情報処理学会論文誌　プログラミング　Vol. 1　No. 2　9–27 (Sep. 2008)

CPUの条件実行機能に対応した型付きアセンブリ言語

飯塚大輔[†1]　前田俊行[†1]　米澤明憲[†1]

型付きアセンブリ言語（TAL）は，高級言語コンパイラのターゲット言語となりうる，型安全な低級言語である．TALの型システムは，型付けされたプログラムが，クロージャ，タプル，またはユーザによって定義された抽象データ型のような高級言語の構造を破壊しないということを保証する．オリジナルのTALは一般的な理論上のRISCアーキテクチャに対して設計されており，具体的な実装はIA-32アーキテクチャ以外は考えられていない．本論文は，組み込み機器で広く使われているARMアーキテクチャに対応するTALについて説明する．ARMの重要な特徴の1つは条件実行機構である．条件実行とは，命令が実行されるかどうかが，実行時にそれぞれの命令に付随する条件フラグとCPUのステータスレジスタの条件ビットの値を比較することによって決定される，という機構である．より詳細にいうと，本論文は，条件実行をサポートするために，4ビットの条件ビットの値を型に写像する写像型を用いて，TALを拡張する．我々の型システムでは，1つのレジスタは，条件ビットの値によって同時に異なった型を持つことができる．

Typed Assembly Language for Conditional Execution

Daisuke Iizuka,[†1] Toshiyuki Maeda[†1]
and Akinori Yonezawa[†1]

Typed Assembly Language (TAL) is a type-safe low-level language that serves as a target language for compilers of high-level languages. TAL's type system ensures that well-typed programs do not violate high-level language abstractions, such as closures, tuples, and user-defined abstract data types. The original TAL was designed for a generic, theoretical RISC architecture, and no specific real-world architectures have been considered except for the IA-32 architecture. We present a TAL for the ARM architecture that is widely used in embedded computing environments. One key feature of ARM is Conditional Execution facility. Conditional Execution is the mechanism whether an instruction is executed or not is determined dynamically by a comparison of the conditional flag, which is specified in the instruction, and the value of the condition bits in the status register of the CPU. More specifically, we present an extension of TAL to support Conditional Execution with *mapping types* which map a value of the condition bits to a type. In our type system, a single register can have different types simultaneously, depending on the condition bits.

1. はじめに

静的に強く型付けされたプログラミング言語[2),7),13)]は，プログラムが実行時にメモリエラーを生じないことを型検査により実行前に保証することができる．このため，多くのアプリケーションが静的に強く型付けされた言語で記述されるようになった．

しかし，従来の静的型付け言語のコンパイル手法では，高級言語のレベルで型検査を行った後，アセンブリ言語や機械語などの低級言語へ変換するため，この変換に誤りがあった場合には，変換後のプログラムは実行時にメモリエラーを生じるかもしれない．また，変換後のプログラムに対しては型検査を行えないため，プログラムの実際の利用者は，元のソースプログラムがなければ，その安全性を検証するのは難しい．

型を保存するコンパイル手法（type-preserving compilation）[3),20),25)]は，高級言語の型情報を低級言語のレベルまで保存することで，この問題を解決しようとする試みである．具体的には，コンパイルに用いる種々の中間言語を型付き言語とし，高級言語での型情報を中間言語での型情報に変換する．

型付きアセンブリ言語（TAL: Typed Assembly Language）[20)]は，型を保存するコンパイル手法において，機械語のレベルまで型情報を保存するために提案されたアセンブリ言語である．このTALは，アセンブリ・機械語のレベルで，メモリ安全性（プログラムが不正なメモリアクセスを行わないこと）と制御フロー安全性（プログラムが不正なコード実行を行わないこと）を保証することができる．実際，Morrisettらは，IA-32アーキテクチャ[9)]向けのTALであるTALx86を設計・実装し，また，Popcornという言語（C言語に似た言語）から，TALx86へのコンパイラを実装した[18)]．

しかしながら，現存するTALの実装は，もっぱらIA-32アーキテクチャを対象としており[12),18)]，他のCPUアーキテクチャに関してはほとんど扱われていない．

そこで本論文は，ARMアーキテクチャ[23)]CPUに対応するためのTALの拡張手法を示す．ARMアーキテクチャは組み込み計算機環境で広く使われており，このアーキテクチャ

†1 東京大学大学院情報理工学系研究科
Graduate School of Information Science and Technology, The University of Tokyo

上での TAL を設計することは実用上意味がある．また ARM アーキテクチャは，従来の TAL の理論では考えられていなかった機構である条件実行機能[8] を備えており，これに対応する理論拡張を行う必要がある．条件実行機能とは，機械語命令に実行条件が付与されていて，CPU の実行状態に応じて，命令を実行するか否かを決定する機構である．

本論文では，条件実行機能をサポートするために，実行条件に関する CPU の実行状態を型に写像する写像型を導入する．この写像型により，従来の TAL と異なり，たとえば 1 つのレジスタが型検査時に複数の型を持つことができるようになる．これにより，条件実行機能を用いているアセンブリプログラムにおいても，実行時の条件により異なる制御フローが発生する可能性があるにもかかわらず，型検査をすることが可能となる．

以降の本論文の構成は次のとおりである．まず従来の TAL の手法について 2 章で説明する．次に条件実行機能について 3 章で説明し，条件実行機能の下での従来の TAL の手法の問題点を 4 章で述べる．次いで 5 章で我々の手法を説明し，6 章で具体的な型システムを説明する．また，7 章では我々の型システムの表現力について議論する．その後，8 章で関連研究について述べ，9 章で結論を述べる．

2. 従来の型付きアセンブリ言語

型付きアセンブリ言語（TAL）は，Morrisett らによって提案された，メモリ安全性と制御フロー安全性を型検査により保証することができる，静的に強く型付けされたアセンブリ言語である[20]．

TAL における型検査の概略を，以下のコード例を用いて説明する．

```
1: inc:
2:      add   r1, r0, #1
3:      jmp   r1
```

上記のコードにおいて，ラベル `inc` から始まる命令列は，まず，レジスタ `r0` の中身の値と即値 `1` の和をレジスタ `r1` に保存し（2 行目），次に，レジスタ `r1` の中身の値が表すアドレスへとジャンプする（3 行目）．ところが，このジャンプは実行時に不正なコード実行を生じてしまうかもしれない．なぜなら，レジスタ `r1` の中身は 2 行目で行われた整数演算の結果であり，つねに正当な命令列を指すとは限らないからである．

上記のコードを TAL を用いて記述すると，以下のコードのようになる．

```
1: inc: {r0:int, r1:{r0:int}}
2:      add   r1, r0, #1
3:      jmp   r1
```

まず 1 行目の `{r0:int, r1:{r0:int}}` は，ラベル型と呼ばれ，ラベル `inc` に実行が到達するときに満たされていなければならない条件を型として表している．具体的には，ラベル `inc` から始まる命令列を実行するためには，`r0` は整数型，`r1` はラベル型を保持していなければならないことを表している．なお，これらの型情報は，TAL のコードを直接記述する場合には，プログラマが注釈として記述する必要があるが，高級言語からの型を保存したコンパイルにおいては，高級言語の型情報からほぼ自動的に生成できることが示されている[12),18),20)]．

TAL の型検査は，基本ブロック単位で，次のように行われる．まず，基本ブロックのラベルに付与されたラベル型をもとに，型検査器の状態を初期化する．上記のコード例では，ラベル `inc` から始まるコード列を型付けするために，型検査器の状態を，`{r0:int, r1:{r0:int}}` のように初期化する．次に，2 行目の `add` 命令の型付けを行う．具体的には，まずレジスタ `r0` が整数型を持つことを検査する．次いでレジスタ `r1` の型を整数型に更新して，次の命令の型付けに移る．したがって，2 行目の命令の型付けは成功し，型検査器の状態は `{r0:int, r1:int}` のようになる．最後に，3 行目の `jmp` 命令の型付けを行う．具体的には，レジスタ `r1` がラベル型を持ち，そのラベル型が示す条件を型検査器の状態が満たしていることを検査する．ところが，型検査器の状態によると，レジスタ `r1` は整数型を持つので，3 行目の命令の検査はパスせず，したがって型エラーとなる．

3. 条件実行機能

条件実行機能[8] とは，命令を実行するか否かを，CPU の実行状態によって決定する命令実行機構である．条件実行機能の下での命令実行の流れは以下のとおりである．CPU は実行する命令をメモリから読み出すと，まず命令に含まれている条件コードを確認する．もし，この条件コードが現在の CPU の実行状態に合致するならば，CPU は実際にその命令を実行する．一方，もし合致していなければ，CPU はその命令を無視し，次の命令を実行しようとする．

条件実行機能を実現している CPU アーキテクチャの 1 つが ARM アーキテクチャである[23]．ARM アーキテクチャにおいては，条件実行機能に用いられる CPU の実行状態は，カレントプログラムステータスレジスタ（CPSR）と呼ばれるレジスタの上位 4 ビットで表される．これらのビットは条件フラグと呼ばれ，それぞれ N，Z，C，V（ネガティブ，ゼロ，キャリー，オーバフロー）と名付けられている．これらの条件フラグは，比較命令など

表 1　条件コードの定義
Table 1　Definition of condition code.

条件コード	条件類	意味	条件フラグの状態
0000	eq	=	Z セット
0001	ne	≠	Z クリア
0010	cs/hs	キャリーセット / 符号なし ≧	C セット
0011	cc/lo	キャリークリア / 符号なし <	C クリア
0100	mi	マイナス / 負	N セット
0101	pl	プラス / 正またはゼロ	N クリア
0110	vs	オーバフロー	V セット
0111	vc	オーバフローなし	V クリア
1000	hi	符号なし >	C セットおよび Z クリア
1001	ls	符号なし ≦	C クリアまたは Z セット
1010	ge	符号付き ≧	N セットおよび V セット，または N クリアおよび V クリア（N == V）
1011	lt	符号付き <	N セットおよび V クリア，または N クリアおよび V セット（N != V）
1100	gt	符号付き >	Z クリアで，N セットおよび V セットか， N クリアおよび V クリアのいずれか（Z == 0, N == V）
1101	le	符号付き ≦	Z セット，または N セットおよび V クリア，または N クリアおよび V セット（Z == 1 または N != V）
1110	al	常時（無条件）	-
1111	-	特別扱い	-

のデータ処理命令の操作の結果に基づいて更新される．

一方，ARM アーキテクチャにおける機械語命令（32 ビット）の 31 ビット目から 28 ビット目には，**表 1** によって示される条件コードが含まれている．上述の条件フラグが，条件コードで指定された条件を満たしている場合のみ，その命令は実行される．条件フラグが条件を満たしていない場合は，その命令は NOP 命令（何も実行しない命令）として解釈される．なお，表 1 における条件類とは，ARM アセンブリの命令ニーモニックの末尾に追加して条件コードを表すための，ニーモニック拡張部分である．なお，条件類が指定されていない場合は，条件類 *al*（無条件実行）が指定されたものと見なされる．

以下，ARM アーキテクチャの条件実行機能の挙動を例を通して説明する．**図 1** は，条件実行機能を用いた ARM アセンブリプログラムの例である．この図 1 のプログラムの実行の流れは次のとおりである．まず，レジスタ `r1` とレジスタ `r2` の値が比較され，その比較結果により，CPU の条件フラグの値が更新される（2 行目）．3 行目以降の命令の実行は，この更新された条件フラグの値によって変化する．たとえば，3 行目と 5 行目の命令は，表 1 で示されるとおり，Z（ゼロ）がクリアされていて，かつ N（ネガティブ）の値と V（オーバフロー）の値が同じであるときのみ実行される．一方，4 行目と 6 行目の命令は，条件フラグ Z（ゼロ）がセットされているときのみ実行される．このように，条件実行機能を用いたプログラムでは，条件フラグに依存して実行の流れが決まるため，1 つの基本ブロックの

```
1: label:
2:      cmp   r1, r2
3:      movgt r0, #1
4:      moveq r0, label
5:      addgt r0, r0, r0
6:      beq   r0
7:      ...
```

図 1　条件実行機能を用いた ARM アセンブリプログラムの例
Fig. 1　An example ARM assembly program with conditional execution.

ように見えても，実際には複数の制御フローが存在することがある．

4. 条件実行機能の下での従来の TAL の問題点

従来の TAL の手法を，条件実行機能の存在する CPU アーキテクチャへ応用するには 1 つ問題がある．それは，従来の TAL の型付けは基本ブロック単位で行われるのに対し，条件実行機能を用いたプログラムでは，1 つの基本ブロック内に複数の制御フローが存在しうる点である．

図 2 は，従来の TAL の手法を図 1 のコードに適用した例である．このコードは，実行時にはエラーを生じないにもかかわらず，従来の TAL の型検査では型エラーと判定されてしまう．

具体的には，従来の TAL の型検査は次のように行われる．まず 1 行目より，型検査器の状態が {r1:int, r2:int} と初期化される．次に 2 行目の比較命令の型検査を行う．具体的には，レジスタ r1 とレジスタ r2 の型がともに整数型であることを検査する．次いで 3 行目の mov 命令では，即値 1 がレジスタ r0 に保存されているため，型検査器の状態を {r0:int, r1:int, r2:int} のように更新する．同様に 4 行目では，ラベル（コードアドレス）label がレジスタ r0 に保存されているため，型検査器の状態は {r0:{r1:int, r2:int}, r1:int, r2:int} のように更新される．ここで，5 行目の加算命令の型検査を行うが，この検査はパスしない．なぜなら，この加算命令の型検査では，レジスタ r0 が整数型を持つことが検査されるが，4 行目までの型検査により，型検査器はレジスタ r0 の型は整数型と判断しているためである．

図 2 の例では，実行時にはエラーを生じないプログラムが型エラーと判定されたが，逆のケース（実行時にはエラーを生じるが，型エラーと判定されないケース）も存在する．

```
1: label: {r1:int, r2:int}
2:      cmp   r1, r2
3:      movgt r0, #1
4:      moveq r0, label
5:      addgt r0, r0, r0
6:      beq   r0
7:      ...
```

図 2　TAL で条件実行機能を扱おうとする例
Fig. 2　An example program using conditional execution with TAL.

このように，条件実行機能の下で従来の TAL の型検査が保守的になったり，逆に健全性を失ったりする理由は，1 つの基本ブロック内であっても，条件実行機能によって複数の制御フローが隠されているかもしれないことを考慮していないためである．

条件実行機能による複数の隠れた制御フローの存在をなくす最も単純な方法は，条件類の付いた命令を，従来の条件分岐命令を用いて変換する方法である．**図 3** は図 2 の例における条件類のついた各命令を，条件分岐命令を用いて変換した例である．この例では，まず元の例の movgt 命令を条件分岐命令 ble を用いて変換し，続く命令列に対しては新たにラベル L1 を導入し，別の基本ブロックにしている．moveq 以下の命令の変換も同様である．

しかし，図 3 のように変換したとしても，従来の TAL の型検査ではうまくいかない．変換後に新しく導入されたラベル L2 の型を考えると，12 行目の add 命令で r0 を用いているので，ラベル L2 の型中では r0 は整数型でなければならない（10 行目）．一方 8 行目では，ラベル label がレジスタ r0 に保存されているため，従来の TAL の型検査では，型検

```
1:  label: {r1:int,r2:int}
2:       cmp   r1, r2
3:       ble   L1
4:       mov   r0, #1
5:       b     L1
6:  L1: {}
7:       bne   L2
8:       mov   r0, label
9:       b     L2
10: L2: {r0:int}
11:      ble   L3
12:      add   r0, r0, r0
13:      b     L3
14: L3: {r0:{r1:int,r2:int}}
15:      beq   r0
16:      ...
```

図 3　条件分岐命令を用いて条件類の付いた命令を変換した例
Fig. 3　An example of converting instructions annotated with conditional kinds to branch instructions.

```
1:  label: {r1:int,r2:int}
2:       cmp   r1, r2
3:       bgt   Lgt
4:       beq   Leq
5:       b     Lf
6:  Lgt: {}
7:       mov   r0, #1
8:       add   r0, r0, r0
9:       b     Lf
10: Leq: {}
11:      mov   r0, label
12:      b     r0
13: Lf:
14:      ...
```

図 4　基本ブロック内の同じ条件類が付与されている命令をまとめる例
Fig. 4　An example of grouping instructions by conditional kinds.

査器の状態は {r0:{r1:int, r2:int}} のようになる．したがって，9 行目の分岐命令は，従来の TAL の型検査をパスしない．なぜならこの分岐命令では，L2 のラベルの型が示すとおり，レジスタ r0 は整数型を持たなければならないからである．

また別の変換方法として，それぞれの命令単位で条件分岐命令を用いて変換するのではなく，基本ブロック内の同じ条件類が付与されている命令をまとめる方法が考えられる．**図 4** は，図 2 中の命令を，条件類 gt と eq についてそれぞれ基本ブロックにまとめたものである．

この方法は，図 2 の例ではうまくいくように見えるが，実際には問題が解決されるわけではない．たとえば，**図 5** は，図 2 中の各命令をそれぞれ基本ブロックに分けたものであるが，これを上述の方法で変換しようとしても，そもそもまとめられる命令が各基本ブロック中になく，図 3 と同様の問題が基本ブロックのレベルで生じてしまい，うまくいかない．結局，従来の TAL の型システムの範囲内でこの問題を解決するためには，基本ブロックをまたがる解析を行う必要があり，容易ではない．

```
1:  label: {r1:int,r2:int}
2:       cmp   r1, r2
3:       b     L1
4:  L1:
5:       movgt r0, #1
6:       b     L2
7:  L2:
8:       moveq r0, label
9:       b     L3
10: L3:
11:      addgt r0, r0, r0
12:      beq   r0
13:      ...
```

図 5　同じ条件類が付与されている命令をまとめる方法がうまくいかない例
Fig. 5　An example program whose instructions cannot be grouped by conditional kinds.

5. 条件実行機能へ対応するための TAL の拡張手法

5.1 写像型の導入

前章で述べた，条件実行機能により生じる隠された制御フローの問題を解決するため，我々は，条件実行機能にかかわる CPU の実行状態に対応して異なる型を扱うことができるように TAL の型システムを拡張する．具体的にはレジスタの型を，条件フラグに応じて型が決定されるような写像型として扱う．写像型 ξ の定義は次のとおりである．

$$\xi \equiv \{\texttt{0000} \mapsto \tau_{\texttt{0000}}, \texttt{0001} \mapsto \tau_{\texttt{0001}}, \cdots, \texttt{1111} \mapsto \tau_{\texttt{1111}}\}$$

ここで，0000, . . . , 1111 は条件フラグ（N，Z，C，V）の状態を表しており，$\tau_{\texttt{0000}}, \ldots, \tau_{\texttt{1111}}$ は条件フラグのとりうる状態に対応する型を表している．各命令の型付けでは，型検査器は条件コード（条件類）が示す状態に対応した型を扱う．たとえば，図 2 のプログラムに対して写像型を用いた我々の型検査を適用すると次のようになる．まず，型検査器はラベル型より状態を {r1: { **** ↦ int }, r2: { **** ↦ int } } のように初期化する（1 行目）．ただし，ここで * は 0 と 1 の両方を表すものとする．すなわち，上述のラベル型は，条件フラグがどんな値であっても，レジスタ r1 とレジスタ r2 が整数型を持つことを表している．次に，2 行目の比較命令の型検査を行うが，上述のとおりレジスタ r1 とレジ

スタ r2 は必ず整数型を持つので，この型検査はパスする．次いで 3 行目の mov 命令の型検査を行う．ここで，この mov 命令は条件類（gt）が指定されているので，この条件類に対応した部分だけレジスタ r0 の写像型を更新する．具体的には，表 1 より，型検査器の状態は { r0 : { 00*0 ↦ int, 10*1 ↦ int }, ... } のように更新される（ただし，ここではレジスタ r1 と r2 の型は，以降の型検査に影響しないため，省略した）．同様に，4 行目の mov 命令の型検査では，条件類 eq に対応する部分だけ扱われるため，型検査器の状態は { r0 : { 00*0 ↦ int, 10*1 ↦ int, *1** ↦ { ... } }, ... } のように更新される．gt に関する条件フラグと eq に関する条件フラグは排他であり互いに干渉しないため，この時点でレジスタ r0 は 2 つの型を同時に持つことになる．ここで，5 行目の加算命令の型検査を行うが，4 章で示した従来の TAL の型付けとは異なり，我々の型検査はパスする．なぜなら，この加算命令には条件類 gt が指定されているため，我々の型検査はレジスタ r0 の写像型のうち，条件フラグの状態が 00*0，10*1 に対応する部分のみを扱うからである．上述した型検査器の状態によると，レジスタ r0 中で条件類 gt に対応する部分の型は，すべて整数型となっている．6 行目の条件分岐命令の型検査も，条件類 eq に対応する部分（*1**）のみ扱うため，同様にパスする．

5.2　比較命令の扱い

前節のように，写像型を導入することで，条件実行機能により 1 つの基本ブロック中に隠された複数の制御フローに応じた型検査を正確に行うことができる．ただし，比較命令に関しては，条件実行機能にかかわる CPU の実行状態（条件フラグ）を変更するため，本節で述べるような扱いが必要である．

たとえば**図 6** のプログラムの型検査を考える．条件実行機能に対応した我々の型検査では，4 行目の型検査の前では，型検査器の状態は { r0 : { 00*0 ↦ int, 10*1 ↦ int, *1** ↦ { ... } }, ...} のようになる．ここで，4 行目の比較命令の型検査を行うが，このとき，レジスタ r1 とレジスタ r2 の検査に加え，レジスタ r0 の型の更新も行う必要がある．なぜなら，この比較命令により CPU の条件フラグが変化する可能性があるからである．仮にもしここで型を更新しなかったとすると，条件類 eq に対応するレジスタ r0 の型はラベル型であるから，5 行目の分岐命令は型検査にパスしてしまうが，実際には 4 行目の比較命令で条件フラグが変わっているかもしれないため，実行時には不正なコード実行が生じる可能性がある．

```
1: label: {r1:int,r2:int}
2:      movgt r0, #1
3:      moveq r0, label
4:      cmp   r1, r2
5:      beq   r0
6:      ...
```

図 6　TAL で条件実行機能を扱おうとする別の例
Fig. 6　Another example program using conditional execution with TAL.

具体的には，比較命令の型検査では，レジスタの写像型の単一化を行う．ここでいう単一化とは，条件フラグのとりうるすべての状態について，同一の型を持つように変換することである．たとえば，写像型が任意の状態に対してすでに同一の型を持つならば，単一化後もその写像型は変化しない．一方，もし異なる型を持つ状態が 1 つでもあるならば，その写像型のすべての状態に対して型 ⊤ を割り当てる．たとえば，図 6 の 4 行目の比較命令の型検査では，型検査器の状態が { r0 : ⊤, ...} のように単一化されるため，5 行目の分岐命令の型検査で型エラーとなる．なぜなら，この分岐命令はレジスタ r0 の値を読み込むが，レジスタ r0 は任意の条件フラグにおいて型 ⊤ を持つからである．

6. 条件実行機能に対応した型付きアセンブリ言語

本章では，前章までに述べた手法に基づいて拡張した TAL の文法，操作的意味論，型付け規則について説明する．我々の TAL は，文献 20) の TAL をベースに拡張したものである．また，条件実行機能をサポートしている ARM アーキテクチャへの対応を考え，命令セットなどを ARM アーキテクチャのそれに近付けてある．

6.1　文　　法

我々の TAL の抽象機械の文法，型の文法は**図 7** のとおりである．抽象機械の状態 P は，ヒープ（H），レジスタファイル（R），命令列（I），条件フラグ（C）の 4 つ組で表される．ヒープはラベル（l）からヒープ値（h）への写像であり，ヒープ値はワード値（w）のタプルまたは命令列である．ワード値は，整数やラベルなど，レジスタに直接保存できる値を表す．レジスタファイルはレジスタ（r）からワード値への写像である．

レジスタは，通常のレジスタ $\mathtt{r}_1$〜$\mathtt{r}_n$ に加えて，関数呼び出し命令で用いられるリンクレジスタ lr が存在する．実際の ARM アーキテクチャでは，関数呼び出し命令が実行されると，関数からの返りアドレスがリンクレジスタに保存されるが，我々の TAL では，関数呼び出し命令に明示的に返りアドレスを指定するようにしている．

レジスタファイル型 Γ の定義より分かるように，レジスタは直接に型（τ）を持つのでは

型	τ	$::= \alpha \mid int \mid \forall[\Delta].\Gamma \mid \langle \tau_1^{\varphi_1}, \ldots, \tau_n^{\varphi_n} \rangle \mid \exists \alpha.\tau \mid \top$
初期化フラグ	φ	$::= 0 \mid 1$
ヒープ型	Ψ	$::= \{l_1 : \tau_1, \ldots, l_n : \tau_n\}$
型環境	Δ	$::= \cdot \mid \alpha, \Delta$
レジスタファイル型	Γ	$::= \{r_1 : \xi_1, \ldots, r_n : \xi_n, \mathtt{lr} : \xi_{lr}\}$
レジスタ写像型	ξ	$::= \{s_1 \mapsto \tau_1, \ldots\}$
レジスタ	r	$::= \mathtt{r}_1 \mid \cdots \mid \mathtt{r}_n \mid \mathtt{lr}$
ワード値	w	$::= l \mid i \mid ?\tau \mid w[\tau] \mid pack[\tau, w]\ as\ \tau'$
値	v	$::= r \mid w \mid v[\tau] \mid pack[\tau, v]\ as\ \tau'$
ヒープ値	h	$::= \langle w_1, \ldots, w_n \rangle \mid \mathtt{code}[\Delta]\Gamma.I$
ヒープ	H	$::= \{l_1 \mapsto h_1, \ldots, l_n \mapsto h_n\}$
レジスタファイル	R	$::= \{r_1 \mapsto w_1, \ldots, r_n \mapsto w_n, \mathtt{lr} \mapsto w_{lr}\}$
条件状態	s	$::= 0000 \mid 0001 \mid 0010 \mid 0011 \mid 0100 \mid 0101 \mid 0110 \mid 0111$
		$1000 \mid 1001 \mid 1010 \mid 1011 \mid 1100 \mid 1101 \mid 1110 \mid 1111$
条件類	$cond$	$::= \{s_1, s_2, s_3, \ldots\}$
命令	ι	$::= aop(cond)\ r_d, r_s, v \mid \mathtt{cmp}\ r, v \mid \mathtt{unpack}(cond)\ [\alpha, r_d], v \mid$
		$\mathtt{malloc}(cond)\ r[\vec{\tau}] \mid \mathtt{b}(cond)\ v \mid \mathtt{mov}(cond)\ r_d, v \mid$
		$\mathtt{ld}(cond)\ r_d[r_s, i] \mid \mathtt{st}(cond)\ r_s[r_d, i]$
算術演算	aop	$::= \mathtt{add} \mid \mathtt{sub} \mid \mathtt{mul}$
命令列	I	$::= \iota; I \mid \mathtt{bl}(cond)\ v,\ v_{next} \mid \mathtt{b}\ v$
条件フラグ	C	$::= \mathtt{NZCV}$
	N,Z,C,V	$::= 0 \mid 1$
抽象機械状態	P	$::= (H, R, I, C)$

図 7　我々の TAL の文法
Fig. 7　Syntax of our TAL.

表 2　条件類とその略称
Table 2　Conditional kinds and their abbreviated names.

略称	条件類
eq	{ *1** }
ne	{ *0** }
cs	{ **1* }
cc	{ **0* }
mi	{ 1*** }
pl	{ 0*** }
vs	{ ***1 }
vc	{ ***0 }
hi	{ *01* }
ls	{ **0* ∨ *1** }
ge	{ 0**0 ∨ 1**1 }
lt	{ 0**1 ∨ 1**0 }
gt	{ 00*0 ∨ 10*1 }
le	{ *1** ∨ 0**1 ∨ 1**0 }
al	{ **** }

ただし，* は 0 または 1 のどちらでもよいことを表す.

なく，5 章で述べたように，写像型（ξ）を持つ．写像型は，条件状態（s）から型への写像である．条件状態は，条件実行機能にかかわる実行状態を表し，ここでは 0000〜1111 の 16 種類存在する．

命令列は，文字どおり命令（ι）の列である．ただし，命令列の末尾は必ず，関数呼び出し命令（bl）か，無条件分岐命令（b）である．プログラムの実行は，6.2 節で述べる命令の操作的意味論に従って，抽象機械の状態 P を変換していくことに対応する．また，条件実行機能を表現するために，比較命令（cmp）と無条件分岐命令（b）以外の命令には，条件類（$cond$）が指定できる．条件類は，条件状態の集合として定義され，抽象機械状態中の条件フラグが，命令に指定された条件類に含まれているときのみその命令は実行される．なお，実際の ARM アセンブリプログラムで用いられる表記と我々の TAL の条件類の対応は**表 2** のとおりである．

なお，実際の ARM アーキテクチャでは，複数データ処理を一括して行う命令や，プログラムカウンタをオペランドに使用した命令が存在するが，そのような命令は，我々の TAL の命令を組み合わせることによって表現することが可能である．各々の命令の動作については，6.2 節で述べる．

6.2　操作的意味論

我々の TAL の操作的意味論は，**表 3** のとおりである．基本的に各命令は，命令に指定された条件類に条件フラグ C が含まれているときに実行され，それ以外のときには状態に影響を与えず，単に次の命令の実行へ移るのみである．ただし，比較命令 `cmp` と，命令列末尾の無条件分岐命令 `b` には，つねに実行されることを表す条件類 `al` が指定されているものと見なす．以下，各命令の操作的意味論について説明する．

比較命令 `cmp` は，指定された 2 つのオペランド（r と v）の値を比較し，その比較結果を条件フラグ C に反映させる命令である．条件フラグを更新する関数 $setC(w_1, w_2)$ は，`NZCV` それぞれのフラグに対する更新をまとめたものである．具体的には，w_1 と w_2 の比較結果が，負であれば `N` がセットされ，ゼロであれば `Z` がセットされ，キャリーが生じれば `C` がセットされ，オーバフローが生じれば `V` がセットされる．

算術命令（`add`，`sub`，`mul`）は，第 2 オペランド（r_s）と第 3 オペランド（v）を演算した結果を第 1 オペランド（r_d）に保存する．なお，実際の ARM アーキテクチャには，演算と同時に条件フラグを更新するような算術命令も存在するが，本質的には比較命令との組合せで実現できるため，本論文では省略した．`mov` 命令は，第 2 オペランド（v）の値を第 1 オペランド（r_d）に保存するのみである．

ヒープアクセス命令 `ld`，`st` は，第 2 オペランドで指定されたアドレス（ラベル）に存在するタプルの，第 3 オペランド i（即値）で指定されたオフセットの要素に対するメモリ読み込み，メモリ書き込みを行う．指定されたラベルにタプルが存在しない場合や，命令列が存在する場合には，状態遷移はスタックする．これは不正なメモリアクセスを意味する．

`malloc` 命令はヒープに新たなタプルを確保する命令である．確保されたタプルを指すラベルは，既存のヒープ中のどのラベルとも異なるラベルが割り当てられる．また，割り当てられたタプルの各要素は，未初期化の値を表す値 $?\tau$ で初期化される．なお，この `malloc` 命令は ARM アーキテクチャには存在しないため，我々の TAL を実際に実装する際には，メモリ確保のための関数呼び出しなどに置き換える必要がある．また，確保したタプルを解放する命令はないため，ガーベジコレクションなどのメモリ管理機構を利用する必要がある．

`unpack` 命令は，依存型[17] として pack されている値を，unpack して利用できるようにする命令である．この命令は，高級言語での関数クロージャを TAL に変換する際に必要となる[20]．この `unpack` 命令も実際の ARM アーキテクチャには存在しないが，型に関する操作しか行わないため，`malloc` 命令とは異なり，単に無視する（削除する）だけでよい．

分岐命令 `b` は，オペランド v で指定されたラベルが指す命令列へジャンプする．指定されたラベルに命令列が存在しない場合や，タプルが存在する場合には，状態遷移はスタックする．これは不正なコード実行を意味する．

`bl` 命令は，関数呼び出し命令である．リンクレジスタ `lr` に第 2 オペランドで指定されたラベル（v_{next}）を保存し，第 1 オペランド（v）で指定されたラベルにジャンプする．実際の ARM アーキテクチャでは，我々の TAL とは異なり第 2 オペランドは存在せず，リンクレジスタには，次の命令のアドレスが保存される．我々の TAL は，命令列の間の位置関

表 3　我々の TAL の操作的意味論
Table 3　Operational semantics of our TAL.

$(H, R, I, C) \mapsto P$ where	
if $I =$	then $P =$
`cmp` $r, v; I'$	(H, R, I', C') where $C' = setC(R(r), \hat{R}(v))$
`b` v	$(H, R, I'[\psi/\Delta], C)$ where $\hat{R}(v) = l[\psi]$ and $H(l) =$ `code`$[\Delta]\Gamma.I'$
else if $C \in cond$ and $I =$	then $P =$
$aop(cond)$ $r_d, r_s, v; I'$	$(H, R[r_d \mapsto R(r_s)\ aop\ \hat{R}(v)], I', C)$
`unpack` $(cond)$ $[\alpha, r_d], v; I'$	$(H, R[r_d \mapsto w], I'[\tau/\alpha], C)$ where $\hat{R}(v) = pack[\tau, w]\ as\ \tau'$
`malloc` $(cond)$ $r_d, [\tau_1, \ldots, \tau_n]; I'$	$(H[l \mapsto \langle ?\tau_1, \ldots, ?\tau_n\rangle], R[r_d \mapsto l], I', C)$ where $l \notin H$
`mov` $(cond)$ $r_d, v; I'$	$(H, R[r_d \mapsto \hat{R}(v)], H, I', C)$
`ld` $(cond)$ $r_d, [r_s, i]; I'$	$(H, R[r_d \mapsto w_i], I', C)$ where $R(r_s) = l$ and $H(l) = \langle w_0, \ldots, w_{n-1}\rangle$ and $0 \le i < n$
`st` $(cond)$ $r_s, [r_d, i]; I'$	$(H[l \mapsto \langle w_0, \ldots, w_{i-1}, R(r_s), w_{i+1}, \ldots, w_{n-1}\rangle], R, I', C)$ where $R(r_d) = l$ and $H(l) = \langle w_0, \ldots, w_{n-1}\rangle$ and $0 \le i < n$
`b` $(cond)$ $v; I'$	$(H, R, I[\psi/\Delta], C)$ where $\hat{R}(v) = l[\psi]$ and $H(l) =$ `code`$[\Delta]\Gamma.I'$
`bl` $(cond)$ v, v_{next}	$(H, R[lr \mapsto \hat{R}(v_{next})], I'[\psi/\Delta], C)$ where $\hat{R}(v) = l[\psi]$ and $H(l) =$ `code`$[\Delta]\Gamma.I'$,
else if $C \notin cond$ and $I =$	then $P =$
`bl` $(cond)$ v, v_{next}	$(H, R, I'[\psi/\Delta], C)$ where $\hat{R}(v_{next}) = l_{next}[\psi]$ and $H(l_{next}) =$ `code`$[\Delta]\Gamma'.I'$
otherwise	then $P =$
$\iota; I'$	(H, R, I', C)

$$\text{where}\quad \hat{R}(v) = \begin{cases} R(r) & when\ v = r \\ w & when\ v = w \\ \hat{R}(v')[\tau] & when\ v = v'[\tau] \\ pack[\tau, \hat{R}(v')]\ as\ \tau' & when\ v = pack[\tau, v']\ as\ \tau' \end{cases}$$

$$[\tau_1, \ldots, \tau_n/\alpha_1, \ldots, \alpha_n] = [\tau_1/\alpha_1] \ldots [\tau_n/\alpha_n]$$

表 4　型付け規則の種類
Table 4　Judgments.

型付け規則	意味
$\Delta \vdash \tau$	型 τ は well-formed である
$\vdash \Psi$	ヒープ型 Ψ は well-formed である
$\Delta \vdash \Gamma$	レジスタファイル型 Γ は well-formed である
$\Delta \vdash \tau_1 = \tau_2$	型 τ_1 と τ_2 は等しい
$\Delta \vdash \Gamma_1 = \Gamma_2$	レジスタファイル型 Γ_1 と Γ_2 は等しい
$\Delta \vdash \tau_1 \leq \tau_2$	τ_1 は τ_2 のサブタイプである
$\Delta \vdash \xi$	写像型 ξ は well-formed である
$\Delta \vdash \xi_1 \leq \xi_2$	写像型 ξ_1 は ξ_2 のサブタイプである
$\Delta \vdash \Gamma_1 \leq \Gamma_2$	レジスタファイル型 Γ_1 は Γ_2 のサブタイプである
$\vdash H : \Psi$	ヒープ H はヒープ型 Ψ を持つ
$\Psi \vdash R : \Gamma@C$	レジスタファイル R は条件フラグ C の下でレジスタファイル型 Γ を持つ
$\Psi \vdash h : \tau$ hval	ヒープ値 h は型 τ を持つ
$\Psi; \Delta \vdash w : \tau$ wval	ワード値 w は型 τ を持つ
$\Psi; \Delta \vdash w : \tau^{\varphi}$ fwval	ワード値 w は初期化フラグ付きの型 τ^{φ} を持つ (i.e., w は型 τ を持つ．もしくは w は $?\tau$ であり初期化フラグ φ は 0 である)
$\Psi; \Delta; \Gamma \vdash v : \tau@cond$	値 v は条件類 $cond$ を満たす状態において型 τ を持つ
$\Psi; \Delta; \Gamma \vdash \iota \Rightarrow \Delta'; \Gamma'$	命令 ι の事前条件は $\Psi; \Delta; \Gamma$ であり，事後条件は $\Psi; \Delta'; \Gamma'$ である
$\Psi; \Delta; \Gamma \vdash I$	命令列 I は well-formed である
$\vdash P$	プログラム P は well-formed である

係（どの命令列がどの命令列とつながっているかなど）を扱っておらず，「次の命令のアドレス（ラベル）」を直接知ることはできないため，明示的に第 2 オペランドを指定する必要がある．命令列間の位置関係を扱うように TAL を拡張することは容易であるが，条件実行機能の観点からは本質的ではないため，本論文では扱わない．

6.3　型付け規則

表 4 は我々の TAL の型付け規則の種類の一覧である．これらの型付け規則は次の定理を満たす．

定理 1 $\vdash P$ ならば，$P \mapsto P'$ なる P' が存在し，$\vdash P'$ を満たす．

つまり，我々の型付け規則において，P が well-formed であると判定されれば，P はスタックしない，すなわち，不正なメモリアクセスや不正なコード実行を生じない．

6.3.1　抽象機械の状態の型付け

図 8 は抽象機械の状態の型付け規則である．状態（H，R，I，C）が well-formed であるとは，ヒープ H がヒープ型 Ψ を持ち，このヒープ型の下で，レジスタファイル R がレジスタファイル型 Γ を持ち，これらのヒープ型とレジスタファイル型の下で，命令列 I が well-formed であると定義される．

$$\boxed{\vdash P \quad \vdash H : \Psi \quad \Psi \vdash R : \Gamma@C}$$

$$\frac{\vdash H : \Psi \qquad \Psi \vdash R : \Gamma@C \qquad \Psi; \cdot; \Gamma \vdash I}{\vdash (H, R, I, C)} \quad (\textsc{program})$$

$$\frac{\vdash \Psi \qquad \Psi \vdash h_i : \tau_i \ hval \qquad \Psi = \{l_1 : \tau_1, \ldots, l_n : \tau_n\}}{\vdash \{l_1 \mapsto h_1, \ldots, l_n \mapsto h_n\} : \Psi} \quad (\textsc{heap})$$

$$\frac{\Psi; \cdot \vdash lr : \xi_{lr}(C) \qquad \Psi; \cdot \vdash w_i : \tau_i \ wval \qquad \tau_i = \xi_i(C) \qquad (1 \leq i \leq n \leq m)}{\Psi \vdash \{r_1 \mapsto w_1, \ldots, r_m \mapsto w_m, lr \mapsto w_{lr}\} : \{r_1 : \xi_1, \ldots, r_n \mapsto \xi_n, lr \mapsto \xi_{lr}\}@C} \quad (\textsc{register file})$$

図 8　抽象機械状態の型付け規則
Fig. 8　Typing rules for abstract machine states.

条件実行機能に対応するために，レジスタファイル R の型付けでは，条件フラグ C を考慮する必要がある．具体的には，レジスタファイル型 Γ 中に含まれているすべてのレジスタ（r_i）について，レジスタファイル中の対応するレジスタが保存している値（$R(r_i)$）が，レジスタファイル型で示された写像型中の条件フラグ C に対応する型（$\Gamma(r_i)(C)$）を持つことを検査する．

6.3.2　命令列の型付け

図 9 と**図 10** は命令列の型付け規則である．

分岐命令 `b` の型付けは，オペランド v が条件類 $cond$ の下でラベル型 $\forall[].\Gamma_2$ を持ち，現在のレジスタファイル型 Γ_1 が Γ_2 のサブタイプであることを検査する（分岐命令の型付け規則は，命令列の末尾の場合と命令列の途中の場合の 2 通りあるが，基本的に検査することは同一である）．

条件類 $cond$ の下での値 v の型付け規則は，**図 11** と**図 12** で定義される．この値の型付け規則で条件類が考慮されるのはレジスタの型付けの部分である．具体的には，レジスタ r が条件類 $cond$ の下で型 τ を持つということは，レジスタファイル型中の $cond$ に含まれるすべての条件状態において型 τ を持つことと定義される．この判定には，**図 13** で定義さ

$\boxed{\Psi;\Delta;\Gamma \vdash I}$

$$\frac{\Delta \vdash \Gamma_1 \leq \Gamma_2 \qquad \Psi;\Delta;\Gamma_1 \vdash v : \forall[].\Gamma_2@al}{\Psi;\Delta;\Gamma_1 \vdash \mathtt{b}\ v} \quad (\textsc{last branch})$$

$$\frac{\begin{array}{c}\Delta \vdash \Gamma_1\{lr : \xi_{lr}\} \leq \Gamma_2 \qquad \xi_{lr} = update(\Gamma(lr), \mathtt{al}, \forall[].\Gamma_3) \qquad \Delta \vdash \Gamma_1 \leq \Gamma_3 \\ \Psi;\Delta;\Gamma_1 \vdash v : \forall[].\Gamma_2@cond \qquad \Psi;\Delta;\Gamma_1 \vdash v_{next} : \forall[].\Gamma_3@\mathtt{al}\end{array}}{\Psi;\Delta;\Gamma_1 \vdash \mathtt{bl}(cond)\ v, v_{next}} \quad (\textsc{branch link})$$

$$\frac{\Psi;\Delta;\Gamma \vdash \iota \Rightarrow \Delta';\Gamma' \qquad \Psi;\Delta';\Gamma' \vdash I}{\Psi;\Delta;\Gamma \vdash \iota; I} \quad (\textsc{instruction})$$

図 9　命令列の型付け規則
Fig. 9　Typing rules for instruction sequences.

れる関数 $get(\xi, cond)$ を用いている．この関数は，条件類 $cond$ に含まれる写像型 ξ 中のすべての条件状態に対応する型に対して，型の単一化操作を行い得られた型 τ_{unify} を返すものである．この関数は型の読み出しのみを行い，写像型の更新は行わない．なお，レジスタファイル型のサブタイプ関係は，**図 14** で定義される．

関数呼び出し命令 `bl` の型付けでは，第 1 オペランド（v）は，条件類 $cond$ の下で，ラベル型でなければならない．また，第 2 オペランド（v_{next}）は，任意の条件状態の下で，ラベル型を持たなければならない．ここで，第 1 オペランドの型 $\forall[].\Gamma_2$ は呼び出し先のラベルの型を，第 2 オペランドの型 $\forall[].\Gamma_3$ は戻り先のラベルの型を表す．もし条件類 $cond$ が満たされないならばこの命令は第 2 オペランドの指す戻り先への分岐命令と等しいので，事前条件のレジスタファイル型 Γ_1 は，第 2 オペランドのラベル型が示すレジスタファイル型 Γ_3 のサブタイプとなっていなければならない．また，条件類 $cond$ が満たされている場合には，リンクレジスタ lr へ戻り先のラベルを保存してから第 1 オペランドの指す呼び出し先へジャンプするので，事前条件のレジスタファイル型 Γ_1 中のリンクレジスタ lr の型を第 2 オペランドのラベル型で更新したものは，第 1 オペランドのラベル型が示すレジスタファイル型 Γ_2 のサブタイプになっていなければならない．このレジスタファイル型 Γ_1 の

$\boxed{\Psi;\Delta;\Gamma \vdash \iota \Rightarrow \Delta';\Gamma'}$

$$\frac{\begin{array}{c}\Psi;\Delta;\Gamma \vdash r : int@al \qquad \Psi;\Delta;\Gamma \vdash v : int@al \\ \xi_i = unify(\Gamma(r_i), al) \qquad (1 \leq i \leq n) \qquad \xi_{lr} = unify(\Gamma(r_{lr}), al)\end{array}}{\Psi;\Delta;\Gamma \vdash \mathtt{cmp}\ r, v \Rightarrow \Delta;\Gamma\{r_1 : \xi_1, \ldots, r_n : \xi_n, r_{lr} : \xi_{lr}\}} \quad (\textsc{compare})$$

$$\frac{\Psi;\Delta;\Gamma \vdash r_s : int@cond \qquad \Psi;\Delta;\Gamma \vdash v : int@cond \qquad \xi_{r_d} = update(\Gamma(r_d), cond, int)}{\Psi;\Delta;\Gamma \vdash aop(cond)\ r_d, r_s, v \Rightarrow \Delta;\Gamma\{r_d : \xi_{r_d}\}} \quad (\textsc{arithmetic})$$

$$\frac{\Psi;\Delta;\Gamma \vdash v : \tau@cond \qquad \xi_{r_d} = update(\Gamma(r_d), cond, \tau)}{\Psi;\Delta;\Gamma \vdash \mathtt{mov}(cond)\ r_d, v \Rightarrow \Delta;\Gamma\{r_d : \xi_{r_d}\}} \quad (\textsc{move operation})$$

$$\frac{\begin{array}{c}\Psi;\Delta;\Gamma \vdash r_s : \langle \tau_0^{\varphi_0}, \ldots, \tau_i^1, \ldots, \tau_{n-1}^{\varphi_{n-1}} \rangle@cond \\ \xi_{r_d} = update(\Gamma(r_d), cond, \tau_i) \qquad (0 \leq i < n)\end{array}}{\Psi;\Delta;\Gamma \vdash \mathtt{ld}(cond)\ r_d, [r_s, i] \Rightarrow \Delta;\Gamma\{r_d : \xi_{r_d}\}} \quad (\textsc{load operation})$$

$$\frac{\begin{array}{c}\Psi;\Delta;\Gamma \vdash r_d : \langle \tau_0^{\varphi_0}, \ldots, \tau_{n-1}^{\varphi_{n-1}} \rangle@cond \qquad \Psi;\Delta;\Gamma \vdash r_s : \tau_i@cond \\ \xi_{r_d} = update(\Gamma(r_d), cond, \langle \tau_0^{\varphi_0}, \ldots, \tau_i^1, \ldots, \tau_{n-1}^{\varphi_{n-1}} \rangle) \qquad (0 \leq i < n)\end{array}}{\Psi;\Delta;\Gamma \vdash \mathtt{st}(cond)\ r_s, [r_d, i] \Rightarrow \Delta;\Gamma\{r_d : \xi_{r_d}\}} \quad (\textsc{store operation})$$

$$\frac{\Delta \vdash \Gamma_1 \leq \Gamma_2 \qquad \Psi;\Delta;\Gamma_1 \vdash v : \forall[].\Gamma_2@cond}{\Psi;\Delta;\Gamma_1 \vdash \mathtt{b}(cond)\ v \Rightarrow \Delta;\Gamma_1} \quad (\textsc{branch})$$

$$\frac{\Psi;\Delta;\Gamma \vdash v : \exists\alpha.\tau@cond \qquad (\alpha \notin \Delta) \qquad \xi_{r_d} = update(\Gamma(r_d), cond, \tau)}{\Psi;\Delta;\Gamma \vdash \mathtt{unpack}(cond)\ [\alpha, r_d], v \Rightarrow \alpha, \Delta;\Gamma\{r_d : \xi_{r_d}\}} \quad (\textsc{unpack})$$

$$\frac{\Delta \vdash \tau_i \qquad \xi_{r_d} = update(\Gamma(r_d), cond, \langle \tau_1^0, \ldots, \tau_n^0 \rangle)}{\Psi;\Delta;\Gamma \vdash \mathtt{malloc}(cond)\ r_d, [\tau_1, \ldots, \tau_n] \Rightarrow \Delta;\Gamma\{r_d : \xi_{r_d}\}} \quad (\textsc{malloc})$$

図 10　命令の型付け規則
Fig. 10　Typing rules for instructions.

$\boxed{\Psi;\Delta;\Gamma \vdash v : \tau@cond}$

$$\frac{get(\Gamma(r), cond) = \tau}{\Psi;\Delta;\Gamma \vdash r : \tau@cond}$$

$$\frac{\Psi;\Delta \vdash w : \tau\ wval}{\Psi;\Delta \vdash w : \tau@cond}$$

$$\frac{\Delta \vdash \tau \qquad \Psi;\Delta;\Gamma \vdash v : \forall[\alpha,\Delta'].\Gamma'@cond}{\Psi;\Delta;\Gamma \vdash v[\tau] : \forall[\Delta'].\Gamma'[\tau/\alpha]@cond}$$

$$\frac{\Delta \vdash \tau \qquad \Psi;\Delta;\Gamma \vdash v : \tau'[\tau/\alpha]@cond}{\Psi;\Delta;\Gamma \vdash pack[\tau,v]as\exists\alpha.\tau' : \exists\alpha.\tau'@cond}$$

$$\frac{}{\Psi;\Delta;\Gamma \vdash v : \top@cond}$$

$$\frac{\Delta \vdash \tau_1 = \tau_2 \qquad \Psi;\Delta;\Gamma \vdash v : \tau_2@cond}{\Psi;\Delta;\Gamma \vdash v : \tau_1@cond}$$

図 11　値の型付け規則
Fig. 11　Typing rules for values.

更新には，図 13 で定義される関数 $update(\xi, cond, \tau)$ を用いている．この関数は，写像型 ξ の条件類 $cond$ に含まれる条件状態に対応する型を τ に更新し，更新した新たな写像型を返すものである．

比較命令 `cmp` の型付けでは，まず，第 1 オペランド（r）と第 2 オペランド（v）の型が双方とも整数型を持つことを検査する．そのうえで，レジスタファイル型中のすべてのレジスタの写像型に対して，すべての条件状態の型を単一化する．この単一化には，図 13 で定義される関数 $unify(\xi, cond)$ を用いる．この関数は，関数 $get(\xi, cond)$ と関数 $update(\xi, cond, \tau)$ を用いて定義されており，写像型 ξ について，条件類 $cond$ に含まれる条件状態に対応する型を単一化し，その単一化した型で ξ を更新し，更新した新たな写像型を返す．この比較命令

$\boxed{\Psi \vdash h : \tau \text{ hval} \quad \Psi;\Delta \vdash w : \tau \text{ wval}}$
$\boxed{\Psi;\Delta \vdash w : \tau^{\varphi} \text{ fwval}}$

$$\frac{\Psi;\cdot \vdash w_i : \tau_i^{\varphi_i}\ fwval}{\Psi \vdash \langle w_1,\ldots,w_n\rangle : \langle \tau_1^{\varphi_1},\ldots,\tau_n^{\varphi_n}\rangle\ hval}$$

$$\frac{\Delta \vdash \Gamma \qquad \Psi;\Delta;\Gamma \vdash I}{\Psi \vdash \mathtt{code}[\Delta]\Gamma.I : \forall[\Delta].\Gamma\ hval}$$

$$\frac{\cdot \vdash \tau_1 = \tau_2 \qquad \Psi \vdash h : \tau_2\ hval}{\Psi \vdash h : \tau_1\ hval}$$

$$\frac{\Delta \vdash \tau_1 \le \tau_2 \qquad \Psi(l) = \tau_1}{\Psi;\Delta \vdash l : \tau_2\ wval}$$

$$\frac{}{\Psi;\Delta \vdash i : int\ wval}$$

$$\frac{\Delta \vdash \tau \qquad \Psi;\Delta \vdash w : \forall[\alpha,\Delta'].\Gamma\ wval}{\Psi;\Delta \vdash w[\tau] : \forall[\Delta'].\Gamma[\tau/\alpha]\ wval}$$

$$\frac{\Delta \vdash \tau \qquad \Psi;\Delta \vdash w : \tau'[\tau/\alpha]\ wval}{\Psi;\Delta \vdash pack[\tau,w]as\exists\alpha.\tau' : \exists\alpha.\tau'\ wval}$$

$$\frac{}{\Psi;\Delta \vdash w : \top} \qquad \frac{\Delta \vdash \tau}{\Psi;\Delta \vdash ?\tau : \tau^0\ fwval} \qquad \frac{\Psi;\Delta \vdash w : \tau\ wval}{\Psi;\Delta \vdash w : \tau^{\varphi}\ fwval}$$

$$\frac{\Delta \vdash \tau_1 = \tau_2 \qquad \Psi;\Delta \vdash w : \tau_2\ wval}{\Psi;\Delta \vdash w : \tau_1\ wval}$$

図 12　ヒープ値とワード値の型付け規則
Fig. 12　Typing rules for heap values and word values.

$get(\xi, cond)$　$update(\xi, cond, \tau)$
$unify(\xi, cond)$

$$
\begin{aligned}
get(\xi, cond) \quad &\equiv \quad \bigsqcup_{\forall s \in cond} \xi(s) \\
&\text{where} \\
&\quad \bigsqcup_i \tau_i \equiv \tau_1 \sqcup \tau_2 \sqcup \cdots \\
&\quad \tau_a \sqcup \tau_b \equiv \\
&\qquad \tau_a \quad (\text{iff} \ \ \tau_a = \tau_b) \\
&\qquad \top \quad (\text{otherwise}) \\
update(\xi, cond, \tau) \quad &\equiv \quad \xi\{s \mapsto \tau\} \ \ (\forall s \in cond) \\
unify(\xi, cond) \quad &\equiv \quad update(\xi, cond, \tau_{unify}) \\
&\text{where} \quad \tau_{unify} = get(\xi, cond)
\end{aligned}
$$

図 13　関数 get，関数 $update$，関数 $unify$ の定義
Fig. 13　Definitions of the functions get, $update$ and $unify$.

$\Delta \vdash \xi$　$\Delta \vdash \xi_1 \leq \xi_2$　$\Delta \vdash \Gamma_1 \leq \Gamma_2$

$$\frac{\Delta \vdash \tau_i}{\Delta \vdash \{s_1 : \tau_1, \ldots, s_n : \tau_n\}}$$

$$\frac{\Delta \vdash \tau_i \leq \tau_i' \quad (for\ 1 \leq i \leq n) \qquad \Delta \vdash \tau_i \quad (for\ n < i \leq m) \qquad (m \geq n)}{\Delta \vdash \{s_1 : \tau_1, \ldots, s_m : \tau_m\} \leq \{s_1 : \tau_1, \ldots, s_n : \tau_n\}}$$

$$\frac{\Delta \vdash \xi_i \leq \xi_i' \quad (for\ 1 \leq i \leq n) \qquad \Delta \vdash \xi_i \quad (for\ n < i \leq m) \qquad (m \geq n)}{\Delta \vdash \{r_1 : \xi_1, \ldots, r_m : \xi_m\} \leq \{r_1 : \xi_1', \ldots, r_n : \xi_n'\}}$$

図 14　写像型とレジスタファイル型のサブタイプ関係
Fig. 14　Subtyping relations for mapping types and register file types.

は条件類が付かないので，写像型のすべての条件状態に対して型の単一化を行う必要がある．

算術命令の型付けでは，まず，第 2 オペランド（r_s）と第 3 オペランド（v）が，条件類 $cond$ に含まれる条件状態において，整数型を持つことを検査する．次に，第 1 オペランド（r_d）のレジスタの写像型中の，条件類 $cond$ に含まれる条件状態に対応する型を整数型に更新する．また，`mov` 命令の型付けでは，第 1 オペランド（r_d）のレジスタの写像型中で，条件類 $cond$ に対応する型を，条件類 $cond$ における第 2 オペランド（v）の型で更新する．

ヒープからのデータ読み出し命令 `ld` の型付けでは，まず第 2 オペランド（r_s）が，条件類 $cond$ に含まれる条件状態においてタプル型を持つことを検査する．次にそのタプル型が第 3 オペランド（即値 i）で指定されたオフセット以上の長さを持つこと，またタプル型の i 番目の要素が初期化済みであることを検査する．最後に，第 1 オペランド（r_d）の写像型中で，条件類 $cond$ に含まれる条件状態に対応する型を，タプル型中の i 番目要素の型に更新する．

ヒープへのデータ書き込み命令 `st` の型付けでは，`ld` 命令と同様，第 2 オペランド（r_d）が，条件類 $cond$ に含まれる条件状態においてタプル型を持つこと，そのタプル型が第 3 オペランド（即値 i）で指定されたオフセット以上の長さを持つことを検査する．そのうえで，そのタプル型の i 番目の要素の型と，第 1 オペランド（r_s）の写像型中で条件類 $cond$ に対応する型が等しいことも検査する．また，第 1 オペランドは $cond$ の状態において，タプルの i 番目の型と同じ型でなければならない．最後に，第 2 オペランドの写像型中で，条件類 $cond$ に対応する部分を，i 番目の要素の初期化フラグを 1 にセットしたタプル型で更新する．

`unpack` 命令の型付けでは，まず第 3 オペランド（v）が，条件類 $cond$ に対応する条件状態において，存在型を持つことを検査する．次に，第 1 オペランド（α）がフレッシュな型変数であることを検査したうえで，この型変数を事後条件の型変数集合に加える．最後に，第 2 オペランド（r_d）の写像型中で，条件類 $cond$ に含まれる条件状態に対応する型を，存在型で指定された型 τ で更新する．

ヒープ確保命令 `malloc` の型付けでは，第 1 オペランド（r_d）の写像型中で，条件類 $cond$ に対応する条件状態の型を，確保するタプルの型で更新する．確保されるタプルは未初期化なので，タプル型の各要素の初期化フラグは 0 である．

7. 我々の型付きアセンブリ言語の表現力についての議論

従来の TAL [20] で型検査にパスするプログラムは，6 章で我々が提案した TAL でも記述可能であるうえ，型付けも可能であるという意味で，我々の TAL は，従来の TAL よりも表現力がある．これは，我々の TAL において，分岐命令を除いた各命令に付与できる条件類を `al`（表 2 参照）に制限し，またレジスタファイル型において，各レジスタの型をつねに単一化された状態に制限すれば，従来の TAL と本質的に等しくなるからである．このた

めたとえば，Morrisett らが示した，System F[6),22)] から従来の TAL への変換手法[20)] を用いることで，System F から我々の TAL へ変換することも可能である．

一方，4 章，5 章で述べたように，我々の TAL の型システムでは，CPU の実行状態によりレジスタが異なる型を持ちうる場合でも，効果的に型検査を行うことが可能である．**図 15** は，CPU の実行状態によってレジスタが異なる型を持つアセンブリプログラムを実際に出力しうる C 言語プログラムである．図 15 を，gcc[24)] の最適化オプション `-O2` でコンパイルすると，**図 16** のようなアセンブリプログラムが生成される（gcc バージョン 4.1.2 で確認．ただしここではタプル型に対する初期化フラグは省略した）．このアセンブリプログラムでは，3 行目の `ldne` 命令でレジスタ `r3` に `<int,int>` 型の値がロードされ，4 行目

```
struct st { int f1; int f2; };

int f(struct st** s, int** i, int c)
{
    if (c)
        return (*s)->f2;
    else
        return **i;
}
```

図 15　CPU の実行状態によりレジスタが異なる型を持つようにコンパイルされうる C 言語プログラム
Fig. 15　An example C program which can be compiled as one register may have different types depending on CPU states.

```
1:  f: {r0:<<int,int>>,r1:<<int>>}
2:      cmp     r2, #0
3:      ldne    r3, [r0, 0]
4:      ldeq    r3, [r1, 0]
5:      ldne    r0, [r3, 1]
6:      ldeq    r0, [r3, 0]
7:      b       lr
```

図 16　CPU の実行状態によりレジスタが異なる型を持つように生成されたアセンブリプログラム
Fig. 16　The generated assembly program one of whose registers has different types depending on CPU states.

の `ldeq` 命令でレジスタ `r3` に `<int>` 型の値がロードされる（レジスタ `r0`，`r1` はそれぞれ，関数 `f` の第 1 引数，第 2 引数を保持している）．このため，従来の TAL の型検査では，4 章で示した例と同様，5 行目の `ldne` 命令の型付けに失敗するため型エラーとなる．一方，我々の型システムでは，写像型によってレジスタが CPU の実行状態に応じて異なる型を持つことができるため，このようなプログラムでも型検査することが可能である．

8. 関 連 研 究

TAL は Morrisett らによって提案されて以来[20)]，様々な拡張が研究されてきたが，条件実行機能に対応する拡張を提案したのは本論文が最初である．STAL（Stack-based Typed Assembly Language）[19)] は，メモリスタックに対応するように拡張された TAL である．多くのプログラミング言語のコンパイラは，メモリスタックを利用して関数呼び出しを実現しているため，この拡張は実用性の点から重要である．STAL のメモリスタック対応拡張手法は，我々の型システムの条件実行機能対応拡張手法と干渉しないため，我々の型システムを STAL の手法で拡張することは容易と考えられる．

我々の写像型の概念は，条件フラグという値に依存した型であるという点では，依存型[15),16),26),28)] と見ることもできる．DTAL（Dependently-typed Assembly Language）[27)] は，依存型を初めて TAL に導入した．しかし，配列の境界検査が主目的であるため，本論文で示したような写像型を DTAL で実現することはできない．CTAL_0 [12)] は，C 言語プログラムを TAL へコンパイルすることを目的として依存型を TAL へ導入した．CTAL_0 の型システムでは，値に依存して異なる型を表すことができるが，条件実行機能は考慮されていない．

プログラム最適化の分野では，条件実行機能の下でのプログラム解析の研究が行われてきた[5),21)]．条件実行機能を利用したプログラム最適化では，プログラム中のいくつかの基本ブロックを大きな基本ブロックに統合し，命令レベル並列性を高めることで，実行性能の向上を目指す[14)]．この統合された基本ブロックに対する解析において，従来のデータフロー解析をそのまま用いると，解析結果が保守的になってしまうため，条件実行機能を考慮した解析を行う必要がある．Eichenberger らは，条件実行機能を考慮した生存変数解析手法を提案し，これを用いたレジスタ割当ての枠組みを提案した[4)]．Johnson らは，条件実行機能の下でのデータフロー解析の手法を提案した[11)]．Carter らは，条件実行機能を考慮した SSA 形式である，PSSA を提案した[1)]．これらの研究は，効率の良いアセンブリプログラムを生成することを目的としているため，生成されたアセンブリプログラムの型検査に直接応

用することは難しい．

9. まとめと今後の課題

本論文は，CPU の条件実行機能に対応した型付きアセンブリ言語（TAL）を提案した．条件実行機能とは，機械語の各命令に実行条件が付与されていて，CPU の実行状態に応じて，命令を実行するか否かが決定される仕組みであり，実際に，組み込み計算環境で広く使われている ARM アーキテクチャにも採用されている．条件実行機能を用いたプログラムでは，1 つの基本ブロック中であっても，CPU の実行状態によっては，複数の制御フローが隠れていることがあるが，従来の TAL の型システムでは，この隠れた制御フローを扱うことが難しかった．これに対し本論文は，従来の TAL を拡張し，実行条件に関する CPU の実行状態を型に対応させる写像型を導入した．この写像型の導入により，基本ブロック中に隠された複数の制御フローに応じて，1 つのレジスタが複数の型を持つことができるようになるため，条件実行機能を用いたアセンブリプログラムの型検査が可能となる．今後の課題は，より柔軟な条件実行機能をサポートするために，型システムを改良することである．たとえば，現在の我々の TAL では比較命令に対して条件実行機能を利用することはできないが，この制限を緩めることで，より複雑なプログラムも扱えるようになる．また，より複雑な条件実行機能を持った CPU アーキテクチャ[10] へ対応するためにも，型システムの改良は必要であると考えられる．

参 考 文 献

1) Carter, L., Simon, B., Calder, B., Carter, L. and Ferrante, J.: Predicated Static Single Assignment, *IEEE PACT*, pp.245–255 (1999).
2) Chailloux, E., Manoury, P. and Pagano, B.: *Développement d'applications avec Objective Caml*, O'Reilly (2000).
3) Dimock, A., Muller, R., Turbak, F.A. and Wells, J.B.: Strongly Typed Flow-Directed Representation Transformations, *ICFP*, pp.11–24 (1997).
4) Eichenberger, A.E. and Davidson, E.S.: Register allocation for predicated code, *MICRO*, pp.180–191 (1995).
5) Fang, J.Z.: Compiler Algorithms on If-Conversion, Speculative Predicates Assignment and Predicated Code Optimizations, *LCPC*, pp.135–153 (1996).
6) Girard, J.-Y.: Une extension de l'interprétation de Gödel à l'analyse, et son application à l'élimination des coupures dans l'analyse et la théorie des types, *2nd Scandinavian Logic Symposium*, pp.63–92 (1971).
7) Gosling, J., Joy, B., Steele, G. and Bracha, G.: *The Java Language Specification, 3rd Edition*, Addison-Wesley (2005).
8) Hsu, P.Y.T. and Davidson, E.S.: Highly Concurrent Scalar Processing, *ISCA*, pp.386–395 (1986).
9) Intel Corporation: *Intel 64 and IA-32 Architectures Software Developer's Manual*.
10) Intel Corporation: *Intel Itanium Architecture Software Developer's Manual*.
11) Johnson, R. and Schlansker, M.S.: Analysis Techniques for Predicated Code, *MICRO*, pp.100–113 (1996).
12) Kosakai, T., Maeda, T. and Yonezawa, A.: Compiling C Programs into a Strongly Typed Assembly Language, *ASIAN*, pp.17–32 (2007).
13) Lindholm, T. and Yellin, F.: *The Java Virtual Machine Specification, 2nd Edition*, Addison-Wesley (1999).
14) Mahlke, S.A., Lin, D.C., Chen, W.Y., Hank, R.E. and Bringmann, R.A.: Effective compiler support for predicated execution using the hyperblock, *MICRO*, pp.45–54 (1992).
15) Martin-Löf, P.: *Intuitionistic Type Theory*, Bibliopolis (1984).
16) Martin-Löf, P.: Constructive mathematics and computer programming, *Proc. discussion meeting of the Royal Society of London on Mathematical logic and programming languages*, Upper Saddle River, NJ, USA, pp.167–184, Prentice-Hall, Inc. (1985).
17) Mitchell, J.C. and Plotkin, G.D.: Abstract Types Have Existential Type, *ACM Trans. Program. Lang. Syst.*, Vol.10, No.3, pp.470–502 (1988).
18) Morrisett, G., Crary, K., Glew, N., Grossman, D., Samuels, R., Smith, F., Walker, D., Weirich, S. and Zdancewic, S.: TALx86: A Realistic Typed Assembly Language, *the 1999 ACM SIGPLAN Workshop on Compiler Support for System Software*, pp.25–35 (1999).
19) Morrisett, J.G., Crary, K., Glew, N. and Walker, D.: Stack-Based Typed Assembly Language, *Types in Compilation*, pp.28–52 (1998).
20) Morrisett, J.G., Walker, D., Crary, K. and Glew, N.: From system F to typed assembly language, *ACM Trans. Program. Lang. Syst.*, Vol.21, No.3, pp.527–568 (1999).
21) Park, J.C. and Schlansker, M.: On predicated execution, Technical report, Hewlett Packard Laboratories (1991).
22) Reynolds, J.C.: Towards a theory of type structure, *Programming Symposium, Proc. Colloque sur la Programmation*, pp.408–423 (1974).
23) Someren, A.V. and Atack, C.: *The ARM Risc Chip: A Programmer's Guide*, Addison-Wesley (1993).
24) Stallman, R.M. and the GCC Developer Community: *Using GCC: The GNU Com-*

piler Collection Reference Manual, Free Software Foundation (2003).
25) Tarditi, D., Morrisett, J.G., Cheng, P., Stone, C., Harper, R. and Lee, P.: TIL: A Type-Directed Optimizing Compiler for ML, *PLDI*, pp.181–192 (1996).
26) Xi, H.: Dependent ML: An approach to practical programming with dependent types, *J. Funct. Program.*, Vol.17, No.2, pp.215–286 (2007).
27) Xi, H. and Harper, R.: A Dependently Typed Assembly Language, *ICFP*, pp.169–180 (2001).
28) Xi, H. and Pfenning, F.: Dependent Types in Practical Programming, *POPL*, pp.214–227 (1999).

付　　録

A.1　定理 1 の証明の概略

補題 1　（*Unified register file*）

$\Psi \vdash R : \Gamma@C$ ならば，任意のレジスタ r，条件類 *cond* に関して，$\Gamma' = \Gamma\{r : unify(\Gamma(r), cond)\}$ とおくと，$\Psi \vdash R : \Gamma'@C$ が成り立つ

証明　$\Psi \vdash R : \Gamma@C$ より，$R = \{r_1 \mapsto w_1, \ldots, r_m \mapsto w_m, lr \mapsto w_{lr}\}$，$\Gamma = \{r_1 : \xi_1, \ldots, r_n : \xi_n, lr : \xi_{lr}\}$ とすると，$\Psi;\cdot \vdash w_i : \xi_i(C)$ *wval*（$1 \le i \le n$ または $i = lr$）が成り立つ．ここで，$\xi'_i = unify(\xi_i, cond)$ とすると，$\xi'_i(C) = \xi_i(C)$ または $\top$ であるから，任意の i，任意の *cond* について，$\Psi;\cdot \vdash w_i : \xi'_i(C)$ *wval* が成り立つ．したがって，$\Psi \vdash R : \Gamma'@C$ が成り立つ．　□

補題 2　（*Unconditional register file*）

$\Psi \vdash R : \Gamma@C$，$\forall r \in \Gamma.\Gamma(r) = unify(\Gamma(r), \mathtt{al})$ ならば，任意の条件フラグ C' に関して，$\Psi \vdash R : \Gamma@C'$ が成り立つ．

証明　$\Psi \vdash R : \Gamma@C$ より，$R = \{r_1 \mapsto w_1, \ldots, r_m \mapsto w_m, lr \mapsto w_{lr}\}$，$\Gamma = \{r_1 : \xi_1, \ldots, r_n : \xi_n, lr : \xi_{lr}\}$ とすると，$\Psi;\cdot \vdash w_i : \xi_i(C)$ *wval*（$1 \le i \le n$ または $i = lr$）が成り立つ．ここで，$\xi_i = unify(\xi_i, \mathtt{al})$ であるから，$\xi_i(C') = unify(\xi_i, \mathtt{al})(C')$ が成り立つ．ここで，関数 $unify$ の定義より，$unify(\xi_i, \mathtt{al})(C') = get(\xi_i, \mathtt{al})$ であるから，$\xi_i(C') = \xi_i(C)$ または $\top$ である．よって，$\Psi;\cdot \vdash w_i : \xi_i(C')$ *wval* が成り立つ．したがって，$\Psi \vdash R : \Gamma@C'$ が成り立つ．　□

補題 3　（*Type substition*）

(1)　$\Psi;\alpha,\Delta;\Gamma \vdash I$ ならば，$\Psi;\Delta;\Gamma[\tau/\alpha] \vdash I[\tau/\alpha]$ が成り立つ．

(2)　$\Psi;\alpha,\Delta;\Gamma \vdash v : \tau'@cond$ ならば，$\Psi;\Delta;\Gamma[\tau/\alpha] \vdash v[\tau/\alpha] : \tau'[\tau/\alpha]$ が成り立つ．

証明　それぞれの導出に関する帰納法で証明する．　□

補題 4　（$\hat{R}$ *typing*）

$\Psi \vdash R : \Gamma@C$，$\Psi;\cdot;\Gamma \vdash v : \tau@cond$，$C \in cond$ ならば，$\Psi;\cdot \vdash \hat{R}(v) : \tau$ *wval* が成り立つ．

証明　値 v の構造に関する帰納法で証明する．　□

補題 5　（*Register file weakening*）

$\Delta \vdash \Gamma \le \Gamma'$ かつ $\Psi \vdash R : \Gamma$ ならば，$\Psi \vdash R : \Gamma'$ が成り立つ．

証明　導出に関する帰納法で証明する．　□

補題 6　（*Conditional canonical forms*）

$\vdash H : \Psi$，$\Psi \vdash R : \Gamma@C$，$\Psi;\cdot;\Gamma \vdash v : \tau@cond$，$C \in cond$ が成り立つならば，

(1)　$\tau = int$ ならば，$\hat{R}(v) = i$ が成り立つ．

(2)　$\tau = \forall[\Delta'].\Gamma'$ ならば，$\hat{R}(v) = l[\psi]$，$H(l) = \mathtt{code}[\Delta\Delta']\Gamma''.I$，$\Gamma' = \Gamma''[\psi/\Delta]$，$\Psi;\Delta\Delta';\Gamma'' \vdash I$ が成り立つ．

(3)　$\tau = \langle \tau_1^{\varphi_1}, \ldots, \tau_n^{\varphi_n} \rangle$ ならば，$\hat{R}(v) = l$，$H(l) = \langle w_1, \ldots, w_n \rangle$，$\Psi;\cdot \vdash w_i : \tau_i^{\varphi_i}$ *fwval* が成り立つ．

(4)　$\tau = \exists\alpha.\tau'$ ならば，$\hat{R}(v) = pack[\tau'', w]\, as\, \exists\alpha.\tau'$，$\Psi;\cdot \vdash w : \tau'[\tau''/\alpha]$ *wval* が成り立つ．

証明　$\hat{R}$ の定義と，ワード値の型付けの導出に関する帰納法で証明する．　□

補題 7　（*Heap update*）

$\vdash H : \Psi$，$\Psi\{l : \tau\} \vdash h : \tau$ *hval*，$\cdot \vdash \tau \le \Psi(l)$ ならば，

(1)　$\vdash H\{l \mapsto h\} : \Psi\{l : \tau\}$

(2)　$\Psi \vdash R : \Gamma@C$ ならば，$\Psi\{l : \tau\} \vdash R : \Gamma@C$ が成り立つ．

(3)　$\Psi;\Delta;\Gamma \vdash I$ ならば，$\Psi\{l : \tau\};\Delta;\Gamma \vdash I$ が成り立つ．

証明　それぞれの導出に関する帰納法で証明する．　□

補題 8　（*Heap extension*）

$\vdash H : \Psi$，$\Psi\{l : \tau\} \vdash h : \tau$ *hval*，$l \notin H$ ならば，

(1)　$\vdash H\{l \mapsto h\} : \Psi\{l : \tau\}$

(2)　$\Psi \vdash R : \Gamma@C$ ならば，$\Psi\{l : \tau\} \vdash R : \Gamma@C$ が成り立つ．

(3)　$\Psi;\Delta;\Gamma \vdash I$ ならば，$\Psi\{l : \tau\};\Delta;\Gamma \vdash I$ が成り立つ．

証明　それぞれの導出に関する帰納法で証明する．　□

補題 9　（*Preservation*）

$\vdash P$ かつ $P \mapsto P'$ ならば $\vdash P'$ が成り立つ．

証明　$P = (H, R, I, C)$，$P' = (H', R', I', C')$ とおき，I の先頭の命令に関する場合分けで証明する．

(1)　*cmp* 命令の場合

$\vdash P$ より，$\Psi; \cdot; \Gamma \vdash \mathit{cmp}\ r, v; I'$ が成り立つ．したがって，*cmp* 命令の型付け規則より，$\Psi; \cdot; \Gamma' \vdash I'$ が成り立つ．ただし，$\Gamma' = \Gamma\{r : \xi\}(\forall r \in Dom(\Gamma))$，$\xi = unify(\Gamma(r), \mathit{al})$ である．ここで補題 *1*，補題 *2* より，$\Psi \vdash R : \Gamma'@C'$ が成り立つ．一方，$R = R'$，$H = H'$ であるから，$\vdash P'$ が成立する．

(2)　算術命令の場合

$\vdash P$ より，$\Psi; \cdot; \Gamma \vdash aop(cond)\ r_d, r_s, v; I'$ が成り立つ．したがって，算術命令の型付け規則より，$\Psi; \cdot; \Gamma' \vdash I'$ が成り立つ．ただし，$\Gamma' = \Gamma\{r_d : \xi_{r_d}\}$，$\xi_{r_d} = update(\Gamma(r_d), cond, int)$ である．もし，$C \in cond$ であった場合は，$R'(r_d)$ は何らかの整数値 j を持つが，$C \in cond$ と関数 *update* の定義より，$\xi_{r_d}(C) = int$ が成り立つので，$\Psi, \cdot \vdash R'(r_d) : \Gamma'(r_d)(C)\ wval$ が成り立つ．したがって，$\Psi \vdash R' : \Gamma'@C$ も成り立つ．よって，$\vdash P'$ が成り立つ．一方，$C \notin cond$ であった場合は，$\forall r \in \Gamma.\Gamma(r)(C) = \Gamma'(r)(C)$ であるから，$\Psi \vdash R' : \Gamma'@C$ が成り立つ．よって，$\vdash P'$ が成り立つ．

(3)　*mov* 命令の場合

$\vdash P$ より，$\Psi; \cdot; \Gamma \vdash \mathit{mov}(cond)\ r_d, v; I'$ が成り立つ．したがって，*mov* 命令の型付け規則より，$\Psi; \cdot; \Gamma' \vdash I'$ が成り立つ．ただし，$\Gamma' = \Gamma\{r_d : \xi_{r_d}\}$，$\Psi, \cdot; \Gamma \vdash v : \tau@cond$，$\xi_{r_d} = update(\Gamma(r_d), cond, \tau)$ である．もし，$C \in cond$ であった場合には，ここで補題 *4* より，$\Psi; \cdot \vdash \hat{R}(v) : \tau\ wval$ が成り立つ．また，関数 *update* の定義より，$\xi_{r_d}(C) = \tau$ であるから，$\Psi; \cdot \vdash R'(r_d) : \Gamma'(r_d)(C)\ wval$ が成り立つので，$\Psi \vdash R' : \Gamma'@C$ が成り立つ．したがって，$\vdash P'$ が成り立つ．一方，$C \notin cond$ であった場合は，$\forall r \in \Gamma.\Gamma(r)(C) = \Gamma'(r)(C)$ であるから，$\Psi \vdash R' : \Gamma'@C$ が成り立つ．よって，$\vdash P'$ が成り立つ．

(4)　*ld* 命令の場合

$\vdash P$ より，$\Psi; \cdot; \Gamma \vdash \mathit{ld}(cond)\ r_d, [r_s, i]; I'$ が成り立つ．したがって，*ld* 命令の型付け規則より，$\Psi; \cdot; \Gamma\{r_d : \xi_{r_d}\} \vdash I'$ が成り立つ．ただし，$\xi_{r_d} = update(\Gamma(r_d), cond, \tau_i)$，$\Psi, \cdot; \Gamma \vdash r_s : \langle \ldots, \tau_i^1, \ldots \rangle@cond$ である．もし，$C \in cond$ であった場合には，補題 *6* より，$\Psi; \cdot \vdash w_i : \tau_i$ が成り立つ．したがって，$\xi_{r_d}(C) = \tau_i$ であるから，$\Psi \vdash R' : \Gamma'@C$ が成り立つ．よって，$\vdash P'$ が成り立つ．一方，$C \notin cond$ であった場合は，$\forall r \in \Gamma.\Gamma(r)(C) = \Gamma'(r)(C)$ であるから，$\Psi \vdash R' : \Gamma'@C$ が成り立つ．よって，$\vdash P'$ が成り立つ．

(5)　*st* 命令の場合

$\vdash P$ より，$\Psi; \cdot; \Gamma \vdash \mathit{st}(cond)\ r_s, [r_d, i]; I'$ が成り立つ．したがって，*st* 命令の型付け規則より，$\Psi; \cdot; \Gamma' \vdash I'$ が成り立つ．ただし，$\Gamma' = \Gamma\{r_d : \xi_{r_d}\}$，$\Psi; \cdot; \Gamma \vdash r_d : \langle \ldots, \tau_i^{\varphi_i}, \ldots \rangle@cond$，$\Psi; \cdot; \Gamma \vdash r_s : \tau_i@cond$，$\xi_{r_d} = update(\Gamma(r_d), cond, \langle \ldots, \tau_i^1, \ldots \rangle)$ である．もし，$C \in cond$ であった場合には，補題 *6* より，$\hat{R}(r_d) = l$，$H(l) = \langle \ldots \rangle$ が成り立つ．ここで $H' = H\{l \mapsto \langle \ldots, \hat{R}(r_s), \ldots \rangle\}$ であるから，$\Psi' = \Psi\{l : \langle \ldots, \tau_i^1, \ldots \rangle\}$ とおくと，$\vdash H' : \Psi'$ が成り立つ．また，$C \in cond$ と関数 *update* の定義より，$\xi_{r_d}(C) = \langle \ldots, \tau_i^1, \ldots \rangle$ が成り立つ．よって，$\Psi; \cdot \vdash R'(r_d) : \Gamma'(r_d)(C)\ wval$ が成り立つので，$\Psi \vdash R' : \Gamma'@C$ が成り立つ．ここで，補題 *7* より，$\Psi' \vdash R' : \Gamma'@C$，$\Psi'; \cdot; \Gamma' \vdash I'$ が成り立つ．したがって，$\vdash P'$ が成り立つ．一方，$C \notin cond$ であった場合には，$\forall r \in \Gamma.\Gamma(r)(C) = \Gamma'(r)(C)$ であるから，$\Psi \vdash R' : \Gamma'@C$ が成り立つ．よって，$\vdash P'$ が成り立つ．

(6)　*unpack* 命令の場合

$\vdash P$ より，$\Psi; \cdot; \Gamma \vdash \mathit{unpack}(cond)\ [\alpha, r_d], v; I'$ が成り立つ．したがって，*unpack* 命令の型付け規則より，$\Psi; \alpha; \Gamma\{r_d : \xi_{r_d}\} \vdash I'$ が成り立つ．ただし，$\xi_{r_d} = update(\Gamma(r_d), cond, \tau)$，$\Psi; \cdot; \Gamma \vdash v : \exists\alpha.\tau@cond$ である．もし，$C \in cond$ であった場合には，補題 *6* より，$\hat{R}(v) = pack[\tau', w]\, as\, \exists\alpha.\tau$，$\Psi; \cdot \vdash w : \tau[\tau'/\alpha]\ wval$ が成り立つ．よって，$\Gamma' = \Gamma\{r_d : \xi_{r_d}\}$ とおくと，$\Psi \vdash R' : \Gamma'[\tau'/\alpha]@C$ が成り立つ．また，補題 *3* より，$\Psi; \cdot; \Gamma'[\tau'/\alpha] \vdash I'$ が成り立つ．したがって，$\vdash P'$ が成り立つ．一方，$C \notin cond$ であった場合には，$\forall r \in \Gamma.\Gamma(r)(C) = \Gamma'(r)(C)$ であるから，$\Psi \vdash R' : \Gamma'@C$ が成り立つ．よって，$\vdash P'$ が成り立つ．

(7)　*malloc* 命令の場合

$\vdash P$ より，$\Psi; \cdot; \Gamma \vdash \mathit{malloc}(cond)\ r_d, [\tau_1, \ldots, \tau_n]; I'$ が成り立つ．したがって，*malloc* 命令の型付け規則より，$\Psi; \cdot; \Gamma' \vdash I'$ が成り立つ．ただし，$\Gamma' = \Gamma\{r_d : \xi_{r_d}\}$，$\xi_{r_d} = update(\Gamma(r_d), cond, \langle \tau_1^0, \ldots, \tau_n^0 \rangle)$ である．もし，$C \in cond$ であった場合には，操作的意味論より，$H' = H\{l \mapsto \langle \ldots \rangle\}$ $(l \notin H)$，$R'(r_d) = l$ である．よって，$\Psi' = \Psi\{l \mapsto \langle \tau_1^0, \ldots, \tau_n^0 \rangle\}$ とおくと，$\vdash H' : \Psi'$ が成り立つ．また，$\Psi \vdash R' : \Gamma'@C$ より，補題 *8* から $\Psi' \vdash R' : \Gamma'@C$，$\Psi'; \cdot; \Gamma' \vdash I'$ が成り立つ．したがって，$\vdash P'$ が成り立つ．一方，$C \notin cond$ であった場合には，$\forall r \in \Gamma.\Gamma(r)(C) = \Gamma'(r)(C)$ である

から，$\Psi \vdash R' : \Gamma'@C$ が成り立つ．よって，$\vdash P'$ が成り立つ．

(8) `b` 命令の場合

(a) 命令列の途中の場合

もし $C \in cond$ であった場合には，操作的意味論より，$I' = I''[\psi/\Delta]$ である．ただし，$\hat{R}(v) = l[\psi]$，$H(l) = \texttt{code}[\Delta]\Gamma''.I''$ である．ここで，$\vdash P$ が成り立つことより，$\Psi \vdash H : \Psi$ が成り立つので，ヒープ値の型付け規則より，$\Psi \vdash \texttt{code}[\Delta]\Gamma''.I'' : \forall[\Delta].\Gamma''\ hval$ が成り立つ．したがって，$\Psi; \Delta; \Gamma'' \vdash I''$ が成り立つ．一方，$\Psi; \cdot; \Gamma \vdash \texttt{b}(cond)\ v \Rightarrow \Delta; \Gamma$ が成り立つので，$\cdot \vdash \Gamma \leq \Gamma_2$，$\Psi; \cdot; \Gamma \vdash v : \forall[].\Gamma_2@cond$ が成り立つ．ここで，$C \in cond$ と補題 *6* より，$\Gamma_2 = \Gamma''[\psi/\Delta]$ が成り立つので，補題 *3* より，$\Psi; \cdot; \Gamma_2 \vdash I''[\psi/\Delta]$ が成り立つ．また，補題 *5* より，$\Psi \vdash R : \Gamma_2$ が成り立つ．よって，$\Gamma' = \Gamma_2$ とおけば，$\vdash P'$ が成り立つ．

一方，$C \notin cond$ であった場合には，操作的意味論より，$I = \texttt{b}(cond)\ v; I'$ である．ここで，$\vdash P$ が成り立つことより，$\Psi; \cdot; \Gamma \vdash I'$ が成り立つ．よって，$\vdash P'$ が成り立つ．

(b) 命令列の末尾の場合

命令列の途中の場合の証明で，$cond$ を `al` としたものと同じ．

(9) `bl` 命令の場合

もし，$C \in cond$ であったとすると，操作的意味論より，$I' = I''[\psi/\Delta]$，$R' = R[lr \mapsto \hat{R}(v_{next})]$ である．ただし，$\hat{R}(v) = l[\psi]$，$H(l) = \texttt{code}[\Delta]\Gamma''.I''$ である．ここで，$\vdash P$ が成り立つことより，$\Psi \vdash H : \Psi$ が成り立つので，ヒープ値の型付け規則より，$\Psi \vdash \texttt{code}[\Delta]\Gamma''.I'' : \forall[\Delta].\Gamma''\ hval$ が成り立つ．したがって，$\Psi; \Delta; \Gamma'' \vdash I''$ が成り立つ．一方，$\Psi; \cdot; \Gamma \vdash \texttt{bl}(cond)\ v, v_{next}$ が成り立つので，$\cdot \vdash \Gamma\{lr : \xi_{lr}\} \leq \Gamma_2$，$\Psi; \cdot; \Gamma \vdash v : \forall[].\Gamma_2@cond$ が成り立つ（ただし，$\xi_{lr} = update(\Gamma(lr), \texttt{al}, \forall[].\Gamma_3)$ である）．ここで，$C \in cond$ と補題 *6* より，$\Gamma_2 = \Gamma''[\psi/\Delta]$ が成り立つので，補題 *3* より，$\Psi; \cdot; \Gamma_2 \vdash I''[\psi/\Delta]$ が成り立つ．また，$\Psi \vdash R' : \Gamma\{lr : \xi_{lr}\}$ と補題 *5* より，$\Psi \vdash R' : \Gamma_2$ が成り立つ．よって，$\vdash P'$ が成り立つ．

一方，$C \notin cond$ であったとすると，操作的意味論より，$I' = I''[\psi/\Delta]$，$R' = R$ である．ただし，$\hat{R}(v_{next}) = l_{next}[\psi]$，$H(l_{next}) = \texttt{code}[\Delta]\Gamma''.I''$ である．ここで，$\vdash P$ が成り立つことより，$\Psi \vdash H : \Psi$ が成り立つので，ヒープ値の型付け規則より，$\Psi \vdash \texttt{code}[\Delta]\Gamma''.I'' : \forall[\Delta].\Gamma''\ hval$ が成り立つ．したがって，$\Psi; \Delta; \Gamma'' \vdash I''$ が成り立つ．一方，$\Psi; \cdot; \Gamma \vdash \texttt{bl}(cond)\ v, v_{next}$ が成り立つので，$\cdot \vdash \Gamma \leq \Gamma_2$，$\Psi; \cdot; \Gamma \vdash v_{next} : \forall[].\Gamma_2@\texttt{al}$ が成り立つ．ここで，$C \in \texttt{al}$ と補題 *6* より，$\Gamma_2 = \Gamma''[\psi/\Delta]$ が成り立つので，補題 *3* より，$\Psi; \cdot; \Gamma_2 \vdash I''[\psi/\Delta]$ が成り立つ．また，補題 *5* より，$\Psi \vdash R' : \Gamma_2$ が成り立つ．よって，$\vdash P'$ が成り立つ．

□

補題 10　（*Progress*）

$\vdash P$ ならば $P \mapsto P'$ なる P' が存在する．

証明　$P = (H, R, I, C)$ とおいて，I の先頭の命令に関する場合分けで証明する．

(1) `cmp` 命令の場合

操作的意味論より，$P' = (H, R, I', C')$ とすれば成り立つ．ただし，$C = setC(R(r), \hat{R}(v))$，$I = \texttt{cmp}\ r, v; I'$ とする．

(2) 算術命令の場合

$\vdash P$ が成り立つことから，$\Psi; \cdot; \Gamma \vdash \iota; I' \Rightarrow \cdot; \Gamma\{r_d : \xi_{r_d}\}$，$\iota = aop(cond)\ r_d, r_s, v$ が成り立つ．したがって，算術命令の型付け規則より，$\Psi; \cdot; \Gamma \vdash r_s : int@cond$，$\Psi; \cdot; \Gamma \vdash v : int@cond$ が成り立つ．ここで，もし $C \in cond$ であれば，補題 *6* より，$R(r_s)$，$\hat{R}(v)$ ともに整数値であるから，$P' = (H, R[r_d \mapsto R(r_s) aop \hat{R}(v)], I', C)$ とすれば成り立つ．一方，$C \notin cond$ であれば，$P' = (H, R, I', C)$ とすれば成り立つ．

(3) `mov` 命令の場合

操作的意味論より，もし $C \in cond$ ならば，$P' = (H, R[r_d \mapsto \hat{R}(v)], I', C)$ とすれば成り立つ（ただし，$I = \texttt{mov}(cond)\ r_d, v; I'$ とする）．一方，$C \notin cond$ ならば，$P' = (H, R, I', C)$ とすれば成立する．

(4) `ld` 命令の場合

$\vdash P$ が成り立つことから，$\Psi; \cdot; \Gamma \vdash \iota; I' \Rightarrow \cdot; \Gamma\{r_d : \xi_{r_d}\}$，$\iota = \texttt{ld}\ r_d, [r_s, i]$ が成り立つ．したがって，`ld` 命令の型付け規則より，$\Psi; \cdot; \Gamma \vdash r_s : \langle \tau_0^{\varphi_0}, \ldots, \tau_i^1, \ldots \tau_{n-1}^{\varphi_{n-1}} \rangle @cond$，$0 \leq i < n$ が成り立つ．ここで，もし $C \in cond$ であれば，補題 *6* より，$\hat{R}(r_s) = l$，$H(l) = \langle w_0, \ldots, w_{n-1} \rangle$ が成り立つので，$P' = (H, R[r_d \mapsto w_i], I', C)$ とすれば成り立つ．一方，$C \notin cond$ であれば，$P' = (H, R, I', C)$ とすれば成り立つ．

(5) `st` 命令の場合

$\vdash P$ が成り立つことから，$\Psi; \cdot; \Gamma \vdash \iota; I' \Rightarrow \cdot; \Gamma\{r_d : \xi_{r_d}\}$，$\iota = \texttt{st}\ r_s, [r_d, i]$ が成り立つ．したがって，`st` 命令の型付け規則より，$\Psi; \cdot; \Gamma \vdash r_d : \langle \tau_0^{\varphi_0}, \ldots, \tau_{n-1}^{\varphi_{n-1}} \rangle$，$0 \leq i < n$ が成り立つ．ここで，もし $C \in cond$ であれば，補題 *6* より，$\hat{R}(r_d) = l$，

$H(l) = \langle w_0, \ldots, w_{n-1} \rangle$ が成り立つので，$P' = (H', R, I', C)$ とすれば成り立つ（ただし，$H' = H[l \mapsto \langle \ldots, w_{i-1}, R(r_s), w_{i+1}, \ldots \rangle]$ である）．

(6) **unpack** 命令の場合

$\vdash P$ が成り立つことから，$\Psi; \cdot; \Gamma \vdash \iota; I' \Rightarrow \alpha; \Gamma\{r_d : \xi_{r_d}\}$，$\iota = \mathbf{unpack}(cond)\ [\alpha, r_d], v$ が成り立つ．したがって，**unpack** 命令の型付け規則より，$\Psi; \cdot; \Gamma \vdash v : \exists\alpha.\tau @ cond$ が成り立つ．ここで，もし $C \in cond$ であれば，補題 *6* より，$\hat{R}(v) = pack[\tau', w]\ as\ \exists\alpha.\tau$ が成り立つので，$P' = (H, R[r_d \mapsto w], I'[\tau'/\alpha], C)$ とすれば成立する．一方，$C \notin cond$ であれば，$P' = (H, R, I', C)$ とすれば成立する．

(7) **malloc** 命令の場合

操作的意味論より，もし $C \in cond$ であれば，$P' = (H', R[r_d \mapsto l], I', C)$ とすれば成立する（ただし，$H' = H[l \mapsto \langle ?\tau_1, \ldots, ?\tau_n \rangle]$，$l \notin H$ である）．一方，$C \notin cond$ であれば $P' = (H, R, I', C)$ とすれば成立する．

(8) **b** 命令の場合

(a) 命令列の途中の場合

$\vdash P$ が成り立つことから，$\Psi; \cdot; \Gamma \vdash \iota; I' \Rightarrow \cdot; \Gamma$，$\iota = \mathbf{b}(cond)\ v$ が成り立つ．したがって，**b** 命令の型付け規則より，$\Psi; \cdot; \Gamma \vdash v : \forall[].\Gamma_2 @ cond$ が成り立つ．ここで，もし $C \in cond$ であれば，補題 *6* より，$\hat{R}(v) = l[\psi]$，$H(l) = \mathbf{code}[\Delta]\Gamma''.I''$ が成り立つので，$P' = (H, R, I''[\psi/\Delta], C)$ とすれば成り立つ．一方，$C \notin cond$ であれば，$P' = (H, R, I', C)$ とすれば成り立つ．

(b) 命令列の末尾の場合

$\vdash P$ が成り立つことから，$\Psi; \cdot; \Gamma \vdash \mathbf{b}\ v$ が成り立つ．したがって，**b** 命令の型付け規則より，$\Psi; \cdot; \Gamma \vdash v : \forall[].\Gamma_2 @ \mathbf{al}$ が成り立つ．ここで，$C \in \mathbf{al}$ と補題 *6* より，$\hat{R}(v) = l[\psi]$，$H(l) = \mathbf{code}[\Delta]\Gamma''.I''$ が成り立つので，$P' = (H, R, I''[\psi/\Delta], C)$ とすれば成り立つ．

(9) **bl** 命令の場合

$\vdash P$ が成り立つことから，$\Psi; \cdot; \Gamma \vdash \mathbf{bl}(cond)\ v, v_{next}$ が成り立つ．したがって，**bl** 命令の型付け規則より，$\Psi; \cdot; \Gamma \vdash v : \forall[].\Gamma_2 @ \mathbf{cond}$，$\Psi; \cdot; \Gamma \vdash v_{next} : \forall[].\Gamma_3 @ \mathbf{al}$ が成り立つ．もし，$C \in cond$ であれば，補題 *6* より，$\hat{R}(v) = l[\psi]$，$H(l) = \mathbf{code}[\Delta]\Gamma''.I''$ が成り立つので，$P' = (H, R', I''[\psi/\Delta], C)$ とすれば成り立つ（ただし，$R' = R[lr \mapsto \hat{R}(v_{next})]$ である）．一方，$C \notin cond$ であれば，$C \in \mathbf{al}$ と補題 *6* より，$\hat{R}(v_{next}) = l'[\psi']$，$H(l') = \mathbf{code}[\Delta']\Gamma'''.I'''$ が成り立つので，$P' = (H, R, I'''[\psi'/\Delta'], C)$ とすれば成り立つ．

□

定理 1　（再掲）

$\vdash P$ ならば，$P \mapsto P'$ なる P' が存在し，$\vdash P'$ を満たす．

証明　補題 *10* より，$P \mapsto P'$ なる P' が存在する．ここで補題 *9* より，$\vdash P'$ が成り立つ．

□

(平成 20 年 2 月 17 日受付)
(平成 20 年 5 月 8 日採録)

飯塚　大輔

1982 年生．2007 年東京大学理学部情報科学科卒業．現在，東京大学大学院情報理工学研究科コンピュータ科学専攻修士課程在学中．

前田　俊行（正会員）

1977 年生．2006 年東京大学大学院情報理工学系研究科博士課程修了．博士（情報理工学）．現在，東京大学大学院情報理工学研究科コンピュータ科学専攻助教．

米澤　明憲（正会員）

1947 年生．1970 年東京大学工学部計数工学科卒業．1977 年マサチューセッツ工科大学計算機科学科博士課程修了．Ph.D. in Computer Science．1988 年東京大学理学部情報科学科教授に着任．日本ソフトウエア科学会理事長，ドイツ国立情報科学技術研究所科学顧問，政府情報科学技術委員会委員，通産省リアルワールドコンピューティングプロジェクト評価推進委員，内閣府総合規制改革会議委員，同教育分野主査等を歴任．現在，情報システム研究機構監事，（独）産業技術総合研究所情報セキュリティ研究センター副センター長，東京大学情報基盤センター長を兼務．エンジンゼロワン文化戦略会議教育委員会委員，日本学術会議連携会員．マイクロソフト本社 Trust-worthy Computing Academic Advisory Board メンバ．