x)$ であるとき，$\models E : \Gamma$ と書く．

以上の定義を用いて，以下の定理を証明する．

定理 4.1. $\mathcal{GKA}^{\mathcal{M}} \vdash \Gamma \triangleright M : \tau$，$\models E : \Gamma$ かつ $E \triangleright M \Downarrow v$ ならば $\models v : \tau$．

証明. 評価関係 $E \triangleright M \Downarrow v$ の導出の大きさに関する帰納法によって示す．いくつかの場合のみ示す．

© 2008 Information Processing Society of Japan

Values

(axiom) $\models c^b : b$　　(∧-R) $\dfrac{\models v_1 : \tau_1 \qquad \models v_2 : \tau_2}{\models (v_1, v_2) : \tau_1 \wedge \tau_2}$

(∨-Ri) $\dfrac{\models v : \tau_i}{\models \mathbf{in}i(v) : \tau_1 \vee \tau_2}$　　(closure) $\dfrac{\emptyset \rhd cls(E, \lambda x.M) : \tau}{\models cls(E, \lambda x.M) : \tau}$

Closures

(⊃-R) $\dfrac{\mathcal{GKA}^{\mathcal{M}} \vdash \Gamma, x{:}\tau_1 \rhd M : \tau_2}{\Gamma \rhd cls(\emptyset, \lambda x.M) : \tau_1 \supset \tau_2}$

(Cut) $\dfrac{\models v : \tau_1 \qquad \Gamma, x{:}\tau_1 \rhd cls(E, \lambda y.M) : \tau_2}{\Gamma \rhd cls((E, x{:}v), \lambda y.M) : \tau_2}$

図 5　値に対する型付け規則
Fig. 5　Typing rules for runtime values.

$E \rhd \lambda x.M' \Downarrow cls(E, \lambda x.M')$ の場合．$\mathcal{GKA}^{\mathcal{M}}$ の規則 (⊃-R) より，$\mathcal{GKA}^{\mathcal{M}} \vdash \Gamma \rhd \lambda x.M' : \tau_1 \supset \tau_2$ である．inversion 原理により，$\mathcal{GKA}^{\mathcal{M}} \vdash \Gamma, x{:}\tau_1 \rhd M' : \tau_2$．仮定より $\models E : \Gamma$ であるから，値に対する型付け規則 (⊃-R)，(Cut)，(closure) より，$\models cls(E, \lambda x.M') : \tau_1 \supset \tau_2$．

$E \rhd \mathbf{app}\ x\ y\ \mathbf{is}\ z\ \mathbf{in}\ M_1 \Downarrow v$ の場合．$E = E_1, x{:}cls(E_2, \lambda w.M_2), y{:}v_1$ とおく．$\mathcal{GKA}^{\mathcal{M}}$ の規則 (⊃-L) より，$\Gamma = \Gamma_1, x{:}\tau_1 \supset \tau_2, y{:}\tau_1$ とおくと $\mathcal{GKA}^{\mathcal{M}} \vdash \Gamma \rhd \mathbf{app}\ x\ y\ \mathbf{is}\ z\ \mathbf{in}\ M_1 : \tau_3$．仮定より $\models E : \Gamma$ であるから，$\models E_1 : \Gamma_1$，$\models cls(E_2, \lambda w.M_2) : \tau_1 \supset \tau_2$，かつ $\models v_1 : \tau_1$ である．評価規則 (⊃-L) より，$E_2, w : v_1 \rhd M_2 \Downarrow v_2$．また，値に対する型付け規則 (⊃-R)，(Cut)，(closure) より，$\models E_2 : \Gamma_2$ となるある Γ_2 が存在し，$\mathcal{GKA}^{\mathcal{M}} \vdash \Gamma_2, w{:}\tau_1 \rhd M_2 : \tau_2$．したがって帰納法の仮定より，$\models v_2 : \tau_2$．一方，評価規則 (⊃-L) より，$E_1, x{:}cls(E_2, \lambda w.M_2), y{:}v_1, z{:}v_2 \rhd M_1 \Downarrow v_3$．$\mathcal{GKA}^{\mathcal{M}}$ の規則 (⊃-L) より，$\mathcal{GKA}^{\mathcal{M}} \vdash \Gamma_1, x{:}\tau_1 \supset \tau_2, y{:}\tau_1, z{:}\tau_2 \rhd M_1 : \tau_3$．したがって帰納法の仮定より，$\models v_3 : \tau_3$．　□

文献 17) と同様の手法によって，A^M-正規形に関する操作的意味論を $\mathcal{GKA}^{\mathcal{M}}$ に関する証明変換アルゴリズムに対応付けることができ，A^M-正規形の意味を論理学的な枠組みによって Cut 除去操作として定式化することができると期待できる．その詳細は本論文の範囲を越えるため，今後の課題とし，ここではその方針の概要を述べる．値の型に関する判定 $\models v : \tau$ をクロージャに対する補助判定 $\Gamma \rhd cls(E, M) : \tau$ と統一し，ある型環境 Γ の下で値の型を与える $\Gamma \rhd v : \tau$ という形の判定と見なすと，規則 (closure) を除く図 5 の各規則はそれぞれ同じ名前を持つ $\mathcal{GK}^{\mathcal{M}}$ の導出規則に 1 対 1 に対応することが分かる．このことは，$\models v : \tau$ に対応する証明 $\mathcal{GK}^{\mathcal{M}} \vdash \emptyset \rhd v : \tau$ が存在することを意味する．したがって，A^M-正規形の操作的意味論は，項 A と対応する証明を，値 v に対応する証明に変換するアルゴリズムと見なすことができる．

5. ラムダ計算からのコンパイル

本章では，Standard ML の評価戦略の下で，型付きラムダ計算から A^M-正規形へ直接コンパイルするアルゴリズムを与える．このアルゴリズムは，ソース言語から A^M-正規形へ直接変換する高効率なアルゴリズムであり，高水準コンパイラのコンパイルフェーズの 1 つとしてそのまま利用されることを想定して構築されている．さらに，このアルゴリズムが，ラムダ計算からシーケント計算への変換と 3 章で述べた A^M-簡約との合成変換と等価であることを示す．

ラムダ計算の定義を以下のように与える．

$$\begin{aligned} M ::= {} & c^b \mid x \mid \lambda x.M \mid (M, M) \\ & \mid \mathbf{in1}(M) \mid \mathbf{in2}(M) \mid M\ M \mid M.i \\ & \mid \mathbf{case}\ M\ \mathbf{of}\ x.M, x.M \end{aligned}$$

ラムダ計算の型システムである自然演繹システムを**図 6** に示す．自然演繹から $\mathcal{GK}^{\mathcal{M}}$ への変換アルゴリズム $\overline{M}$ を**図 7** に定義する．この変換アルゴリズムは，すでに知られている自然演繹からシーケント計算への証明変換アルゴリズム[8),20),25)] に対し，以下の変更を加えたものである．

- $\mathcal{GK}^{\mathcal{M}}$ に合わせて規則 (∨-L) の場合を変更した．
- 2 章で提示した，すべての演算および値の構築は変数のみを対象とする，という方針に則り，各右規則の変換に追加の Cut 規則を挿入した．

この変換アルゴリズムの正しさは，自然演繹からシーケント計算への証明変換に関する既存の研究成果から明らかである．

定理 5.1. $\mathcal{N} \vdash \Gamma \rhd M : \tau$ ならば $\mathcal{GK}^{\mathcal{M}} \vdash \Gamma \rhd \overline{M} : \tau$．

型付きラムダ計算を A^M-正規形に直接コンパイルするアルゴリズムは，Flanagan らの A-正規化アルゴリズム[7)] を参考に型なしの変換アルゴリズムとして構築する．Flanagan らは，Danvy らが提案したメタレベルプログラミングによる方法[5),6)] を用いて，ラムダ計算から A-正規形への線形時間コンパイルアルゴリズムを Scheme で記述している．我々もこ

© 2008 Information Processing Society of Japan

$$(\text{axiom})\quad \Gamma \triangleright c^b : b \qquad (\text{taut})\quad \Gamma, x{:}\tau \triangleright x : \tau \qquad (\supset\text{-R})\quad \frac{\Gamma, x{:}\tau_1 \triangleright M : \tau_2}{\Gamma \triangleright \lambda x.M : \tau_1 \supset \tau_2}$$

$$(\wedge\text{-R})\quad \frac{\Gamma \triangleright M_1 : \tau_1 \qquad \Gamma \triangleright M_2 : \tau_2}{\Gamma \triangleright (M_1, M_2) : \tau_1 \wedge \tau_2} \qquad (\vee\text{-R}i)\quad \frac{\Gamma \triangleright M : \tau_i \qquad (i \in \{1,2\})}{\Gamma \triangleright \mathbf{in}i(M) : \tau_1 \vee \tau_2}$$

$$(\supset\text{-L})\quad \frac{\Gamma \triangleright M_1 : \tau_1 \supset \tau_2 \qquad \Gamma \triangleright M_2 : \tau_1}{\Gamma \triangleright M_1\ M_2 : \tau_2}$$

$$(\wedge\text{-L}i)\quad \frac{\Gamma \triangleright M : \tau_1 \wedge \tau_2 \qquad (i \in \{1,2\})}{\Gamma \triangleright M.i : \tau_i}$$

$$(\vee\text{-L})\quad \frac{\Gamma \triangleright M : \tau_1 \vee \tau_2 \qquad \Gamma, y{:}\tau_1 \triangleright M_1 : \tau_3 \qquad \Gamma, z{:}\tau_2 \triangleright M_2 : \tau_3}{\Gamma \triangleright \mathbf{case}\ M\ \mathbf{of}\ y.M_1, z.M_2 : \tau_3}$$

図 6　自然演繹証明システム $\mathcal{N}$
Fig. 6　Natural deduction proof system $\mathcal{N}$.

$$\begin{aligned}
\overline{c^b} &= \mathbf{let}\ x = c^b\ \mathbf{in}\ x \\
\overline{x} &= x \\
\overline{\lambda x.M} &= \mathbf{let}\ x = \lambda x.\overline{M}\ \mathbf{in}\ x \\
\overline{(M_1, M_2)} &= \mathbf{let}\ x = (\overline{M_1}, \overline{M_2})\ \mathbf{in}\ x \\
\overline{\mathbf{in}i(M)} &= \mathbf{let}\ x = \mathbf{in}i(\overline{M})\ \mathbf{in}\ x \\
\overline{M_1\ M_2} &= \mathbf{let}\ x = \overline{M_1}\ \mathbf{in}\ \mathbf{let}\ y = \overline{M_2}\ \mathbf{in}\ \mathbf{app}\ x\ y\ \mathbf{is}\ z\ \mathbf{in}\ z \\
\overline{M.i} &= \mathbf{let}\ x = \overline{M}\ \mathbf{in}\ \mathbf{proj}\ x\ \mathbf{on}\ (x_1, x_2)\ \mathbf{in}\ x_i \\
\overline{\mathbf{case}\ M\ \mathbf{of}\ x_1.M_1, x_2.M_2} &= \mathbf{let}\ y = \overline{M}\ \mathbf{in}\ \mathbf{let}\ f = \lambda x_1.\overline{M_1}\ \mathbf{in}\ \mathbf{let}\ g = \lambda x_2.\overline{M_2}\ \mathbf{in} \\
&\quad\ \mathbf{case}\ y\ \mathbf{of}\ f, g\ \mathbf{is}\ z\ \mathbf{in}\ z
\end{aligned}$$

図 7　$\mathcal{N}$ から $\mathcal{GK}^{\mathcal{M}}$ への証明変換アルゴリズム
Fig. 7　Proof transformation from $\mathcal{N}$ to $\mathcal{GK}^{\mathcal{M}}$.

れに準じ，A^M 項を受け取り A^M 項を返すメタ関数とそれに対するメタ関数適用を導入し，それらを用いて A^M-正規化アルゴリズムを定義する．メタ項を以下のように定義する．

$D ::= M \mid \delta X.D \mid D \odot M$

メタ簡約 $\xrightarrow{D}$ を以下のように定義する．

$C[(\delta X.D_1) \odot M] \xrightarrow{D} C[[M/X]D_1]$

$$\begin{aligned}
[\![c^b]\!]k &= \mathbf{let}\ x = c^b\ \mathbf{in}\ k \odot x \\
[\![x]\!]k &= k \odot x \\
[\![\lambda x.M]\!]k &= \mathbf{let}\ y = \lambda x.[\![M]\!](\delta X.X)\ \mathbf{in}\ k \odot y \\
[\![(M_1, M_2)]\!]k &= [\![M_1]\!](\delta X.[\![M_2]\!](\delta Y.\mathbf{let}\ x = (X, Y)\ \mathbf{in}\ k \odot x)) \\
[\![\mathbf{in}i(M)]\!]k &= [\![M]\!](\delta X.\mathbf{let}\ x = \mathbf{in}i(X)\ \mathbf{in}\ k \odot x) \\
[\![M_1\ M_2]\!]k &= [\![M_1]\!](\delta X.[\![M_2]\!](\delta Y.\mathbf{app}\ X\ Y\ \mathbf{is}\ z\ \mathbf{in}\ k \odot z)) \\
[\![M.i]\!]k &= [\![M]\!](\delta X.\mathbf{proj}\ X\ \mathbf{on}\ (x_1, x_2)\ \mathbf{in}\ k \odot x_i) \\
[\![\mathbf{case}\ M\ \mathbf{of}\ x.M_1, y.M_2]\!]k &= [\![M]\!](\delta X.\mathbf{let}\ f = \lambda x.[\![M_1]\!](\delta X_1.X_1)\ \mathbf{in} \\
&\qquad \mathbf{let}\ g = \lambda y.[\![M_2]\!](\delta X_2.X_2)\ \mathbf{in} \\
&\qquad \mathbf{case}\ X\ \mathbf{of}\ f, g\ \mathbf{is}\ w\ \mathbf{in}\ k \odot w)
\end{aligned}$$

図 8　A^M-正規化アルゴリズム
Fig. 8　A^M-normalization algorithm.

$\xrightarrow{D}$ の反射的推移的閉包を $\xrightarrow{D*}$ と書く．この簡約関係は明らかに強正規性を持つ．

A^M-正規化アルゴリズムは，ラムダ式 M およびメタ関数 k を受け取りメタ項 D を返す $[\![M]\!]k = D$ という形の関数として定義する．アルゴリズムの定義を**図 8** に示す．アルゴリズムが返すメタ項から，すべてのメタ関数とメタ関数適用をメタ簡約によって除去したものが，M の A^M-正規形となる．

3 章で述べたように，A^M-正規化は何らかの評価戦略に基づいて行う必要がある．図 8 で定義したアルゴリズムは，A^M-正規化を行うにあたり Standard ML の評価戦略を前提としている．他の評価戦略に基づく A^M-正規化アルゴリズムは，本章の結果を参考に構築することができる．

以上を用いて，A^M-正規化アルゴリズムの各ステップが，ラムダ計算の項から λ^M 項への変換と λ^M 項の A^M-正規化の合成に対応することを示す．

補題 5.2. 任意の M，k について，$\overline{M} \xrightarrow{*} P[x]$ となる P，x が存在し，$[\![M]\!]k \xrightarrow{D*} P[k \odot x]$.

証明. M の構造に関する帰納法によって示す．以下，いくつかの場合のみ示す．

$M = (M_1, M_2)$ の場合．M_1，M_2 に対する帰納法の仮定および補題 3.1 より，ある P_1，P_2，x_1，x_2 に対して以下の等式が成立する．

$[\![(M_1, M_2)]\!]k$
$= [\![M_1]\!](\delta X.[\![M_2]\!](\delta Y.\mathbf{let}\ y = (X, Y)\ \mathbf{in}\ k \odot y))$

© 2008 Information Processing Society of Japan

$\xrightarrow{D*} P_1[(\delta X.[\![M_2]\!](\delta Y.\mathbf{let}\ y=(X,Y)\ \mathbf{in}\ k\odot y))\odot x_1]$
$\xrightarrow{D} P_1[[\![M_2]\!](\delta Y.\mathbf{let}\ y=(x_1,Y)\ \mathbf{in}\ k\odot y)]$
$\xrightarrow{D*} P_1[P_2[(\delta Y.\mathbf{let}\ y=(x_1,Y)\ \mathbf{in}\ k\odot y)\odot x_2]]$
$\xrightarrow{D} P_1[P_2[\mathbf{let}\ y=(x_1,x_2)\ \mathbf{in}\ k\odot y]]$

かつ

$\overline{(M_1,M_2)} = \mathbf{let}\ y=(\overline{M_1},\overline{M_2})\ \mathbf{in}\ y$
$\xrightarrow{*} \mathbf{let}\ y=(P_1[x_1],P_2[x_2])\ \mathbf{in}\ y$
$\xrightarrow{*} P_1[P_2[\mathbf{let}\ y=(x_1,x_2)\ \mathbf{in}\ y]]$

したがって，

$P = P_1[P_2[\mathbf{let}\ y=(x_1,x_2)\ \mathbf{in}\ [\ ]]]$
$x = y$

ととればよい．

$M = M_1\ M_2$ の場合．M_1，M_2 に対する帰納法の仮定および補題 3.1 より，ある P_1，P_2，x_1，x_2 に対して以下の等式が成立する．

$[\![M_1\ M_2]\!]k$
$= [\![M_1]\!](\delta X.[\![M_2]\!](\delta Y.\mathbf{app}\ X\ Y\ \mathbf{is}\ z\ \mathbf{in}\ k\odot z))$
$\xrightarrow{D*} P_1[(\delta X.[\![M_2]\!](\delta Y.\mathbf{app}\ X\ Y\ \mathbf{is}\ z\ \mathbf{in}\ k\odot z))\odot x_1]$
$\xrightarrow{D} P_1[[\![M_2]\!](\delta Y.\mathbf{app}\ x_1\ Y\ \mathbf{is}\ z\ \mathbf{in}\ k\odot z)]$
$\xrightarrow{D*} P_1[P_2[(\delta Y.\mathbf{app}\ x_1\ Y\ \mathbf{is}\ z\ \mathbf{in}\ k\odot z)\odot x_2]]$
$\xrightarrow{D} P_1[P_2[\mathbf{app}\ x_1\ x_2\ \mathbf{is}\ z\ \mathbf{in}\ k\odot z]]$

かつ，

$\overline{M_1\ M_2}$
$= \mathbf{let}\ y_1=\overline{M_1}\ \mathbf{in}\ \mathbf{let}\ y_2=\overline{M_2}\ \mathbf{in}$
$\quad \mathbf{app}\ y_1\ y_2\ \mathbf{is}\ z\ \mathbf{in}\ z$
$\xrightarrow{*} \mathbf{let}\ y_1=P_1[x_1]\ \mathbf{in}\ \mathbf{let}\ y_2=P_2[x_2]\ \mathbf{in}$
$\quad \mathbf{app}\ y_1\ y_2\ \mathbf{is}\ z\ \mathbf{in}\ z$
$\xrightarrow{*} P_1[\mathbf{let}\ y_1=x_1\ \mathbf{in}\ P_2[\mathbf{let}\ y_2=x_2\ \mathbf{in}$
$\quad \mathbf{app}\ y_1\ y_2\ \mathbf{is}\ z\ \mathbf{in}\ z]]$
$\xrightarrow{*} P_1[P_2[\mathbf{app}\ x_1\ x_2\ \mathbf{is}\ z\ \mathbf{in}\ z]]$

したがって，

$P = P_1[P_2[\mathbf{app}\ x_1\ x_2\ \mathbf{is}\ z\ \mathbf{in}\ [\ ]]]$
$x = z$

ととればよい．

$M = \mathbf{case}\ M_0\ \mathbf{of}\ x_1.M_1, x_2.M_2$ の場合．M_0，M_1，M_2 に対する帰納法の仮定および補題 3.1 より，ある P_0，P_1，P_2，y_0，y_1，y_2 に対して以下の等式が成立する．

$[\![\mathbf{case}\ M_0\ \mathbf{of}\ x_1.M_1,x_2.M_2]\!]k$
$= [\![M_0]\!](\delta X.\ \mathbf{let}\ f=\lambda x_1.[\![M_1]\!](\delta X_1.X_1)\ \mathbf{in}$
$\quad \mathbf{let}\ g=\lambda x_2.[\![M_2]\!](\delta X_2.X_2)\ \mathbf{in}$
$\quad \mathbf{case}\ X\ \mathbf{of}\ f,g\ \mathbf{is}\ w\ \mathbf{in}\ k\odot w)$
$\xrightarrow{D*} P_0[(\delta X.\ \mathbf{let}\ f=\lambda x_1.P_1[(\delta X_1.X_1)\odot y_1]\ \mathbf{in}$
$\quad \mathbf{let}\ g=\lambda x_2.P_2[(\delta X_2.X_2)\odot y_2]\ \mathbf{in}$
$\quad \mathbf{case}\ X\ \mathbf{of}\ f,g\ \mathbf{is}\ w\ \mathbf{in}\ k\odot w)\odot y_0]$
$\xrightarrow{D*} P_0[\ \mathbf{let}\ f=\lambda x_1.P_1[y_1]\ \mathbf{in}$
$\quad \mathbf{let}\ g=\lambda x_2.P_2[y_2]\ \mathbf{in}$
$\quad \mathbf{case}\ y_0\ \mathbf{of}\ f,g\ \mathbf{is}\ w\ \mathbf{in}\ k\odot w]$

かつ

$\overline{\mathbf{case}\ M_0\ \mathbf{of}\ x_1.M_1,x_2.M_2}$
$= \mathbf{let}\ z=\overline{M_0}\ \mathbf{in}\ \mathbf{let}\ f=\lambda x_1.\overline{M_1}\ \mathbf{in}$
$\quad \mathbf{let}\ g=\lambda x_2.\overline{M_2}\ \mathbf{in}$
$\quad \mathbf{case}\ z\ \mathbf{of}\ f,g\ \mathbf{is}\ w\ \mathbf{in}\ w$
$\xrightarrow{*} \mathbf{let}\ z=P_0[y_0]\ \mathbf{in}\ \mathbf{let}\ f=\lambda x_1.P_1[y_1]\ \mathbf{in}$
$\quad \mathbf{let}\ g=\lambda x_2.P_2[y_2]\ \mathbf{in}$
$\quad \mathbf{case}\ z\ \mathbf{of}\ f,g\ \mathbf{is}\ w\ \mathbf{in}\ w$
$\xrightarrow{*} P_0[\ \mathbf{let}\ z=y_0\ \mathbf{in}\ \mathbf{let}\ f=\lambda x_1.P_1[y_1]\ \mathbf{in}$
$\quad \mathbf{let}\ g=\lambda x_2.P_2[y_2]\ \mathbf{in}$
$\quad \mathbf{case}\ z\ \mathbf{of}\ f,g\ \mathbf{is}\ w\ \mathbf{in}\ w]$
$\longrightarrow P_0[\ \mathbf{let}\ f=\lambda x_1.P_1[y_1]\ \mathbf{in}$
$\quad \mathbf{let}\ g=\lambda x_2.P_2[y_2]\ \mathbf{in}$
$\quad \mathbf{case}\ y_0\ \mathbf{of}\ f,g\ \mathbf{is}\ w\ \mathbf{in}\ w]$

したがって

© 2008 Information Processing Society of Japan

$$P = P_0[\,\textbf{let}\ f = \lambda x_1.P_1[y_1]\ \textbf{in}$$
$$\textbf{let}\ g = \lambda x_2.P_2[y_2]\ \textbf{in}$$
$$\textbf{case}\ y_0\ \textbf{of}\ f, g\ \textbf{is}\ w\ \textbf{in}\ [\ ]]$$
$$x = w$$

ととればよい．□

この補題の直接の帰結として，以下の系を得る．

系 5.3. 任意の M について，$[\![M]\!](\delta X.X) \xrightarrow{D*} M'$ ならば $\overline{M} \xrightarrow{*} M' \not\longrightarrow$.

以上の結果および定理 5.1，系 3.4 より，このコンパイルアルゴリズムは型を保存する．

定理 5.4. $\mathcal{N} \vdash \Gamma \triangleright M : \tau$ かつ $[\![M]\!](\delta X.X) \xrightarrow{D*} M'$ ならば，$\mathcal{GKA}^{\mathcal{M}} \vdash \Gamma \triangleright M' : \tau$.

このコンパイルアルゴリズムが自然演繹システムから $\mathcal{GKA}^{\mathcal{M}}$ への証明変換と対応することを示すためには，メタ項に対しても型を与えたうえで，メタ簡約も含めたすべてのコンパイルステップが型を保存することを示す必要がある．この命題の厳密な証明は，文献 17) における A-正規化アルゴリズムの証明論的解釈と同様の方法によって導くことが可能である．その詳細および，本論文で新たに導入された規則を含む証明系の証明論的性質や，それら性質と中間コードの意味論に関する詳細な分析は，本論文の範囲を越えるため，別の機会に報告する予定である．

このコンパイルアルゴリズムは，A^M-正規形への変換だけでなく，4 章で定義した操作的意味論を通じて，ソース言語の意味を定義していると見なすことができる．また，4 章で示唆した証明論的な枠組みによる意味の定式化によって，証明変換とカット除去操作を通じて，ソース言語に証明論的な意味を与えることができると期待できる．このようにして与えられるソース言語の意味と，実際のコンパイルや実行環境とは独立に与えられる意味（たとえば評価文脈や CPS 変換など）との関連を分析することは，今後の課題である．

6. 実　　装

著者らが開発している SML# コンパイラ 22) の中間言語の 1 つである YAANormal は，本研究で提案する計算系を基に設計されている．本章では，中間言語 YAANormal の定義および YAANormal へのコンパイルを行うフェーズ YAANormalization の実装の概要について述べる．

6.1 中間言語 YAANormal

YAANormal（Yet Another A-Normal ⋆1）は SML# のバックエンドの中間言語の 1 つで，クロージャ変換を行った後，プラットフォーム非依存な抽象命令列を生成する前に必要な解析や最適化を行うために用いる．YAANormal は本研究で提案する計算系を理論的基礎としているものの，構文は 3 章での形式的定義とは異なる，実装の容易さや拡張性を考慮したより柔軟な構文を採用している．Standard ML による YAANormal の定義の一部を**図 9** に

```
type varInfo =
    {id: id, displayName: string, ty: ty, ...}

datatype anvalue =
      ANCONSTANT of constant
    | ANVAR of varInfo
    | ...

datatype anexp =
      ANVALUE of anvalue
    | ANRECORD of {fieldList: anvalue list, ...}
    | ANSELECT of {record: anvalue, offset: anvalue, ...}
    | ANCLOSURE of {funLabel: anvalue, env: anvalue}
    | ANAPPLY of {closure: anvalue, argList: anvalue list, ...}
    | ANPRIMAPPLY of {primName: string, argList: anvalue list, ...}
    | ...

datatype andecl =
      ANVAL of {varList: varInfo list, exp: anexp, ...}
    | ANRETURN of {valueList: anvalue list, ...}
    | ANSWITCH of {value: anvalue,
                   branches: {constant: anvalue, branch: program} list,
                   default: program, ...}
    | ANMERGE of {label: id, ...}
    | ANMERGEPOINT of {label: id, varList: varInfo list, ...}
    | ...

withtype program = andecl list
```

図 9　中間言語 YAANormal
Fig. 9　Intermediate language YAANormal.

⋆1 ANormal という名前は SML# コンパイラ内部で別の中間言語の名前としてすでに使用されていたため，このような名前となった．

© 2008 Information Processing Society of Japan

示す．

YAANormal のプログラム全体（program）は andecl のリストである．andecl の各項はそれぞれ 1 つの演算処理を表し，おおよそ $\mathcal{GKA}^{\mathcal{M}}$ の各左規則に対応する．プログラム全体が andecl のリストとなっているのは，A^M-正規形は実質的に演算処理を **in** によって連結したリストと見なすことができることに由来する．線形リストと等価な構造を項の定義に組み込むよりも，ライブラリ関数が豊富に提供されている汎用のリスト型を用いて演算の並びを表現したほうが実用上取り扱いやすい．A^M-正規形の末尾位置に現れる T は呼び出し元に値を返す処理と見なし，ANRETURN という演算として他の演算処理と同列に扱う．したがって，プログラムを表す andecl のリストは，基本的には ANRETURN で終わる．

anexp は具体的な演算内容を表し，各 anexp と演算結果を変数に代入する項 ANVAL との組合せが，それぞれ **case** 項を除く A^M-正規形の各項に対応する．ただし，直和型は YAANormal より前のコンパイルフェーズによってレコード型にコンパイルされているため，YAANormal は A^M-正規形の $\mathbf{in}i(x)$ に相当する値構築子を持たない．また，YAANormal はクロージャ変換が行われた後の中間言語であるから，A^M-正規形の λ 項に対応する YAANormal の項 ANCLOSURE はコードを含まず，代わりにクロージャ変換された関数を指すラベルとクロージャ環境をとる．したがって，anexp の各項は引数として anvalue のみをとり，anexp の中に anexp や andecl はネストして現れない．

anvalue は変数または定数からなり，各演算が直接引数としてとれるものを表す項である．形式的な A^M-正規形では演算は変数上でのみ行うため，演算が直接引数としてとれるのは変数だけであった．YAANormal では定数畳み込みなどの最適化処理や即値を持つ演算命令への対応などの実用上の利便性から，定数も演算の直接の引数として認めている．

ANSWITCH および ANMERGEPOINT はそれぞれ制御フローの分岐と合流点を表す項であり，A^M-正規形の **case** 項に相当する．A^M-正規形の形式的な意味論では，**case** 項が分岐元と合流点の両者の役割を同時に担っている．しかし，実際のコンパイラでは，末尾呼び出しの最適化や無用命令の除去などによって，分岐はするものの分岐先から分岐元へ帰ってこないようなコードが生成される可能性がある．分岐後の合流の有無を明示的に取り扱うには，分岐元と合流先を別の項として分離したほうが実用上都合がよい．また，ANMERGEPOINT は必ず基本ブロックの先頭となるため，コード生成時における基本ブロックの発見にも有用である．

ANSWITCH は分岐元を表すだけでなく，各分岐先のコードを含む．これらのコードは，A^M-正規形において **case** 項に先行する各分岐先を表す λ 抽象に相当する．A^M-正規化アルゴリズムによって導入される条件分岐式の各分岐先に対する λ 抽象は，A^M-正規化アルゴリズムによって初めて導入され，その唯一の呼び出し元および唯一の帰り先は静的に決定している．したがって，条件分岐とこれらの λ 抽象を本物の関数として取り扱う必要はない．YAANormal では，これらの λ 抽象を ANSWITCH とともに導入される特殊なコードブロックとし，制御フローグラフにおける基本ブロックのようにジャンプによって分岐および合流を行うようにコンパイルする．分岐元と分岐先の間での引数および返り値の授受は，それぞれ局所変数の参照・代入によって行うようにコードを生成する．

ANMERGE は分岐先から分岐元へのリターンを表す項である．形式的定義では分岐先も λ 抽象として取り扱っているため，形式的には ANMERGE は ANRETURN と同じ意味を持つ．しかし，コード生成時には，前者は軽量なジャンプ命令に，後者はスタック操作を含む関数からのリターン命令にコンパイルされる．

ANMERGE の戻り先は，原則として分岐元の ANSWITCH 項に対応する ANMERGEPOINT である．ただし，戻り先の ANMERGEPOINT の直後に ANMERGE が現れるような場合は，無駄なジャンプの連続を防ぐために，ANMERGE から ANMERGEPOINT を越えてその先の ANMERGEPOINT に直接ジャンプするようにする最適化が考えられる．YAANormal では，このような最適化を行うことができるように，各 ANMERGEPOINT に一意のラベルを与え，ANMERGE がどの ANMERGEPOINT にジャンプするかをラベルによって識別するようにしている．

ソースコードとその A^M-正規形，および YAANormal の例を**図 10** に示す．

6.2　コンパイルフェーズ YAANormalization

YAANormalization は，ラムダ計算を基礎とする中間言語 RBUCalc から YAANormal への変換を行うコンパイルフェーズである．YAANormalization のメインルーチンは，RBUCalc の式を受け取り YAANormal のプログラム（andecl のリスト）を返す関数として実装している．

YAANormal のプログラムの構造を andecl のリストとしたことによって，5 章での A^M-正規化アルゴリズムの定義のような継続渡し形式での複雑なプログラミングを行うことなく，単純な再帰呼び出し形式でアルゴリズムの定義と等価な実装が可能となった．継続渡し形式のアルゴリズムは，以下のようにしてリストを再帰的に連結するアルゴリズムに読み替えることができる．

- アルゴリズム全体をラムダ式 M を受け取り，コンパイル結果のリスト M' と変数 x を返す関数と見なす．
- アルゴリズムの定義のうち，$[\![M]\!](\delta X.D)$ は以下のように解釈する．

© 2008 Information Processing Society of Japan

Source Code:
```
fn x => (if x then 1 else 2) * 3 + 4
```

A^M-normal form:

$\lambda x.$ **let** $f = \lambda u.1$ **in let** $g = \lambda v.2$ **in**
　case x **of** f, g **is** y **in**
　let $z = y * 3$ **in let** $w = z + 4$ **in** w

YAANormal:
```
{
  args = [x],
  body = [
    ANSWITCH {value = ANVAR x,
              branches = [{constant = ANCONSTANT 0,
                           branch = [
                             ANVAL {varList=[y],
                                    exp=ANVALUE (ANCONSTANT 2)},
                             ANMERGE {label = l, ...}
                           ]}],
              default = [
                ANVAL {varList=[y], exp = ANVALUE (ANCONSTANT 1),
                       ...},
                MERGE {label = l, ...}
              ], ...},
    ANMERGEPOINT {label = l, ...},
    ANVAL {dst = [z],
           exp = ANPRIMAPPLY {primName="*",
                              argList=[ANVAR y, ANCONSTANT 3],
                              ...}},
    ANVAL {dst = [w],
           exp = ANPRIMAPPLY {primName="+",
                              argList=[ANVAR z, ANCONSTANT 4],
                              ...}},
    ANRETURN {varList = w, ...}
  ], ...}
```

図 10　YAANormal への変換例
Fig. 10　Example of transformation to YAANormal.

```
fun normalizeExp rbuexp =
    case rbuexp of
      ...
    | RBUAPPM {funExp, argExpList, ...} =>
      let
        val (decls1, funValue) = normalizeExp funExp
        val (decls2, argValues) = normalizeExp argExpList
        val var = newVar ()
        val appDecl =
            ANVAL {varList = [var],
                   exp = ANAPPLY {closure = funExp, argList = argValues,
                                  ...},
                   ...}
      in
        (decls1 @ decls2 @ appDecl, ANVAR var)
      end
```

図 11　A^M-正規化アルゴリズムの実装例
Fig. 11　Example of implementation of A^M-normalization algorithm.

(1)　M をコンパイルして M'，x を得る．
(2)　X に x を代入する．
(3)　D をコンパイルして D'，x' を得る．
(4)　M' と D' を連結したものと x' を返す．

- $k \odot x$ は空のリストと x を返すものと見なす．

コンパイルアルゴリズムの実装例を**図 11** に示す．なお，SML# コンパイラの YAANormalization では，末尾呼び出しの取扱いなどの処理が加わるため，実際のコードはこの例よりも複雑なものとなっている．しかし，その本質はこの例と同等である．

リストの連結には 1 回あたり $O(n)$ の時間計算量を要するため，長大なリストを短いリストの前に連結する操作を再帰的に繰り返した最悪の場合を考えると，リストを用いたコンパイルアルゴリズムは $O(n^2)$ の時間計算量となり，線形時間で完了する継続渡し形式のコンパイルアルゴリズムに劣る．しかし，現実のプログラムでは大きなリストを繰り返し連結するような深いネスト構造で書かれていることはほとんどなく，実用上の問題はないと思われる．

© 2008 Information Processing Society of Japan

7. 関連研究

本研究が対象とした A-正規形変換[7] は，歴史的には，CPS 変換から導かれたものである．CPS 変換は，Plotkin の先駆的研究[19] 以来多数の研究がなされ[5),6),12),21]，関数型言語のコンパイラ中間言語として広く用いられ用いられている[1),10]．文献 5) では，文献 7) で示されたと同様の洞察に基づき，CPS 式を継続計算の引数を持たない中間言語への変換を与えている．文献 12) では，CPS 変換や関連する変換を，モナドを基礎とするメタ言語[16] を使用して特徴付けている．これら研究はいずれもラムダ計算やその意味論の枠組みでなされているのに対して，我々の研究は，制御フローの合流を含めた A-正規変換の論理学的意味付けを与えるものである．

条件分岐における継続の複製に関する問題はすでに指摘されており，それに対する解決法は文献 6)，7)，10)，12) などで議論されている．本研究は，これらの解決法を論理学的な解釈を通じて理解するための方法を提示するものである．

8. まとめと今後の課題

本論文では，文献 17) で示された証明論と A-正規形の対応を基礎に，シーケント計算の分析を通じて，制御フローの合流を表現できる計算系 λ^M および証明システム $\mathcal{GK}^{\mathcal{M}}$ を構築した．λ^M を正規化することで A^M-正規形を導出し，A^M-正規形に対応する証明システム $\mathcal{GKA}^{\mathcal{M}}$ を構築した．A^M-正規形の操作的意味論を定義し，$\mathcal{GKA}^{\mathcal{M}}$ がその意味論に関して健全であることを示した．また，型付きラムダ計算を A^M-正規形にコンパイルする A^M-正規化アルゴリズムを定義し，その正しさを示した．さらに，本論文で提案した A^M-正規形を基に設計・実装された SML# コンパイラの中間言語 YAANormal，および YAANormal へのコンパイルフェーズ YAANormalization について報告し，その実用性を示した．

今後の課題として，A^M-正規形の操作的意味論と証明変換との対応，および A^M-正規化アルゴリズムと証明変換との対応の詳細な分析があげられる．この分析を通じて，プログラムの意味を論理学的な枠組みによってカット除去プロセスとして定式化することができ，証明論的アプローチによる言語の意味の定義を提示することができると期待できる．さらに，このようにして定義された意味と，評価文脈などのすでに知られている方法による意味の定義との関連を分析することも有意義な課題である．

また，本研究と同様の手法によって CPS 変換に関する証明論的性質を調べることも今後の課題としてあげられる．CPS 変換は，計算と論理の関係に関する様々な性質を調べるために用いられている[4),9),24]．また，CPS 変換に関する型理論的な分析も行われている[11),15]．これらと本研究との関連を調べることによって，新たな洞察を得ることができる可能性がある．

謝辞　本論文に関して有益なコメントをいただいた査読者に感謝いたします．

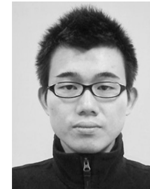
参考文献

1) Appel, A.W.: *Compiling with Continuations*, Cambridge University Press (1992).
2) Birkedal, L., Tofte, M. and Turner, N.D.: The ML Kit (Version 1), Technical Report DIKU-report 93/14, Department of Computer Science, University of Copenhagen (1993).
3) Birkedal, L., Tofte, M. and Vejlstrup, M.: From Region Inference to von Neumann Machines via Region Representation Inference, *Proc. 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pp.171–183, ACM Press (1996).
4) Curien, P.-L. and Herbelin, H.: The duality of computation, *ICFP '00: Proc. 5th ACM SIGPLAN international conference on Functional programming*, New York, NY, USA, pp.233–243, ACM (2000).
5) Danvy, O.: Back to direct style, *Proc. European Symposium on Programming*, Lecture Notes in Computer Science, Vol.582, pp.130–150 (1992).
6) Danvy, O. and Filinski, A.: Representing control: A study of the CPS transformation, *Mathematical Structures in Computer Sciences*, Vol.4, pp.361–391 (1992).
7) Flanagan, C., Sabry, A., Duba, B. and Felleisen, M.: The essence of compiling with continuation, *Proc. ACM PLDI Conference*, pp.237–247 (1993).
8) Gallier, J.: Constructive Logics Part I: A tutorial on proof systems and typed λ-calculi, *Theoretical Computer Science*, Vol.110, pp.249–339 (1993).
9) Griffin, T.: A Formulae-as-Types Notion of Control, *Conference Record of the 17th Annual ACM Symposium on Principles of Programming Languages*, pp.47–58 (1990).
10) Guy L. Steele, J.: Rabbit: A Compiler for Scheme, Technical report, Cambridge, MA, USA (1978).
11) Harper, R. and Lillibridge, M.: Polymorphic Type Assignment and CPS Conversion, *LISP AND SYMBOLIC COMPUTATION*, Vol.6, No.4, pp.361–380 (1993).
12) Hatcliff, J. and Danvy, O.: A Generic Account of Continuation-Passing Styles, *Conf. Record 21st Ann. ACM SIGPLAN-SIGACT Symp. on Principles of Programming Languages, POPL'94, Oregon, PL, USA, 17–21 Jan. 1994*, pp.458–471, ACM Press, New York (1994).
13) Kahn, G.: Natural Semantics, *Proc. Symposium on Theoretical Aspects of Com-*

© 2008 Information Processing Society of Japan

puter Science, pp.22–39, Springer Verlag (1987).
14) Kleene, S.: *Introduction to Metamathematics*, 7th edition, North-Holland (1952).
15) Meyer, A.R. and Wand, M.: Continuation Semantics in Typed Lambda-Calculi, *Proc. 3rd Workshop on Logics of Programs, Brooklyn*, NY, USA, 17–19 June 1985, Parikh, R. (Ed.), Vol.193, pp.219–224, Springer-Verlag, Berlin (1985).
16) Moggi, E.: Computational lambda-calculus and monads, *Proc. Symposium on Logic in Computer Science* (1989).
17) Ohori, A.: A Curry-Howard Isomorphism for Compilation and Program Execution, *Proc. Typed Lambda Calculi and Applications*, Springer LNCS 1581, pp.258–179 (1999).
18) Ohori, A.: A proof theory for machine code, *ACM Trans. Prog. Lang. Syst.*, Vol.29, No.6, p.36 (2007).
19) Plotkin, G.: Call-by-name, call-by-value and the λ-calculus, *Theoretical Computer Science*, Vol.1, pp.125–159 (1975).
20) Pottinger, G.: Normalization as a homomorphic image of cut-elimination, *Ann. Math. Logic*, Vol.12, pp.323–357 (1977).
21) Sabry, A. and Felleisen, M.: Reasoning about programs in continuation-passing style, *J. Lisp and Symbolic Computation*, Vol.6, No.3, pp.287–358 (1993).
22) SML# Compiler. http://www.pllab.riec.tohoku.ac.jp/smlsharp/
23) Sumii, E.: MinCaml: A simple and efficient compiler for a minimal functional language, *FDPE '05: Proc. 2005 workshop on Functional and declarative programming in education*, New York, NY, USA, ACM, pp.27–38 (2005).
24) Wadler, P.: Call-by-value is dual to call-by-name, *ICFP '03: Proc. 8th ACM SIGPLAN international conference on Functional programming*, New York, NY, USA, pp.189–201, ACM (2003).
25) Zucker, J.: The correspondence between cut-elimination and normalization, *Ann. Math. Logic*, Vol.7, pp.1–112 (1974).

(平成 20 年 4 月 22 日受付)
(平成 20 年 7 月 14 日採録)



上野　雄大

1981 年生．2006 年北陸先端科学技術大学院大学情報科学研究科博士前期課程修了．同年より東北大学情報科学研究科博士課程に在学中．2006 年東北大学電気通信研究所産学官連携研究員．プログラミング言語に関する研究に従事．

大堀　　淳（正会員）

1957 年生．1981 年東京大学文学部哲学科卒業．1989 年ペンシルバニア大学大学院計算機科学科博士課程修了．Ph.D. 1981 年沖電気入社．英国王立協会特別研究員（グラスゴー大学），沖電気関西総合研究所特別研究室長，京都大学数理解析研究所助教授，北陸先端科学技術大学院大学教授を経て，現在，東北大学電気通信研究所教授．プログラミング言語に興味を持つ．

© 2008 Information Processing Society of Japan