

配列のためのホーア型理論の拡張

渡邊 裕貴, 前田 俊行, 米澤 明憲

ホーア型理論 (Hoare Type Theory) は、メモリの読み書きなどの副作用を持つ命令をモナドとして扱うことのできる、型付きラムダ計算の一種である。この型システムの特徴は、モナドに含まれる副作用のある命令に対してホーア論理に基づいた動作の検証を行う点である。これにより、ラムダ計算に基づく関数型プログラミングと副作用を利用する命令型プログラミングを組合せたプログラムの動作を検証することができるようになっていく。

ホーア型理論では副作用のある命令としてメモリ領域の確保と読み書きを扱えるが、従来の型システムでは書換えられる領域の数が型チェック時に固定されており、領域の数が動的に決定されるようなプログラムを検証することができなかった。本論文では、このようなプログラムの型チェックを可能とするために、アサーションにおけるメモリ領域の状態の表し方を、アドレスごとに個別の式を用いるような形に再定義する。新しい型システムでは、動的に数が決まる連続したメモリ領域を配列として扱えるようになる。

1 はじめに

関数型プログラミングと命令型プログラミングはどちらも重要なプログラミングパラダイムであるが、両者の性質は相異なっている。前者の本質は関数の評価であり、純粋な関数型プログラミング言語では副作用の概念は意図的に退けられている。副作用のないプログラムは互いの独立性を高め (参照透過性)、プロ

グラムの検証を容易にする。また関数型プログラミングで扱うことのできる高階関数 (引数や戻り値が関数であるような関数) は、より複雑なプログラムの構築を可能とする。一方命令型プログラミングの本質は副作用を伴う命令の逐次的実行である。副作用の存在はプログラムの参照透過性をなくし、検証を困難にする。しかし、現実のプログラミングにおいてファイルの読み書きやユーザに対する入出力などの副作用の扱いは完全には避けられないものである。

ホーア型理論 [8][7][6] は、関数型プログラミングにおける検証手法の一つである依存型付きラムダ計算 [1][2] に、命令型プログラミングの検証手法であるホーア論理 [3] を組込んだ型システムである。このシステムは、メモリ操作を扱う簡単な命令型プログラミング言語にラムダ計算を組合せた独自の対象言語に対して型チェックを行えるようになっていく。

この言語では、副作用のある命令としてメモリ領域の確保と読み込み・書き込みを扱うことができる。例えば、メモリアドレス p に値 0 を書き込んだあとユニット値を返す命令は以下のように書かれる。

$$[p]_{\text{nat}} := 0; ()$$

このような命令を直接ラムダ計算の中で扱うと参照透過性が失われてしまうため、副作用のある命令はラムダ計算においてはモナド [11][5][12] の中に閉じ込められた形でのみ出現する。モナドは自然数や真偽値と同様に値として扱われるため、モナドを関数の戻り値として返すこともできる。一つのアドレスを受け取り、そのアドレスに 0 を書き込んでユニット値を返す命令を表すモナドを返す関数は、以下のように書か

れる。

$$\lambda p:\text{nat}. \text{dia}([p]_{\text{nat}} := 0; ())$$

ただし、 dia は命令をモナド化する演算子である。

ホーア型理論の型システムではこのようなモナドに対して、いわゆるホーアトリプル¹の形をした型を与える。すなわち、モナド化された命令は、その命令が実行される前の事前条件と、命令が返す結果の型と、命令が実行された後の事後条件の三つからなる型を持つ。命令をこのように型付けすることによって、命令が不正なメモリアクセス（確保していないメモリ領域への読み書きなど）を行わないことを保証するとともに、命令がプログラマの望む挙動を確かに行うどうかを検証できるようになる。

命令の事前条件とはその命令が実行される直前に満たされるべき条件であり、事後条件とは（実行前に事前条件が確かに満たされていたという仮定の下で）命令が実行された直後に満たされる条件である。これらの条件は、メモリの状態を表す専用の述語を含む論理式によって表わされる。ホーア型理論では、各命令に対して与えられた事前・事後条件が妥当であるかどうか、型チェックの一環として検証される。なお、事前・事後条件を表す論理式は特にアサーションと呼ばれる。

例えば上述のモナド $\text{dia}([p]_{\text{nat}} := 0; ())$ は、以下のような型を持つ。

$$\{p \in \text{mem}\} u:\text{unit} \{ \text{hid}(\text{upd}_{\text{nat}}(\text{init}, p, 0), \text{mem}) \}$$

最初の $p \in \text{mem}$ は事前条件のアサーションで、命令が実行される前にアドレス p が予め確保されていないことを表している。次の $u:\text{unit}$ は、命令が返す結果がユニット値であることを表している。最後の $\text{hid}(\text{upd}_{\text{nat}}(\text{init}, p, 0), \text{mem})$ は事後条件のアサーションで、命令実行前のメモリ (init) に対してアドレス p に自然数 0 を書き込んだものが命令実行後のメモリ (mem) になっていることを表している。このように命令に対してアサーションを与えることで、命令の実行後に確かにアドレス p に値 0 が書き込まれていること（およびそれ以外のアドレスは書き換えられないこと）が検証できる。

さて、従来のホーア型理論では、メモリの書換えを表すために上の例のように upd という記号を用いて

いた。これはある状態のメモリのアドレス一つを書換えることで新たな状態のメモリを得る記号であり、複数のアドレスの書換えを表すには以下のように upd を入れ子にする必要があった。

$$\text{hid}(\text{upd}(\text{upd}(\text{init}, p, 0), p + 1, 1), \text{mem})$$

この書き方は、命令が書換えるアドレスの数が固定されている場合はその数だけ upd を入れ子にすればよいので問題ないが、書換えるアドレスの数が動的に決まる場合は型チェック時に必要な upd の数が分からないためアサーションを書き表すことができなかった。このため、連続する複数のアドレスを配列として確保して読み書きするようなプログラムへの応用は限られていた。

そこで本論文では、アサーションにおけるメモリ領域の状態を入れ子を用いずに表せる書き方に変更することで、このような配列を扱うプログラムの検証を可能にする。従来の upd を使った記法は廃止され、代わりにある時点でのあるアドレスの状態を表す記号 seleq が新たなプリミティブの一つとして定義される。この新しい定義を用いると、例えばアドレス p に値 0 が入っているということを表すアサーションは以下ようになる。

$$\text{seleq}_{\text{nat}}(\text{mem}, p, 0)$$

この式を全称量子化等で修飾することで、個数に変数で指定された複数の連続するアドレスの状態を例えば以下のように表すことができるようになる。

$$\forall i:\text{nat}. i < n \supset \text{seleq}_{\text{nat}}(\text{mem}, p + i, 0)$$

これにより、要素数が動的に与えられるような配列を初期化したりソートしたりするプログラムを型チェックすることができるようになる。

本論文のこれ以降の構成は以下のようになっている。まず第 2 節では本論文における新しいホーア型理論の対象言語と型システムの構文を説明する。続いて第 3 節で型システムの定義を全般的に説明し、第 4 節で型システムの持つ性質のうち重要なものを示す。第 5 節で対象言語の操作的意味論を定義し、型システムの健全性を示す。第 6 節で型チェックの決定可能性について触れたのち、第 7 節で配列を扱う具体的なプログラムの例を取り上げ、最後に第 8 節で本論文の内容をまとめる。

本論文で定義するホーア型理論は [8] での定義を基にしていくつかの構文や型付け規則を変更したものとなっており、型システムが持つ性質やその証明も多くは [8] のものと同様になっている。この型システムが持つ重要な性質の一つは健全性であるが、それを証明するための補題のうち [8] で未証明だったものは本論文でも未証明のまま残されている。この補題はホーア型理論の他の拡張 [7][6] では示されているため、本論文による拡張をこれらの拡張と組み合わせることで最終的に証明が可能になると考えられる。

2 対象言語と型システムの構文

本論文で定義するホーア型理論の対象言語と型システムの構文を図 1 に示す。これは [8] での定義にいくつかの変更を加えたものになっている。

ホーア型理論の項には、ユニット値 $()$ や真偽値、および自然数とそれに対する可算・乗算・比較演算が含まれている。[8] では自然数の比較演算は等号 (eq) のみであったが、配列の境界判定を容易にするため不等号演算子 (le, lt) が追加されている。ラムダ抽象化 $(\lambda x.M)$ と関数適用 (KM) の項は通常ラムダ計算と同じである。モナド化された命令は $\text{dia } E$ によって表わされる。項は [14] に倣って elim 項と intro 項の 2 種類に分類されており、 intro 項を elim 項として使用するにはキャスト $(M : A)$ によって型を明示的に与える必要がある。この分類は、項の型チェックの仕方の違いに関係している。項の中に現れる命令はすべてモナド化されるため、項の評価は副作用を伴わないことに注意されたい。

基本命令は、メモリ領域 (ホーア型理論では特にヒープと呼ばれる) の確保・読み込み・書き込みのほか、条件分岐とループのための構文を含んでいる。

- ヒープ確保の命令 $(x = \text{alloc } M)$ は、 M 個の連続したヒープ領域を確保し、先頭のアドレスに変数 x を束縛する。[8] ではこの命令は単に一つの領域を確保することしかできなかったが、本論文では複数の連続した領域を配列として確保できるように再定義されている。
- ヒープ読み込みの命令 $(x = [M]_A)$ は指定したアドレス M に入っている値に変数 x を束縛す

る。 A は値の型である。

- ヒープ書き込みの命令 $([M]_A = N)$ は指定されたアドレス M に項 N の評価結果の値を書き込む。値はその種類によらずアドレス一つ分のヒープに書き込まれる。
- 条件分岐命令 $(x = \text{if}_A(M, E_1, E_2))$ は、項 M の評価結果が true であるか false であるかによって E_1 と E_2 のどちらかを実行し、その結果に変数 x を束縛する。
- ループ命令 $(x = \text{loop}_A^I(M, y.N, y.F))$ は、 M の値をループカウンタの初期値としてループを実行する。ループは条件式 N が true に評価される間繰り返される。命令 F がループの内容であり、毎回の実行結果が次のループカウンタの値になる。 N と F に含まれる自由変数 y はこれらが評価・実行される際にループカウンタに置き換えられる。ループ終了時、最終的なループカウンタの値に変数 x が束縛される。 I はループ不変式であり、 A はループカウンタの型である。
- 不動点演算子命令 $(x = \text{fix}_A(f.y.F, M))$ は、命令 F に含まれる自由変数 f, y をそれぞれ関数 $\lambda f.\lambda y. \text{dia } F$ の不動点と項 M の評価結果に置き換えたうえで F を実行し、その結果に変数 x を束縛する。 A は不動点の型である。

ループ命令は条件分岐命令と不動点演算子命令の組み合わせで書き直せるので本質的には不要であるが、プログラムをより簡潔に書くことができるので定義に含められている。

基本命令以外の命令には、単に項の評価結果を返すもの (M) と命令の逐次実行 $(c; E)$ の他に、モナドから命令を取り出して実行する命令 $(\text{let dia } x = K \text{ in } E)$ がある。これはモナド K に含まれる命令を実行した結果に変数 x を束縛してから命令 E を実行するものである。

型には自然数・真偽値・ユニット値の型のほかに関数の型と命令モナドの型がある。関数の型 $(\Pi x:A.B)$ は依存型になっており、これは基本的には A から B への関数を意味するが、引数として渡される (型 A をもつ) 値を表す変数 x が型 B の中に出現できる点が通常関数の型と異なる。命令のモナドの型

| | |
|---------|---|
| Elim 項 | $K, L ::= x \mid KM \mid M : A$ |
| Intro 項 | $M, N, O ::= K \mid \lambda x.M \mid \text{dia } E \mid () \mid \text{true} \mid \text{false} \mid \text{zero} \mid \text{succ } M \mid$ $M + N \mid M \times N \mid \text{eq}(M, N) \mid \text{le}(M, N) \mid \text{lt}(M, N)$ |
| 基本命令 | $c ::= x = \text{alloc } M \mid x = [M]_A \mid [M]_A = N \mid$ $x = \text{if}_A(M, E_1, E_2) \mid x = \text{loop}_A^I(M, y.N, y.F) \mid x = \text{fix}_A(f.y.F, M)$ |
| 命令 | $E, F ::= M \mid \text{let dia } x = K \text{ in } E \mid c; E$ |
| 型 | $A, B, C ::= \alpha \mid \text{bool} \mid \text{nat} \mid \text{unit} \mid \Pi x:A.B \mid \{P\}x:A\{Q\}$ |
| 基本論理式 | $p ::= \top \mid \perp \mid \text{id}_A(M, N) \mid \text{seleq}_A(h, M, N)$ |
| 論理式 | $P, Q, R ::= p \mid P \wedge Q \mid P \vee Q \mid P \supset Q \mid \neg P \mid \forall x:A.P \mid \exists x:A.P \mid$ $\forall \alpha:\text{type}.P \mid \exists \alpha:\text{type}.P \mid \forall h:\text{heap}.P \mid \exists h:\text{heap}.P$ |
| 変数環境 | $\Delta ::= \cdot \mid \Delta, x:A \mid \Delta, \alpha$ |
| ヒープ環境 | $\Psi ::= \cdot \mid \Psi, h$ |
| 論理式環境 | $\Gamma ::= \cdot \mid \Gamma, P$ |
| 値 | $v, l ::= \lambda x.M \mid \text{dia } E \mid () \mid \text{true} \mid \text{false} \mid \text{zero} \mid \text{succ } v$ |
| ヒープ値 | $\chi ::= \cdot \mid \chi, l \mapsto_A v$ |
| 継続 | $\kappa ::= \cdot \mid x:A.E; \kappa$ |
| 制御式 | $\rho ::= \kappa \triangleright E$ |
| 抽象機械 | $\mu ::= \chi, \rho$ |

ただし、 h はヒープ変数を、 α は型変数を表す。

図 1 ホーア型理論の構文

$\{P\}x:A\{Q\}$ は、前節で述べたように、命令の事前条件 P 、実行結果の型 A 、事後条件 Q の三つからなる。これも依存型になっており、実行結果の値を表す変数 x が事後条件 Q の中に出現することができる。なお、ホーア型理論にはメモリポインタを表すための特別な型は存在せず、ポインタは単なる自然数として扱われる。

論理式は、命令の事前条件・事後条件を表すために用いられる。述語 $\text{id}_A(M, N)$ は二つの項 M, N が等しい値に評価されることを表す。また述語 $\text{seleq}_A(h, M, N)$ はヒープ h においてアドレス M に項 N の値が入っていることを表す。 A はこれらの項の型である。またここでのヒープ h はヒープ値ではなく、プログラム実行中のある時点におけるヒープを指し示すための変数（ヒープ変数）である。その他の論理式には、連言 (\wedge) や選言 (\vee) などの基本的な論理演算子のほかに、値・型・ヒープに対する全称量子子・存在量子子がある。型に対する量子子は後述の share を書き表せるようにするために [7] にならって本論文で新たに

加えられた。

元のホーア型理論 [8] では論理式の中でメモリの状態を表すための特別な式構造 (upd) が用いられていたが、本論文では [7] で加えられた述語 seleq のみでメモリに関する条件を表す論理式を記述する。またいくつかの糖衣構文を [7] に倣って図 2 に示す通り定義する。

式 $M \in h$ はヒープ h において M が有効なアドレスであることを表す。式 $\text{share}(h_1, h_2, M)$ は二つのヒープ h_1, h_2 のアドレス M の状態が同じであることを表す。式 $\text{hid}(h_1, h_2)$ はヒープ h_1, h_2 の状態が完全に同じであることを表す。

値・ヒープ値・継続・制御式・抽象機械は、プログラム実行の意味論を定義するために使われる。ヒープ値は実行時の具体的なヒープを表す、アドレスから値への有限関数である。継続は一つの値を受け取る命令 ($x:A.E$) の並びである。制御式は継続に命令を与えたもので、この命令の実行結果を継続に渡すことを意味する。抽象機械は制御式とヒープ値の組であり、そ

$$\begin{aligned}
M \in h & := \exists \alpha : \text{type}. \exists v : \alpha. \text{seleq}_\alpha(h, M, v) \\
M \notin h & := \neg(M \in h) \\
P \Leftrightarrow Q & := (P \supset Q) \wedge (Q \supset P) \\
\text{seleq}_A(h, M, -) & := \exists x : A. \text{seleq}_A(h, M, x) \\
\text{share}(h_1, h_2, M) & := \forall \alpha : \text{type}. \forall v : \alpha. \text{seleq}_\alpha(h_1, M, v) \Leftrightarrow \text{seleq}_\alpha(h_2, M, v) \\
\text{hid}(h_1, h_2) & := \forall x : \text{nat}. \text{share}(h_1, h_2, x) \\
M \leq N & := \exists x : \text{nat}. \text{id}_{\text{nat}}(M + x, N) \\
M < N & := \text{succ } M \leq N
\end{aligned}$$

図 2 論理式に対する糖衣構文

の制御式をそのヒープ値の下で実行することを表す。

3 型システムの定義

ここではホーア型理論の型システムについて説明する。紙面の都合上、型システムの形式的定義の全てをここに載せることはできないので重要な部分についてのみ取り上げる。型システムの完全な定義は [13] の付録に示されている。

ホーア型理論の型システムにはいくつかの判定カテゴリがある。変数環境・型・論理式の well-formedness に関する判定はそれぞれ以下の形をしている。

- $\vdash \Delta \text{ ctx}$
- $\Delta \vdash A \Leftarrow \text{type } [A']$
- $\Delta \vdash P \Leftarrow \text{prop } [P']$

ここで \vdash の後にある Δ, A, P がそれぞれ判定の対象となる変数環境・型・論理式である。 A', P' はそれぞれ A, P の正規形である。正規形については第 3.2 節で述べる。

Elim 項と intro 項に関する型チェックの判定はそれぞれ以下の形をしている。

- $\Delta \vdash K \Rightarrow A [M']$
- $\Delta \vdash M \Leftarrow A [M']$

これは変数環境 Δ の元で項 K, M が型 A を持つという判定を表す。Intro 項 M の型 A は型チェックの前に与えられるが、elim 項 K の型は型チェックを行う中で推論される。括弧内の M' は項の正規形である。

命令に関する型チェックの判定は以下の 2 種類がある。

- $\Delta; P \vdash E \Rightarrow x : A.Q [E']$
- $\Delta; P \vdash E \Leftarrow x : A.Q [E']$

これは変数環境 Δ と事前条件 P の下で、命令 E の実行結果の型が A になり、事後条件 Q を満たすという判定を表す。このとき、 Q に含まれる自由変数 x は実行結果の値を示す。二つ目の形式では事後条件 Q は型チェックの前に与えられるが、一つ目の形式では Q は型チェックを行う中で推論される。括弧内の E' は E の正規形である。

アサーションが正しいかどうかを判定するためには、以下の形のシーケントが用いられる。

- $\Delta; \Psi; \Gamma_1 \vdash \Gamma_2$

これは環境 Δ, Ψ の元で論理式 Γ_1 から Γ_2 が証明可能であるという判定を表す。

最後に、制御式と抽象機械に対する型付け規則の形を示す。

- $\Delta; P \vdash \kappa \triangleright E \Leftarrow x : A.Q$
- $\vdash \chi, \kappa \triangleright E \Leftarrow x : A.Q$

制御式と抽象機械の型付けについては第 5 節で述べる。

3.1 命令の型チェック

モナドの型 ($\{P\}x:A\{Q\}$) の事前条件 (P) と事後条件 (Q) の中では、init と mem という二つの特殊なヒープ変数が用いられている。事前条件 P の中で mem が用いられた場合、mem はその命令が実行される直前のヒープを表す。また事後条件 Q の中で init が用いられた場合も、命令が実行される直前のヒープを表す。そして Q の中で mem が用いられたときは命令

が実行された直後のヒープを表す。例えば、命令を実行する前と後とでアドレス M の内容が変わっていないということを表す事後条件は $\text{share}(\text{init}, \text{mem}, M)$ という論理式で表せる。

これらの特殊なヒープ変数は、複数の命令の事前・事後条件を組み合わせる際にも重要な役割を持つ。例えば二つの連続する命令があった時、前の命令の実行直後のメモリと後の命令の実行直前のメモリは一致するので、前の命令の事後条件 (Q_1 とする) に現れる mem と後の命令の事前条件 (P_2 とする) に現れる mem は同じメモリ状態を指すことになる。よって後の命令の事前条件が満たされているかを検証するには単にシーケント $Q_1 \vdash P_2$ が導出可能であることを確かめればよい。また、前の命令の事後条件に現れる mem と後の命令の事後条件に現れる init も同じメモリ状態を指すので、これらを同じ変数に置き換えて二つの事後条件を組み合わせると、二つの命令を合わせて一つの命令と見なしたときの事後条件が得られる。すなわち、二つの命令を合わせた全体の事後条件 $Q_1 \circ P_2$ は以下のようになる。

$$Q_1 \circ P_2 := \exists h:\text{heap}. [\text{mem} \mapsto h]Q_1 \wedge [\text{init} \mapsto h]P_2$$

さて、命令を閉じ込めたモナドの型チェックは、以下の型付け規則に従って、含まれている命令の型チェックに帰着される。

$$\frac{\Delta; \text{hid}(\text{init}, \text{mem}) \wedge P \vdash E \Leftarrow x:A.Q [E']}{\Delta \vdash \text{dia } E \Leftarrow \{P\}x:A\{Q\} [\text{dia } E']} \{\}I$$

命令の型チェックは、与えられた事前条件 (より正確には前の命令の事後条件) の下でその命令を実行したときに、与えられた型の結果が得られかつ与えられた事後条件が満たされていることをチェックすることで行われる。そのためには、与えられた事前条件と命令から推論された事後条件から、与えられた事後条件を導けるかどうかを、以下の型付け規則によりチェックする。

$$\Delta; P \vdash E \Rightarrow x:A.R [E']$$

$$\frac{\Delta, x:A; \text{init}, \text{mem}; R \vdash Q}{\Delta; P \vdash E \Leftarrow x:A.Q [E']} \text{-consequent}$$

個々の命令に対する型チェックは全て $\Delta; P \vdash E \Rightarrow x:A.Q [E']$ の形式になっており、各命令のチェックにおいてその命令に対する (最強の) 事後条件が推論される。そしてこの推論された事後条件から

与えられた事後条件を導けることを、シーケント $\Delta, x:A; \text{init}, \text{mem}; R \vdash Q$ を導出することによって検証するのである。

命令の型チェックで推論される事後条件は、型付け規則によって決められている。例えばアドレス M の内容を項 N の評価結果に変更する命令 $[M]_A = N$ の事後条件は以下のように定義されている。

$$\text{seleq}_A(\text{mem}, M, N) \wedge$$

$$\forall n:\text{nat}. \neg \text{id}_{\text{nat}}(n, M) \supset \text{share}(\text{init}, \text{mem}, n)$$

これは、命令が実行された後にはアドレス M に項 N の評価結果が入っているととも、 M 以外のアドレスについてはメモリの内容は変わっていないことを示している。

本論文で再定義された alloc 命令の型付け規則は図 3 のように定義されている。 $Q_{\text{alloc}}(x, M)$ は alloc 命令自身の事後条件を表す。 alloc 命令に与えられる事前条件 P と alloc 命令自身の事後条件 $Q_{\text{alloc}}(x, M)$ の合成が、 alloc 命令の次に実行される命令 E の事前条件となる。この合成された事前条件の下で E が型を持つならば、 alloc 命令と E を合わせた命令 ($x = \text{alloc } M; E$) も型を持つというのがこの規則の解釈である。また E の事後条件として Q が推論された時、 ($x = \text{alloc } M; E$) の事後条件として $\exists x:\text{nat}. Q$ が推論される。

3.2 正規形と相続置換

いくつかの型チェックの判定を行う中で、項や型の正規形が生成される。項の正規形とは、その項及びそれに含まれる全ての部分項について β 簡約と η 展開が完全に行われた状態の項である。

アサーションが正しいかどうかをチェックする中で、複数の項が等しい値に評価されるかどうかをチェックすることなどが必要となるが、項の等価性の判定は一般に容易ではない。そこで項同士を直接比較するのではなく項の正規形を比較することで、項の等価性を判定する。

例えば二つの項 $\text{succ zero} \times \text{succ zero}$ と $\text{succ zero} + \text{zero}$ はどちらも succ zero という正規形を持つため、等しい項と見なされる。

型や論理式についても、それに含まれている項を正

$$\frac{\Delta \vdash M \Leftarrow \text{nat} [M'] \quad \Delta, x:\text{nat}; P \circ Q_{\text{alloc}}(x, M) \vdash E \Rightarrow y:B.Q [E']}{\Delta; P \vdash (x = \text{alloc } M; E) \Rightarrow y:B.(\exists x:\text{nat}.Q) [x = \text{alloc } M'; E']}$$

ただし

$$Q_{\text{alloc}}(x, M) = \forall n:\text{nat}.(n < x \vee x + M \leq n \supset \text{share}(\text{init}, \text{mem}, n)) \wedge (x \leq n \wedge n < x + M \supset n \notin \text{init} \wedge \text{seleq}_{\text{unit}}(\text{mem}, n, ()))$$

図 3 alloc 命令の型付け規則

規化することによって、型や論理式の正規形を考えることができる。

項を正規化するためには、簡単に言えば正規形が得られるまで項を β 簡約・ η 展開し続ければよい。 β 簡約を繰り返し行うために、ホーア型理論では相続置換 (hereditary substitution) という特殊な置換方法を用いる。この置換法が通常の変数の置換と異なるのは、置換によって新たに β 簡約可能な項ができた場合、それは自動的に簡約される点である。これにより、相続置換の結果得られる項はそれ以上 β 簡約できない形になっており、これを η 展開したものは正規形になっていることが保証される。

しかし β 簡約を繰り返すことは任意の項に対して終了するとは限らない (無限に β 簡約し続けられる項も存在する)。よって、型チェックの中で出てくるすべての項を正規化していたのでは、型チェックが停止しない可能性があるため、後述の決定可能性を保証できない。

そこで相続置換は、最終的に正規形が得られることが保証できる項に対してしか定義されない。相続置換を行う際に、置換される項の型の大きさに基づいて β 簡約の停止性がチェックされる。停止性を保証できない場合は、型エラーと見なされて型チェックは中止される。

通常の変数の置換は $[x \mapsto N]M$ という式で表わされるのに対し、相続置換は $[x \mapsto N]_A(M)$ という式で表される (項 M に含まれる自由変数 x を項 N に置換したものを意味する)。ここで A は N の型であり、相続置換はこの型に基づいて β 簡約の停止性をチェックする。相続置換の中で β 簡約可能な項が出現した際、その新たな項の型が現在置換を行っている項の型に含まれている場合のみ簡約が停止すると判断し、簡約を繰り返すのである。こうすると、簡約

を繰り返す中で項の型の構文的構造が必ず小さくなってゆくので、簡約の停止性を保証できる。

相続置換の具体的な定義は [13] を見られたい。

3.3 モナド置換

項の評価において β 簡約は変数の置換によって行われるが、命令の実行においても命令に含まれる変数を別な命令の実行結果で置換する操作が必要となる。ホーア型理論ではこのような命令に対する置換をモナド置換と呼んでいる。命令 F の中に現れる型 A の自由変数 x を命令 E の実行結果で置換するモナド置換は $\langle x:A \mapsto E \rangle F$ で表される。モナド置換は [8] と同様に図 4 のように定義される。

モナド置換は最終的に通常の変数の置換に帰着するが、モナド置換の中で行われる変数の置換を相続置換で行う形式も考えることができる。このようなモナド相続置換は $\langle x \mapsto E \rangle_A(F)$ で表される。モナド相続置換の定義は省略する。

3.4 アサーション論理

型システムの中で、アサーションが正しいかどうかを検証する部分は特にアサーション論理という。ホーア型理論のアサーション論理はシーケントの導出可能性判定によっている。ここで定義されるシーケントの導出規則は一般的なシーケント計算に倣っているが、ホーア型理論にはさらに `bool` や `nat` などの型の値に対する規則がある [8]。これらの規則には数学的帰納法を正当化する規則 (図 5) も含まれており、これによって自然数の比較による配列の境界判定などのアサーションの検証が可能になる。

本論文ではヒープ上の一つのアドレスの状態を表す述語 `seleq` を加えたため、それに関する導出規則も定義し直されている。述語 `seleq` に関する導出規則は

$$\begin{aligned}
\langle x:A \mapsto M \rangle F & := [x \mapsto M : A]F \\
\langle x:A \mapsto \text{let dia } y = K \text{ in } E \rangle F & := \text{let dia } y = K \text{ in } \langle x:A \mapsto E \rangle F \\
\langle x:A \mapsto c; E \rangle F & := c; \langle x:A \mapsto E \rangle F
\end{aligned}$$

図 4 モナド置換の定義

$$\frac{\Delta \vdash M \leftarrow \text{nat} [M] \quad \Delta; \Psi; \Gamma_1, P \vdash [x \mapsto \text{succ } x]_{\text{nat}}(P), \Gamma_2}{\Delta; \Psi; \Gamma_1, [x \mapsto \text{zero}]_{\text{nat}}(P) \vdash [x \mapsto M]_{\text{nat}}(P), \Gamma_2}$$

図 5 数学的帰納法を正当化する導出規則

$$\overline{\Delta; \Psi; \Gamma_1, \text{seleq}_A(h, M, N_1), \text{seleq}_A(h, M, N_2) \vdash \text{id}_A(N_1, N_2), \Gamma_2}$$

図 6 述語 seleq に関する導出規則

図 6 のとおりである。この規則は一つのアドレスには高々一つの値しか入らないことを保証する。

4 型システムの性質

本論文で拡張したホーア型理論も、元のホーア型理論 [8] と同様の性質を持つ。ここでは、後述の定理を証明するための補題をいくつか取り上げる。

まず一つ目はアサーションの取り換えの補題である。

補題 1 (アサーションの取り換え) 命令 E の型付け $\Delta; P \vdash E \leftarrow x:A.Q [E']$ が得られている時、

1. $\Delta, x:A; \text{init}, \text{mem}; Q \vdash R$ ならば
 $\Delta; P \vdash E \leftarrow x:A.R [E']$
2. $\Delta; \text{init}, \text{mem}; R \vdash P$ ならば
 $\Delta; R \vdash E \leftarrow x:A.Q [E']$
3. $\Delta; \text{init}, \text{mem} \vdash R \leftarrow \text{prop} [R]$ ならば
 $\Delta; R \circ P \vdash E \leftarrow x:A.(R \circ Q) [E']$

また型システムの健全性を示すために必要となる代入補題 (substitution lemma; 図 7) も成り立つ。

配列の境界判定に関するアサーションを検証するには自然数の大小比較などの基本的な算術に関する公式がアサーション論理の中で扱えることが必要である。これを保証するのが以下の定理である。(これ以降、述語 id に付する型などは適宜省略する)

定理 3 (基本的な算術公式) 以下の形のシーケントはアサーション論理で導出可能である。

1. $\Delta; \Psi; \Gamma_1, \text{id}(M_1 + N, M_2 + N) \vdash \text{id}(M_1, M_2), \Gamma_2$
2. $\Delta; \Psi; \Gamma_1 \vdash \text{id}(M + N, N + M), \Gamma_2$
3. $\Delta; \Psi; \Gamma_1 \vdash \text{id}((M_1 + M_2) \times N, M_1 \times N + M_2 \times N), \Gamma_2$
4. $\Delta; \Psi; \Gamma_1, M \leq N, N \leq M \vdash \text{id}(M, N), \Gamma_2$
5. $\Delta; \Psi; \Gamma_1 \vdash M \leq N, N \leq M, \Gamma_2$
6. $\Delta; \Psi; \Gamma_1, M \leq N, N < M \vdash \Gamma_2$
7. $\Delta; \Psi; \Gamma_1, \text{id}(\text{eq}(M, N), \text{true}) \vdash \text{id}(M, N), \Gamma_2$

これらのシーケントの導出には、図 5 の数学的帰納法に関する導出規則を用いる。例えばシーケント 1 は、導出すべきシーケントをカット則により $\Delta; \Psi; \Gamma_1, \text{id}(M_1, M_2) \supset \text{id}(M_1, M_2) \vdash \text{id}(M_1 + N, M_2 + N) \supset \text{id}(M_1, M_2), \Gamma_2$ に置き換えた後、 $P = \text{id}(M_1 + x, M_2 + x) \supset \text{id}(M_1, M_2)$ として図 5 の規則を適用する。すると導出すべきシーケントは以下の二つになる。

- $\Delta; \Psi; \Gamma_1 \vdash \text{id}(M_1, M_2) \supset \text{id}(M_1, M_2), \Gamma_2$
- $\Delta; \Psi; \Gamma_1, \text{id}(M_1 + n, M_2 + n) \supset \text{id}(M_1, M_2) \vdash \text{id}(\text{succ}(M_1 + n), \text{succ}(M_2 + n)) \supset \text{id}(M_1, M_2), \Gamma_2$

これらは他の基本的な導出規則より導出可能である。

またシーケント 7 の導出には二重の帰納法が必要である。論理式 $\forall n:\text{nat}. \text{id}_{\text{bool}}(\text{eq}(m, n), \text{true}) \supset$

補題 2 (一般代入補題) $\Delta \vdash A \Leftarrow \text{type } [A']$ と $\Delta \vdash M \Leftarrow A' [M']$ が満たされているとき、

1. $\Delta, x:A', \Delta_1 \vdash K \Rightarrow B [N']$ ならば
 $\Delta, [x \mapsto M']_A(\Delta_1) \vdash [x \mapsto M : A]K \Rightarrow [x \mapsto M']_A(B) [[x \mapsto M']_A(N')]$
2. $\Delta, x:A', \Delta_1 \vdash N \Rightarrow B [N']$ ならば
 $\Delta, [x \mapsto M']_A(\Delta_1) \vdash [x \mapsto M : A]N \Rightarrow [x \mapsto M']_A(B) [[x \mapsto M']_A(N')]$
3. $\Delta, x:A', \Delta_1; P \vdash E \Leftarrow y:B.Q [E']$ かつ y が M の自由変数でないならば $\Delta, [x \mapsto M']_A(\Delta_1); [x \mapsto M']_A(P) \vdash [x \mapsto M : A]E \Leftarrow y:[x \mapsto M']_A(B).[x \mapsto M']_A(Q) [[x \mapsto M']_A(E')]$
4. $\Delta, x:A', \Delta_1; \vdash B \Leftarrow \text{type } [B']$ ならば $\Delta, [x \mapsto M']_A(\Delta_1) \vdash [x \mapsto M : A]B \Leftarrow \text{type } [[x \mapsto M']_A(B')]$
5. $\Delta, x:A', \Delta_1; \Psi \vdash P \Leftarrow \text{prop } [P']$ ならば
 $\Delta, [x \mapsto M']_A(\Delta_1); \Psi \vdash [x \mapsto M : A]P \Leftarrow \text{prop } [[x \mapsto M']_A(P')]$
6. $\Delta; P \vdash E \Leftarrow x:A'.Q [E']$ かつ $\Delta, x:A'; Q \vdash F \Leftarrow y:B.R [F']$ (ただし x は B と R の自由変数でない) ならば $\Delta; P \vdash \langle x:A \mapsto E \rangle F \Leftarrow y:B.R [\langle x \mapsto E' \rangle_A (F')]$

図 7 一般代入補題

$\text{id}(m, n)$ において、まず m に関する帰納法を適用し、続いて n に関する帰納法を用いる。

最も導出が複雑なのはシーケント 5 である。まず論理式 $\exists p:\text{nat}. \text{id}(m+p, N) \vee \exists q:\text{nat}. \text{id}(N+q, m)$ において m に関する帰納法を用いる。 $m = \text{zero}$ の場合を示すのは容易である。 $m = k$ の場合から $m = \text{succ } k$ の場合を示すには、以下の二つのシーケントを導かなくてはならない。

- $\exists p:\text{nat}. \text{id}(m+p, N) \vdash \exists r:\text{nat}. \text{id}(\text{succ}(m+r), N) \vee \exists s:\text{nat}. \text{id}(N+s, \text{succ } m)$
- $\exists q:\text{nat}. \text{id}(N+q, m) \vdash \exists r:\text{nat}. \text{id}(\text{succ}(m+r), N) \vee \exists s:\text{nat}. \text{id}(N+s, \text{succ } m)$

後者は、 q をインスタンス化した後 $\text{succ } q$ を s の具体値として与えればよい。前者は p に関する場合分けを要する。具体値は、 $p = \text{zero}$ の場合は $s = \text{succ } \text{zero}$ 、 $p > \text{zero}$ の場合は $\text{succ } r = p$ となる。この場合分けを実際に行うには p に関する帰納法を用いる。

5 対象言語の意味論と型システムの健全性

ホーア型理論の対象言語の実行時の状態は制御式・抽象機械を用いて表される。そして対象言語の意味論(評価規則)は抽象機械の遷移規則として定義される。ここでは本論文で再定義された alloc 命令の遷移規則を図 8 に示す。他の遷移規則の定義については [13] を参照されたい。

型システムの健全性を論ずるにはこれらに対する型付けを考える必要がある。制御式の型付け規則は以下の形をしている。

$$\Delta; P \vdash \kappa \triangleright E \Leftarrow x:A.Q$$

これは環境 Δ と事前条件 P の下で制御式 $\kappa \triangleright E$ の実行結果が型 A を持ち、事後条件 Q を満たすことを表す。また抽象機械の型付け規則の形は以下のようになっている。

$$\vdash \chi, \kappa \triangleright E \Leftarrow x:A.Q$$

これら二つの規則を関係付ける唯一の型付け規則は
$$\frac{\vdash \chi, \kappa \triangleright E \Leftarrow x:A.Q}{\vdash [\chi] \vdash \kappa \triangleright E \Leftarrow x:A.Q}$$
 である。ただし、 $[\chi]$ はヒープ値 χ を対応する論理式に変換する記号であり、以下のように定義される。

$$[\cdot] := \top$$

$$[\chi, l \mapsto_A v] := [\chi] \wedge \text{seleq}_A(\text{mem}, l, v)$$

ホーア型理論の健全性は、抽象機械の遷移に関する二つの定理 (preservation と progress) によって示される。

補題 4 (制御式の置き換え) 制御式の型付けに関して以下が成り立つ。

1. $\Delta; P \vdash \kappa \triangleright E \Leftarrow x:A.Q$ ならば、 $\Delta; P \vdash E \Leftarrow y:B.R [E']$ を満たす y, B, R, E' が存在する。さらに $\Delta_1 \supset \Delta$ なる $\Delta_1, P_1, \kappa_1, E_1$ が $\Delta_1; P_1 \vdash \kappa_1 \triangleright E_1 \Leftarrow y:B.R$ を満たすならば

$$\frac{M \hookrightarrow_m M'}{\chi, \kappa \triangleright (x = \text{alloc } M; E) \hookrightarrow_e \chi, \kappa \triangleright (x = \text{alloc } M'; E)}$$

$$\frac{\{i \mid l \leq i < l + v\} \cap \text{dom}(\chi) = \emptyset}{\chi, \kappa \triangleright (x = \text{alloc } v; E) \hookrightarrow_e (\chi, l \mapsto_{\text{unit}} (), \dots, l + v - 1 \mapsto_{\text{unit}} ()), \kappa \triangleright [x \mapsto l : \text{nat}]E}$$

図 8 alloc の遷移規則

$\Delta_1; P_1 \vdash (\kappa_1; \kappa) \triangleright E_1 \Leftarrow x:A.Q$ である。

2. $\Delta; P \vdash (y:B.F; \kappa) \triangleright E \Leftarrow x:A.Q$ ならば $\Delta; P \vdash \kappa \triangleright \langle y:B \mapsto E \rangle F \Leftarrow x:A.Q$ である。

証明は κ の構造に関する帰納法による。

定理 5 (Preservation) 型を持つ項・制御式は遷移後も同じ型を持つ。すなわち、

1. $K_0 \hookrightarrow_k K_1$ かつ $\vdash K_0 \Rightarrow A [N']$ ならば $\vdash K_1 \Rightarrow A [N']$
2. $M_0 \hookrightarrow_m M_1$ かつ $\vdash M_0 \Leftarrow A [M']$ ならば $\vdash M_1 \Leftarrow A [M']$
3. $\mu_0 \hookrightarrow_e \mu_1$ かつ $\vdash \mu_0 \Leftarrow x:A.Q$ ならば $\vdash \mu_1 \Leftarrow x:A.Q$

項に関する preservation は遷移規則の導出に関する帰納法で示される。制御式に関する preservation は型付け規則の導出に関する帰納法で示される。ここでは $\mu_0 = \chi_0, \kappa_0 \triangleright (y = \text{alloc } v; E)$ の場合のみを示す。この場合は、遷移後の制御式は $\mu_1 = (\chi, l \mapsto (), \dots, l + v - 1 \mapsto ()), \kappa_0 \triangleright [x \mapsto l : \text{nat}]E$ と置ける。 μ_0 の型付けに関する仮定より、 $\cdot; [\chi_0] \vdash \kappa_0 \triangleright (y = \text{alloc } v; E) \Leftarrow x:A.Q$ を得る。補題 4 より、 $\cdot; [\chi_0] \vdash (y = \text{alloc } v; E) \Leftarrow z:C.S [E']$ なる z, C, S, E' が存在し、更に alloc の型付け規則 (図 3) より $y:\text{nat}; [\chi_0] \circ Q_{\text{alloc}}(y, v) \vdash E \Leftarrow z:C.S [E']$ を得る。補題 2 より $\cdot; [\chi_0] \circ Q_{\text{alloc}}(l, v) \vdash [y \mapsto l : \text{nat}]E \Leftarrow z:C.S [E']$ を得、更にシーケント $\cdot; \text{mem}; [\chi_0, l \mapsto (), \dots, l + v - 1 \mapsto ()] \vdash [\chi_0] \circ Q_{\text{alloc}}(l, v)$ が導出可能である (導出の仕方は省略) ことから、補題 1 より $\cdot; [\chi_0, l \mapsto (), \dots, l + v - 1 \mapsto ()] \vdash [y \mapsto l : \text{nat}]E \Leftarrow z:C.S [E']$ を得る。制御式の型付け規則より $\cdot; [\chi_0, l \mapsto (), \dots, l + v - 1 \mapsto ()] \vdash \cdot \triangleright [y \mapsto l : \text{nat}]E \Leftarrow z:C.S$ となり、補題 4 より $\cdot; [\chi_0, l \mapsto (), \dots, l + v - 1 \mapsto ()] \vdash \kappa_0 \triangleright [y \mapsto l : \text{nat}]E \Leftarrow x:A.Q$ となるので、抽象機械の型付け規則より $\vdash \mu_1 \Leftarrow x:A.Q$ が導かれる。

(証明終)

続いて progress 定理を示すために重要な命題がヒープ健全性である。この命題はアサーション論理を構成しているシーケント計算の健全性を示す。

予想 6 (ヒープ健全性) 任意のヒープ χ と自然数 l について

1. $\cdot; \text{mem}; [\chi] \vdash \text{seleq}_A(\text{mem}, l, -)$ ならば $l \mapsto_A v \in \chi$ なる値 v が存在する。
2. $\cdot; \text{mem}; [\chi] \vdash l \in \text{mem}$ ならば $l \mapsto_A v \in \chi$ なる型 A と値 v が存在する。

ヒープ健全性は本論文の依拠する [8] では証明されていないため、本論文でも未証明の予想とする。シーケント計算の健全性は一般にはカット除去定理から導かれるが、ホーア型理論のシーケント計算はヒープの状態を表す述語や自然数の数学的帰納法を正当化する規則などを含んでいるため、通常の方法でカット除去定理を示すのは難しいと考えられる。

なお、ヒープ健全性はホーア型理論の他の拡張では領域理論を用いて証明されている [7][6]。これらの拡張におけるヒープ健全性の証明を本論文の拡張に直接適用することは、アサーション論理におけるヒープの扱いの違い故に難しい。しかしこれらの拡張を基にして本論文での拡張を組み合わせることでヒープ健全性が証明されたホーア型理論において配列を扱うことも可能になると考えられる。

最後に progress 定理を示す。

定理 7 (Progress)(ヒープ健全性が成り立つという仮定の下で) 型を持つ項・抽象機械は “stuck” していない。すなわち、

1. $\vdash K_0 \Rightarrow A [N']$ ならば、 $K_0 = v : A$ なる v, A が存在するか、 $K_0 \hookrightarrow_k K_1$ なる K_1 が存在する。
2. $\vdash M_0 \Rightarrow A [M']$ ならば、 $M_0 = v$ なる v が存在するか、 $M_0 \hookrightarrow_m M_1$ なる M_1 が存在する。

3. $\vdash \chi_0, \kappa_0 \triangleright E_0 \Leftarrow x:A.Q$ ならば、 $\kappa_0 \triangleright E_0 = \cdot \triangleright v$ なる v が存在するか、 $\chi_0, \kappa_0 \triangleright E_0 \hookrightarrow_e \chi_1, \kappa_1 \triangleright E_1$ なる χ_1, κ_1, E_1 が存在する。

証明は K_0, M_0, E_0 の構造に関する場合分けによる。ここでは $E_0 = (x = \text{alloc } M; E)$ の場合のみ示す。この場合、 M に progress を適用すると、 M は値 v であるか、または $M \hookrightarrow_m M'$ となる M' が存在する。後者の場合は、 $E_1 = (x = \text{alloc } M'; E)$ とすると遷移規則 (図 3) より明らかに $\chi_0, \kappa_0 \triangleright E_0 \hookrightarrow_e \chi_0, \kappa_0 \triangleright E_1$ となる。前者の場合は χ に含まれる全てのアドレスの最大値より 1 大きい自然数を l とすると、 $\{i \mid l \leq i < l + v\} \cap \text{dom}(\chi_0) = \emptyset$ であるから、 $\chi_1 = \chi_0, l \mapsto (), \dots, l + v - 1 \mapsto ()$ および $\kappa_0 = \kappa_1$ および $E_1 = [x \mapsto l : \text{nat}]E$ とすると遷移規則より $\chi_0, \kappa_0 \triangleright E_0 \hookrightarrow_e \chi_1, \kappa_1, \triangleright E_1$ となる。(証明終)

6 型チェックの決定可能性

型システムのもう一つの重要な定理は、型チェックの決定可能性である。

定理 8 (部分的決定可能性) シークエントの導出可能性判定が決定可能であるという仮定の下で、ホーア型理論の型チェックは決定可能である。

アサーション論理で用いられているシークエント計算は一階述語論理に基づいているため、シークエントの導出可能性判定は一般には決定不能である。そのため、型チェックの決定可能性はアサーション論理を除いた部分に関してのみ示される。実際の型チェッカーの実装においては、シークエントの導出判定の部分に関してはある程度人手を用いる定理証明支援系の利用などが想定される。

7 プログラムの例

本論文で拡張したホーア型理論で扱えるプログラムの例として、自然数の配列の全ての要素をインクリメントするプログラムを図 9 に挙げる。この例で定義される `incarray` 関数は、配列の先頭アドレスへのポインタと配列の要素数 (どちらも `nat` 型) を引数として受け取り、命令のモナドを返す。このモナドの命令が実際に実行されると、関数の引数で指定された配

列の要素がインクリメントされる。

このモナドの型の事前条件は、この命令が実行される前に指定された配列がヒープに既に確保されていることを要求している。すなわち、事前条件の論理式はアドレス p から始まる n 個のヒープ領域に自然数の値が入っていることを全称量子化や不等号を用いて表している。ここで、 p と n は関数の引数として受け取った自然数を表す変数であり、関数を表す依存型 ($\Pi p:\text{nat}.\Pi n:\text{nat}.$) によって導入される。また事後条件は、命令の実行後にこの配列の各要素がインクリメントされていることと、配列以外のヒープ領域の内容は変化しないことを表している。これも、全称量子化や存在量子化を組合せることで表すことができる。

拡張前のホーア型理論では、第 1 節で触れたように、複数のアドレスの書き換えには `upd` という記号を書き換えるアドレスの数だけ入れ子にしたものを事後条件の中で用いていたが、その書き方では書き換えるアドレスの数 (の最大値) が予め分かっている場合しか表せなかった。本論文による拡張では、図 9 にあるように、述語 `seleq` と全称量子化等の組み合わせによって入れ子を用いずに複数のアドレスの書き換えを表すことができる。書き換えられるアドレスの範囲は不等号を用いて指定することができるので、型チェック時に書き換えるアドレスの個数が具体的に分からなくとも、アドレスの個数を項として表すことさえできればよい。

実際に配列をインクリメントする命令は、一つのループによって構成されている。ループカウンタ m は 0 から始まり、ループが一周するごとに 1 ずつ増える。 m が n より小さい間ループは実行され、ループ本体の中で m 個目の要素がインクリメントされる。ループが終わるとダミーの結果としてユニット値を返してこの命令は終了する。

ループの中に現れる I はループ不変式である。これはループカウンタの値が m から m' に変化した際に成り立つ事後条件を表している。この例では、 m' が常に n 以下であること、配列はずっとヒープ上に確保されていること、 m 個目から m' 個目までの要素がインクリメントされること、そしてそれ以外のヒープ領域は変化しないことを表している。

$$\begin{aligned}
& \text{incarray} : \prod p:\text{nat}.\prod n:\text{nat}. \\
& \quad \{\forall k:\text{nat}.k < n \supset \text{seleq}_{\text{nat}}(\text{mem}, p+k, -)\} \\
& \quad \text{dummy}:\text{unit} \\
& \quad \{\forall k:\text{nat}. (k < n \supset \exists v:\text{nat}.\text{seleq}_{\text{nat}}(\text{init}, p+k, v) \wedge \text{seleq}_{\text{nat}}(\text{mem}, p+k, \text{succ } v)) \wedge \\
& \quad \quad (k < p \vee p+n \leq k \supset \text{share}(\text{init}, \text{mem}, k))\} \\
& = \lambda p.\lambda n.\text{dia}(\\
& \quad m' = \text{loop}_{\text{nat}}^I(\text{zero}, \\
& \quad \quad m.\text{lt}(m, n), \\
& \quad \quad m.v = [p+m]_{\text{nat}}; [p+m]_{\text{nat}} = \text{succ } v; \text{succ } m); \\
& \quad () \\
&)
\end{aligned}$$

ただし

$$\begin{aligned}
I & = m' \leq n \wedge \forall k:\text{nat}. \\
& \quad (k < n \supset \text{seleq}_{\text{nat}}(\text{mem}, p+k, -)) \wedge \\
& \quad (m \leq k \wedge k < m' \supset \exists v:\text{nat}.\text{seleq}_{\text{nat}}(\text{init}, p+k, v) \wedge \text{seleq}_{\text{nat}}(\text{mem}, p+k, \text{succ } v)) \wedge \\
& \quad (k < p+m \vee p+m' \leq k \supset \text{share}(\text{init}, \text{mem}, k))
\end{aligned}$$

図 9 配列の要素をインクリメントする例

このプログラムに対する型チェックでは、モナドに含まれる命令の動作が与えられた事前・事後条件を満たしているかどうかを検証される。そのためには、型システムの導出規則の定義に従っていくつかのシーケントを導出する必要がある。この例では、以下の事実を表すシーケントの導出が必要となる。

- モナドの型で与えられた命令全体の事前条件が、ループの事前条件 (ループ不変式 I において $m = m' = \text{zero}, \text{init} = \text{mem}$ としたものを満たしていること
- $[m \mapsto m_0, m' \mapsto m_1]I$ と $[m \mapsto m_1, m' \mapsto m_2]I$ を合成すると $[m \mapsto m_0, m' \mapsto m_2]I$ が得られること
- I から $[m \mapsto m', \text{init} \mapsto \text{mem}]I$ が導けること
- I が成り立っている元でループの内容を実行した後もやはり I が成り立っていること
- ループの事後条件 (I にループ終了時のループカウンタを代入したもの) が命令全体の事後条件を満たしていること

これらのシーケントの具体的な式及び導出は紙面の都合上省略する。

8 関連研究とまとめ

本論文では、ホーア型理論のアサーション論理におけるヒープの状態の表し方を変更することで、連続するヒープ領域を配列として扱うプログラムの型チェックを可能にした。連続するヒープ領域に関するアサーションは、プログラム実行時のある時点における一つのアドレスの状態を表す述語 seleq に全称量子化や不等号を組み合わせて配列のアドレス範囲を指定することで行われる。拡張されたホーア型理論は従来と同様に部分的決定可能性と健全性を持つ。ただしメモリに関するアサーションを正当化するヒープ健全性は未証明のままである。

ホーア型理論に関する他の拡張には、[7] や [6] がある。前者は Separation Logic [10][9] に基づいてホーア型理論を再定式化するとともに、多相型を導入した。後者は Extended Calculus of Constructions [4] をホーア型理論の型システムに取り入れ、より抽象化された形で論理式を扱えるようにした。これらの研究は主に項の型の柔軟性の拡張を目的としており、ヒープ上のデータ構造の拡張を目的とする本論文とは方向性が異なっている。しかしこれらの研究を本論

文の内容と組み合わせることで、配列を扱うアサーションの論理式をより簡潔に表すことができるとともにリンクリストなどの配列以外のヒープ上のデータ構造も扱えるようになると考えられる。またこれらの拡張ではヒープ健全性の証明がなされているので、これらとの組み合わせにより健全性が完全に証明されたホア型理論で配列を扱うこともできるようになると考えられる。

参考文献

- [1] T. Coquand and G. Huet. The calculus of constructions. *Information and Computation*, Vol. 76, No. 2-3, pp. 95-120, 1988.
- [2] R. Harper, F. Honsell, and G. Plotkin. A framework for defining logics. *Journal of the ACM*, Vol. 40, No. 1, pp. 143-184, 1993.
- [3] C.A.R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, Vol. 12, No. 10, pp. 576-585, 1969.
- [4] Z. Luo. *An Extended Calculus of Constructions*. PhD thesis, University of Edinburgh, 1990.
- [5] E. Moggi. Notions of computation and monads. *Information and Computation*, Vol. 93, No. 1, pp. 55-92, 1991.
- [6] A. Nanevski, A. Ahmed, G. Morrisett, and L. Birkedal. Abstract predicates and mutable adts in hoare type theory. Technical Report TR-14-06, Harvard Univ., 2006.
- [7] A. Nanevski, A. Ahmed, G. Morrisett, and L. Birkedal. Polymorphism and separation in hoare type theory. Technical Report TR-10-06, Harvard Univ., 2006.
- [8] A. Nanevski and G. Morrisett. Dependent type theory of stateful higher-order functions. Technical Report TR-24-05, Harvard Univ., 2005.
- [9] P. O'Hearn, J. Reynolds, and H. Yang. Local reasoning about programs that alter data structures. *LNCS*, Vol. 2142, pp. 1-19, 2001.
- [10] J.C. Reynolds. Separation logic: A logic for shared mutable data structures. In *17th LICS*, pp. 55-74, 2002.
- [11] P. Wadler. Comprehending monads. In *ACM Conference on Lisp and Functional Programming*, pp. 61-78, 1990.
- [12] P. Wadler and P. Thiemann. The marriage of effects and monads. *ACM Transactions on Computational Logic*, Vol. 4, No. 1, pp. 1-32, 2003.
- [13] Y. Watanabe. Extending hoare type theory for arrays (senior thesis), 2009.
- [14] K. Watkins, I. Cervesato, F. Pfenning, and D. Walker. A concurrent logical framework: The propositional fragment. *LNCS*, Vol. 3085, pp. 355-377, 2004.