強く型付けされたオペレーティングシステム

Strongly Typed Operating System

前田 俊行 Toshiyuki MAEDA

米澤 明憲 Akinori YONEZAWA

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

tosh@is.s.u-tokyo.ac.jp yonezawa@is.s.u-tokyo.ac.jp

従来のオペレーティングシステム (OS) は、C 言語など、強く型付けされていない、安全でないプログラミング言語で記述されてきた。このため、OS の安全性を保証・検証することは、著しく複雑で困難であった。そこで本研究では、OS カーネルの記述に適した、強く型付けされた言語 (型付きアセンブリ言語) を設計・実装し、その言語で OS カーネルを記述することにより、OS の安全性を保証することを目指す。

1 はじめに

いまやコンピュータ (PC・携帯電話等) は広く一般に普及し、それらを相互に接続したネットワークは、既に我々の生活に欠かすことのできない社会基盤の一つとなった。このためソフトウェアの安全性を確保することの重要性が広く認識されるようになり、多くのソフトウェアが強く型付けされた言語 (Java [9], C# [5], Objective Caml [11] 等) で書かれるようになった。これは、強く型付けされた言語で書かれ、型検査をパスしたプログラムは、実行時にエラーが生じないことが保証されるためである。

しかしその一方で、やはり多くのプログラムが依然として C や C++ などの強く型付けされていない言語で書かれている。特に既存の OS (Linux, FreeBSD, Windows XP, Solaris 等) は、ほぼ例外なく C とアセンブリ言語で書かれている。

このため、既存の OS の安全性を保証・検証することは非常に困難である。例えば、モデル検査 [7] の手法を用いた安全性の検証では、検査状態数が指数的に増加するため、非常に小さな部分しか検証できない、もしくは、近似を行うため検査の健全性が保証できない等の問題がある [7, 16]。また、プログラムの安全性を、人手もしくは定理証明支援器で証明する手法は、安全性の証明そのものが非常に複雑で困難な上、元のプログラムが変更されると証明を再度行わなければならない等の問題がある [3, 2]。

このように、安全性の保証・検証が困難となるにもかかわらず、OS が強く型付けされた言語で書かれてこなかった理由の一つは、強く型付けされた言語では、OS を書くために必要なハードウェアに近いローレベルな操作、例えばレジスタ操作やメモリ管理 (いわゆる malloc/free) 等を記述できないと考えられてきたことにある。

本研究では、これらローレベルな操作が記述可能な、強く型付けされた言語を設計・実装し、その言語に基づいて OS を実装することにより、安全性が保証された OS を実現することを目指す。具体的には、型付きアセンブリ言語 (TAL) [10, 12] と Alias Type [14] のアイデアを組み合わせ、更に、任意長の配列を扱えるように拡張した言語を設計・実装し、これを用いることで OS の基本的な機能であるメモリ管理やスレッド管理などが実現できることを示す。

本論文の以降の構成は次の通りである。まず 2 節では、本研究の方針と手法について述べる。3 節では、我々が設計した OS 記述のための強く型付けされた言語を示す。4 節では、3 節で示した言語を用いて、メモリ管理やスレッド管理等のコードがどのように記述できるかを具体例で示す。5 節では関連研究について議論する。6 節ではまとめと今後の課題について述べる。

2 強く型付けされた OS の実現方針・手法

本研究は、型付アセンブリ言語をベースに、OS の基本的な機能を記述できるように拡張した言語を設計・実装し、実際にその言語で OS を実装することによって、その言語の実用性を示すとともに、型検査を通して安全性が検証可能な、強く型付けされた OS を実現することを目指す。

型付きアセンブリ言語をベースに用いる理由は 3 つある。第 1 の理由は、OS の記述に必要とされるローレベルな操作 (レジスタ操作等) を記述できるためである。第 2 の理由は、型情報を機械語コードに付随させることで、機械語のレベルで型検査を行えるためである。第 3 の理由は、様々な高級言語から型付きアセンブリ言語へのコンパイラを実装することで、プログラマに高級言語選択の自由を与えることができるためである。

なお本研究では、最も基本的な安全性である、メモリ安全性 (プログラムが不正にメモリを操作しないこと) と制御フロー安全性 (プログラムが不正にコードを実行しないこと) を保証することを目的とする。本来は、より複雑で高度な安全性 (例: デッドロックが生じないこと等) も保証するのが望ましいが、これらの高度な安全性を保証するための理論は、メモリ安全性と制御フロー安全性を保証する型システムを土台として構築されることが多いため、最初の段階としては、対象をメモリ安全性と制御フロー安全性に絞るのが適切である。

また、現在実現しようとしている OS の機能は、メモリ管理、スレッド管理、デバイス管理の 3 つである。これらの機能の実現方法については、2.1 節以降で説明する。

一方、実現対象としていない機能としては、ブート、仮想メモリなどが挙げられる。ブートに関しては、実現自体は可能と考えられるが、OS 起動時の実行環境は大抵通常とは異なっており、別途特別な型システムを設計しなければならない。しかし、ブートは OS 起動時にしか使われず、労力に見合う効果が少ないと思われるため、対象から外している。また仮想メモリに関しては、ページテーブルの操作自体は容易に実現可能であるが、その副作用、つまりアドレス空間の変更を扱うのが困難なため、今のところ仮想メモリの実現は対象としていない。現実的には、OS が自身のアドレス空間を変更することは稀なため、これはあまり大きな制限にはならない。

2.1 メモリ管理の実現方法

型理論の観点からは、メモリ管理とは、あるメモリ領域の型を他の型に変更することと同じである。例えば、元々ポインタ型であったメモリ領域を整数型に変更することは、ポインタとして使われていたメモリ領域を解放し、整数として再利用することを型で表現したものと見ることができる。図 1 はこのことを表した C の関数である。この関数は、引数 x が指すメモリ領域を整数として再利用している (3 行目)。

```
1:  void pointer_to_int(int** x)
2:  {
3:      int* p = (int*)x;
4:      *p = 1;
5:  }
```

図 1: 型を変更する関数の例 (ポインタとして使われていたメモリ領域を整数として再利用している)

しかし、安易にこのような型の変更を許してしまうと、型安全性が失われてしまうため、従来の強く型付けされた言語は一度確保されたメモリ領域の型を変更することを禁止してきた。例えば、図 1 の関数を用いると、図 2 のようなプログラムが書けるが、整数をポインタとして使おうとしているため実行するとエラーが生じる可能性が高い (4 行目)。これは、関数 pointer_to_int の中の変数 p と、関数 dangerous_func の引数 x が実は同じメモリ領域を参照していること (エイリアシングしていること) が型システムから知ることができないのが原因である。

```
1:  int dangerous_func(int** x)
2:  {
3:      pointer_to_int(x);
4:      return **x;
5:  }
```

図 2: 安易に型の変更を許したため型安全性が失われる例

これに対し我々は Alias Type [14] の手法を用いて、エイリアシングを追跡できるような型システムを設計した。

Alias Type の基本的なアイデアは、ポインタ型に持たせる情報を変えることにある。従来の型システムでのポインタ型は、ポインタの指す先のメモリ領域の型の情報を保持しているのに対し、Alias Type でのポインタ型は、ポインタが指すメモリのアドレスを保持する。このため、ポインタのエイリアシング関係を容易に把握できるようになっている。また、メモリ領域の型はポインタ型とは別に、メモリ型として保持される。メモリ型は、メモリのアドレスとそのアドレスのメモリ領域の型の対応を保持するマップであり、型安全性を保つために、アドレスが重複しないよう、型システムで制限されている。

例えば、図 3 は、図 1 のプログラムを Alias Type のアイデアに基づいて書き直した擬似コードである。まず、関数の引数 x の型が、int**から ptr(p) に変更されている (2 行目)。この ptr(p) は、変数 x がアドレス p を指すポインタであることを表している。また、関数の最初と最後に “[” と “]” に囲まれた部分が追加されているが、これがメモリ型を表している。関数の最初に追加されたメモリ型は、関数が実行される前のメモリの状態を (1 行目)、関数の最後に追加されたメモリ型は、関数が実行された後のメモリの状態を表している (6 行目)。ここで、関数が実行される前のアドレス p のメモリ領域はポインタ型 ptr(q) であり、更にそのアドレス q のメモリ領域の型は整数であることから、変数 x が整数を指すポインタを指すポインタであることが分かる。次に、関数の本体を見ると、変数 x が指すメモリ領域に整数値 1 を保存している (4 行目)。このため、関数実行後のメモリ型の、アドレス p に対応する型が整数に変わっている (6 行目)。メモリ型の保持するアドレスの情報が重複しないようにされているため、つまり、アドレス p とアドレス q が異なるアドレスを指していることがメモリ型から分かるため、型が変更されても型安全性は損なわれない。

```
1:  [ p --> ptr(q), q --> int ]
2:  void pointer_to_int(ptr(p) x)
3:  {
4:      *x = 1;
5:  }
6:  [ p --> int, q --> int ]
```

図 3: Alias Type のアイデアに基づいた、型を変更する関数の擬似コード例

また、図 4 は、図 2 のプログラムを Alias Type のアイデアに基づいて書き直した擬似コードである。このコードを型検査すると、5 行目で型エラーが生じる。なぜなら、4 行目で関数 pointer_to_int を呼び出しているため、アドレス p の指すメモリ領域の型がポインタ型 (ptr(q)) から整数型 (int) に変わっているからである。

このように、Alias Type はメモリの再利用を可能とする強力な手法であるが、従来の Alias Type の手法では、柔軟なメモリ管理を実現することはできなかった。これは任意長の配列を扱うことができなかったためである。機械語やアセンブリ言語などの

```
1:  [ p --> ptr(q), q --> int ]
2:  int dangerous_func(ptr(p) x)
3:  {
4:      pointer_to_int(x);
5:      return **x;
6:  }
7:  [ p --> int, q --> int ]
```

図 4: 型エラーが生じる擬似コードの例

ローレベルな視点では、メモリは単なるバイトデータの配列の集合である。例えば、典型的な IA-32 [8] システムでは、通常、 OS は起動時に、利用可能なメモリ領域の情報として、バイトデータの配列の先頭アドレスとそのサイズを BIOS から受け取る。型検査時には当然このサイズを知ることはできないので、実行時にサイズが決まるような任意長の配列を扱えなければ、柔軟なメモリ管理はおろか、利用可能なメモリ領域を知ることすらままならない。

これに対し我々の言語では、任意長の配列を扱うできるように拡張したため、柔軟なメモリ管理が可能である。実際の拡張については 3 節で、その拡張を利用したメモリ管理の実装例については 4.1 節で説明する。

2.2 スレッド管理の実現方法

現在主流の実装では、スレッドの実体は、実行中の命令を表すプログラムカウンタと、アクセス可能なデータを表すスタックの組である。従って、これらを扱えるように型システムを構築すればよい。

ここで問題となるのが、やはりエイリアシングである。例えばスタックは、1 つのスタックに対して通常複数のポインタが存在し、それらのポインタを通してスタックの要素の型が変更されるので、型システムは、これらのポインタのエイリアス関係を追跡できなければならない。

このエイリアシングの問題は、本質的には、前節 (2.1 節) で述べた問題と同じであるため、Alias Type の手法で解決できるが、スタックも実行時にしかサイズが分からないため、任意長の配列を扱えるように Alias Type を拡張することが必要である。

また、存在型 (existential type) を導入することによって、複数のスレッドが互いのメモリ領域にアクセスしないことを保証することもできる (詳細は 3 節、4.2 節を参照)。

2.3 デバイス管理の実現方法

デバイス管理は一見複雑だが、基本的にはメモリとデバイス間でデータを転送しているだけである。このため、メモリ安全性と制御フロー安全性に限れば、デバイス管理を実現することは難しくない。

3 OS 記述のための強く型付けされた言語

本節では、我々が設計した、スレッド管理やメモリ管理が可能な強く型付けされた言語を紹介する。なお、実際の言語はアセンブリ言語であるが、本質的ではない部分が複雑なためここではより抽象化した言語で説明する。我々の言語の文法は図 5、6 の通りである。以降では、まず我々の言語の抽象機械について説明した後に、型について説明する。次に操作的意味論を説明し、最後に型付け規則について説明する。

3.1 抽象機械

我々の言語では、抽象機械の状態 S は、メモリ M と命令列 I からなる。メモリ M は、整数 n から配列 a へのマップである。配列 a は、複数個のタプル h からなり、タプル h は、複数個の値 v からなる。(なお、$\mathtt{roll}_t\,(h)$、$\mathtt{pack}_{[c_1,\ldots,c_n|M]as\ t}\,(h)$、$\mathtt{union}_{at,n}\,(a)$ は、再帰型、存在型、バリアント型の形式的議論の際に便利なため導入したもので、本質的には重要でない。)
値 v は、変数 x、整数 n、関数のいずれかである。

関数は $\mathtt{fix}\ f\,[\Delta|C|\Sigma]\,(x_1:\sigma_1,\ldots,x_n:\sigma_n)\,.I$ のように書き、このとき関数の引数は $x_1,\ldots,x_n$ で、関数の本体は命令列 I である。なお、$\sigma_1,\ldots,\sigma_n$ はそれぞれ引数の型を表す。$[\Delta|C|\Sigma]$ はこの関数を呼出すときの制約条件を表している (この制約条件については後で関数型を説明するときに詳述する)。

なお、我々の言語は continutation passing style である。これには 2 つの理由がある。1 つ目の理由は、機械語やアセンブリ言語のレベルでの関数呼出しが実は continutation passing style であるからである。通常、関数呼び出しはジャンプ命令を用いて実装されるため、何らかの手段で関数の返りアドレスを関数に知らせなければならない (さもなければ関数から戻って来られない)。例えば、IA-32 の関数呼出しでは、関数の返り番地は、関数の引数とともにスタックに積まれる。これは返り番地を継続として関数に渡しているのに他ならない。2 つ目の理由は、型検査において、関数内で変更されたメモリの型情報を関数の戻り先へ反映するのに、continuation passing style が都合がよいからである。continuation passing style では、関数からのリターンも関数適用 (継続適用) として扱えるので、関数内で生じた型情報の変更を自然に戻り先に反映することができる (型検査については、3.4 節を参照)。

3.2 型

整数の型は整数値型 i で表される。整数値型 i は、整数 n と型変数 α からなる。例えば、ある変数 x が型 3 を持つならば、変数 x の値は 3 である。また、変数 y、z が型 α を持つならば、これらの変数の具体的な値は型検査時には分からないが、$y = z$ であることが分かる。

メモリ型 Σ は、アドレスを表す整数値型 i から配列型 at へのマップと型変数 ϵ の組み合わせからなる。例えば、メモリ型 $\{\mathtt{0x}c0345810 \mapsto at\}$ は、アドレス $\mathtt{0x}c0345810$ に型 at の配列が存在するだけのメモリを表す (他のアドレスは存在しない)。一方、メモリ型 $\{\mathtt{0x}c0345810 \mapsto at\} \otimes \epsilon$ は、アドレス $\mathtt{0x}c0345810$ に型 at の配列が存在し、他のアドレスにもデータが存在する (かもしれない) メモリを表す。型変数 ϵ がアドレス $\mathtt{0x}c0345810$ 以外のアドレスの存在の可能性を示唆している。また、メモリ型のマップの定義域の各アドレスは暗黙的に互いに異なるものと仮定される。例えば、メモリ型 $\{\alpha \mapsto at_1\} \otimes \{\beta \mapsto at_2\}$ においては、$\alpha \neq \beta$ が仮定されている。

配列型 at は、配列を表す型である。$t\ \mathtt{array}\,(i)$ は、各要素の型が型 t でサイズが整数値型 i であるような配列の型を表す。サイズに型変数を取ることができるため、実行時にサイズが決まるような任意長の配列を扱うことができる。一方、$at_1\ \mathtt{when}\ C_1 + \ldots + at_n\ \mathtt{when}\ C_n$ は、バリアント型を表す。この型を持つ配列は、at_1 から at_n のいずれかの型を取り、もし整数制約条件 C_i(後述) が満たされるならば、型 at_i を持つと見なすことができる。(なお、整数制約 $C_1,\ldots,C_n$ が、互いに同時に満たされることがあってはならない。さもなければ型安全性が損なわれる。現在の型システムでは、この条件は、後述の型付け規則においてチェックされる。)

配列の要素の型 t は、タプルを表す型である。$\langle\sigma_1,\ldots,\sigma_n\rangle$ は、各要素の型が σ_i であるようなタプルの型を表す。$\exists\,[\Delta|C|\Sigma]\,.t$ は依存型にパックされたタプルの型を表す。(詳細は、関数型を説明した後に述べる。) $\mu\alpha\,[\Delta]\,.t\,(c_1,\ldots,c_n)$ は再帰型を表す。ま

$$
\begin{array}{llll}
(\mathit{value}) & v & ::= & x \mid n \mid \mathtt{fix}\ f\,[\Delta|C|\Sigma]\,(x_1:\sigma_1,\ldots,x_n:\sigma_n)\,.I \\
(\mathit{tuple}) & h & ::= & \langle v_1,\ldots,v_n\rangle \mid \mathtt{roll}_t\,(h) \mid \mathtt{pack}_{[c_1,\ldots,c_n|M]as\ t}\,(h) \\
(\mathit{array}) & a & ::= & \langle h_1,\ldots,h_n\rangle \mid \mathtt{union}_{at,n}\,(a) \\
(\mathit{instruction}) & I & ::= & \mathtt{let}\ x = v.n; I \mid v_1.n = v_2; I \\
 & & & \mid \mathtt{let}\ x = v_1\ \mathit{aop}\ v_2; I \\
 & & & \mid v\,(v_1,\ldots,v_n) \\
 & & & \mid \mathtt{if}\ v_1\ \mathit{cop}\ v_2\ \mathtt{then}\ I_1\ \mathtt{else}\ I_2 \\
 & & & \mid \mathtt{let}\ x = v\,[c/\delta]\,; I \\
 & & & \mid \mathtt{coerce}\,(\iota)\,; I \\
(\mathit{coerce}) & \iota & ::= & \mathtt{roll}_{\mu\alpha[\Delta].t(c_1,\ldots,c_n)}\,(i) \\
 & & & \mid \mathtt{unroll}\,(i) \\
 & & & \mid \mathtt{pack}_{[c_1,\ldots,c_n|\Sigma_1]as\exists[\Delta|C|\Sigma_2].t}\,(i) \\
 & & & \mid \mathtt{unpack}\,(i)\ \mathtt{with}\ \Delta \\
 & & & \mid \mathtt{union}_{at_1\ \mathtt{when}\ C_1+\ldots+at_n\ \mathtt{when}\ C_n,n'}\,(i) \\
 & & & \mid \mathtt{case}_n\,(i) \\
 & & & \mid \mathtt{split}\,(i_1,i_2)\ \mathtt{with}\ \alpha \\
 & & & \mid \mathtt{concat}\,(i_1,i_2) \\
 & & & \mid \mathtt{tuple\_split}\,(i_1,i_2)\ \mathtt{with}\ \alpha \\
 & & & \mid \mathtt{tuple\_concat}\,(i_1,i_2) \\
(\mathit{arithmetic}) & \mathit{aop} & ::= & + \mid - \mid * \\
(\mathit{comparison}) & \mathit{cop} & ::= & = \mid \neq \mid < \mid > \mid \leq \mid \geq \\
(\mathit{memory}) & M & ::= & \cdot \mid \{n \mapsto a\}M \\
(\mathit{state}) & S & ::= & (M, I)
\end{array}
$$

図 5: 構文 (abstract machine)

た、タプルの要素の型 σ は、整数値型 i か、関数型を取る。

関数型は $\forall[\Delta|C|\Sigma].\Gamma \to 0$ のように書かれる。Γ は、型環境、つまり変数 x から型 σ へのマップであり、ここでは関数に渡すべき引数の型を表している。$\to 0$ は、我々の言語が continuation passing style であり、関数に返り値がないことを明示している。$[\Delta|C|\Sigma]$ は、この関数を呼出すときに満たしていなければならない制約条件を表している。以下この条件について説明する。まず、Δ は型変数の集合であり、関数がこれらの変数に関して多相的であることを意味している。次に、C は整数制約を表す。型検査器は、関数呼出し箇所でこの整数制約が満たされているかをチェックする。このため、関数の本体の型検査は、この整数制約が満たされているという仮定の下で行われる。なお、整数制約は制約変数 γ を含むことができる。例えば、整数制約 $i = j, \gamma$ は、i と j の値が等しいという制約に加え、型検査時には分からないが、何らかの制約 γ が存在していることを意味する。最後は、既に説明したメモリ型 Σ である。整数制約と同様、型検査器は、関数呼出し箇所でこのメモリ型の表すメモリの条件が満たされているかをチェックし、関数本体の型検査では、メモリがこのメモリ型を持つものと仮定する。例えば、$\forall[\alpha,\beta,\epsilon|\beta \leq 128|\epsilon \otimes \{\alpha \mapsto \langle 0\rangle\ \texttt{array}\,(\beta)\}].(x:\alpha) \to 0$ は、全要素が 0 でサイズが 128 以下の配列へのポインタを引数として取る関数の型を表す。

また、存在型 $\exists[\Delta|C|\Sigma].t$ は、整数制約 C を満たしメモリ型 Σ を持つようなメモリが存在し、その下で型 t を持つようなタプルを表す。例えば、$\exists[\alpha,\beta|\beta \leq 128|\{\alpha \mapsto \langle 0\rangle\ \texttt{array}\,(\beta)\}].\langle\alpha\rangle$ は、全要素が 0 でサイズが 128 以下の配列へのポインタを要素に持つようなタプルの型を表し、更に、そのような配列が必ず存在することを表している。

3.3 命令と操作的意味論

関数の本体である命令列 I は文字通り命令の列である。我々の言語では、命令は大きく 2 種類に分けられる。一つは、抽象機械の状態を更新する通常の命令、もう一つは、型検査時に明示的に型情報を操作するための仮想的な命令である。図 7、 8 がこれらの命令の操作的意味論である。(なお本論文において、$t[b/a]$ は、t 中に現れる自由変数 a を b で置換することを表す (変数が衝突する場合は、暗黙的に α 変換がなされるものとする)。また、$t[b_1,b_2/a_1,a_2]$ は、$t[b_1/a_1][b_2/a_2]$ の省略記法である。)

通常の命令は全部で 6 種類ある。$\texttt{let}\ x = v.n$ はロード命令で、v が表すアドレスにあるタプルの n 番目の要素をロードして、変数 x をその値に束縛する。$v_1.n = v_2$ はストア命令で、v_1 が表すアドレスにあるタプルの n 番目に値 v_2 をストアする。$\texttt{let}\ x = v_1\ aop\ v_2$ は算術命令で、変数 x を $v_1\ aop\ v_2$ の結果に束縛する。我々の言語には参照やポインタ値はなく、アドレスは単なる整数なので、タプルや配列のアドレス計算もこの算術命令で行うことができる。$v\,(v_1,\ldots,v_n)$ は関数適用命令で、$v_1,\ldots,v_n$ を引数として関数 v を呼出す。既に述べたとおり、我々の言語は continuation passing style であるため、関数は戻らない。$\texttt{if}\ v_1\ cop\ v_2\ \texttt{then}\ I_1\ \texttt{else}\ I_2$ は分岐命令で、$v_1\ cop\ v_2$ を満たす場合は命令列 I_1 を、満たさない場合は命令列 I_2 を実行する。

型情報を明示的に操作する仮想的な命令は全部で 11 種類ある。これらの命令は型検査時にのみ解釈されるため、実行時のオーバーヘッドは存在しない。

$\texttt{let}\ x = v\,[c/\delta]$ は、型適用命令で、v が表す関数の型で束縛されている型変数 δ を c で置換する。型変数 δ は、整数値型変数 (α)、メモリ型変数 (ϵ)、制約変数 (γ) の何れかである。

$\texttt{roll}_{\mu\alpha[\Delta].t(c_1,\ldots,c_n)}\,(i)$ と $\texttt{unroll}\,(i)$ は再帰型を扱う命令で、アドレス i が指す先の再帰型を一段展開したり (unroll)、また逆に元に戻したりする (roll)。

$\texttt{pack}_{[c_1,\ldots,c_n|\Sigma]as\ t}\,(i)$ と $\texttt{unpack}\,(i)\ \texttt{with}\ \Delta$ は存在型を扱う命令で、アドレス i が指す先の型を存在型にパックしたり (pack)、逆に存在型を展開したりする (unpack)。存在型にパックされたメモリは、その存在型が展開されるまでアクセスすることができなくなる。

$\texttt{union}_{at,n}\,(i)$ と $\texttt{case}_n\,(i)$ は、バリアント型を扱う命令で、アドレス i が指す先の型をバリアント型に変換したり (union)、逆にバリアント型から型を一つ選んでその型へ変換したりする (case) (詳細は、型付け規則の説明時に述べる)。

$\texttt{split}\,(i_1,i_2)\ \texttt{with}\ \alpha$ と $\texttt{concat}\,(i_1,i_2)$ は、配列型を扱う命令で、1 つの配列を 2 つの配列に分割したり (split)、逆に 2 つの配列を 1 つの配列に結合したりする (concat)。これらの命令は、配列の要素にアクセスするときに必要となる (詳細は 3.4.2 節で述べる)。また、メモリ管理を実現する上でも利用することができる (4.1 節で具体例を示す)。

$\texttt{tuple\_split}\,(i_1,i_2)\ \texttt{with}\ \alpha$ と

$$
\begin{array}{lllll}
(integer) & i & ::= & n \mid \alpha \\
(type\ var) & \delta & ::= & \alpha, \epsilon, \gamma \\
(type\ vars) & \Delta & ::= & \cdot \mid \delta, \Delta \\
(small\ type) & \sigma & ::= & i \mid \forall\, [\Delta | C | \Sigma]\, .\Gamma \to 0 \\
(type) & t & ::= & \langle \sigma_1, \ldots, \sigma_n \rangle \mid \exists\, [\Delta | C | \Sigma]\, .t \\
 & & & \mid \mu\alpha\, [\Delta]\, .t\, (c_1, \ldots, c_n) \\
(array\ type) & at & ::= & t\ \texttt{array}\, (i) \\
 & & & \mid at_1\ \texttt{when}\ C_1 + \ldots + at_n\ \texttt{when}\ C_n \\
(memory\ type) & \Sigma & ::= & \emptyset \mid \Sigma \otimes \{i \mapsto at\} \mid \Sigma \otimes \epsilon \\
(type\ env) & \Gamma & ::= & \cdot \mid \Gamma, x : \sigma \\
(constructor) & c & ::= & i \mid \Sigma \mid C \\
(constraints) & C & ::= & \cdot \mid C, e_1\ cop\ e_2 \mid C, \gamma \\
(expression) & e & ::= & i \mid e_1\ aop\ e_2
\end{array}
$$

図 6: 構文 (型)

$\texttt{tuple\_concat}\,(i_1, i_2)$ は、split と concat のタプル版である。これらの命令は、複数のサイズ 1 の配列から、複数要素のタプル 1 つを生成するのに使われる、メモリ管理を実現する上で必要となる命令である (具体例は 4.1 節)。

3.4 型付け規則

図 9、10、11 は型付け規則である。まず、$\vdash (M, I)$ は抽象機械の状態の妥当性判定規則である。もし (M, I) が、この判定にパスするならば、ある状態 (M', I') が存在して、$(M, I) \mapsto_P (M', I')$ となる状態遷移が必ず存在する。これはつまり実行時にエラーが生じないことを意味する。ただし、我々はまだこの性質 (型安全性) を証明していない。この抽象機械の状態の妥当性判定は、メモリの妥当性判定と命令の妥当性判定の 2 つの規則からなる。以降ではこれらの規則についてそれぞれ説明する。

3.4.1 メモリの妥当性判定

メモリの妥当性判定規則は$\vdash M : \Sigma$ である (MEMORY)。この規則は、メモリ M がメモリ型 Σ を持つかどうかをチェックする。具体的には、M と Σ の定義域 (アドレスの集合) が一致し、更にそれぞれのアドレスについて、M が示す配列が、Σ が示す配列型を持つかをチェックする。

配列 a が配列型 at を持つかどうかを判定する規則は $\Delta; \Gamma; C \vdash a : at$ である。この規則は、バリアント型を扱うものもあるため 2 種類存在する (ARRAY と ARRAY_UNION)。

型付け規則 ARRAY は、配列の各要素が型 t を持ち、更に配列のサイズが型で指定されたサイズと一致するかをチェックする。例えば、$\Delta; \Gamma; C \vdash \langle h_1, h_2 \rangle : t\ \texttt{array}\,(i)$ の判定では、整数制約解消系を用いて $i = 2$ が満たされるかどうかチェックする。これを $\Delta; C \models i = 2$ と書く。なお、本論文では、$\Delta; C \models C'$(条件 C の下で C' が満たされるか) の判定規則については述べていないが、一般に整数制約は、線形であれば必ず解けることが知られている。

型付け規則 ARRAY_UNION は、バリアント型を扱う型付け規則である。この規則は、配列型がバリアント型に正しく変換されているかをチェックしている。バリアント型の型付けについては、3.4.2 節で説明する。なお、この規則は、型安全性の証明に必要となるもので、実際の型検査に使われることはない。

配列の要素 h が型 t を持つかどうかの判定規則は、$\Delta; \Gamma; C \vdash h : t$ である。再帰型と存在型を扱うため、この規則は 3 種類存在する (TUPLE と TUPLE_ROLL と TUPLE_PACK)。

型付け規則 TUPLE は、タプルの各要素がタプル型で指定された要素の型を持つかをチェックする。この規則は、一般の言語のタプルの型付けと同様である。

型付け規則 TUPLE_ROLL と TUPLE_PACK はそれぞれ再帰型と存在型を扱う規則である。再帰型の型付けは、一般の言語と同様である。存在型の型付

$$
\begin{array}{lcl}
(M\{n_1 \mapsto \langle\langle v_1, \ldots, v_n\rangle\rangle\}, \texttt{let}\ x = n_1.n_2; I) & \mapsto_P & (M\{n_1 \mapsto \langle\langle v_1, \ldots, v_n\rangle\rangle\}, I\,[v_{n_2}/x]) \\
(M\{n_1 \mapsto \langle\langle v_1, \ldots, v_{n_2}, \ldots, v_n\rangle\rangle\}, n_1.n_2 = v; I) & \mapsto_P & (M\{n_1 \mapsto \langle\langle v_1, \ldots, v, \ldots, v_n\rangle\rangle\}, I) \\
(M, \texttt{let}\ x = n_1\ aop\ n_2; I) & \mapsto_P & (M, I\,[n/x])\ \ where\ n = n_1\ aop\ n_2 \\
(M, v\,(v_1, \ldots, v_n)) & \mapsto_P & (M, \theta\,(I)) \\
 & & where \\
 & & v = \texttt{fix}\ f\,[\cdot|C|\Sigma]\,(x_1 : \sigma_1, \ldots, x_n : \sigma_n)\,.I \\
 & & \theta = [v/f]\,[v_1, \ldots, v_n/x_1, \ldots, x_n] \\
(M, \texttt{if}\ n_1\ cop\ n_2\ \texttt{then}\ I_1\ \texttt{else}\ I_2) & \mapsto_P & if\ n_1\ cop\ n_2\ then\ \ (M, I_1) \\
 & & else\ \ (M, I_2) \\
(M, \texttt{let}\ x = v\,[c/\delta]\,; I) & \mapsto_P & (M, \theta\,(I)) \\
 & & where \\
 & & v = \texttt{fix}\ f\,[\Delta|C|\Sigma]\,(x_1 : \sigma_1, \ldots, x_n : \sigma_n)\,.I \\
 & & \delta \in \Delta \\
 & & C' = [c/\delta]\,C \quad \Sigma' = [c/\delta]\,\Sigma \\
 & & \sigma'_i = [c/\delta]\,\sigma_i \quad I' = [c/\delta]\,I \\
 & & \theta = [\texttt{fix}\ f\,[\Delta - \delta|C'|\Sigma']\,(x_1 : \sigma'_1, \ldots, x_n : \sigma'_n)\,.I'/x] \\
(M, \texttt{coerce}\,(\iota)\,; I) & \mapsto_P & (M', \theta\,(I)) \\
 & & where \\
 & & \iota\,(M) \mapsto_\iota M', \theta
\end{array}
$$

図 7: 操作的意味論 (instructions)

$$
\begin{array}{lll}
\mathtt{roll}_t\,(n)\,(M\,\{n \mapsto \langle h\rangle\}) & \mapsto_\iota & M\,\{n \mapsto \langle \mathtt{roll}_t\,(h)\rangle\}\,,[] \\
\mathtt{unroll}\,(n)\,(M\,\{n \mapsto \langle \mathtt{roll}_t\,(h)\rangle\}) & \mapsto_\iota & M\,\{n \mapsto \langle h\rangle\}\,,[] \\
\mathtt{pack}_{[c_1,\ldots,c_n|\Sigma]as\ t}\,(n)\,(M\,\{n \mapsto \langle h\rangle\}\,M') & \mapsto_\iota & M\left\{n \mapsto \left\langle \mathtt{pack}_{[c_1,\ldots,c_n|M']as\ t}\,(h)\right\rangle\right\},[] \\
 & & \textit{where} \\
 & & \Sigma = \{n_1 \mapsto at_1\} \otimes \ldots \otimes \{n_m \mapsto at_m\} \\
 & & M' = \{n_1 \mapsto a_1, \ldots, n_m \mapsto a_m\} \\
\mathtt{unpack}\,(n)\,\mathtt{with}\ \Delta\,(M'') & \mapsto_\iota & MM'\,\{n \mapsto h\}\,,[c_1,\ldots,c_n/\Delta] \\
 & & \textit{where} \\
 & & M'' = M\left\{n \mapsto \left\langle \mathtt{pack}_{[c_1,\ldots,c_n|M']as\exists[\Delta'|C|\Sigma].t}\,(h)\right\rangle\right\} \\
\mathtt{union}_{at,n'}\,(n)\,(M\,\{n \mapsto a\}) & \mapsto_\iota & M\,\{n \mapsto \mathtt{union}_{at,n'}\,(a)\}\,,[] \\
\mathtt{case}_{n'}\,(n)\,(M\,\{n \mapsto \mathtt{union}_{at,n''}\,(a)\}) & \mapsto_\iota & M\,\{n \mapsto a\}\,,[] \\
\mathtt{split}\,(n_1,n_2)\ \mathtt{with}\ \alpha\,(M') & \mapsto_\iota & M\,\{n \mapsto \langle h_1,\ldots,h_{n_2}\rangle\}\,\{n'' \mapsto \langle h_{n_2+1},\ldots,h_n\rangle\}\,,[] \\
 & & \textit{where} \\
 & & M' = M\,\{n_1 \mapsto \langle h_1,\ldots,h_n\rangle\} \\
\mathtt{concat}\,(n_1,n_2)\,(M') & \mapsto_\iota & M\,\{n_1 \mapsto \langle h_1,\ldots,h_n,h'_1,\ldots,h'_m\rangle\}\,,[] \\
 & & \textit{where} \\
 & & M' = M\,\{n_1 \mapsto \langle h_1,\ldots,h_n\rangle\}\,\{n_2 \mapsto \langle h'_1,\ldots,h'_m\rangle\} \\
 & & \textit{and}\ n_1 + n * \textit{sizeof}\,(h_1) = n_2 \\
 & & \textit{and}\ \textit{sizeof}\,(h_i) = \textit{sizeof}\,\left(h'_j\right) \\
\mathtt{tuple\_split}\,(n_1,n_2)\ \mathtt{with}\ \alpha\,(M') & \mapsto_\iota & M\,\{n \mapsto \langle\langle v_1,\ldots,v_{n_2}\rangle\rangle\}\,\{n'' \mapsto \langle\langle v_{n_2+1},\ldots,v_n\rangle\rangle\}\,,[] \\
 & & \textit{where} \\
 & & M' = M\,\{n_1 \mapsto \langle v_1,\ldots,v_n\rangle\} \\
\mathtt{tuple\_concat}\,(n_1,n_2)\,(M') & \mapsto_\iota & M\,\{n_1 \mapsto \langle\langle v_1,\ldots,v_n,v'_1,\ldots,v'_m\rangle\rangle\}\,,[] \\
 & & \textit{where} \\
 & & M' = M\,\{n_1 \mapsto \langle\langle v_1,\ldots,v_n\rangle\rangle\}\,\{n_2 \mapsto \langle\langle v'_1,\ldots,v'_m\rangle\rangle\} \\
 & & \textit{and}\ n_1 + n = n_2
\end{array}
$$

図 8: 操作的意味論 (coerce)

けについては、3.4.2 で説明する。なお、この規則は、ARRAY_UNION と同様、型安全性の証明に必要となるもので、実際の型検査に使われることはない。

タプルの要素 v が型 σ を持つかどうかの判定規則は、$\Delta;\Gamma;C \vdash v : \sigma$ である。変数、整数、関数の 3 種類を扱うために規則も 3 種類存在する (VALUE_VARIABLE、VALUE_INTEGER、VALUE_FUNCTION)。
型付け規則 VALUE_INTEGER では、整数制約解消系を用いて、整数 n が型 i と等しいかどうかをチェックしている ($\Delta;C \models n = i$)。型付け規則 VALUE_VARIABLE と VALUE_FUNCTION は、一般の言語の変数や関数の型付けと同様である。

3.4.2 命令の妥当性判定

命令の妥当性判定規則は $\Delta;\Gamma;C;\Sigma \vdash I$ である。ただし、型検査時にのみ解釈される仮想命令の型付けは、$\Delta;C;\Gamma \vdash \iota \Rightarrow \Delta';C';\Gamma'$ として別に扱っている (ただし、型適用命令の型付け規則 APPLY は例外である)。

型付け規則 LOAD は、ロード命令の型付けである。この規則では、まず値 v が整数型を持ち、その整数がアドレスとして有効であるかをチェックする。次に、そのアドレスが指す先がサイズ 1 の配列であるかをチェックし、そして、その配列の要素タプルのサイズが、整数 n より大きいかをチェックする。以上のチェックが全てパスしたら、型環境 Γ を拡張して変数 x が型 σ_n を持つようにし、その型環境で、次の命令を型付けする。

型付け規則 STORE は、ストア命令の型付けである。この規則では、型付け規則 LOAD と同様、まず値 v_1 が整数型を持ち、その整数がアドレスとして有効であるかをチェックする。次に、そのアドレスが指す先がサイズ 1 の配列であるかをチェックし、そして、その配列の要素タプルのサイズが、整数 n より大きいかをチェックする。以上のチェックが全てパスしたら、タプルの n 番目の要素の型を、v_2 の型 σ に変更し、そのメモリ型の下で、次の命令を型付けする。

型付け規則 LOAD と STORE では、操作をサイズ 1 の配列のみに許しているため、サイズが 1 以外の配列の要素を読み書きするためには、まず `split` を用いてサイズ 1 の配列を作る必要がある。これは一見無用な制限に見えるが、実は必要である。例えば任意の整数の配列は型 $\exists\,[\alpha|\cdot|\cdot]\,.\,\langle\alpha\rangle\ \mathtt{array}\,(s)$ で表されるが、この配列への直接アクセスを型で上手く表現することができない (ロードの場合は、ロード先の変数 x の型が表せない。ストアの場合はストア後の配列の型が表せない)。

型付け規則 ARITH は、算術演算命令の型付けである。この規則は、一般の言語と同様であるが、演算の結果を整数制約に加えて、次の命令を型付けする点が異なる。なお、非線形な整数制約が生成される可能性があるのは、この規則だけである。

型付け規則 JUMP は、関数呼出しの型付けである。引き数や多相型の扱いなど、基本的には一般の言語と同様の型付けであるが、型の指定する制約 C' を満たしているかをチェックしたり ($\Delta;C \models C'$)、メモリに関する条件 (Σ) を満たしているかをチェックする点が異なる。

型付け規則 BRANCH は、条件分岐の型付けである。この規則も、一般の言語の型付けとほぼ同様であるが、条件判定の結果を整数制約 C に加えて、分岐先の命令の型付けを行う点が異なる。この規則によって、配列の境界検査や、バリアント型のパターンマッチの型付けが可能となる。なお、$\neg cop$ は、比較演算子 cop の自然な意味での逆の演算子を表す。

型付け規則 APPLY は、型適用の型付けである。この規則もやはり、一般の言語とほぼ同様である。なお、$\Delta - \delta$ は、集合 Δ から要素 δ を除いた残りの集合を表す。

型付け規則 ROLL と UNROLL は、再帰型の操作の型付け規則である。これらの規則は、一般の言語の再帰型の型付け (特にパラメトリック再帰型) と同様である。

型付け規則 PACK と UNPACK は、存在型の操作の型付け規則である。これらの規則も、ほぼ一般の言語と同様であるが、大きく異なるのが、メモリ型の扱いである。

型付け規則 PACK は、アドレス i が指す先の型を存在型にパックするが、このとき、メモリの一部を存在型にパックすることが可能で、パックされたメモリはメモリ型から削除され、以降の命令の型付けでは、存在しないものと仮定される。これは、もしパックされたメモリをメモリ型から削除しなければ、以降の命令で変更されたメモリの状態が、存在型のメモリの状態と異なってしまい、型安全性が失われてしまうからである。

型付け規則 UNPACK は、型付け規則 PACK とは逆に、アドレス i が指す先の存在型をアンパック

$$\frac{\vdash M : \Sigma \qquad \cdot;\cdot;\cdot;\Sigma \vdash I}{\vdash (M, I)} \quad (\textsc{state})$$

$$\frac{\begin{array}{c} M = \{n_1 \mapsto a_1, \ldots, n_m \mapsto a_m\} \\ \Sigma = \{n_1 \mapsto at_1, \ldots, n_m \mapsto at_m\} \\ \cdot;\cdot;\cdot \vdash a_i : at_i \\ (\text{and } n_i \mapsto at_i \text{ does not overlap with others}) \end{array}}{\vdash M : \Sigma} \quad (\textsc{memory})$$

$$\frac{\Delta;\Gamma;C \vdash h_j : t \qquad \Delta;C \models n = i}{\Delta;\Gamma;C \vdash \langle h_1, \ldots, h_n \rangle : t\ \mathtt{array}\,(i)} \quad (\textsc{array})$$

$$\frac{\begin{array}{c} at = at_1\ \mathtt{when}\ C_1 + \ldots + at_n\ \mathtt{when}\ C_n \\ \Delta;\Gamma;C \vdash a : at_{n'} \qquad \Delta;C \models C_{n'} \end{array}}{\Delta;\Gamma;C \vdash \mathtt{union}_{at,n'}\,(a) : at} \quad (\textsc{array\_union})$$

$$\frac{\Delta;\Gamma;C \vdash v_j : \sigma_j}{\Delta;\Gamma;C \vdash \langle v_1, \ldots, v_n \rangle : \langle \sigma_1, \ldots, \sigma_n \rangle} \quad (\textsc{tuple})$$

$$\frac{\begin{array}{c} t = \mu\alpha\,[\Delta'].t\,(c_1, \ldots, c_n) \\ \Delta;\Gamma;C \vdash h : t\,[\mu\alpha\,[\Delta'].t/\alpha]\,[c_1, \ldots, c_n/\Delta'] \end{array}}{\Delta;\Gamma;C \vdash \mathtt{roll}_t\,(h) : t} \quad (\textsc{tuple\_roll})$$

$$\frac{\begin{array}{c} t = \exists\,[\Delta'|C'|\Sigma'].t' \\ \Delta;\Gamma;C \vdash h : t'\,[c_1, \ldots, c_n/\Delta'] \\ \vdash M : \Sigma'\,[c_1, \ldots, c_n/\Delta'] \\ \Delta;C \models C'\,[c_1, \ldots, c_n/\Delta'] \end{array}}{\Delta;\Gamma;C \vdash \mathtt{pack}_{[c_1,\ldots,c_n|M]as\ t}\,(h) : t} \quad (\textsc{tuple\_pack})$$

$$\frac{\Gamma\,(x) = \sigma}{\Delta;\Gamma;C \vdash x : \sigma} \quad (\textsc{value\_variable})$$

$$\frac{\Delta;C \models n = i}{\Delta;\Gamma;C \vdash n : i} \quad (\textsc{value\_integer})$$

$$\frac{\begin{array}{c} \sigma_f = \forall\,[\Delta'|C'|\Sigma'].\Gamma' \to 0 \\ \Gamma' = x_1 : \sigma_1, \ldots, x_n : \sigma_n \\ \Delta\Delta';\Gamma\Gamma', f : \sigma_f; CC'; \Sigma' \vdash I \end{array}}{\Delta;\Gamma;C \vdash \mathtt{fix}\ f\,[\Delta'|C'|\Sigma']\,(x_1 : \sigma_1, \ldots, x_n : \sigma_n).I : \sigma_f} \quad (\textsc{value\_function})$$

図 9: 型付け規則 (machine state)

$$\frac{\begin{array}{c}\Delta;\Gamma;C\vdash v:i\\ \Sigma=\Sigma'\otimes\{i'\mapsto\langle\ldots,\sigma_n,\ldots\rangle\ \texttt{array}\,(j)\}\\ \Delta;C\models i=i'\qquad\Delta;C\models j=1\qquad\Delta;\Gamma,x:\sigma_n;C;\Sigma\vdash I\end{array}}{\Delta;\Gamma;C;\Sigma\vdash\texttt{let}\ x=v.n;I}\ (\textsc{load})$$

$$\frac{\begin{array}{c}\Delta;\Gamma;C\vdash v_1:i\qquad\Delta;\Gamma;C\vdash v_2:\sigma\\ \Sigma=\Sigma'\otimes\{i'\mapsto\langle\ldots,\sigma_n,\ldots\rangle\ \texttt{array}\,(j)\}\\ \Delta;C\models i=i'\qquad\Delta;C\models j=1\\ \Sigma''=\Sigma'\otimes\{i\mapsto\langle\ldots,\sigma,\ldots\rangle\ \texttt{array}\,(1)\}\\ \Delta;\Gamma;C;\Sigma''\vdash I\end{array}}{\Delta;\Gamma;C;\Sigma\vdash v_1.n=v_2;I}\ (\textsc{store})$$

$$\frac{\begin{array}{c}\Delta;\Gamma;C\vdash v_1:i_1\qquad\Delta;\Gamma;C\vdash v_2:i_2\\ \Delta'=\Delta,\alpha\quad(\alpha\text{ is fresh})\\ \Gamma'=\Gamma,x:\alpha\\ C'=C,\alpha=i_1\ op\ i_2\\ \Delta';\Gamma';C';\Sigma\vdash I\end{array}}{\Delta;\Gamma;C;\Sigma\vdash\texttt{let}\ x=v_1\ op\ v_2;I}\ (\textsc{arith})$$

$$\frac{\begin{array}{c}\Delta;\Gamma;C\vdash v:\forall\,[\cdot|C'|\Sigma]\,.\,(\sigma_1,\ldots,\sigma_n)\to 0\\ \Delta;\Gamma;C\vdash v_j:\sigma_j\\ \Delta;C\models C'\end{array}}{\Delta;\Gamma;C;\Sigma\vdash v\,(v_1,\ldots,v_n)}\ (\textsc{jump})$$

$$\frac{\begin{array}{c}\Delta;\Gamma;C\vdash v_j:i_j\\ \Delta;\Gamma;C,i_1\ cop\ i_2;\Sigma\vdash I_1\\ \Delta;\Gamma;C,i_1\ \neg cop\ i_2;\Sigma\vdash I_2\end{array}}{\Delta;\Gamma;C;\Sigma\vdash\texttt{if}\ v_1\ cop\ v_2\ \texttt{then}\ I_1\ \texttt{else}\ I_2}\ (\textsc{branch})$$

$$\frac{\begin{array}{c}\Delta;\Gamma;C\vdash v:\forall\,[\Delta'|C'|\Sigma']\,.\,(\sigma_1,\ldots,\sigma_n)\to 0\\ \delta\in\Delta'\\ \Gamma'=\Gamma,x:\forall\,[\Delta'-\delta|C''|\Sigma'']\,.\,(\sigma'_1,\ldots,\sigma'_n)\to 0\\ C''=[c/\delta]\,C\qquad\Sigma''=[c/\delta]\,\Sigma\\ \sigma'_i=[c/\delta]\,\sigma_i\qquad\Delta;\Gamma';C;\Sigma\vdash I\end{array}}{\Delta;\Gamma;C;\Sigma\vdash\texttt{let}\ x=v\,[c/\delta]\,;I}\ (\textsc{apply})$$

$$\frac{\begin{array}{c}\Delta;C;\Sigma\vdash\iota\Rightarrow\Delta';C';\Sigma'\\ \Delta';\Gamma;C';\Sigma'\vdash I\end{array}}{\Delta;\Gamma;C;\Sigma\vdash\texttt{coerce}\,(\iota)\,;I}\ (\textsc{coerce})$$

図 10: 型付け規則 (instructions)

するが、このとき、存在型にパックされているメモリを復帰してメモリ型を拡張し ($\Sigma'[\Delta''/\Delta']$)、以降の命令を型付けする。

なお、型付け規則 ROLL、UNROLL、PACK、UNPACK の 4 つは、型付け規則 LOAD、STORE と同様な理由で、操作はサイズ 1 の配列のみに許される。

型付け規則 UNION と CASE は、バリアント型の操作の型付け規則である。型付け規則 UNION は、アドレス i が指す先の型をバリアント型に変換することができるかをチェックする。具体的には、挿入しようとしているバリアント型の枝 (n' 番目) から得られる制約 $C_{n'}$ が満たされているかをチェックする ($\Delta; C \models C_{n'}$)。また、型付け規則 CASE は、逆に、アドレス i が指す先のバリアント型を、そのバリアント型の中から n 番目の枝を選んでその型 at_n に変換する。このとき、選んだ枝に対応する制約 C_n が満たされているかどうかをチェックする ($\Delta; C \models C_n$)。また、型付け規則 UNION と CASE では、バリアント型のそれぞれの枝の制約条件が、互いに同時に満たされることがないことをチェックする ($\Delta; C \not\models C_i \wedge C_j \quad (i \neq j)$)。

型付け規則 SPLIT と CONCAT は、配列型の分割と結合の型付け規則である。

型付け規則 SPLIT は、まず、分割しようとしている配列のサイズ (j_1) が、十分に大きいかをチェックする ($\Delta; C \models i_2 \leq j_1$)。次に実際に配列を分割し、新たにできた 2 つの配列でメモリ型を拡張し、更に 2 つの配列のアドレス間の関係を整数制約に加え、次の命令を型付けする。なお、*sizeof* (t) は、タプル型 t の要素数である。(再帰型や存在型の場合は、内側の t に再帰的に *sizeof* を適用する。) ここで、タプル型の要素数は固定であるため、新たに生成される整数制約は必ず線形となる。

型付け規則 CONCAT は、型付け規則 SPLIT とは逆に、アドレス i_1 の指す配列とアドレス j_1 の指す配列を結合するが、このとき、2 つの配列が連続しているかどうかチェックする ($\Delta; C \models j_1 = i_1 + sizeof\ (t) * i_2$)。そして、新たな配列のサイズを整数制約に加え、次の命令を型付けする。

型付け規則 TUPLE_SPLIT と TUPLE_CONCAT は、型付け規則 SPLIT と CONCAT とほとんど同じであるが、こちらはタプルの分割と結合を扱う規則である。

4 OS の基本機能の実装

4.1 メモリ管理の実装例

本節では、メモリ管理の実装例として、データ配列からメモリを確保し解放する例を示す (図 12、ただし、煩雑な型注釈等は省略してある)。この例は、与えられたメモリ領域 (データ配列) からサイズ 2 のタプルを確保し、その後解放するだけの非常に簡単な関数であるが、より柔軟で複雑なメモリ管理 (malloc/free) を実装する上で必須となる操作を含んでいる 。

この関数 `simple` におけるメモリ確保は以下のように行われる。まず、与えられた配列が、タプルを確保するのに十分なサイズを持つかどうかを調べる (1〜3 行目)。次に、与えられた配列を、サイズ 2 の配列と残りの配列の 2 つに分割する (5 行目)。更に確保したサイズ 2 の配列をサイズ 1 の配列に分割し、これをタプルとして結合する (6〜8 行目)。

一方、メモリの解放は、メモリの確保とは逆に以下のように行われる。まず、タプルをサイズ 1 の配列に分割し、これを配列として結合する (12〜14 行目)。最後に、この配列を、メモリの確保時に余った配列と結合し (15 行目)、関数の呼出元に戻る (17 行目)。

なお、ここで説明した簡単な例以外に、関数のスタックフレームや malloc/free のような仕組みが実装できることを、我々は確認している。

4.2 スレッド管理の実装

本節では、スレッド管理の実装例として、2 つのスレッドを切り替える関数を示す (図 13、ただし、前節同様、煩雑な型注釈等は省略してある。また、実際には関数のスタックフレームも扱わなければ実用的なスレッドとはならないので、この例は不完全であるが、例が煩雑になるのを避けるため、スタックフレームについては省略している。また、ここでは 1 CPU 上でのスレッドについて考察しており、SMP は対象にしていない)。

この関数 `switch` におけるスレッド切替は以下のように行われる。まず、この関数を呼出した関数の継続 (= 戻り番地 = 現在のスレッドのプログラムカウンタ) を、スレッド構造体に保存する (1 行目)。次に、現在のスレッドのメモリ (メモリ型 ϵ) とともに、スレッド構造体をパックする (2 行目)。これで、現在のスレッドのメモリを他のスレッドが直接アクセスすることを防げる。続いて、次のスレッドのスレッド構造体を展開する (4 行目)。これで、次のスレッド

$$\frac{t = \mu\alpha\,[\Delta'].t'\,(c_1,\ldots,c_n) \qquad \Sigma = \Sigma' \otimes \{i \mapsto t'\,[\mu\alpha\,[\Delta'].t'/\alpha]\,[c_1,\ldots,c_n/\Delta']\ \texttt{array}\,(j)\} \qquad \Delta, C \models j = 1}{\Delta; C; \Sigma \vdash \texttt{roll}_t\,(i) \Rightarrow \Delta; C; \Sigma' \otimes \{i \mapsto t\ \texttt{array}\,(1)\}}\ (\textsc{roll})$$

$$\frac{\Sigma = \Sigma' \otimes \{i \mapsto \mu\alpha\,[\Delta'].t'\,(c_1,\ldots,c_n)\ \texttt{array}\,(j)\} \qquad \Delta, C \models j = 1}{\Delta; C; \Sigma \vdash \texttt{unroll}\,(i) \Rightarrow \Delta; C; \Sigma' \otimes \{i \mapsto t'\,[\mu\alpha\,[\Delta'].t'/\alpha]\,[c_1,\ldots,c_n/\Delta']\ \texttt{array}\,(1)\}}\ (\textsc{unroll})$$

$$\frac{\begin{array}{c}\Sigma = \Sigma'' \otimes \{i \mapsto t\,[c_1,\ldots,c_n/\Delta']\ \texttt{array}\,(j)\} \otimes \Sigma'\,[c_1,\ldots,c_n/\Delta'] \\ \Delta, C \models C'\,[c_1,\ldots,c_n/\Delta'] \qquad \Delta, C \models j = 1\end{array}}{\Delta; C; \Sigma \vdash \texttt{pack}_{[c_1,\ldots,c_n|\Sigma'[c_1,\ldots,c_n/\Delta']]as\exists[\Delta'|C'|\Sigma'].t}\,(i) \Rightarrow \Delta; C; \Sigma'' \otimes \{i \mapsto \exists\,[\Delta'|C'|\Sigma'].t\ \texttt{array}\,(1)\}}\ (\textsc{pack})$$

$$\frac{\Sigma = \Sigma'' \otimes \{i \mapsto \exists\,[\Delta'|C'|\Sigma'].t\ \texttt{array}\,(j)\} \qquad \Delta, C \models j = 1}{\Delta; C; \Sigma \vdash \texttt{unpack}\,(i)\ \texttt{with}\ \Delta'' \Rightarrow \Delta\Delta''; CC'\,[\Delta''/\Delta']; \Sigma'' \otimes \{i \mapsto t\,[\Delta''/\Delta']\ \texttt{array}\,(1)\} \otimes \Sigma'\,[\Delta''/\Delta']}\ (\textsc{unpack})$$

$$\frac{\begin{array}{c}at = at_1\ \texttt{when}\ C_1 + \ldots + at_n\ \texttt{when}\ C_n \qquad \Sigma = \Sigma' \otimes \{i \mapsto at_{n'}\} \\ \Delta; C \models C_{n'} \qquad \Delta; C \not\models C_i \wedge C_j \quad (i \neq j)\end{array}}{\Delta; C; \Sigma \vdash \texttt{union}_{at,n'}\,(i) \Rightarrow \Delta; C; \Sigma' \otimes \{i \mapsto at\}}\ (\textsc{union})$$

$$\frac{\begin{array}{c}\Sigma = \Sigma' \otimes \{i \mapsto at_1\ \texttt{when}\ C_1 + \ldots + at_{n'}\ \texttt{when}\ C_{n'}\} \\ \Delta; C \models C_n \qquad \Delta; C \not\models C_i \wedge C_j \quad (i \neq j)\end{array}}{\Delta; C; \Sigma \vdash \texttt{case}_n\,(i) \Rightarrow \Delta; C; \Sigma' \otimes \{i \mapsto at_n\}}\ (\textsc{case})$$

$$\frac{\begin{array}{c}\Sigma = \Sigma' \otimes \{i_1 \mapsto t\ \texttt{array}\,(j_1)\} \qquad \Delta; C \models i_2 \leq j_1 \\ \Delta' = \Delta, \alpha, j \quad (\alpha \text{ and } j \text{ are fresh}) \\ C' = C, \alpha = i_1 + \mathit{sizeof}\,(t) * i_2,\ j_1 = i_2 + j\end{array}}{\Delta; C; \Sigma \vdash \texttt{split}\,(i_1, i_2)\ \texttt{with}\ \alpha \Rightarrow \Delta'; C'; \Sigma' \otimes \{i_1 \mapsto t\ \texttt{array}\,(i_2)\} \otimes \{\alpha \mapsto t\ \texttt{array}\,(j)\}}\ (\textsc{split})$$

$$\frac{\begin{array}{c}\Sigma = \Sigma' \otimes \{i_1 \mapsto t\ \texttt{array}\,(i_2)\} \otimes \{j_1 \mapsto t\ \texttt{array}\,(j_2)\} \\ \Delta; C \models j_1 = i_1 + \mathit{sizeof}\,(t) * i_2 \\ \Delta' = \Delta, j \quad (j \text{ is fresh}) \qquad C' = C, j = i_2 + j_2\end{array}}{\Delta; C; \Sigma \vdash \texttt{concat}\,(i_1, j_1) \Rightarrow \Delta'; C'; \Sigma' \otimes \{i_1 \mapsto t\ \texttt{array}\,(j)\}}\ (\textsc{concat})$$

$$\frac{\begin{array}{c}\Sigma = \Sigma' \otimes \{i_1 \mapsto \langle\sigma_1,\ldots,\sigma_n\rangle\ \texttt{array}\,(j)\} \qquad \Delta; C \models i_2 \leq n \qquad \Delta; C \models j = 1 \\ \Delta' = \Delta, \alpha \quad (\alpha \text{ is fresh}) \qquad C' = C, \alpha = i_1 + i_2\end{array}}{\begin{array}{c}\Delta; C; \Sigma \vdash \texttt{tuple\_split}\,(i_1, i_2)\ \texttt{with}\ \alpha \Rightarrow \\ \Delta'; C'; \Sigma' \otimes \{i_1 \mapsto \langle\sigma_1,\ldots,\sigma_{i_2}\rangle\ \texttt{array}\,(1)\} \otimes \{\alpha \mapsto \langle\sigma_{i_2+1},\ldots,\sigma_n\rangle\ \texttt{array}\,(1)\}\end{array}}\ (\textsc{tuple\_split})$$

$$\frac{\begin{array}{c}\Sigma = \Sigma' \otimes \{i_1 \mapsto \langle\sigma_1,\ldots,\sigma_n\rangle\ \texttt{array}\,(j_1)\} \otimes \{i_2 \mapsto \langle\sigma'_1,\ldots,\sigma'_m\rangle\ \texttt{array}\,(j_2)\} \\ \Delta; C \models j_1 = 1 \qquad \Delta; C \models j_2 = 1 \qquad \Delta; C \models i_2 = i_1 + n\end{array}}{\Delta; C; \Sigma \vdash \texttt{tuple\_concat}\,(i_1, i_2) \Rightarrow \Delta; C; \Sigma' \otimes \{i_1 \mapsto \langle\sigma_1,\ldots,\sigma_n,\sigma'_1,\ldots,\sigma'_m\rangle\ \texttt{array}\,(1)\}}\ (\textsc{tuple\_concat})$$

図 11: 型付け規則 (coerce instructions)

```
fix simple[mem, size, ϵ | · |
  ϵ ⊗ {mem ↦ ⟨int⟩ array(size)}]
 (mem_p : mem, sz : size, ret : ret_t).

 1:         if sz < 2 then
 2:                ret ()
 3:         else
 4:
 5:         split(mem, 2) with rest_mem;
 6:         split(mem, 1) with mem';
 7:         unpack(mem); unpack(mem');
 8:         tuple_concat(mem, mem');
 9:
10:         // do something...
11:
12:         tuple_split(mem, 1) with mem';
13:         pack(mem); pack(mem');
14:         concat(mem, mem');
15:         concat(mem, rest_mem);
16:
17:         ret ()
```

図 12: 非常に簡単なメモリ管理の実装例 (ret_t の定義は省略

のメモリにアクセスすることが可能となる。そして、次のスレッドのスレッド構造体に保存してあった継続 (= 戻り番地 = 次のスレッドプログラムカウンタ) をロードし (5 行目)、最後に、その継続を適用してスレッドの切替を完了する (7 行目)。

```
fix switch[c_thd, n_thd, α, ϵ | · |
  ϵ ⊗ {c_thd ↦ ⟨α⟩ array(1)}
   ⊗{n_thd ↦ ∃[ϵ'| · |ϵ'] . ⟨n_ret_t⟩ array(1)}]
 (cur : c_thd, next : n_thd, ret : ret_t).

1:         cur.1 = ret;
2:         pack(c_thd);
3:
4:         unpack(n_thd);
5:         let next_ret = n_thd.1;
6:
7:         next_ret (next, cur)
```

図 13: 簡単なスレッド切替の実装例 (ret_t や n_ret_t の定義は省略。また整数制約についても省略。)

5 関連研究

5.1 型理論とメモリ管理

線形型 [13] システムは、あるメモリ領域へのアクセスが 1 回しか行われないように制限された型システムである。このため、線形型システムにおいては、エイリアシングの問題が発生しないため、メモリ領域の型の変更を安全に行うことができる。この線形型に基づいた型付きアセンブリ言語もいくつか存在する [4, 1]。しかし、基本的にエイリアシングが使えなくなるため、言語の表現力が大きく制限されてしまうという問題がある。

Alias Type は、2 節で述べたように、線形型のアイデアを一歩進めて、エイリアシング情報を型情報として保持することで、メモリ領域の型を安全に変更できるようにし、メモリの再利用を可能とする手法である。しかし、Walker らの言語は、実行時にサイズが決まるような任意長の配列をサポートしていないため、柔軟なメモリ管理を記述するのは困難である [14]。これに対し我々の言語は、任意長の配列をサポートしているため、より柔軟なメモリ管理を記述することができる。

Hawblitzel らは、線形型や Alias Type を拡張し

て、柔軟なメモリ管理を実現しようとしている [6]。我々の手法と彼らの手法の共通点は、整数制約を導入している点である。異なるのは、彼らは、固定長のタプル型を用いて、実行時にしかサイズの分からない配列の型を表現しようとしている点である。このため、そのままでは実行時に型情報を参照する必要が生じてしまう。この問題を回避するため、彼らは、実行時に実行しなくても良い関数 (型情報のみを操作し、必ず終了する関数) を解析で検出する仕組みを提案している。例えば、彼らは配列の split を関数として定義し、実行時に実行しなくても良いことを示している。これに対し、我々は任意長の配列やその split 等をプリミティブとして扱っているため、そもそもこのような問題は生じない。また、彼らの言語は λ 計算をベースにしており、高階関数などが存在するため、メモリ管理を実装するためのローレベルな言語として適当であるかは必ずしも明らかでない。

Xi らは、依存型に基づいた型付きアセンブリ言語、Dependently Typed Assembly Language (DTAL) を提案している [15]。DTAL も整数制約を導入しているが、彼らの主目的は、配列の境界チェックを型付けすることで、メモリ領域の型の変更など、柔軟なメモリ管理を実現することはできない。

5.2 安全な OS

以降では、OS の安全性を検証するための関連研究について議論するが、現段階で、本研究は、メモリ安全性と制御フロー安全性という基礎的な安全性の保証を対象としているのに対し、本節で紹介する関連研究は、より高度な安全性を保証しようとしているため、比較対象としては不公平かもしれない、が敢えて比較を試みる。

モデル検査 [7] の手法を用いた安全性の検証は、通常、状態数が指数爆発するため、非常に小さなプログラムしか検証できなかったり、もしくは、近似や抽象化を行うため、検査の健全性が必ずしも明らかでない等の問題がある。例えば、Yang らはモデル検査の手法に基づいて Linux カーネルのファイルシステムのバグを発見しているが [16]、この検査の健全性は証明されていない。これに対し、我々の手法では (メモリ安全性や制御フロー安全性といった基礎的な安全性の保証に限定されるが) 健全性は保証される (ただし、型安全性の証明が完了すれば)。しかし、我々の手法では、基本的に OS を我々の言語で一から実装しなければならないという問題がある。このことから、モデル検査の手法は、既に存在している OS 実装のバグ検出器としては (不完全ではあるが) 有用であるといえる。

また、OS の実装がある性質を満たしていることを人手、もしくは定理証明支援器を用いて証明する手法がある [3, 2]。この手法の問題点は、まず一般の OS 開発者には敷居が高い点にある。また、OS の実装と安全性の証明を別々に行わなければならないため、OS の実装が変更される度に、それに合わせて証明をやり直さなければならない等、二度手間になる恐れがある。これに対し、我々のアプローチでは、安全性は型検査器が自動的に検証するので、プログラマは、ただプログラムを書くだけでよい。ただし、モデル検査手法との比較で既に述べたように、我々の手法では既存の OS 実装の安全性を検証することはできない。

6 まとめと今後の課題

本研究は、メモリ管理やスレッド管理などの OS の基本的な機能が、強く型付けされた言語で記述可能であることを示し、強く型付けされた OS の実現することを目指している。

本論文で述べたとおり、Alias Type のアイデアを元に、任意長の配列を扱えるように拡張した我々の言語を用いれば、メモリ管理やスレッド管理が記述可能となることが確認できた。

現状は、IA-32 を対象に実際に我々の言語を実装し、簡単な OS を実際に構築しているところであり、今後は、まず割込み機構の実現を目指す。基本的には、割込み処理命令列が任意のタイミングで実行されるものとして型システムを構築するだけで実現できるが、割込み処理命令列と通常の命令列とのデータの共有が難しくなるため、やはり割込み禁止フラグを型システムで扱う必要があると考えられる。また、型安全性の証明が未完であるので、これを完成することを目指す。

参考文献

[1] David Aspinall and Adriana Compagnoni, Heap Bounded Assembly Language. *Journal of Automated Reasoning*, 2003, vol.31, iss.3-4, pp.261-302(42). Kluwer Academic Publishers. Special issue on Proof-Carrying Code. 2004.

[2] Gustavo Betarte, Cristina Cornes, Nora Szasz and Alvaro Tasistro, Specification of a Smart Card Op-

erating System. In *Proc. TYPES 1999*, LNCS 1956, 2000, pp.77-93.

[3] W. R. Bevier, Kit: A Study in Operating System Verification. A verified operating system kernel. *IEEE Transactions on Software Engineering*, 15(11):1382-1396, 1989.

[4] James Cheney and Greg Morrisett, A Linearly Typed Assembly Language. Technical Report, Department of Computer Science, Cornell University, 2003.

[5] C#. http://msdn.microsoft.com/net/ecma/

[6] Chris Hawblitzel, Edward Wei, Heng Huang, Eric Krupski and Lea Wittie, Low-Level Linear Memory Management. In *Proc. SPACE 2004*.

[7] G. J. Holzmann, The model checker SPIN. *IEEE Transactions on Software Engineering*, 23(5):279-295, 1997.

[8] IA-32 Intel Architecture. http://developer.intel.com

[9] Java. http://java.sun.com/

[10] Greg Morrisett, David Walker, Karl Crary and Neal Glew, From System F to Typed Assembly Language. *ACM Transactions on Programming Languages and Systems*, 21(3):528-569, 1999.

[11] Objective Caml. http://caml.inria.fr/

[12] Greg Morrisett, Karl Crary, Neal Glew, Dan Grossman, Richard Samuels, Frederick Smith, David Walker, Stephanie Weirich and Steve Zdancewic, TALx86: A Realistic Typed Assembly Language. In *Proc. the 1999 ACM SIGPLAN Workshop on Compiler Support for System Software*.

[13] David N. Turner, Philip Wadler and Christian Mossion, Once upon a Type. In *ACM International Conference on Functional Programming and Computer Architecture*, 1995.

[14] David Walker and Greg Morrisett, Alias Types for Recursive Data Structures. In *Proc. Types in Compilation 2000*.

[15] Hongwei Xi and Robert Harper, A Dependently Typed Assembly Language. In *Proc. ICFP 2001*.

[16] Junfeng Yang, Paul Twohey, Dawson Engler and Madanlal Musuvathi, Using Model Checking to Find Serious File System Errors. In *Proc. OSDI 2004*.