修 士 論 文

型理論に基づくプログラミング言語の効率的な実装に関する研究

北陸先端科学技術大学院大学
情報科学研究科情報処理学専攻

挽地 篤志

2002 年 3 月

修 士 論 文

型理論に基づくプログラミング言語の効率的な実装に関する研究

指導教官　大堀淳 教授

審査委員主査　大堀淳 教授
審査委員　田島敬史 助教授
審査委員　小野寛晰 教授

北陸先端科学技術大学院大学
情報科学研究科情報処理学専攻

010092 挽地 篤志

提出年月: 2002 年 2 月

概 要

本稿では、既存の型理論の研究のコンパイラへの応用可能性について吟味する。それとともに、中間言語 A-normal form から機械言語 SLAM へのコンパイルアルゴリズム Z について提案する。また、型理論に基づいたコンパイラの実装を行う。

目 次

図 目 次

表 目 次

第1章　序論

1.1　導入

従来、コンパイラの開発は ad hoc になされてきた。そのため、高級言語から低級言語にコンパイルできたとしても、実行時に誤りを含む可能性があった。それに対して近年、論理学とプログラミング言語の関係が分かってきた。論理学とプログラミング言語における命題と型、証明とプログラムの間には関係がある。論理学において P が命題 A の証明であるなら、プログラミング言語において P は型 A の値を計算するプログラムであると言える。本稿ではこの論理学とプログラミング言語の関係を用い、コンパイラの分析と実装を行う。以下、既存の研究とその問題点、本研究の目的、本稿の構成について述べる。

1.2　既存の研究と問題点

コンパイラとは、ソース言語から中間言語を経て目的言語へと変換するプログラムのことを言う。

コンパイラの研究には、中間言語に CPS を使ったコンパイラ [Appel et al.: 89]、A-normal form を使ったコンパイラ [Flanagan et al.: 93] がある。Appel は、中間言語に CPS を使ったコンパイラを設計した。CPS は継続 (Continuation) と環境 (Closure) を引数にとる中間言語である。CPS の欠点は、継続を引数に取ることによってコード量が増大することである。それに対し Flanagan は、中間言語に A-normal form を使ったコンパイラを設計した。A-normal form は中間の値に名前を付けるという特徴を持つ中間言語である。A-normal form の利点は、コード量が増大しないということである。これらの研究により、A-normal form は CPS よりも良い変換を与えることが分かった。しかしこれらのコンパイラは型無しの言語を対象にしているため、論理学との関係を適用することができない。

それに対して最近、型付きのコンパイラの研究がなされてきた。型付きのコンパイラの研究には、型主導のコンパイルの研究 [Morrisett et al.: 98] がある。Morrisett の研究は、型付き CPS を使用している。しかし、型付き A-normal form を使ったコンパイラの方がより良い変換をあたえることが分かっている。よって本稿では、型付きの A-normal form を用い、型を保存したコンパイラの設計と実装を行う。

論理学とプログラミング言語の間には以下のような関係がある。コンパイラのソース言語に用いる型付きラムダ計算と、論理学における自然演繹 (Natural Deduction) という体系は同一のものとみなすことができる。型付きラムダ計算と自然演繹における、型と

命題、プログラムと証明は同じ性質を持つ。自然演繹においてPが命題Aの証明ならば、型付きλ計算においてPが型τの値を計算するプログラムである。これをカリー・ハワード同型性質 (Curry-Howard Isomorphism)[Curry:80][Howard:80][Gallier:93] と言う。

近年、この性質を用いることによって、プログラム変換を証明変換ととらえた分析がおこなわれてきた。型付きλ計算から型付き A-normal form へのプログラム変換は、自然演繹から Genzen 流のシーケント計算の一種である GK[Kleene:52] への証明変換 (これを NG と呼ぶ) と、GK から GK のサブシステムである GKA への証明変換 (これを S と呼ぶ) の合成によって表される [ohori:99]。型付き A-normal from は、GK を A-normal 正規化した GKA と一致する。

コンパイラは最終的に逐次シーケント計算 (Sequential Sequent Calculus(SSC))[ohori:99] と同じ性質を持つ論理抽象機械 (Stack-based Logical Abstract Machine(SLAM)) へプログラム変換を行う。型付き A-normal form から SLAM へのプログラム変換は、GKA から逐次シーケント計算への証明変換と同じである。しかし、この証明変換はまだ定義されていない。

1.3 目的

本研究の目的は、A-normal form の論理学的解釈を基礎に、A-normal form を中間言語とする型主導コンパイル方式を研究することである。実際には以下の 3 点を行う。まず既存の型理論の研究のコンパイラへの応用可能性を吟味する。次に A-normal form から SLAM へのプログラム変換における型の保存を証明する。最期に理論に基づいて実装を行なう。

1.4 構成

本稿の内容の概略について述べる。2 章では、コンパイラ実装のための既存の研究について述べる。2.1 節ではソース言語の説明する。ここでは、計算機言語のモデルである型つきラムダ計算、論理学の体系である自然演繹、型付きラムダ計算と自然演繹の間のカリー・ハワード同型性質について述べる。2.2 節では中間言語の説明をする。ここでは、中間言語のモデルである A-normal form、論理学の体系である GK と GKA について述べる。2.3 節では機械言語の説明をする。ここでは、機械言語のモデルであるスタックベースの論理抽象機械 (SLAM)、論理学の体系である逐次シーケント計算 (SSC) について述べる。2.4 節ではソース言語から中間言語へのコンパイルアルゴリズムに対応した、自然演繹から GKA へ証明変換 NG と S について述べる。3 章の成果では、中間言語から機械言語へのコンパイルアルゴリズムについて述べる。3.1 節では GKA から SSC への証明変換に対応したコンパイルアルゴリズム Z について述べる。プログラミング言語と論理学の対応関係について、図 1.1 に表す。

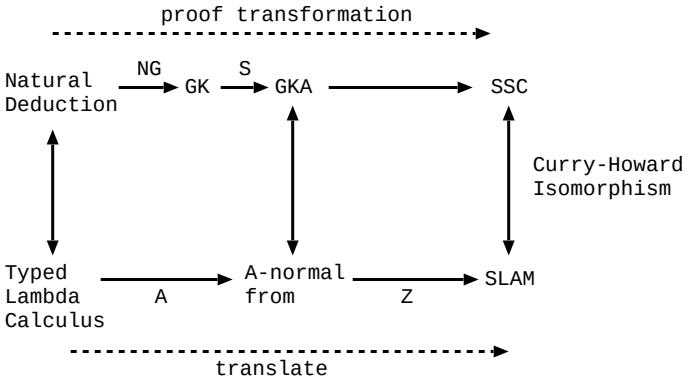

図 1.1: コンパイルにおけるカリー・ハワード同型対応

4 章では、コンパイラの実装について述べる。4.1 節では、どのようなシステム上でコンパイラを実装したかについて述べる。4.2 節では、コンパイラの概要と実行の流れに付いて述べる。4.3 節では、コンパイル結果について述べる。4.4 節では、コンパイル結果について解説する。4.5 節では、作成したコンパイラのソースコードについて述べる。5 章では、結論と今後の課題について述べる。

第2章　準備

本章では、コンパイラ実装のための既存の研究について説明する。具体的には、プログラミング言語と論理学の体系、コンパイルアルゴリズムと証明簡約、カリー・ハワード同型性質の三点について説明する。

2.1　ソース言語

本節では、型付きラムダ計算、自然演繹、カリー・ハワード同型性質について述べる。

計算機言語のモデルである型付きラムダ計算と、論理学おける自然演繹は、元々独自に研究されてきた分野である。しかし近年、型付きラムダ計算と自然演繹の対応が研究され、その一致が証明された。その対応であるカリー・ハワード同型性質についても述べる。

0 まず、型付きラムダ計算の型、項、型システムについて述べる。型 τ を以下に示す。

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

b は原始型 (atomic types) を表す。$\supset$ は、$\wedge$ や $\vee$ よりも強い結合力を持つ。型付きラムダ項 M を以下に示す。

$$M \ ::= \ c^b \mid x \mid \lambda x : \tau.M \mid M \ M \mid (M, M) \mid M.1 \mid M.2 \mid in1(M : \tau) \mid in2(M : \tau)$$
$$\mid case \ M \ of \ x.M, x.M$$

c^b は原始型 b をもつ定数を表す。x は変数を表す。$\lambda x \ : \ \tau.M$ はラムダ抽象 (lambda abstraction) と呼ばれ、型 τ を持つ変数 x を受け取り M を実行する名前の無い関数を表す。ラムダ抽象 $\lambda.x \ M$ の M は最も大きくとるものとする。$M_1 \ M_2$ はラムダ適用 (lambda application) と呼ばれ、引数 M_2 に関数 M_1 を適用することを表す。関数適用は左結合する。(M_1, M_2) は M_1 と M_2 の組を表す。$M.1$、$M.2$ は、M の 1 番目、2 番目の取り出しを表す。$in1(M : \tau)$、$in2(M : \tau)$ は、型 τ を持つ M の 1 番目、2 番目への埋め込みを表す。$case \ M_0 \ of \ x_1.M_1, x_2.M_2$ は、M_0 を評価し、1 番目への埋め込み $(in1(a))$ なら a を x_1 に束縛して M_1 の実行、2 番目への埋め込み $(in2(a))$ なら a を x_2 に束縛して M_2 の実行を表す。

Γ は、変数から型 τ への関数を表す。型環境 Γ の元で項 M が型 τ を持つことを、$\Gamma \triangleright M : \tau$ と書く。この式を型判定と呼ぶ。

表 2.1: 型付き λ 計算の型システム Λ

$(axiom)$	$\Gamma \triangleright c^b : b$
$(taut)$	$\Gamma, x : \tau \triangleright x : \tau$
$(\supset\!: I)$	$\dfrac{\Gamma, x : \tau_1 \triangleright M : \tau_2}{\Gamma \triangleright \lambda x : \tau_1 . M : \tau_1 \supset \tau_2}$
$(\supset\!: E)$	$\dfrac{\Gamma \triangleright M_1 : \tau_1 \supset \tau_2 \qquad \Gamma \triangleright M_2 : \tau_1}{\Gamma \triangleright M_1\ M_2 : \tau_2}$
$(\wedge : I)$	$\dfrac{\Gamma \triangleright M_1 : \tau_1 \qquad \Gamma \triangleright M_2 : \tau_2}{\Gamma \triangleright (M_1, M_2) : \tau_1 \wedge \tau_2}$
$(\wedge : E_i)$	$\dfrac{\Gamma \triangleright M : \tau_1 \wedge \tau_2}{\Gamma \triangleright M.i : \tau_i} \qquad (\{1,2\} \in i)$
$(\vee : I_i)$	$\dfrac{\Gamma \triangleright M : \tau_i}{\Gamma \triangleright in_i(M : \tau_1 \vee \tau_2) : \tau_1 \vee \tau_2} \qquad (\{1,2\} \in i)$
$(\vee : E)$	$\dfrac{\Gamma \triangleright M_1 : \tau_1 \vee \tau_2 \qquad \Gamma, x : \tau_1 \triangleright M_2 : \tau_3 \qquad \Gamma, y : \tau_2 \triangleright M_3 : \tau_3}{\Gamma \triangleright case\ M\ of\ x.M_2\ ,\ y.M_3 : \tau_3}$

型付きラムダ計算の推論規則は以下の形で示す。

$$\frac{\Gamma_1 \triangleright M_1 : \tau_1 \quad \cdots \quad \Gamma_n \triangleright M_n : \tau_n}{\Gamma \triangleright M : \tau}\ (\mathrm{I})$$

この推論規則は、推論規則 I により、型環境 Γ_1 のもとで M_1 が型 τ_1 を持ち、かつ、型環境 Γ_n のもとで M_n が型 τ_n(ただし $1 < n$) を持つことが証明可能なら、型環境 Γ のもとで M が型 τ を持つことが証明可能であることを表す。

型付きラムダ計算の型システムを表 2.1 に示す。型付きラムダ計算の型システムは、型判定を導出する証明システムである。この型システムを Λ と呼び、$\Gamma \triangleright M : \tau$ がこの証明システムで証明可能であるなら $\Lambda \vdash \Gamma \triangleright M : \tau$ と書く。

次に、自然演繹について述べる。自然演繹は、プログラミング言語のモデルである型付きラムダ計算に対応する証明系である。以下に、命題論理式と直観主義論理に対する命題論理学の自然演繹の体系について述べる。

命題論理式 A を以下に示す。

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

b は命題定数を表す。

論理式で使う記号と推論規則の記法について述べる。$\supset$ は、$\wedge$ や $\vee$ よりも大きく取る。Δ は、論理式 A の有限な重複の集合とする。Δ に A を加えた集合を $\Delta \cup A$ と表す。a が Δ の要素であることを $a \in \Delta$ と表す。仮定の集合 Δ から A が証明可能であることを $\Delta \triangleright A$ と書く。自然演繹におけるの証明は以下の形で書く。

$$\begin{array}{c} \vdots \\ \Delta \triangleright A \end{array}$$

表 2.2: 自然演繹システム N

$(axiom)$	$\Delta \triangleright \alpha \qquad (\alpha \in Ax)$
$(taut)$	$\Delta \cup A \triangleright A$
$(\supset\!: I)$	$\dfrac{\Delta \cup A \triangleright B}{\Delta \triangleright A \supset B}$
$(\supset\!: E)$	$\dfrac{\Delta \triangleright A \supset B \qquad \Delta \triangleright A}{\Delta \triangleright B}$
$(\wedge : I)$	$\dfrac{\Delta \triangleright A \qquad \Delta \triangleright B}{\Delta \triangleright A \wedge B}$
$(\wedge : E_1)$	$\dfrac{\Delta \triangleright A \wedge B}{\Delta \triangleright A}$
$(\wedge : E_2)$	$\dfrac{\Delta \triangleright A \wedge B}{\Delta \triangleright B}$
$(\vee : I_1)$	$\dfrac{\Delta \triangleright A}{\Delta \triangleright A \vee B}$
$(\vee : I_2)$	$\dfrac{\Delta \triangleright B}{\Delta \triangleright A \vee B}$
$(\vee : E)$	$\dfrac{\Delta \triangleright A \vee B \qquad \Delta \cup A \triangleright C \qquad \Delta \cup B \triangleright C}{\Delta \triangleright C}$

これは、仮定の集合 Δ のもとで A が導かれると解釈する。自然演繹におけるの証明規則は以下の形で書く。

$$\frac{\Delta_1 \triangleright A_1 \quad \cdots \quad \Delta_n \triangleright A_n}{\Delta \triangleright A}\ (\mathrm{I})$$

この推論規則は、推論規則 (I) により、$\Delta_1 \triangleright A_1$ から $\Delta_n \triangleright A_n (1 < n)$ が全て証明可能なら、$\Delta \triangleright A$ が証明可能であることを表す。

推論規則には、純粋な命題論理学の証明規則に、非論理的公理 (non logical axiom) を加える。Ax を与えられた非論理的公理の集合とする。推論規則 $(taut)$ と $(axiom)$ は、それぞれ Δ に仮定した命題および公理は証明無しで使用して良いことを表している。

自然演繹の推論規則は、$\supset$、$\wedge$、$\vee$ についての導入 (Introduction) と除去 (Elimination) から成る。自然演繹の証明システムを表 2.2 に示す。この証明システムを N と呼び、$\Delta \triangleright A$ がこの証明システムで証明可能であるなら $N \vdash \Delta \triangleright A$ と書く。

最後に、カリー・ハワード の同型対応について述べる。カリー・ハワード の同型対応とは、プログラミング言語と論理学の間の一致する性質を述べたものである。型付きラムダ計算と自然演繹の間にあるカリー・ハワード の同型性質を以下に述べる。

1. ラムダ式 $\Gamma \triangleright M : \tau$ から M を消し、値を消すことによって得られた multi-setΔ と Γ を置き換えると、非論理的公理を加えた自然演繹システム N を得ることができる。

2. もし $\vdash \Gamma \triangleright M : \tau$ なら、項 M の型 τ は N で証明可能である。

3. 型付き λ 項の β 簡約は、N の証明正規化に一致する。

2.2 中間言語

本節では、中間言語で使う A-normal form と GK、GKA について述べる。

A-normal form は、中間の値に名前付けするという特徴を持つ中間言語で、Flanagan らによって定義された。しかし、Flanagan らのモデルは型無しの A-normal form であったため、カリー・ハワード同型性質が保存されていない。そこで、Kleene によって定義された証明システム GK と、ohori によって定義された GK のサブシステムである証明システム GKA について説明する。型付きの A-normal form は、GK を A-normal 正規化した証明システム GKA に対応する。

まず、A-normal form の型、項、型システムについて述べる。型 τ を以下に示す。

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

b は原子定数を表す。項 M を以下に示す。

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

中間の値に名前付けした項 app、proj、case、let について説明する。$app\ (x\ M_1)\ is\ y\ in\ M_2$ は、M_1 に x を適用したものを y に束縛して、M_2 を実行することを表す。$proj\ x\ on\ (y, z)\ in\ M$ は、組 x の左を y、右を z に束縛して、M を実行することを表す。$case\ x\ of\ y.M_1\ ,\ z.M_2$ は、もし x が左への埋め込み $(in1(a))$ なら a を y に束縛して M_1 を実行し、もし x が右への埋め込み $(in2(a))$ なら a を z に束縛して M_2 を実行することを表す。$let\ x = M_1\ in\ M_2$ は、M_1 を評価したものを x に束縛し、M_2 を実行することを表す。

A-normal form の型システムを表 2.3 に示す。

次に証明システム GK,GKA について述べる。GK は、Gentzen の直観主義論理におけるシーケント計算の一種で、Kleene によって定義された。命題論理式 A は自然演繹と同じである。GK の証明システムを表 2.4 に示す。GKA は、GK を A-normal 正規化したサブシステムで、ohori によって定義された。

型付き A-normal form と GKA の、型と命題、プログラムと証明は対応している。よって型付き A-normal form と GKA の間にカリー・ハワード同型性質が存在する。

表 2.3: A-normal form の型システム

Values.	
$(axiom)$	$\Gamma \triangleright c^b : b$
$(taut)$	$\Gamma, x : \tau \triangleright x : \tau$
$(\wedge : R)$	$\dfrac{\Gamma \triangleright V_1 : \tau_1 \qquad \Gamma \triangleright V_2 : \tau_2}{\Gamma \triangleright (V_1, V_2) : \tau_1 \wedge \tau_2}$
$(\vee : R_i)$	$\dfrac{\Gamma \triangleright V : \tau_i}{\Gamma \triangleright in_i(V) : \tau_1 \vee \tau_2} \qquad (\{1,2\} \in i)$
$(\supset\colon R)$	$\dfrac{\Gamma, x : \tau_1 \triangleright M : \tau_2}{\Gamma \triangleright \lambda x.M : \tau_1 \supset \tau_2}$
General A-normal forms.	
$(\supset\colon L)$	$\dfrac{\Gamma, x : \tau_1 \supset \tau_2 \triangleright M_1 : \tau_1 \qquad \Gamma, x : \tau_1 \supset \tau_2, y : \tau_2 \triangleright M_2 : \tau_3}{\Gamma, x : \tau_1 \supset \tau_2 \triangleright app\ (x\ M_1)\ is\ y\ in\ M_2 : \tau_3}$
$(\wedge : L)$	$\dfrac{\Gamma, x : \tau_1 \wedge \tau_2, y : \tau_1, z : \tau_2 \triangleright M : \tau_3}{\Gamma, x : \tau_1 \wedge \tau_2 \triangleright proj\ x\ on\ (y,z)\ in\ M : \tau_3}$
$(\vee : L)$	$\dfrac{\Gamma, x : \tau_1 \vee \tau_2, y : \tau_1 \triangleright M : \tau_3 \qquad \Gamma, x : \tau_1 \vee \tau_2, z : \tau_2 \triangleright M_2 : \tau_3}{\Gamma, x : \tau_1 \vee \tau_2 \triangleright case\ x\ of\ y.M_1\ ,\ z.M_2 : \tau_3}$
(cut)	$\dfrac{\Gamma \triangleright M_1 : \tau_1 \qquad \Gamma, x : \tau_1 \triangleright M_2 : \tau_2}{\Gamma \triangleright let\ x = M_1\ in\ M_2 : \tau_2}$

表 2.4: GK の証明システム

$(axiom)$	$\Gamma \triangleright \alpha \quad (\alpha \in Ax)$
$(taut)$	$\Gamma, A \triangleright A$
$(\supset\colon R)$	$\dfrac{\Gamma, A_1 \triangleright A_2}{\Gamma \triangleright A_1 \supset A_2}$
$(\supset\colon L)$	$\dfrac{\Gamma, A_1 \supset A_2 \triangleright A_1 \qquad \Gamma, A_1 \supset A_2, A_2 \triangleright A_3}{\Gamma, A_1 \supset A_2 \triangleright A_3}$
$(\wedge : R)$	$\dfrac{\Gamma \triangleright A_1 \qquad \Gamma \triangleright A_2}{\Gamma \triangleright A_1 \wedge A_2}$
$(\wedge : L)$	$\dfrac{\Gamma, A_1 \wedge A_2, A_1, A_2 \triangleright A_3}{\Gamma, A_1 \wedge A_2 \triangleright A_3}$
$(\vee : R_i)$	$\dfrac{\Gamma \triangleright A_i}{\Gamma \triangleright A_1 \vee A_2} \qquad (\{1,2\} \in i)$
$(\vee : L)$	$\dfrac{\Gamma, A_1 \vee A_2, A_1 \triangleright A_3 \qquad \Gamma, A_1 \vee A_2, A_2 \triangleright A_3}{\Gamma, A_1 \vee A_2 \triangleright A_3}$
(cut)	$\dfrac{\Gamma \triangleright A_1 \qquad \Gamma, A_1 \triangleright A_2}{\Gamma \triangleright A_2}$

2.3 機械言語

本節では、ohoriによって定義された、機械言語のモデルであるスタックベースの論理抽象機械(Stack-based Logical Abstract Machine(SLAM))と、それに対応する証明システムである逐次シーケント計算(Sequential Sequent Calculus(SSC))について述べる。

まず、SLAMの型、項、型システムについて述べる。SLAMの型τを以下に示す。

$$\tau \ = \ b \mid (\Delta \Rightarrow \tau) \mid \tau \wedge \tau \mid \tau \vee \tau$$

bは原子定数を表す。$\Rightarrow$は関数型を表す。$\Rightarrow$は$\wedge$や$\vee$よりも大きく取るものとする。SLAMの項Mを以下に示す。

$$\begin{aligned} M \ ::= \ & return \mid swap \mid pop \mid const\ c^b \mid acc\ n \mid pair \mid proj \mid inl \mid inr \mid switch(C, C) \\ & \mid code\ C \mid app\ n \end{aligned}$$

Cは命令の列を表す。SLAMはCの最も左を実行するという規則を持つ。命令MをCの先頭に適用したものを$M;C$と書き、命令列C'をCの先頭に適用したものを$C';C$と書く。

型環境Δは最も右がスタックのトップである。Δに型τを追加することを$\Delta;\tau$と書く。型環境ΔのもとでMが型τを持つことを$\Delta \triangleright M : \tau$と書く。推論規則は以下の形で書く。

$$\frac{\Delta' \triangleright M' : \tau}{\Delta \triangleright M : \tau}\ (\mathrm{I})$$

この推論規則は、推論規則(I)により型環境ΔのもとでMの型τは変わらないということを表す。プログラムは下式の状態を破棄して上式の状態へ遷移することで実行される。

SLAMの型システムを表2.5に示す。以下、各命令について説明する。命令$(return)$はスタックのトップを返す。命令$(swap)$は、スタックのトップから1番目と2番目の要素を入れ換える。命令(pop)は、スタックのトップの要素を取り出して破棄する。命令$(const)$は、スタックに原始型bを追加する。命令$(pair)$は、スタックのトップから1番目と2番目の要素を取り出し、論理積型をスタックに追加する。命令(inl)と命令(inr)は、スタックのトップの要素を取り出し、論理和型をスタックに追加する。命令$(switch)$は、スタックのトップの要素を取り出して評価し、左への埋め込み(inl)ならC_1の型を、右への埋め込み(inr)ならC_2の型を、スタックに追加する。命令$(code)$は、型環境Δ_0のもとでC_0の型τ_0を返す関数型をスタックに追加する。命令(app)は、n個の引数と関数クロージャをスタックから取り出し、n個の引数によって拡張されたクロージャスタックから導かれた関数型$\Delta_1 \Rightarrow \tau_0$をスタックに追加する。命令$(call)$は、スタックのトップから1番目と2番目の要素を取り出し、第1要素Δ_1に第2要素$(\Delta_1 \Rightarrow \tau_0)$を適用することによって得

表 2.5: SLAM の型システム

(return)	$\Delta; \tau \triangleright return : \tau$
	$\dfrac{\Delta; \tau_1 \triangleright C : \tau}{\Delta \triangleright acc(n); C : \tau}$ (ただし、$\Delta(n) = \tau_1$)
(acc)	
	$\dfrac{\Delta; b \triangleright C : \tau}{\Delta \triangleright const(c^b); C : \tau}$
(const)	
	$\dfrac{\Delta; \tau_1 \wedge \tau_2 \triangleright C : \tau}{\Delta; \tau_1; \tau_2 \triangleright pair; C : \tau}$
(pair)	
	$\dfrac{\Delta; \tau_1; \tau_2 \triangleright C : \tau}{\Delta; \tau_1 \wedge \tau_2 \triangleright proj; C : \tau}$
(proj)	
	$\dfrac{\Delta; \tau_1 \vee \tau_2 \triangleright C : \tau}{\Delta; \tau_1 \triangleright inl; C : \tau}$
(inl)	
	$\dfrac{\Delta; \tau_1 \vee \tau_2 \triangleright C : \tau}{\Delta; \tau_2 \triangleright inr; C : \tau}$
(inr)	
	$\dfrac{\Delta; \tau_3 \triangleright C : \tau}{\Delta; \tau_1 \vee \tau_2 \triangleright switch(C_1, C_2); C : \tau}$
(switch)	(ただし、$\vdash \Delta; \tau_1 \triangleright C_1 : \tau_3$かつ$\vdash \Delta; \tau_2 \triangleright C_2 : \tau_3$の場合)
	$\dfrac{\Delta; \tau_2; \tau_1 \triangleright C : \tau}{\Delta; \tau_1; \tau_2 \triangleright swap; C : \tau}$
(swap)	
	$\dfrac{\Delta; \tau_1 \triangleright C : \tau}{\Delta; \tau_1; \tau_2 \triangleright pop; C : \tau}$
(pop)	
	$\dfrac{\Delta; (\Delta_0 \Rightarrow \tau_0) \triangleright C : \tau}{\Delta \triangleright code(C_0); C : \tau}$ (ただし、$\vdash \Delta_0 \triangleright C_0 : \tau_0$の場合)
(code)	
	$\dfrac{\Delta; (\Delta_1 \Rightarrow \tau_0) \triangleright C : \tau}{\Delta; (\Delta_2; \Delta_1 \Rightarrow \tau_0); \Delta_2 \triangleright app(n); C : \tau}$ (ただし、$n = \lvert\Delta_2\rvert$)
(app)	
	$\dfrac{\Delta; \tau_0 \triangleright C : \tau}{\Delta; (\Delta_1 \Rightarrow \tau_0); \Delta_1 \triangleright call(n); C : \tau}$ (ただし、$n = \lvert\Delta_1\rvert$)
(call)	

られる型 τ_0 をスタックに追加する。

逐次シーケント計算 (SSC) は、推論規則における上式と下式が 1 対 1 に対応する機械言語に適した証明システムである。以下に SSC の命題論理式 τ と証明システム SSC について述べる。

命題変数 τ を以下に示す。

$$\tau \ = \ b \mid (\Delta \Rightarrow \tau) \mid \tau \wedge \tau \mid \tau \vee \tau$$

推論規則は以下の形で書く。

$$\frac{\Delta_1 \triangleright \tau_1}{\Delta_2 \triangleright \tau_2} \ (\mathrm{I})$$

これは、推論規則 I により証明 $\Delta_1 \triangleright \tau$ から証明 $\Delta_2 \triangleright \tau$ が導かれると解釈する。

証明システムを表 2.6 に示す。仮定のリストはスタックの型を表す。

SLAM における型とプログラムと、SSC における命題と証明の間には、カリー・ハワード同型性質が存在する。

2.4 ソース言語から中間言語への証明変換

本節では，N から GKA への証明変換について述べる。N から GKA への証明簡約は 2 つの段階がある。

[**Theorem 1**] N の証明から GK の証明への変換は NG の証明簡約によって与えられる。
[**Theorem 2**] GK の証明から GKA の証明への変換は S の証明簡約によって与えられる。

これにより、以下のことが結論付けられる。

[**Collollary 1**] N のすべての証明は GKA のすべての証明に変換できる。

証明のための変数として X を使う。型 τ のメタ変数として σ を使う。Ω は型代入のセットを表す。型代入は $\{X_1 : \sigma_1, \cdot, X_n : \sigma_n\}$ と書き、σ_i の $X_i (1 \leq i \leq n)$ への代入を表す。$GK(\Omega)$ は GK から手にいれた証明システムである。$(axiom)\Gamma \triangleright X : \tau$ は、Ω のもとで $X : \Gamma \triangleright \tau$ と書く。もし $\Gamma \triangleright M : \tau$ が $GK(\Omega)$ で証明可能なら、$GK(\Omega) \vdash \Gamma \triangleright M : \tau$ と書く。δ は λ と同じく抽象を表す。D は Ω のもとで型づけされた項である。もし $\Omega, X : \sigma_1 \vdash D : \sigma_2$ ならば、$\Omega \vdash \delta X : \sigma_1 . D : \sigma_1 \rightarrow \sigma_2$ である。$\odot$ は適用を表す。もし $\Omega \vdash D_1 : \sigma_1 \rightarrow \sigma$ かつ $\Omega \vdash D_2 : \sigma_1$ ならば、$\Omega \vdash D_1 \odot D_2 : \tau$ である。一回の簡約を $\Rightarrow$ と書く。n 回で停止す

表 2.6: SSC の証明システム S_S

(return)	$\Delta; \tau \triangleright \tau$
(acc)	$\dfrac{\Delta; \tau_1 \triangleright \tau}{\Delta \triangleright \tau}$ (ただし、$\tau_1 \in \Delta$)
(const)	$\dfrac{\Delta; b \triangleright \tau}{\Delta \triangleright \tau}$
(pair)	$\dfrac{\Delta; \tau_1 \wedge \tau_2 \triangleright \tau}{\Delta; \tau_1; \tau_2 \triangleright \tau}$
(proj)	$\dfrac{\Delta; \tau_1; \tau_2 \triangleright \tau}{\Delta; \tau_1 \wedge \tau_2 \triangleright \tau}$
(inl)	$\dfrac{\Delta; \tau_1 \vee \tau_2 \triangleright \tau}{\Delta; \tau_1 \triangleright \tau}$
(inr)	$\dfrac{\Delta; \tau_1 \vee \tau_2 \triangleright \tau}{\Delta; \tau_2 \triangleright \tau}$
(switch)	$\dfrac{\Delta; \tau_3 \triangleright \tau}{\Delta; \tau_1 \vee \tau_2 \triangleright \tau}$ (ただし、$\vdash \Delta; \tau_1 \triangleright \tau_3$かつ$\vdash \Delta; \tau_2 \triangleright \tau_3$の場合)
(swap)	$\dfrac{\Delta; \tau_2; \tau_1 \triangleright \tau}{\Delta; \tau_1; \tau_2 \triangleright \tau}$
(pop)	$\dfrac{\Delta; \tau_1 \triangleright \tau}{\Delta; \tau_1; \tau_2 \triangleright \tau}$
(code)	$\dfrac{\Delta; (\Delta_0 \Rightarrow \tau_0) \triangleright \tau}{\Delta \triangleright \tau}$ (ただし、$\vdash \Delta_0 \triangleright \tau_0$の場合)
(app)	$\dfrac{\Delta; (\Delta_1 \Rightarrow \tau_0) \triangleright \tau}{\Delta; (\Delta_2; \Delta_1 \Rightarrow \tau_0); \Delta_2 \triangleright \tau}$
(call)	$\dfrac{\Delta; \tau_0 \triangleright \tau}{\Delta; (\Delta_1 \Rightarrow \tau_0); \Delta_1 \triangleright \tau}$

表 2.7: コンパイルアルゴリズム A

$[\![c^b]\!]k$	$=$	$k \odot c^b$
$[\![x]\!]k$	$=$	$k \odot x$
$[\![\lambda x.M]\!]k$	$=$	$k \odot (\lambda x.[\![M]\!](\delta X.X))$
$[\![(M\ N)]\!]k$	$=$	$[\![M]\!](\delta X.[\![N]\!](\delta Y.let\ x = X\ in\ app\ (x\ Y)\ is\ z\ in\ k \odot z))$
$[\![(M,N)]\!]k$	$=$	$[\![M]\!](\delta X.[\![N]\!](\delta Y.k \odot (X,Y)))$
$[\![M.i]\!]k$	$=$	$[\![M]\!](\delta X.let\ x = X\ in\ proj\ x\ on\ (x_1,x_2)\ in\ k \odot x_i)$
$[\![in_i(M)]\!]k$	$=$	$[\![M]\!](\delta X.k \odot in_i(X))$
$[\![case\ M\ of\ \lambda x.N, \lambda y.L]\!]k$	$=$	$[\![M]\!](\delta X.(let\ z = X\ in\ case\ z\ of\ x.[\![N]\!]k, y.[\![L]\!]k))$

る簡約を $\stackrel{*}{\Rightarrow}$ と書く。コンパイルアルゴリズム A は関数 $[\![\_]\!]\_$ で与えられる。関数 $[\![\_]\!]\_$ は、$\Omega \vdash D : \Gamma_1 \triangleright \tau_1$ のような D と、$\Omega \vdash k : (\Gamma_1 \triangleright \tau_1) \to (\Gamma_2 \triangleright \tau_2)$ のような関数項 k を持ち、$\Omega \vdash D' : \Gamma_2 \triangleright \tau_2$ を返す。

以上より、N から GKA の変換を以下に示す。

[Theorem 3] もし $N \vdash \Gamma \triangleright M : \tau_1$、かつ $\Gamma \subseteq \Gamma'$ のもとで $\Omega \vdash k : (\Gamma' \triangleright \tau_1) \to (\Gamma \triangleright \tau_2)$ ならば、$\Gamma \subseteq \Gamma'$ のもとで $\Omega \vdash [\![M]\!]k : (\Gamma \triangleright \tau_2)$、かつ $S(\Omega) \vdash k \odot NG(M) \stackrel{*}{\Rightarrow} [\![M]\!]k : (\Gamma \triangleright \tau_2)$ である。

また、k が $\delta X.X$ の場合を以下に示す。

[Theorem 4] もし $N \vdash \Gamma \triangleright M : \tau$ ならば、$GKA \vdash \Gamma \triangleright [\![M]\!]\delta X.X : \tau$、かつ $S \vdash \Gamma \triangleright NG(M) \stackrel{*}{\Rightarrow} [\![M]\!]\delta X.X : \tau$

コンパイルアルゴリズム A を表 2.7 に示す。コンパイルアルゴリズム A は証明変換 NG と S の合成に一致する。

第3章　成果

前章では、型付きλ計算、A-normal form、SLAMの型システムと、型付きλ計算からA-normal formへのコンパイルアルゴリズムAを紹介した。しかし実装するにあたり、A-normal formからSLAMへのコンパイルアルゴリズムがまだできていない。そこで本章では、A-normal formからSLAMへのコンパイルと同等の性質を持つ、GKAからSSCへの証明変換Zを定義する。

3.1　中間言語から機械言語への証明変換

本節では、GKからSSCへの証明変換から抽出されるコンパイルアルゴリズムZについて説明する。本節の目的は正しいコンパイルアルゴリズムを得ることである。型付きラムダ計算からA-normal formへの変換のようにA-normal formからSLAMへの変換が可能なら、証明変換からコンパイルアルゴリズムを抽出できる。GKAはGKの一部であるので、GKからSSCの変換を行う。

A-normal formからSLAMへの正確なコンパイルアルゴリズムを得るためには、以下の証明を行えば良い。

[**Theorem 5**] もし$GK \vdash \Gamma \triangleright M : \tau$ならば、$S_S \vdash \Delta_\Gamma \triangleright C_M : \tau$のようなSLAMのプログラム$C_M$がある。

Theorem 4の証明は補助定理を必要とする。

補助定理で使用する記号について説明する。S_Sの命題$(< \tau_1 >\Rightarrow \tau_2)$と$GKA$の命題$(\tau_1 \supset \tau_2)$は同じものであると解釈する。GKAの仮定のリスト$\Gamma$と、$\Gamma$の並びから得られた$S_S$の仮定のリストを一致させる。すると、$\Gamma = \{x_1 : A_1, \cdots, x_n : A_n\}$(ただし$1 < n$)ならば、$\Delta_\Gamma =< \tau_1, \cdots, \tau_n >$である。もし$\Gamma \triangleright M : \tau$が$GKA$の証明システムで証明可能なら、$GKA \vdash \Gamma \triangleright M : \tau$と書く。$\Gamma$の範囲内にxが含まれていることを$x \in dom(\Gamma)$と書く。もし$x \in dom(\Gamma)$なら、xと一致した$\Delta_\Gamma$の中の場所を$lookup(\Delta, x)$と書く。

補助定理はMの導出の最期に現れた規則による場合分けによって行う。補助定理を以下に述べる。

[**Lemma 1**] もし$A \vdash \Gamma \triangleright M : \tau$ならば、$[\![M]\!] : \Delta_\Gamma \Rightarrow \Delta_\Gamma; \tau$となるSLAMのコード$[\![M]\!]$

が存在する。
[**Proof**] $A \vdash \Gamma \triangleright M : \tau$ の導出に関する帰納法による。以下、導出の最後に使われた規則による場合分けを行う。
c^b の場合
ルール (const) より、$const(c^b) : \Delta_\Gamma \Rightarrow \Delta_\Gamma; b$ となる。
x の場合
$x : \tau \in \Gamma$ からルール (acc) より、$acc(lookup(\Gamma, x)) : \Delta_\Gamma \Rightarrow \Delta_\Gamma; \tau$ となる。
$\lambda x.M$ の場合
A の型システムから、$\tau = \tau_1 \supset \tau_2$ かつ $A \vdash \Gamma, x : \tau_1 \triangleright M : \tau_2$ のような τ_1、τ_2 がある。まず帰納法の仮定から、$[\![M]\!] : \Delta_{\Gamma,x:\tau_1} \Rightarrow \Delta_{\Gamma,x:\tau_1}; \tau_2$ となる。A の束縛変数の変換より、x が $dom(\Gamma)$ の中の全ての変数より大きいと仮定することができる。ゆえに、$\Delta_\Gamma, x : \tau_1 = \Delta_\Gamma; \tau_1$ である。よって、$[\![M]\!] : \Delta_\Gamma; \tau_1 \Rightarrow \Delta_\Gamma; \tau_1; \tau_2$ となる。ルール (return) より、$[\![M]\!]; return : \Delta_\Gamma \Rightarrow \Delta_\Gamma; \tau_2$ となる。ルール (code) より、$code([\![M]\!]; return) : \Delta_\Gamma \Rightarrow \Delta_\Gamma; (\Delta_\Gamma; \tau_1 \Rightarrow \tau_2)$ となる。$n = |\Delta_\Gamma|$ とし、ルール (acc) より、$code([\![M]\!]; return); acc(0); \cdots; acc(n-1) : \Delta_\Gamma \Rightarrow \Delta_\Gamma; (\Delta_\Gamma; \tau_1 \Rightarrow \tau_2); \Delta_\Gamma$ となる。最後にルール (app) より、$code([\![M]\!]; return); acc(0); \cdots; acc(n-1); app(n) : \Delta_\Gamma \Rightarrow \Delta_\Gamma; (< \tau_1 >\Rightarrow \tau_2)$ となる。
(M_1, M_2) の場合
A の型システムから、$\tau = \tau_1 \wedge \tau_2$ かつ $A \vdash \Gamma \triangleright M_1 : \tau_1$、かつ、$A \vdash \Gamma \triangleright M_2 : \tau_2$ のような τ_1、τ_2 がある。M_1 のための帰納法の仮定から、$[\![M_1]\!] : \Delta_\Gamma \Rightarrow \Delta_\Gamma; \tau_1$ である。A の機能から、$A \vdash \Gamma, x : \tau_1 \triangleright M_2 : \tau_2$ である。x は $dom(\Gamma)$ の全ての変数よりも大きな変数とする。x の選択から、$\Delta_{\Gamma,x:\tau_1} = \Delta_\Gamma; \tau_1$ である。M_2 のための帰納法の仮定から、$[\![M_2]\!] : \Delta_\Gamma; \tau_1 \Rightarrow \Delta_\Gamma; \tau_1; \tau_2$ となる。ルール (pair) から、$[\![M_1]\!]; [\![M_2]\!]; pair : \Delta_\Gamma \Rightarrow \Delta_\Gamma; \tau_1 \wedge \tau_2$ となる。
$app\ (x\ M_1)\ is\ y\ in\ M_2$ の場合
A の型システムから、$A \vdash \Gamma, x :< \tau_1 >\Rightarrow \tau_2 \triangleright M_1 : \tau_1$ かつ $A \vdash \Gamma, x :< \tau_1 >\Rightarrow \tau_2, y : \tau_2 \triangleright M_2 : \tau_3$ のような τ_1、τ_2、τ_3 がある。$x : \tau \in \Gamma$ から、ルール (acc) より $acc(lookup(\Gamma, x)) : \Delta_\Gamma \Rightarrow \Delta_\Gamma; (< \tau_1 >\Rightarrow \tau_2)$ となる。M_1 のための帰納法の仮定から、$[\![M_1]\!] : \Delta_{\Gamma,x:(<\tau_1>\Rightarrow\tau_2)} \Rightarrow \Delta_{\Gamma,x:<\tau_1>\Rightarrow\tau_2}; \tau_1$ である。よって、$[\![M_1]\!] : \Delta_\Gamma; (< \tau_1 >\Rightarrow \tau_2) \Rightarrow \Delta_\Gamma; (< \tau_1 >\Rightarrow \tau_2); \tau_1$ となる。ルール (call) より、$call(1) : \Delta_\Gamma; (< \tau_1 >\Rightarrow \tau_2); \tau_1 \Rightarrow \Delta_\Gamma; \tau_2 \equiv \Delta_{\Gamma,y:\tau_2}$ となる。M_2 のための帰納法の仮定から、$[\![M_2]\!] : \Delta_{\Gamma,x:(<\tau_1>\Rightarrow\tau_2),y:\tau_2} \Rightarrow \Delta_{\Gamma,x:(<\tau_1>\Rightarrow\tau_2),y:\tau_2}; \tau_3$ である。束縛変数に関する仮定より、y は $dom(\Gamma)$ の中の全ての変数よりも大きい変数とする。y の選択より、$[\![M_2]\!] : \Delta_\Gamma; \tau_2 \triangleright \Delta_\Gamma; \tau_2; \tau_3$ となる。最後に、ルール (swap) とルール (pop) より、$acc(lookup(\Gamma, x)); [\![M_1]\!]; call(1); [\![M_2]\!]; swap; pop : \Delta_\Gamma \Rightarrow \Delta_\Gamma; \tau_3$(ただし、y が Γ の中の全ての変数よりも大きい) となる。
$in1(M)$ の場合
A の型システムから、$\tau = \tau_1 \vee \tau_2$ かつ $A \vdash \Gamma \triangleright M : \tau_1$ のようないくつかの τ_1、τ_2 がある。帰納法の仮定より、$[\![M]\!] : \Delta_\Gamma \Rightarrow \Delta_\Gamma; \tau_1$ である。ルール (inl) より、$[\![M]\!]; inl : \Delta_\Gamma \Rightarrow \Delta_\Gamma; \tau_1 \vee \tau_2$ となる。
$in2(M)$ の場合

A の型システムから、$\tau = \tau_1 \vee \tau_2$ かつ $A \vdash \Gamma \triangleright M : \tau_2$ のようないくつかの τ_1、τ_2 がある。帰納法の仮定より、$[\![M]\!] : \Delta_\Gamma \Rightarrow \Delta_\Gamma; \tau_2$ である。ルール (inl) より、$[\![M]\!]; inl : \Delta_\Gamma \Rightarrow \Delta_\Gamma; \tau_1 \vee \tau_2$ となる。

proj x on (y, z) in M の場合

A の型システムから、$A \vdash \Gamma, x : \tau_1 \wedge \tau_2, y : \tau_1, z : \tau_2 \triangleright M : \tau_3$ のようないくつかの τ_1、τ_2、τ_3 がある。$x : \tau \in \Gamma$ から、ルール (acc) より、$acc(lookup(\Gamma, x)) : \Delta_\Gamma \Rightarrow \Delta_\Gamma; \tau_1 \wedge \tau_2$ となる。ルール (proj) より、$proj : \Delta_\Gamma; \tau_1 \wedge \tau_2 \Rightarrow \Delta_\Gamma; \tau_1; \tau_2$ となる。M のための帰納法の仮定より、$[\![M]\!] : \Delta_{\Gamma, y:\tau_1, z:\tau_2} \Rightarrow \Delta_{\Gamma, y:\tau_1, z:\tau_2}; \tau_3$ である。y、z を $dom(\Gamma)$ の中の全ての変数よりも大きい変数とし、かつ、$y < z$ とする。ゆえに、$\Delta_{\Gamma, y:\tau_1, z:\tau_2} = \Delta_\Gamma; \tau_1; \tau_2$ となる。よって、$[\![M]\!] : \Delta_\Gamma; \tau_1; \tau_2 \Rightarrow \Delta_\Gamma; \tau_1; \tau_2; \tau_3$ となる。最後にルール (swap) とルール (pop) より、$acc(lookup(\Gamma, x)); proj; [\![M]\!]; swap; pop; swap; pop : \Delta_\Gamma \Rightarrow \Delta_\Gamma; \tau_3$(ただし、y、z は $dom(\Gamma)$ の中の全ての変数よりも大きく、かつ、$y < z$) となる。

case x of y.M₁ z.M₂ の場合

A の型システムから、$A \vdash \Gamma, x : \tau_1 \vee \tau_2, y : \tau_1 \triangleright M_1 : \tau_3$ かつ $A \vdash \Gamma, x : \tau_1 \vee \tau_2, z : \tau_2 \triangleright M_2 : \tau_3$ のようないくつかの τ_1、τ_2、τ_3 がある。$x : \tau \in \Gamma$ から、ルール (acc) より $acc(lookup(\Gamma, x)) : \Delta_\Gamma \Rightarrow \Delta_\Gamma; \tau_1 \vee \tau_2$ となる。M_1 のための帰納法の仮定より、$[\![M_1]\!] : \Delta_{\Gamma, x:\tau_1 \vee \tau_2, y:\tau_1} \Rightarrow \Delta_{\Gamma, x:\tau_1 \vee \tau_2, y:\tau_1}; \tau_3$ である。y を $dom(\Gamma, x)$ の中の全ての変数よりも大きい変数とする。ゆえに、$\Delta_{\Gamma, x:\tau_1 \vee \tau_2, y:\tau_1} = \Delta_\Gamma; \tau_1 \vee \tau_2; \tau_1$ となる。よって、$[\![M_1]\!] : \Delta_\Gamma; \tau_1 \vee \tau_2; \tau_1 \Rightarrow \Delta_\Gamma; \tau_1 \vee \tau_2; \tau_1; \tau_3$ となる。M_2 のための帰納法の仮定より、$[\![M_2]\!] : \Delta_{\Gamma, x:\tau_1 \vee \tau_2, z:\tau_2} \Rightarrow \Delta_{\Gamma, x:\tau_1 \vee \tau_2, z:\tau_2}; \tau_3$ である。z を $dom(\Gamma, x)$ の中の全ての変数よりも大きい変数とする。ゆえに、$\Delta_{\Gamma, x:\tau_1 \vee \tau_2, z:\tau_2} = \Delta_\Gamma; \tau_1 \vee \tau_2; \tau_2$ となる。よって、$[\![M_2]\!] : \Delta_\Gamma; \tau_1 \vee \tau_2; \tau_2 \Rightarrow \Delta_\Gamma; \tau_1 \vee \tau_2; \tau_2; \tau_3$ となる。最後にルール (switch) とルール (return) より、$switch(acc(lookup(\Gamma, x)); [\![M_1]\!]; return, acc(lookup(\Gamma, x)); [\![M_2]\!]; return) : \Delta_\Gamma \Rightarrow \Delta_\Gamma; \tau_3$(ただし、y、z は $dom(\Gamma)$ の中の全ての変数より大きい) となる。

let x = M₁ in M₂ の場合

A の型システムから、$A \vdash \Gamma \triangleright M_1 : \tau_1$ かつ $A \vdash \Gamma, x : \tau_1 \triangleright M_2 : \tau_2$ のようないくつかの τ_1、τ_2 がある。$[\![M_1]\!]$ のための帰納法の仮定より $[\![M_1]\!] : \Delta_\Gamma \Rightarrow \Delta_\Gamma; \tau_1$ である。A の機能から、$A \vdash \Gamma, x : \tau_1 \triangleright M_2 : \tau_2$ である。x は $dom(\Gamma)$ の中の全ての変数よりも大きい変数とする。x の選択から、$\Delta_{\Gamma, x:\tau_1} = \Delta_\Gamma; \tau_1$ である。M_2 のための帰納法の仮定から、$[\![M_2]\!] : \Delta_\Gamma; \tau_1 \Rightarrow \Delta_\Gamma; \tau_1; \tau_2$ である。最後にルール (swap) とルール (pop) より、$[\![M_1]\!]; [\![M_2]\!]; swap; pop : \Delta_\Gamma \Rightarrow \Delta_\Gamma; \tau_2$(ただし、x は Γ の中の全ての変数よりも大きい) となる。

[Theorem 5] を [Lemma 1] を用いて証明する。

[Proof] [Lemma 1] より、$[\![M]\!] : \Delta_\Gamma \Rightarrow \Delta_\Gamma; \tau$ のようないくつかの $[\![M]\!]$ がある。C_M はスタックのトップの型を返すので $[\![M]\!]; return$ となる。$S_S \vdash \Delta_\Gamma; \tau \triangleright return : \tau$ から、$S_S \vdash \Delta_\Gamma \vdash C_M : \tau$ となる。

以上より、コンパイルアルゴリズム Z は以下のように結論づけられる。Com(M) は A-

表 3.1: コンパイルアルゴリズム $Z([\![M]\!]; return)$

$[\![c^b]\!]$	=	$const\ c^b$
$[\![x]\!]$	=	$acc(lookup(\Gamma, x))$
$[\![\lambda x.M]\!]$	=	$code([\![M]\!]; return); acc(0); \cdots; acc(n-1); app(n)$
$[\![(M_1, M_2)]\!]$	=	$[\![M_1]\!]; [\![M_2]\!]; pair$
$[\![in1\ M]\!]$	=	$[\![M]\!]; inl$
$[\![in2\ M]\!]$	=	$[\![M]\!]; inr$
$[\![app(x\ M_1)\ is\ y\ in\ M_2]\!]$	=	$[\![M_1]\!]; [\![M_2]\!]; swap; pop$ (ただし、x, yがΓの中のどの変数よりも大きく、 かつ、$x < y$)
$[\![proj\ x\ on\ (y,z)\ in\ M]\!]$	=	$proj; [\![M]\!]; swap; pop; swap; pop$ (ただし、x, y, zがΓの中のどの変数よりも大きく、 かつ、$x < y < z$)
$[\![case\ x\ of\ y.M_1\ z.M_2]\!]$	=	$switch([\![M_1]\!]; swap; pop; return, [\![M_2]\!]; swap; pop; return)$ (ただし、x, y, zがΓの中のどの変数よりも大きく、 かつ、$x < y$、$x < z$)
$[\![let\ x = M_1\ in\ M_2]\!]$	=	$[\![M_1]\!]; [\![M_2]\!]; swap; pop$ (ただし、xがΓの中のどの変数よりも大きい。)

normal form のコード M をコンパイルしたコードとする。

[Corollary 2] コンパイルアルゴリズム Z は、$Com(M) = [\![M]\!]; return \equiv C_M$ である。ただし、$[\![M]\!]$ は表 3.1 で与えられる。

第4章　実装

本章では、これまでに説明した理論に基づき実装したコンパイラに関する説明を行う。具体的には、実装環境、実行の流れ、実行例とその解説、作成したコンパイラのソースについて説明する。

4.1　実装環境

コンパイラは以下の環境で実装した。

Kernel linux-2.4.4-18k

Compiler Standard ML of New Jersey(SML/NJ), Version 110.0.7　[Milner et al.:2001]

Editer XEmacs 21.1 (patch 14)

Shell tcsh 6.10.01-4k

実装にはML言語を使った。SML/NJは、定理証明システムの記述言語を起源とする関数型言語である。今回SML/NJを使用したのは、この言語自体が堅牢な計算機言語の理論によって設計されていることと、必要十分なライブラリがあるためである。ML(Meta Language)は定理証明システム記述言語の総称であり、Standard MLはその一つである。また、NJはSMLにないライブラリ群である。SML/NJの処理系は以下のURLからダウンロード可能である。

URL http://cm.bell-labs.com/cm/cs/what/smlnj/index.html

4.2　全体の流れ

コンパイラは、入力したソースコードを、型付きラムダ計算、A-normal form、SLAMのソースコードへと変換する。以下の手順によって実行される。

1. 字句解析
2. 構文解析

3. 型の検査

4. 型付きλ計算から A-normal form への変換

5. A-normal form から SLAM への変換

型付きλ計算の式、型、A-normal form の式、SLAM の式は実行の途中で出力する。コンパイル過程における型の保存は既に説明したので、型の検査は初めに一度だけ行う。

4.3 実行

本節では、作成したコンパイラの実行の例を説明する。

実行の前段階として、以下を実行する。> はターミナルでの実行、– は sml での実行、– : は今回作成したコンパイラでの実行である。

```
> sml
- CM.make();
- open top;
- parse();
```

これにより、作成したコンパイラの実行環境が整う。

コンパイル結果の出力を以下に示す。入力したソースコード、型付きλ計算のソースコード、型チェックの結果、A-normal form のソースコード、SLAM のソースコードの順に示す。

```
-: (fn x :int => (fn y:int => (x,y))) 2 1;

[typed Lambda Calculus :]
APP(APP(ABS((x:int ).ABS((y:int ).PAIR(VAR(x),VAR(y)))) CONS(2)) CONS(1))

[types :]
INT * INT

[A-Normal Forms :]
ALET
   $2 = AABS(x:int AABS(y:int APAIR(AVAR(x),AVAR(y))))
IN
   AAPP($2 ACONS(2)) IS $3
   IN
      ALET
```

```
         $0 = AVAR($3)
      IN
         AAPP($0 ACONS(1)) IS $1
         IN
            AVAR($1)
         END
      END
   END
END
```

[STAL :]

Scode(	)	Sswap
Scode(	Sapp(0)	Spop
Sacc(1)	Sacc(0)	Sswap
Sacc(0)	Sconst(2)	Spop
Spair	Sapp(1)	Sswap
Sreturn	Sacc(0)	Spop
)	Sacc(0)	Sswap
Sacc(1)	Sconst(1)	Spop
Sapp(1)	Sapp(1)	Sreturn
Sreturn	Sacc(0)	

4.4 実行の解説

この節では前節のコンパイル結果に付いて解説する。

例として実行するソースコードは、本研究の特色である General A-normal forms($\wedge : L)(\vee : L)(\supset : L)(cut)$ を使ったものがよい。今回は $(\supset : L)$ を使用する。

今回実行したソースコードは型付きラムダ計算の以下のコードと同等である。

$\lambda x : int.\lambda y : int.(x, y)\ 2\ 1$

型検査について説明する。上の式を型付きラムダ計算の型システムを使ってチェックする。

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\Gamma, x : int \triangleright x : int \quad \Gamma, y : int \triangleright y : int}{\Gamma, x : int, y : int \triangleright (x, y) : int \wedge int}\ (\supset\!: E)}{\Gamma, x : int \triangleright \lambda y.(x, y) : int \supset int \wedge int}\ (\supset\!: E)}{\Gamma \triangleright \lambda x.\lambda y.(x, y) : int \supset int \supset int \wedge int \quad \Gamma \triangleright 2^{int} : int}\ (\supset\!: I)}{\Gamma \triangleright \lambda x.\lambda y.(x, y)\ 2 : int \supset int \wedge int \quad \Gamma \triangleright 1^{int} : int}\ (\supset\!: I)}{\Gamma \triangleright \lambda x.\lambda y.(x, y)\ 2\ 1 : int \wedge int}\ (\wedge : I)$$

型検査では結果型のみ返す。よって返す型は $(int \wedge int)$ である。

型付きラムダ計算から A-normal form への変換を考える。型検査は終了しているので、ソースに含まれる型付けを除くと、型付きラムダ計算は以下のように表される。

$\lambda x.\lambda y.(x, y)\ 2\ 1$

これに 2.4 節の変換アルゴリズムを適用すると、以下のコードを得る。

$\rightarrow let\ \$2\ =\ \lambda x.\lambda y.(x, y)\ in\ app(\$2\ 2)\ is\ \$3\ in\ let\ \$0\ =\ \$3\ in\ app(\$0\ 1)\ is\ \$1\ in\ \$1\ end\ end\ end$

上で導かれた A-normal form の式に 3.1 節のコンパイルアルゴリズムを適用すると以下の SLAM コードを得る。

$\rightarrow code(\ code(\ acc(1)\ acc(0)\ pair\ return)\ acc(1)\ app(1)\ return\ )$
$app(0)\ acc(0)\ const(2)\ app(1)\ acc(0)\ acc(0)\ const(1)\ app(1)\ acc(0)$
$swap\ pop\ swap\ pop\ swap\ pop\ swap\ pop\ return$

4.5 ソース説明

本節では作成したコンパイラのソースについて説明する。

コンパイラ全体のソースコードはトップループが管理する。トップループは以下を管理する。

1. データ型 (datatype)
2. スタック (stack) と環境 (environment)
3. 字句解析 (lex) と構文解析 (parse)
4. 型検査 (type check)
5. 変換 (translate)

6. 出力 (print)

実装したコンパイラは、データ型、スタック、環境の定義を前段階とし、字句解析、構文解析、型検査、変換、出力を行う。詳細は以下の小節で述べる。

4.5.1 データ型

この小節では、型付きラムダ計算、A-normal form、SLAM の各々のデータ型について説明する。型付きラムダ計算のデータ型は structure TL で、A-normal form のデータ型は structure TA で、SLAM のデータ型は structure TS で実装した。

まず、型付きラムダ計算、A-normal form、SLAM に共通する原子型 b と型 τ を定義する。原子型は b を以下に示す。

$$b ::= int \mid string$$

型 τ を以下に示す。

$$\tau ::= b \mid \tau \rightarrow \tau \mid \tau * \tau \mid \tau + \tau$$

$\tau_1 \rightarrow \tau_2$ は τ_1 から τ_2 への関数型である。$\tau_1 * \tau_2$ は τ_1 と τ_2 の論理積型である。$\tau_1 + \tau_2$ は τ_1 と τ_2 の論理和型である。論理積型は組の型であり、論理和型は取り出し (injection) の型である。

次に各々のデータ型を定義する。

型付きラムダ計算のデータ型は 2.1 節で説明した。structure TL に含まれるデータ型は、型付きラムダ計算の項 M に対応している。型付きラムダ計算のデータ型は、constant、variable、abstract、apply、pair、projection$_i$($\{1,2\} \in i$)、injection$_i$($\{1,2\} \in i$)、case を持つ。

A-normal form のデータ型は 2.2 節で説明した。structure TA に含まれるデータ型は、A-normal form の項 M に対応している。A-normal form のデータ型は、型付きラムダ計算のデータ型に let、fun、val を加えたものである。注意する点は、apply、projection、case、let である。このデータ型は、中間に値を持つため変化している。A-normal form の apply、projection、case、let のデータ型を以下に示す。

```
datatype term =
        Aapp of string * term * string * term
      | Aproj of string * (string * string) * term
      | Acase of string * string * term * string * term
      | Alet of string * term * term
```

SLAM のデータ型は 2.3 節で説明した。structure TS に含まれるデータ型は、SLAM の項 M に対応している。SLAM のデータ型は、return、swap、pop、const、acc、pair、projection$_i(\{1,2\} \in i)$、injection$_i(\{1,2\} \in i)$、switch、code、apply、call を持つ。

4.5.2 スタックと環境

この小節では、型検査で使った環境と、仮想機械 SLAM で使ったスタックについて述べる。環境とスタックは、束縛変数と自由変数を区別するために使われる。環境は structure SEnv で、スタックは structure Stack で実装した。

structure Stack には、スタックのデータ型とスタックを操作する関数を実装している。
スタックのデータ型を以下に示す。

```
datatype 'a stack = NONE | SOME of 'a * 'a stack
```

これは多層型'a を持つ stack が、NONE、もしくは、型'a と型'a のスタックの組を持つ SOME のどちらかを持つ事を表す。
スタックを操作する関数は、push、pop、peek、lookup、proj、inj、depth、reverse、stackToList、listToStackL、listToStack、pushlistL、pushlist、bottompush、bottompushlistL、bottompushlist、union、unionL がある。L はスタックに加えるリストの左がスタックのトップであることを表す。union はスタックとスタックの結合を表す。この中で重要なのは、関数 lookup である。関数 lookup の型を以下に示す。

```
val lookup : ''a stack * ''a -> int
```

関数 lookup は、スタックと変数の組を引数にとり、スタックのトップから数えた変数の番号を返す。
structure Stack は、A から SLAM へのコンパイルの時に使用する。

structure SEnv はスタックよりも高度な機能を持つ。しかし、SEnv では lookup 関数を作成していない。
structure SEnv は、型検査の時に使用する。変数が束縛されたとき、変数をラベルとする型を環境に追加する。そして束縛された変数の型が必要になったときに、環境から変数をラベルとする型を取り出す。つまり、型環境 Γ と Δ の動作を実現するために SEnv を使用する。

4.5.3 字句解析と構文解析

この小節では、字句解析と構文解析について説明する。

字句解析にはSMLで提供されているlexerを使用した。lexerを用いて、コマンドラインからの入力を受け取り、含まれる字句に対応するtokenの列を返す関数を実装する。tokenを以下に示す。

- 整数・文字列
- 記号 (“、.、(、)、,、=> 、|、=)
- 予約文字列 (fn、case、of、proj1、proj2、inj1、inj2、int、string、use)

構文解析にはSMLで提供されているparserを使用した。parserを用いて、tokenを受け取り、TL型のコードを返す関数を実装する。構文を以下に示す。NUMは整数、IDは文字列である。構文はstartから始まる。

```
start ::= exp
        | use ‘‘file’’

file ::= ID
       | file . ID

atmicexp ::= ( exp )
           | ( exp , exp )
           | NUM
           | ID

appexp ::= atmicexp
         | appexp atmicexp

exp ::= appexp
      | fn args -> exp
      | proj1 exp
      | proj2 exp
      | inj1 exp : ty
      | inj2 exp : ty
      | case exp of ID . exp | ID . exp

args ::= ID : ty
       | args . ID : ty

constant ::= int
           | string
```

```
ty ::= constant
     | ty => ty
     | ty * ty
     | ty + ty
```

この中で特別な構文は use である。use は入力ストリームをコマンドラインから ID 名のファイルに変更する。

4.5.4 型検査

この小節では、型付きラムダ計算の型検査について説明する。実装では型付きラムダ計算のソースコードのみ型検査する。型付きラムダ計算のソースコードの型が正しいならば、A-normal form、SLAM の型が正しいことが保証される。型付きラムダ計算の型システムは 2.1 節で、コンパイルにおける型の保存は前章までに説明した。

型検査は structure Typecheckで実装した。structure Typecheckは、TL 型のソースコードを引数に取り結果の型を返す関数 typecheck を持つ。関数 typecheckは、環境と TL 型のソースコードを引数に取り結果の型を返す関数 tycheck を持つ。関数 tycheck では、型付きラムダ計算の型システムに基づき、TL のソースコードの最期に現れた命令による場合分けを行う。TL の特別な命令は、ラムダ抽象 Abs、Case、変数 Var である。Abs と Case の構文が現れたとき、変数名をラベルにした型を環境に追加する。変数 Var が現れたとき、環境をチェックして変数に型を付ける。

4.5.5 変換

この小節では、型付きラムダ計算から A-normal form、SLAM への変換の実装について述べる。型付きラムダ計算から A-normal form への変換は structure Translate で、A-normal form から SLAM への変換は structure Translate2 で実装した。

まず、型付きラムダ計算から A-normal form への変換の実装について述べる。コンパイルアルゴリズムは 2.4 節で説明した。structure Translate では、型付きラムダ計算から A-normal form への変換のための関数 translate を実装した。translate は、TL 型のコードを引数に取り、TA 型のコードを返す。

中間のコードに付ける名前の生成について説明する。A-normal form は中間のコードに名前を付けるという特徴を持つ。そのとき新しい名前は一意でなければならない。実装では、一意な名前をつけるためにシンボル生成 (structure Gensym) を作った。Gensym は、initGensym、gensym という関数を持つ。initGensym を呼び出すことにより初期化され、gensym を呼び出すことにより一意な名前を生成する。Gensym は Translate の中で使用し

た。

次に、A-normal from から SLAM へのコンパイルを表す。コンパイルアルゴリズムは 3 章で説明した。structure Translate2 では、A-normal form から SLAM への変換のための関数 translate を実装した。translate は、TA 型のコードを引数に取り、TS 型のコードを返す。

SLAM での注意する点は、関数 lookup である。関数 lookup は、スタックに含まれる束縛変数と自由変数を区別するために使う。この実装では、lookup はスタックのトップからの番号を持つ。しかし、lookup は他の方法でも実装可能である。他の方法とは、環境に数値を持たせる、スタックのボトムからの番号を持たせる等の方法である。lookup は、SLAM のコードを実行する仮想機械を実装するときに再度考慮しなおすことが望ましい。

4.5.6 出力

この小節では、出力について説明する。出力のために structure Printer を作成した。structure Printer は以下の 4 つの関数を持つ。4 つの関数とは、型チェック結果の出力のための関数 prettyprinterTY、型付き λ 計算の出力のための関数 prettyprinterTS、A-normal form の出力のための関数 prettyprinterTA、SLAM の出力のための関数 prettyprinterTS である。これらの関数は各々の型のソースコードを受け取り文字列を返す。

第5章　結論

本論文では、中間言語 A-normal form から機械言語 SLAM へのコンパイルアルゴリズムを作り、型を保存したコンパイラの実装を行った。

今後、以下のものを作る必要がある。

1. コードの最適化
2. SLAM 仮想機械の実装
3. RLAM(Register-based LAM) への簡約の証明

まず、機械言語でのステップ数を減らすためにコード最適化の理論と実装を行う。例えば、スタックの第一要素と第二要素を交換する命令 (swap) と、スタックの第一要素を捨てる命令 (pop) は、共に使われることが多い。よって、(swap) と (pop) を合成した命令を作る。これにより、ステップ数を減少させることができる。

次に、SLAM 仮想機械のソースコードを実行する SLAM インタプリタを実装する。現状では、ソースコードの変換は行うが評価は行わない。よって、SLAM インタプリタを実装する。

最期に、RLAM への変換の証明と実装を行う。仮想機械にはスタックベースとレジスタベースがある。スタックベースの抽象機械は、SECD マシン [Landin:64] や、JAVA をはじめ多くの機械で使われている。しかし、スタックは実際のメモリとは扱い方が違うので実行が遅いという問題がある。よって、A-normal form から RLAM へのコンパイルアルゴリズムを作り実装を行う。

謝辞

本研究を進めるにあたり御指導して頂いた大堀淳教授に深く感謝致します。また、日頃から研究に関する様々な話題とヒントを頂いた計算機言語学講座の皆様に感謝致します。

関連図書

[1] A.W.Appel and T.Jim. Continuation-Passing, Closure-Passing Style. Principles of Programming Language, CS-TR-183-88, 1989.

[2] C.Flanagan, A.Sabry, B.F.Duba and M.Felleisen. The Essence of Compiling with Continuations. In conference on ACM Programming Language Design and Implementation(PLDI), pages 237-247 , 1993.

[3] J. Gallier. Constructive logics part I: A tutorial on proof systems and typed - calculi. Theoretical Computer Science, 110(2):249–339, March 1993.

[4] W.A.Howard. The formulae-as-types notion of construction. In J.R. Hindley and J.P. Seldin, editors, To H.B. Curry: Essays on combinatory logic, lambda calculus and formalism. Academic Press, 1980.

[5] S.Kleene. Introduction to Metamathematics. North-Holland, 1952, 7th edition.

[6] P.Landin. The mechanical evaluation of expressions. Computer Journal 6(4), 308-320. 1964.

[7] R.Milner, M.Tofte, R.Harper, D,MacQueen. The Definition of Standard ML (Revised), The MIT Press, 2001.

[8] G.Morrisett, D.Walker, K.Crary and N.Glew. From System F to typed assembly language. In Twenty-Fifth ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages, pages 85–97, San Diego, January 1998. Extended version published as Cornell University technical report TR97-1651.

[9] A.Ohori. A Curry-Howard Isomorphism for Compilation and Program Execution (Extended Abstract). In proc. Typed Lambda Calculi and Application, Springer LNCS 1581, pages 258-279, 1999.

[10] A.Ohori. The logical abstract machine: a Curry-Howard isomorphism for machine code. In Proceedings of International Symposium on Functional and Logic Programming, 1999.