修士論文

# 分数所有権に基づくメモリ解放安全性検証器

指導教員　　末永 幸平

京都大学大学院情報学研究科
修士課程通信情報システム専攻

大元 武

平成 28 年 2 月 8 日

# 分数所有権に基づくメモリ解放安全性検証器

大元 武

**内容梗概**

C 言語では，プログラマは `malloc` や `free` などの命令を用いて，手動でメモリ管理を行わなければならない．そのため，メモリリークや，一度解放したメモリ領域へのアクセスなどの，メモリ管理上の誤りを起こすことがある．

Suenaga と Kobayashi は，これらの誤りを静的に検出するための型システムを提案した．この型システムでは，ポインタを表す型を，**所有権**と呼ばれる情報で拡張している．この情報は，プログラマがそのポインタを通してどのような操作を行ってよいか，行わなければならないかを表している．プログラムに型がつけば，メモリ管理に関する誤りが起きないということが証明されている．彼らはこの型システムに基づいて，各変数や各関数にどのような型がつくのか自動で推論する型推論アルゴリズムを提案し，そのアルゴリズムに基づき検証器を実装した．型推論は，各命令から型付け規則に基づき，所有権が満たすべき制約式を生成し，その制約式の充足可能問題に帰着させている．

しかし，彼らの型システムは C 言語のサブセットを対象としているため，制御文など，いくつかの構文を扱うことができない．彼らの検証器は制御文も扱うことができるが，その部分の実装は厳密には理論に基づいたものにないっていない．

そこで本研究では，より信頼度の高い方法で検証器の実装を行う．具体的には，(1) 制御文で言語を拡張し，(2) その構文を扱えるように型システムを拡張する．また，拡張した型システムに基づいて実装を行った．この検証器は，制御文に加えて，単方向リストなどのデータ構造や，相互再帰関数も扱うことができる．

更に実装の利便性を高めるために**型エラースライサー**を検証器に組み込んだ．型エラーが起こった場合，検証器は型エラーに関係する命令の行番号を表示する．この情報は，型エラーの原因を特定するのに役立つ．

また，実装した検証器に対して予備実験を行った．検証器が制御文を含むプログラム，再帰のある構造体を扱ったプログラム，相互再帰関数を含むプログラムに対しても，メモリリークを正しく検出できることを確かめた．

# Static verifier for safe memory deallocation based on fractional ownership

Takeshi OMOTO

**Abstract**

Programming in the C language requires manual memory management using `malloc` and `free`. Such memory management style is prone to bugs such as memory leak and an access to deallocated memory cells.

Suenaga and Kobayashi proposed a type system for statically verifying memory-deallocation safety. In this type system, a *fractional ownership*, auxiliary information that expresses the capability and the obligation on a pointer, is assigned to each pointer type. They proved that a well-typed program does not cause bugs related to memory deallocation. They also proposed a type inference algorithm that automatically infers a type of each variable and each function and implemented a verifier based on the algorithm. The type inference algorithm generates constraints that the ownerships should satisfy and checks the satisfiability of the constraints.

However, their type system is designed only for a subset of the C language; it does not handle some statements such as control statements. Although their verifier supports the control statements, they support these statements in such a way that is not strictly based on the formally defined type inference algorithm.

Our aim is to extend their framework so that it deals with the C language in a more dependable way. Concretely, we extend (1) their language with the control statements and (2) the type system to handle these statements. We implement a verifier based on the extended type system. Our implementation supports the control statements, data structures such as singly linked lists, and definitions of mutually recursive functions.

In order to make the implementation more useful, we incorporate a *type-error slicer* to our verifier. If a program is not well-typed, the verifier displays the line numbers of the commands in the program that are involved in the type error. This feature is useful for finding the cause of the type error.

We conducted preliminary experiments with the implementation. We confirmed that our implementation correctly detects memory leak of programs that include the control statements, that manipulate recursive data structures, and that include mutually recursive functions.

# 分数所有権に基づくメモリ解放安全性検証器

## 目次

# 第1章　はじめに

## 1.1　背景

C 言語 [1] は，1972 年に Dennis M. Ritchie らによって開発されたプログラミング言語である．本来，ソフトウェアは特定のハードウェア上で動作するように設計されている．そのため，他のハードウェア上でそのソフトウェアを動作させるためには，ソフトウェアを書き換えたり修正したりする必要がある．この書き換えや修正のことを移植と呼ぶ．C 言語は，当時開発されていた OS である UNIX の移植性を高めるために開発された．OS を書くために書かれた言語のためハードウェアよりの低水準な記述もできる，C 言語のコンパイラ自体の移植性や拡張性も高い，C 言語で記述された UNIX が広く普及した，などの理由から今でも多くの分野で使用されているプログラミング言語である．

C 言語は上で述べたように古くに開発された言語のため，自動でメモリ管理をする仕組みなどは組み込まれていない．そのため，プログラマが手動でメモリ領域の確保と解放を行う必要がある．C 言語におけるメモリ確保は，静的確保と動的確保の 2 種類ある．静的確保は，プログラム中で確保するメモリサイズを指定する方法である．しかし，この手法は予め使用するメモリサイズを予測して確保するため，無駄に多くメモリ領域を確保してしまったり，確保したメモリ領域が足りないなどの問題が発生する．動的確保は，この問題点を改善したもので，プログラムの実行中に必要な分だけメモリ領域を確保する方法である．この動的確保に使う命令が `malloc` である．`malloc` は確保するメモリサイズのバイト数を引数として取り，指定されたサイズのメモリ領域を確保し，先頭のアドレスを返す．そのアドレスをポインタに代入すれば，そのポインタを通してメモリ領域に対して操作を行うことができる．確保したメモリ領域を使い終わった後は，そのメモリ領域を解放する必要がある．このメモリ領域の開放に使う命令が `free` である．`free` は，ポインタを引数として取り，そのポインタの参照先のメモリ領域を解放する．

C 言語ではこれらの操作をプログラマが手動で行うため，種々の誤りが発生しうる．1 つ目は**メモリリーク**である．これは，`malloc` によって確保したメモリ領域を解放し忘れるという誤りで，動的確保したメモリ領域を解放し忘れるとシステムが新たに確保できるメモリ領域がどんどん少なくなってしまい，その結果システム全体が停止してしまう，という状況を引き起こす．2 つ目は**ダ**

**ブルフリー**である．これは，一度解放したメモリ領域をもう一度解放してしまうというものである．これにより，メモリ管理情報の一貫性が破壊されることがある．他には，一度解放したメモリ領域から読み込みを行ったり，そこへ書き込みを行うなどがある．一度解放したメモリ領域の値は未定義となっており，そこに操作を行うと予期せぬ動作を起こす原因となる．このようにメモリ操作に関する誤りは深刻なエラーを引き起こしてしまう．更に，プログラムが大きくなるに連れてメモリ操作の誤りを発見することが難しくなってしまう．

そこで，Suenaga と Kobayashi はプログラム中のメモリ操作に関する誤りを静的に検出するための型システム [2] を提案した．この型システムでは，所有権と呼ばれる 0 以上 1 以下の有理数でポインタ型を拡張している．例えば，所有権で拡張された型は $\texttt{int ref}_f$ のように表される．この型はポインタについている所有権が $f$ で，このポインタを一回参照した先の型が `int` である，ということを表してる．この所有権の値に応じてポインタを通してプログラマが行ってよい操作や，行わなければならない操作を定義している．所有権の値が 0 の時は，プログラマはポインタを通してどのような操作も行うことができない．所有権が 0 より大きく，1 未満の時は，プログラマはポインタを通して読み込みだけ行うことができる．所有権の値が 1 の時は，プログラマはポインタを通して読み込み，書き込み，解放を行うことができる．また所有権の値が 0 より大きい時は，プログラマはそのポインタの参照先のメモリ領域を解放しなければならない．このポインタ型を使い，プログラムに型がつけばプログラム中でメモリ操作に関する誤りが起きないということが数学的に証明されている．また彼らは型システムに基いて，各変数や各関数にどのような型がつくのかを自動で推論する型推論アルゴリズムを提案した．型推論は，型付け規則に基いて各命令から所有権が満たすべき制約式を生成し，その制約式の充足可能問題に帰着させている．制約式が充足可能であればプログラムに型がつくということがわかる．

## 1.2 本論文の目的

彼らは，提案した型推論アルゴリズムに基いて検証器を実装した．しかし，彼らが論文中で提案した型システムは `for`，`while`，`break`，`continue` などの制御文に対して形式的な定義は与えていない．そのため，検証器の制御文を扱っている部分は，厳密に理論に基づいた実装になっていない．そこで本研究では，

より信頼度の高い方法で検証器の実装を行う．具体的には，彼らが提案した型システムで扱う言語を制御文で拡張し，拡張した言語に対して型付け規則を与える．そして拡張した型システムに基いて検証器の実装を行う．この検証器は制御文だけでなく，単方向リストなど再帰を含むデータ構造や，相互再帰関数なども扱うことができる．更に，検証器の利便性を高めるために型エラースライサーを検証器に組み込んだ．型エラースライサーはプログラムに型がつかないとわかった時，つまり制約式が充足不能とわかった時，充足不能の原因となっている制約式の部分集合である unsat core を用いて，型エラーの原因となっている命令をスライスとして表示する．この情報は，プログラマがプログラム中のメモリ操作の誤りを発見するにの役立つと考えられる．

## 1.3 本論文の構成

本論文の構成は以下の通りである．第 2 章では，Suenaga と Kobayashi によって提案された，メモリ操作に関する誤りを静的に検出するための型システムと型推論アルゴリズムについて述べる．第 3 章では，本研究で検証の対象とする言語 Clight [3] について説明をし，提案された型システムを Clight へ拡張する．第 4 章では，拡張された型システムに基づいた検証器の実装について述べる．第 5 章では，実装した検証器に対して行った予備実験について述べる．第 6 章では，関連研究の紹介と本研究との比較を行う．第 6 章では，本研究のまとめと今後の課題について述べる．

# 第2章　言語と型システム

この章では，Suenaga と Kobayashi が提案した型システムの説明を行う．

## 2.1　言語

まず，型システムで扱う言語について定義する．

**定義 1 (言語)**

言語の構文を図 1 のように定義する．

$$
\begin{array}{rl}
s\ (\mathit{command}) & ::= \mathtt{skip} \mid *x \leftarrow y \mid s_1; s_2 \mid \mathtt{free}(x) \mid \mathtt{let}\ x = \mathtt{malloc}()\ \mathtt{in}\ s \\
& \mid \mathtt{let}\ x = \mathtt{null}\ \mathtt{in}\ s \mid \mathtt{let}\ x = y\ \mathtt{in}\ s \mid \mathtt{let}\ x = *y\ \mathtt{in}\ s \\
& \mid \mathtt{ifnll}\ (x)\ \mathtt{then}\ s_1\ \mathtt{else}\ s_2 \mid f(x_1, \ldots, x_n) \\
& \mid \mathtt{assert}(x_1, x_2) \\
d\ (\mathit{definitions}) & ::= f(x_1, \ldots, x_n) = s
\end{array}
$$

図 1: 言語

プログラムは，*definitions* の集合を $D$ とすると，$(D, s)$ の組で与えられる．関数定義には返り値がないが，参照を渡すことで関数から値を返す動作をエンコードすることができる．

skip は何もしない命令である．$*x \leftarrow y$ は $x$ が参照しているメモリ領域を $y$ で更新する．$s_1; s_2$ は $s_1$ を実行した後 $s_2$ を実行する．free$(x)$ は $x$ が参照しているメモリ領域を解放する．let $x =$ malloc() in $s$ は新しいメモリ領域を確保し，$x$ をそれに束縛して $s$ を実行する．let $x =$ null in $s$ は $x$ を null に束縛して $s$ を実行する．let $x = y$ in $s$ は $x$ を $y$ に束縛して $s$ を実行する．let $x = *y$ in $s$ は $x$ を $y$ の参照先に束縛して $s$ を実行する．ifnll $(x)$ then $s_1$ else $s_2$ は $x$ が null の場合 $s_1$ を実行し，それ以外の場合は $s_2$ を実行する．$f(x_1, \ldots, x_n)$ は引数 $x_1, \ldots, x_n$ で関数 $f$ を呼ぶ．assert$(x_1, x_2)$ は $x_1$ と $x_2$ が同じメモリ領域を参照している場合は何もせず，そうでない場合はプログラムの実行を停止する．

## 2.2 操作的意味論

上記の言語 (図 1) に対して操作的意味論を定義する．操作的意味論とは，実行状態の遷移によって，プログラムに意味を与えるための数学的定義である．この論文の意味論では，プログラムの実行状態は $\langle H, R, E\rangle$ の三つ組で表現される．$\mathcal{H}$ をヒープ領域のアドレスを表す集合とすると，$H$ は $\mathcal{H}$ から $\mathcal{H} \cup \{\mathtt{null}\}$ への写像で定義され，$R$ は変数から $\mathcal{H} \cup \{\mathtt{null}\}$ への写像で定義される．$H$ と $R$ はそれぞれ，ヒープ領域とレジスタをモデル化している．また，$E$ は，評価文脈で $E ::= [\,] \mid E;\, s$ で定義される．評価文脈は命令の実行順序を決めるもので，$[\,]$ の中には次に実行する命令が入る．$E[s]$ は，$E$ の中の $[\,]$ を $s$ で置き換えたものを表す.

**定義 2 (操作的意味論)**
操作的意味論を図 2 のように定義する．

例えば,

$$\frac{R(x) \in dom(H) \cup \{\mathtt{null}\}}{\langle H, R, E[\mathtt{free}(x)]\rangle \longrightarrow_D \langle H \setminus \{R(x)\}, R, E[\mathtt{skip}]\rangle}$$

は，free(x) の意味を表す規則になっている．まず，前提部分の $R(x) \in dom(H) \cup \{\mathtt{null}\}$ は，変数 $x$ がレジスタに登録されているということを表している．この条件のもとで，ヒープ領域が $H$，レジスタが $R$ の状態で，$\mathtt{free}(x)$ を実行すると，実行後のヒープ領域が $H \setminus \{R(x)\}$ になり，レジスタが $R$ になり，評価文脈が $E[\mathtt{skip}]$ になる．$H \setminus \{R(x)\}$ は，$H$ から $\{R(x)\}$ を取り除くという意味で，これで，メモリ領域が解放されたということを表している．

また,

$$\frac{h \notin dom(H) \quad x' \notin dom(R)}{\langle H, R, E[\mathtt{let}\ x\ =\ \mathtt{malloc}()\ \mathtt{in}\ s]\rangle \longrightarrow_D \langle H\{h \mapsto v\}, R\{x' \mapsto h\}, E[[x'/x]s]\rangle}$$

は，$\mathtt{let}\ x\ =\ \mathtt{malloc}()\ \mathtt{in}\ s$ の意味を表す規則になっている．前提部分の $h \notin dom(H)$ は，アドレス $h$ がヒープに登録されていないということ，つまり新しい領域であるということを表している．また，$x' \notin dom(R)$ は，変数 $x'$ がレジスタに登録されていないということ，つまり新しい変数であるということを表している．この条件のもとで，ヒープ領域が $H$，レジスタが $R$ の状態で，$\mathtt{let}\ x\ =\ \mathtt{malloc}()\ \mathtt{in}\ s$ を実行すると，実行後のヒープ領域が $H\{h \mapsto v\}$,

$$\frac{}{\langle H,\, R,\, E[\texttt{skip};\, s]\rangle \longrightarrow_D \langle H,\, R,\, E[s]\rangle}$$

$$\frac{R(x) \in dom(H)}{\langle H,\, R,\, E[*x \leftarrow y]\rangle \longrightarrow_D \langle H\{R(x) \mapsto R(y)\},\, R,\, E[\texttt{skip}]\rangle}$$

$$\frac{R(x) \in dom(H) \cup \{\texttt{null}\}}{\langle H,\, R,\, E[\texttt{free}(x)]\rangle \longrightarrow_D \langle H \setminus \{R(x)\},\, R,\, E[\texttt{skip}]\rangle}$$

$$\frac{x' \notin dom(R)}{\langle H,\, R,\, E[\texttt{let}\ x\ =\ \texttt{null}\ \texttt{in}\ s]\rangle \longrightarrow_D \langle H,\, R\{x' \mapsto \texttt{null}\},\, E[[x'/x]s]\rangle}$$

$$\frac{x' \notin dom(R)}{\langle H,\, R,\, E[\texttt{let}\ x\ =\ y\ \texttt{in}\ s]\rangle \longrightarrow_D \langle H,\, R\{x' \mapsto R(y)\},\, E[[x'/x]s]\rangle}$$

$$\frac{x' \notin dom(R)}{\langle H,\, R,\, E[\texttt{let}\ x\ =\ *y\ \texttt{in}\ s]\rangle \longrightarrow_D \langle H,\, R\{x' \mapsto H(R(y))\},\, E[[x'/x]s]\rangle}$$

$$\frac{h \notin dom(H)\quad x' \notin dom(R)}{\langle H,\, R,\, E[\texttt{let}\ x\ =\ \texttt{malloc}()\ \texttt{in}\ s]\rangle \longrightarrow_D \langle H\{h \mapsto v\},\, R\{x' \mapsto h\},\, E[[x'/x]s]\rangle}$$

$$\frac{}{\langle H,\, R\{x \mapsto \texttt{null}\},\, E[\texttt{ifnull}(x)\ \texttt{then}\ s_1\ \texttt{else}\ s_2]\rangle \longrightarrow_D \langle H,\, R\{x \mapsto \texttt{null}\},\, E[s_1]\rangle}$$

$$\frac{R(x) \neq \texttt{null}}{\langle H,\, R,\, E[\texttt{ifnull}(x)\ \texttt{then}\ s_1\ \texttt{else}\ s_2]\rangle \longrightarrow_D \langle H,\, R,\, E[s_2]\rangle}$$

$$\frac{R(x) = \texttt{null}}{\langle H,\, R,\, E[*x \leftarrow y]\rangle \longrightarrow_D \texttt{NullEx}}$$

$$\frac{R(y) = \texttt{null}}{\langle H,\, R,\, E[\texttt{let}\ x\ =\ *y\ \texttt{in}\ s]\rangle \longrightarrow_D \texttt{NullEx}}$$

$$\frac{R(y) \notin dom(H) \cup \{\texttt{null}\}}{\langle H,\, R,\, E[\texttt{let}\, x\, =\, *y\, \texttt{in}\, s]\rangle \longrightarrow_D \texttt{Error}}$$

$$\frac{R(x) \notin dom(H) \cup \{\texttt{null}\}}{\langle H,\, R,\, E[\texttt{free}(x)]\rangle \longrightarrow_D \texttt{Error}}$$

$$\frac{f(\tilde{y}) = s \in D}{\langle H,\, R,\, E[f(\tilde{x})]\rangle \longrightarrow_D \langle H,\, R,\, E[[\tilde{x}/\tilde{y}]s]\rangle}$$

$$\frac{H,\, R \models P}{\langle H,\, R,\, E[\texttt{assert}(P)]\rangle \longrightarrow_D \langle H,\, R,\, E[\texttt{skip}]\rangle}$$

$$\frac{H,\, R \not\models P}{\langle H,\, R,\, E[\texttt{assert}(P)]\rangle \longrightarrow_D \texttt{AssertFail}}$$

図 2: 操作的意味論

レジスタが $R\{x' \mapsto h\}$，評価文脈が $E[[x'/x]s]$ になる．$H\{h \mapsto v\}$ は，アドレス $h$ の値が $v$ であるという情報を，ヒープに登録する．アドレス $h$ は新しい領域だったので，これで新しいメモリ領域が確保されたということを表している．なお，新しく確保された領域の中にどのような値が入っているかはわからないので，$v$ は $dom(H) \cup \{\texttt{null}\}$ の中の任意の値をとることができる．また，$R\{x' \mapsto h\}$ は，ポインタ $x'$ がアドレス $h$ であるという情報をレジスタに登録している．$E[[x'/x]s]$ は，$s$ の中の $x$ を $x'$ に置き換えて実行するという意味である．

assert の前提条件にある $H,\, R \models P$ は，以下のように定義される．

$$H,\, R \models x = y \quad \textit{iff} \quad R(x) = R(y)$$
$$H,\, R \models x = *y \quad \textit{iff} \quad R(x) = H(R(y))$$

$P$ が $x = y$ の時は，$R(x)$ と $R(y)$ が等しい，つまり $x$ と $y$ が同じメモリ領域を参照している時に成り立つ．$P$ が $x = *y$ の時は，$x$ と $*y$ が同じメモリ領域を参照している時に成り立つ．

エラーの種類は 3 つある．(1) null に対して読み書きを行うと発生する NullEx (2) 既に解放されたメモリ領域に対して読み書きや，もう一度解放を行うと発生

する Error (3) assert を行った際，引数が同じメモリ領域を参照していない際に発生する AssertFail である．提案された型システムで検出できるのは (2) のエラーだけである．

**例 1 (操作的意味論)**

以下のプログラム $s_0$ を例に挙げる．

```
let a = malloc() in
let b = a in
let c = *a in
let d = *b in
assert(a = b);
free(a)
```

プログラムの初期状態は $\langle \emptyset, \emptyset, s_0 \rangle$ である．図 2 から実行時状態は以下のように遷移していく．なお，$s_1 := \texttt{let b = a in } s_2$，$s_2 := \texttt{let c = *a in } s_3$，$s_3 := \texttt{let d = *b in } s_4$，$s_4 := \texttt{assert(a = b); free(a)}$ である．

$\langle \emptyset, \emptyset, \texttt{let } a = \texttt{malloc}() \texttt{ in } s_1 \rangle$
$\longrightarrow_D \langle \{h_0 \mapsto v_0\}, \{a' \mapsto h_0\}, [a'/a]s_1 \rangle$

$\langle \{h_0 \mapsto v_0\}, \{a' \mapsto h_0\}, \texttt{let } b = a' \, s_2 \rangle$
$\longrightarrow_D \langle \{h_0 \mapsto v_0\}, \{a' \mapsto h_0,\ b' \mapsto h_0\}, [b'/b]s_2 \rangle$

$\langle \{h_0 \mapsto v_0\}, \{a' \mapsto h_0,\ b' \mapsto h_0\}, \texttt{let } c = *a' \texttt{ in } s_3 \rangle$
$\longrightarrow_D \langle \{h_0 \mapsto v_0\}, \{a' \mapsto h_0,\ b' \mapsto h_0,\ c' \mapsto v_0\}, [c'/c]s_3 \rangle$

$\langle \{h_0 \mapsto v_0\}, \{a' \mapsto h_0,\ b' \mapsto h_0,\ c' \mapsto v_0\}, \texttt{let } d = *b' \texttt{ in } s_4 \rangle$
$\longrightarrow_D \langle \{h_0 \mapsto v_0\}, \{a' \mapsto h_0,\ b' \mapsto h_0,\ c' \mapsto v_0,\ d' \mapsto v_0\}, [d'/d]s_4 \rangle$

$\langle \{h_0 \mapsto v_0\}, \{a' \mapsto h_0,\ b' \mapsto h_0,\ c' \mapsto v_0,\ d' \mapsto v_0\}, \texttt{assert}(a' = b');\ \texttt{free}(a') \rangle$
$\longrightarrow_D \langle \{h_0 \mapsto v_0\}, \{a' \mapsto h_0,\ b' \mapsto h_0,\ c' \mapsto v_0,\ d' \mapsto v_0\}, \texttt{skip};\ \texttt{free}(a') \rangle$
$\longrightarrow_D \langle \{h_0 \mapsto v_0\}, \{a' \mapsto h_0,\ b' \mapsto h_0,\ c' \mapsto v_0,\ d' \mapsto v_0\}, \texttt{free}(a)' \rangle$
$\longrightarrow_D \langle \emptyset, \{a' \mapsto h_0,\ b' \mapsto h_0,\ c' \mapsto v_0,\ d' \mapsto v_0\}, \texttt{skip} \rangle$

## 2.3 型

次に型システムで扱う型について定義する．

**定義 3 (型)**
型の構文を図 3 のように定義する．

$$\begin{array}{|l|}\hline \tau\ (\mathit{value\ types})\ ::=\ \alpha \mid \tau\ \mathtt{ref}_f \mid \mu\alpha.\tau \\ \sigma\ (\mathit{function\ types})\ ::=\ (\tau_1, \ldots, \tau_n) \to (\tau'_1, \ldots, \tau'_n) \\ \hline \end{array}$$

図 3: 型

$\alpha$ は型変数で，再帰型を作る構築子 $\mu\alpha$ によって束縛される．$\tau\ \mathtt{ref}_f$ はポインタにつく型で，参照した先の値の型が $\tau$ で所有権が $f$ である，ということを表している．所有権 $f$ は 0 以上 1 以下の有理数で，プログラマがそのポインタを通して行って良い操作，行わなければならない操作を表現している．所有権が 0 の時，プログラマはポインタを通してどのような操作も行うことができない．所有権が 0 より大きく 1 未満の時，プログラマはポインタを通して読み込みを行うことができる．所有権が 1 の時，プログラマはポインタを通して，読み込み，書き込み，解放を行うことができる．また，所有権が 0 より大きい時は，プログラマはそのポインタの参照先を解放しなければならない．$(\tau_1, \ldots, \tau_n) \to (\tau'_1, \ldots, \tau'_n)$ は，$n$ 引数関数につく型で，引数の型が $(\tau_1, \ldots, \tau_n)$ で，関数を実行すると，その型が $(\tau'_1, \ldots, \tau'_n)$ になるということを表している．

次に型の意味論について定義する．

**定義 4 (意味論)**
型の意味論を以下のように定義する．

$$[\![\tau\ \mathtt{ref}_f]\!](\epsilon)\ =\ f\quad [\![\tau\ \mathtt{ref}_f]\!](0w)\ =\ [\![\tau]\!](w)\quad [\![\mu\alpha.\tau]\!]\ =\ [\![[\mu\alpha.\tau/\alpha]\tau]\!]$$

$[\![\cdot]\!]$ は，0 の有限列の集合から，有理数への集合の写像で定義される．$[\![\tau\ \mathtt{ref}_f]\!](\epsilon)$ は，ポインタが直接参照しているメモリ領域への所有権を表し，$[\![\tau\ \mathtt{ref}_f]\!](0^k)$ は，ポインタから $k$ 回参照した先のメモリ領域への所有権を表している．

また，$[\![\tau]\!] = [\![\tau']\!]$ が成り立つとき，$\tau \approx \tau'$ と表記する．再帰型 $\mu\alpha.\tau$ に関しては，$\mu\alpha.\tau \approx [\mu\alpha.\tau/\alpha]\tau$ が成り立つ．$[\mu\alpha.\tau/\alpha]\tau$ は，$\tau$ 中の $\alpha$ を $\mu\alpha.\tau$ に置き換えたもので，例えば，$\mu\alpha.(\alpha\ \mathtt{ref}_0)$ と $(\mu\alpha.(\alpha\ \mathtt{ref}_0))\ \mathtt{ref}_0$ は同じ型を表している．更に，型 $\tau$ 中に含まれる所有権が全て 0 の時，$\mathtt{empty}(\tau)$ と表記する．

## 2.4 型付け規則

上で定義した型をもとに型付け規則を定義する．

型付け可能かどうかを判断する型判断は $\Theta; \Gamma \vdash s \Rightarrow \Gamma'$ の形をしている．$\Theta$ は，関数環境で，変数 (関数名) から関数の型への写像である．$\Gamma$ は，型環境で，変数からその変数についている型への写像である．関数環境 $\Theta$，型環境 $\Gamma$ の元で，命令 $s$ を実行すると，実行後の型環境が $\Gamma'$ になる，という意味である．

**定義 5 (型付け規則)**

型付け規則を図 4 のように定義する．

$$\frac{}{\Theta; \Gamma \vdash \mathtt{skip} \Rightarrow \Gamma} \qquad \frac{\Theta; \Gamma \vdash s_1 \Rightarrow \Gamma'' \qquad \Theta; \Gamma'' \vdash s_2 \Rightarrow \Gamma'}{\Theta; \Gamma \vdash s_1; s_2 \Rightarrow \Gamma'}$$

$$\frac{\tau \approx \tau_1 + \tau_2 \qquad \mathtt{empty}(\tau') \qquad f = 1}{\Theta; \Gamma, x : \tau' \,\mathtt{ref}_f, y : \tau \vdash *x \leftarrow y \Rightarrow \Gamma, x : \tau_1 \,\mathtt{ref}_f, y : \tau_2}$$

$$\frac{\mathtt{empty}(\tau) \qquad f_1 = 1 \qquad f_2 = 0}{\Theta; \Gamma, x : \tau \,\mathtt{ref}_{f_1} \vdash \mathtt{free}(x) \Rightarrow \Gamma, x : \tau \,\mathtt{ref}_{f_2}}$$

$$\frac{\Theta; \Gamma, x : \tau \,\mathtt{ref}_1 \vdash s \Rightarrow \Gamma', x : \tau' \,\mathtt{ref}_0 \qquad \mathtt{empty}(\tau) \qquad \mathtt{empty}(\tau')}{\Theta; \Gamma \vdash \mathtt{let}\, x = \mathtt{malloc}()\, \mathtt{in}\, s \Rightarrow \Gamma'}$$

$$\frac{\Theta; \Gamma, x : \tau_1, y : \tau_2 \vdash s \Rightarrow \Gamma', x : \tau_1' \qquad \tau \approx \tau_1 + \tau_2 \qquad \mathtt{empty}(\tau_1')}{\Theta; \Gamma, y : \tau \vdash \mathtt{let}\, x = y\, \mathtt{in}\, s \Rightarrow \Gamma'}$$

$$\frac{\begin{array}{c}\Theta; \Gamma, x : \tau_1, y : \tau_2 \,\mathtt{ref}_f \vdash s \Rightarrow \Gamma', x : \tau_1' \\ f > 0 \qquad \tau \approx \tau_1 + \tau_2 \qquad \mathtt{empty}(\tau_1')\end{array}}{\Theta; \Gamma, y : \tau \,\mathtt{ref}_f \vdash \mathtt{let}\, x = *y\, \mathtt{in}\, s \Rightarrow \Gamma'}$$

$$\frac{\Theta(f) = (\tilde{\tau}) \to (\tilde{\tau}')}{\Theta; \Gamma, \tilde{x} : \tilde{\tau} \vdash f(\tilde{x}) \Rightarrow \Gamma, \tilde{x} : \tilde{\tau}'}$$

$$\frac{\Theta;\, \Gamma,\, x : \tau \vdash s \Rightarrow \Gamma',\, x : \tau'}{\Theta;\, \Gamma \vdash \texttt{let}\, x \,=\, \texttt{null}\ \texttt{in}\, s \Rightarrow \Gamma'}$$

$$\frac{\Theta;\, \Gamma,\, x : \tau' \vdash s_1 \Rightarrow \Gamma' \qquad \Theta;\, \Gamma,\, x : \tau \vdash s_2 \Rightarrow \Gamma'}{\Theta;\, \Gamma,\, x : \tau \vdash \texttt{ifnull}(x)\ \texttt{then}\, s_1\ \texttt{else}\, s_2 \Rightarrow \Gamma'}$$

$$\frac{\Theta;\, \tilde{x} : \tilde{\tau} \vdash s : \tilde{x} : \tilde{\tau}' \qquad \Theta(f) = \tilde{\tau} \to \tilde{\tau}' \qquad (\text{for each } f(\tilde{x}) = s \in D)\ dom(\Theta) = dom(D)}{\vdash D : \Theta}$$

$$\frac{\vdash D : \Theta \quad \Theta;\, \emptyset \vdash s \Rightarrow \emptyset}{\vdash (D,\, s)}$$

$$\frac{\tau_1 + \tau_2 \approx \tau_1' + \tau_2'}{\Theta;\, \Gamma,\, x_1 : \tau_1,\, x_2 : \tau_2 \vdash \texttt{assert}(x_1, x_2) \Rightarrow \Gamma,\, x_1 : \tau_1',\, x_2 : \tau_2'}$$

$$\frac{\Gamma \approx \Gamma_1 \qquad \Gamma' \approx \Gamma_1' \qquad \Theta;\, \Gamma_1 \vdash s \Rightarrow \Gamma_1'}{\Theta;\, \Gamma \vdash s \Rightarrow \Gamma'}$$

図 4: 型付け規則

例えば,

$$\frac{\tau \approx \tau_1 + \tau_2 \qquad \texttt{empty}(\tau') \qquad f = 1}{\Theta;\, \Gamma,\, x : \tau'\, \texttt{ref}_f,\, y : \tau \vdash *x \leftarrow y \Rightarrow \Gamma,\, x : \tau_1\, \texttt{ref}_f,\, y : \tau_2}$$

は，$*x \leftarrow y$ に対する型付け規則になっている．$\tau \approx \tau_1 + \tau_2$ は，$\tau$ が実行前の $y$ の型，$\tau_1$ と $\tau_2$ がそれぞれ実行後の $*x$ と $y$ の型なので，実行前の $y$ の所有権を実行後の $*x$ と $y$ に分配する，ということを表している．$f$ は実行前の $x$ についている所有権で，この値が 1 になっている．これにより，ポインタを通して書き込みを行うには所有権が 1 でないといけない，ということを表している．

同じように $\texttt{free}(x)$ の規則を見ると，実行前の $x$ の所有権が 1 で，実行後の所有権が 0 になっている．これはポインタを通してメモリ領域を解放するには所有権が 1 必要であるということと，解放後には 0 になっているため，もうそのポインタを通して書き込みや解放は行うことができなくなっているということを表している．

ポインタに対する所有権は $\texttt{let}\, x \,=\, \texttt{malloc}()\ \texttt{in}\, s$ で与えられる．$x$ に所有権 1 が与えられた環境の元で $s$ を実行し，$s$ の実行が終わる，つまり $x$ のスコー

プを抜ける時点で，$x$ の所有権は 0 になっていなければならない．これで，メモリリークが起きていないということを表している．

## 2.5 健全性

また，型システムの健全性が証明されている．

**定理 1 (型の健全性 [2])**

$\vdash (D, s)$ なら，以下の 2 条件を満たす．

1. $\langle \emptyset, \emptyset, s \rangle \not\longrightarrow_D^* \texttt{Error}$
2. *If* $\langle \emptyset, \emptyset, s \rangle \longrightarrow_D^* \langle H, R, \texttt{skip} \rangle$, *then* $H = \emptyset$

$\vdash (D, s)$ は，プログラムに型がつくということを表している．1 の条件は，プログラムに型がつくなら，実行状態が Error になることはない，ということを意味している．Error は，図 2 で定義したように，一度解放したメモリ領域にアクセスすると発生するエラーである．この条件により，プログラムに型がつけば，不正なメモリ領域へのアクセスが起きないということが保証されている．2 の条件は，プログラムに型がつき，かつプログラムが停止したなら，停止後の $H$ は空集合になっているということを意味している．$H$ はヒープ領域をモデル化したものなので，ヒープ領域が空になっているということは，全てのメモリ領域が解放されているということである．この条件により，プログラムに型がつき，かつプログラムが停止するなら，メモリリークが起きないということが保証されている．

## 2.6 型推論アルゴリズム

最後に各変数や各関数につく型を自動で推論するためのアルゴリズムについて説明する．定理 1 より，型推論によりプログラムに型が付けばメモリに関するエラーが発生しない，ということが保証されている．実装の簡単化のために，型推論での型の構文を $(\mu\alpha.\alpha\ \texttt{ref}_{f_1})\ \texttt{ref}_{f_2}$ に制限している．型推論は以下のように進んでいく．

1. $n$ 引数の関数 $f$ の型を

$$\begin{aligned}&((\mu\alpha.\alpha\ \texttt{ref}_{\eta_{f,1,1}})\ \texttt{ref}_{\eta_{f,1,2}}, \ldots (\mu\alpha.\alpha\ \texttt{ref}_{\eta_{f,n,1}})\ \texttt{ref}_{\eta_{f,n,2}})\\ &\longrightarrow ((\mu\alpha.\alpha\ \texttt{ref}_{\eta'_{f,1,1}})\ \texttt{ref}_{\eta'_{f,1,2}}, \ldots (\mu\alpha.\alpha\ \texttt{ref}_{\eta'_{f,n,1}})\ \texttt{ref}_{\eta'_{f,n,2}})\end{aligned}$$

とする．$\eta_{f,i,j}$ と $\eta'_{f,i,j}$ は値の分からない所有権である．

同様に，プログラム地点 $p$ での変数 $x$ の型を

$$((\mu\alpha.\alpha\ \mathtt{ref}_{\eta_{p,x,1}})\ \mathtt{ref}_{\eta_{p,x,2}})$$

とする．

2. 型付け規則 (図 4) に従い所有権についての線形不等式を生成する．
3. 線形不等式を解く．不等式に解があれば，そのプログラムには型が付く．

**例 2 (型推論)**

図 5 のプログラムを例に挙げる．コメントには，各プログラム地点の型環境と，型環境と型付け規則から生成された所有権に関する制約式が書かれている．型は `(t ref_f1) ref_f0` の形をしており，`f0` がポインタに直接についている所有権，`f1` がポインタを一回参照した先のポインタについている所有権を表している．

`let a = malloc()` により，ポインタ `a` に所有権 1 が与えられる．`let b = a` により，`a` と `b` は同じメモリ領域を参照するので，`a` が持っていた所有権 1 が `a` と `b` で分配される．`let c = *a` により，`*a` と `c` は同じメモリ領域を参照するので，`*a` が持っていた所有権が`*a` と `c` で分配される．また，ポインタ `a` を参照しているので，`a` には 0 より大きい所有権が必要となる．`let d = *b` も同様である．そして最後に `free(a)` を実行するのだが，この段階では `a` が持っていた所有権は `a` と `b` のに分配されているため，`a` は所有権 1 を持っておらず，`free` を実行することができない．そのため，`free` の前に，`assert(a = b)` を実行する．`assert` は，同じメモリ領域を参照しているポインタ間同士で所有権の受け渡しをするための命令で，`a` と `b` は同じメモリ領域を参照しているので，分配していた所有権を `assert` により `a` に戻すことができる．これにより，`free(a)` が実行でき，全ての所有権が 0 になる．

各命令から所有権に関する制約式が生成されるので，それらをまとめて SMT ソルバで解く．このプログラムの場合，制約式に解が存在するので，プログラムに型がつき，メモリ操作に関する誤りが起きていないことがわかる．

```
let a = malloc() in
  /*
  type_env = [a:(t ref_f1) ref_f0]
  constr = [f0 = 1; f1 = 0]
  */
let b = a in
  /*
  type_env = [a:(t ref_f3) ref_f2; b:(t ref_f5) ref_f4]
  constr = [f0 = f2 + f4; f1 = f3 + f5]
  */
let c = *a in
  /*
  type_env = [a:(t ref_f7) ref_f6; b:(t ref_f5) ref_f4
              c:(t ref_f9) ref_f8]
  constr = [f2 > 0; f3 = f7 + f8]
  */
let d = *b in
  /*
  type_env = [a:(t ref_f7) ref_f6; b:(t ref_f11) ref_f10
              c:(t ref_f9) ref_f8; d:(t ref_f13) ref_f12]
  constr = [f4 > 0; f5 = f11 + f12]
  */
assert(a = b);
  /*
  type_env = [a:(t ref_f15) ref_f14; b:(t ref_f17) ref_f16
              c:(t ref_f9) ref_f8; d:(t ref_f13) ref_f12]
  constr = [f6 + f10 = f14 + f16; f7 + f11 = f15 + f17]
  */
free(a)
  /*
  type_env = [a:(t ref_f19) ref_f18; b:(t ref_f17) ref_f16
              c:(t ref_f9) ref_f8; d:(t ref_f13) ref_f12]
  constr = [f14 = 1; f18 = 0; f19 = 0]
  */
```

図 5: プログラム例

# 第3章　型システムの拡張

この章では，Leroy らによって実装された C 言語のコンパイラ CompCert [4] と，そのコンパイラで使われている中間言語 Clight [3] の説明を行う．そして，前章の型システムを，Clight を対象としたものに拡張する．

## 3.1　CompCert

CompCert [4] は，Leroy らによって実装された C 言語のコンパイラである．CompCert の最大の特徴は，コンパイラの正しさを数学的に証明をしている，という点である．コンパイラは，コンパイル時に多くの変換を行う．しかし，それらの変換が正しく行われているかどうかは，テストによってのみ保証されている．テストによる保証には限界があるため，Leroy らは，言語に意味論を与え，変換を行った際，変換後の言語が変換前の言語の意味を正確に保存しているというという性質を，証明支援系 Coq [5] を用いて証明した．また，Coq には Extraction という Coq で書かれたコードを他の言語に変換する機能がある．この機能により，証明された変換をそのままプログラムにすることができる．CompCert は，この Extraction によって変換された OCaml のコードが大部分を占めている．

## 3.2　Clight

Clight [3] は，CompCert 内で使われている中間言語で，C 言語のサブセットである．Clight は，ポインタや配列や構造体などのデータ構造，`if` や `switch` などの分岐，`while` や `for` などの再帰，`break` や `continue` などの構文，再帰関数や関数ポインタなどの関数に関する機能など，C 言語の機能をほぼ全てサポートしているが，以下の機能は排除されている．

- `long long` や `long double` などの拡張された数字
- 可変長引数の関数
- 式内で副作用を起こす演算子
- ブロック内での変数宣言
- unstructured switch

式内で副作用を起こす演算子とは，インクリメント文などのことである．インクリメントは，$x = x + 1$ のように文で表現される．また，C 言語では，`if`

や while などのブロックの中でも変数を宣言できるが，Clight では，変数は大域変数か関数のローカル変数だけである．

switch 文は，変数の値に応じて処理を分岐させる構文である．

```
switch (x) {
  case 1: s1;
  case 2: s2;
  case 3: s3;
  default: s0;
}
```

一般の switch 文は $x$ の値に応じて 1 つだけ case 文が実行される．しかし，C 言語の場合，s1，s2，s3 に break を書かないと，それ以降全ての case 文を実行する．例えば $x$ の値が 2 だったとすると，case 2 と case 3 が実行される．このような switch のことを，unstructured switch と呼ぶ．Clight では，この unstructured switch は扱えず，全ての case 文に break が挿入されているものとみなされる．

## 3.3 型システムの拡張

第 2 章で説明した型システムを，Clight を対象としたものに拡張を行う．具体的には，Clight の型に所有権の情報を追加し，その型をもとに Clight の構文に型付け規則を与える．

まず，元の Clight の型について説明をする．

**定義 6 (型)**

型を図 6 のように定義する．$t\ list$ は $t$ の 0 個以上の要素の並びを表す．

int や float は，定数につく型で，引数としてそれぞれのサイズを表す，*intsize*，*floatsize* を引数としてとる．3.2 節で説明したように，64bit の整数など，拡張された数は扱うことができない．

array は，配列につく型で，要素の型を表す型 $t$ と配列の要素数 $n$ を引数としてとる．配列の要素数がわからない場合は，ポインタ型で置き換えられる．

function は，関数につく型で，引数の型 $t\ list$ と返り値の型 $t$ を引数としてる．こちらも 3.2 節で説明したように，可変長引数の関数は扱うことができない．そのため，$t\ list$ の長さは関数ごとに固定である．

sturct と union は，構造体につく型で，構造体の名前を表す $id$ とフィール

$$
\begin{aligned}
t\ (types)\ ::= & \texttt{int}(intsize,\ signedness) \mid \texttt{float}(floatsize) \mid \texttt{void} \\
& \mid \texttt{array}(t,\ n) \mid \texttt{pointer}(t) \mid \texttt{function}(t_1\ list,\ t) \\
& \mid \texttt{struct}(id,\ fl) \mid \texttt{union}(id,\ fl) \\
& \mid \texttt{comp\_ptr}(id) \\
intsize\ ::= & \texttt{I8} \mid \texttt{I16} \mid \texttt{I32} \\
floatsize\ ::= & \texttt{F32} \mid \texttt{F64} \\
fl\ (fieldlist)\ ::= & (id,\ t)\ list
\end{aligned}
$$

図 6: 型

ドの型 $fl$ を引数としてとる．$fl$ は，どの構造体のフィールドなのかを表す $id$ と各フィールドの型を表す $t$ の組のリストになっている．

comp_ptr は，構造体内で自己参照しているポインタにつく型で，構造体の名前を表す *id* を引数として取る．C 言語では，再帰のある構造体を作る際，自分自身を指すフィールドは必ずポインタ型でないといけない．そのため，特別な型として comp_ptr を用意することで，フィールドに直接 struct 型がでてくることを防いでいる．

次に，上記の型を第 2 章で説明した所有権の概念で拡張する．ただし本システムでは，array と union をま扱っていないため，省略する．

**定義 7 (型)**
拡張後の型を図 7 のように定義する．

ポインタにつく型 pointer と，再帰のある構造体内で自己参照しているポインタにつく型 comp_ptr には所有権 $o$ を追加する．所有権には 3 種類あり，所有権変数を表わす $\texttt{Ovar}(n)$ と，定数を表す $\texttt{Oconst}(n)$ と， 所有権同士の足し算を表す $\texttt{Oplus}(o, o)$ である．$\texttt{Ovar}(n)$ の $n$ は整数で，それぞれの所有権変数を区別するための識別子として使われる．$\texttt{Oconst}(n)$ の $n$ は 0 か 1 の整数である．所有権は 0 以上 1 以下の有理数だが，制約生成の段階で定数として現れるのは 0 か 1 だけなので，$n$ を整数としても問題ない．

また，function の型にもう一つ $t\ list$ を追加してある．関数の実行により，関数の引数の所有権の値が変わることがあり，関数の実行後に引数がどういう型になるのか記録しておく必要があるため，これを追加した．

$$
\begin{aligned}
t\ (types) \ ::= & \ \texttt{int}(intsize,\ signedness) \mid \texttt{float}(floatsize) \mid \texttt{void} \\
& \mid \texttt{pointer}(t,\ o) \mid \texttt{function}(t_1\ list,\ t_2\ list,\ t) \\
& \mid \texttt{struct}(id,\ fl) \\
& \mid \texttt{comp\_ptr}(id,\ n,\ o) \mid \texttt{Tplus}((id,\ n)\ list) \\
intsize \ ::= & \ \texttt{I8} \mid \texttt{I16} \mid \texttt{I32} \\
floatsize \ ::= & \ \texttt{F32} \mid \texttt{F64} \\
fl\ (fieldlist) \ ::= & \ (id,\ t)\ list \\
o\ (ownership) \ ::= & \ \texttt{Ovar}(n) \mid \texttt{Oconst}(n) \mid \texttt{Oplus}(o, o)
\end{aligned}
$$

図 7: 拡張された型

comp_ptr には更に，整数 $n$ を追加してある．comp_ptr は，再帰のある構造体内で，自分を参照しているポインタにつく型なので，常に同じ名前の構造体を参照していることになる．しかし，ポインタに所有権の情報を追加したため，名前が同じ構造体でも各フィールドの所有権が違う構造体を作ることができる．整数 $n$ は，この名前が同じだがフィールドの所有権が違う構造体を区別するための識別子である．

更に新しい型として，$\texttt{Tplus}((id,\ n)\ list)$ を追加した．これは，再帰のある構造体同士の足し算を表している型で，引数として構造体の名前 $id$ と上で述べた名前が同じだが所有権が違う構造体を区別するための識別子 $n$ の組のリストを引数として取る．

次に，式について説明する．元の Clight の式の定義には，$\&x$ と，$(type)\ e$ も定義されている．それぞれ，変数 $x$ のアドレスをとってくる操作と，式 $e$ の型を $type$ に変換する操作を表している．本システムでは扱っていないため，省略する．

**定義 8 (式)**

式を図 8 のように定義する．

const は定数で，int 型と float 型の 2 種類ある．$x$ は変数で，temp は，コンパイラが用意した一時変数である．$*x$ はポインタの参照で，$e.f$ は構造体のフィールドアクセスである．$unop\ e$ と $e_1\ binop\ e_2$ は，それぞれ単項演算と二項演算で，使える演算子は上記の通りである．3.2 節で述べたようにインクリメン

$$
\begin{aligned}
e\ (expression)\ ::= & \ \texttt{const} \mid x \mid \texttt{temp} \mid \texttt{*}x \mid e.f \\
& \mid unop\ e \mid e_1\ binop\ e_2 \\
unop\ ::= & \ \texttt{-} \mid \texttt{\~{}} \mid \texttt{!} \\
binop\ ::= & \ \texttt{+} \mid \texttt{-} \mid \texttt{*} \mid \texttt{/} \mid \texttt{\%} \\
& \mid \texttt{«} \mid \texttt{»} \mid \texttt{\&} \mid \texttt{|} \mid \texttt{\^{}} \\
& \mid \texttt{<} \mid \texttt{<=} \mid \texttt{>} \mid \texttt{>=} \mid \texttt{==} \mid \texttt{!=}
\end{aligned}
$$

図 8: 式

トのような，副作用を起こす操作は式の段階では定義されていない．

続いて，文について説明する．元の Clight の文には，goto 文と label 文が定義されているが，今回の型システムでは扱っていないため省略してある．また，新しい文として，$\texttt{free}(e)$，$t = \texttt{malloc}(e)$，$\texttt{assert}(e_1,\ e_2)$，$\texttt{assert\_null}(e)$ を追加した．

**定義 9 (文)**

文を図 9 のように定義する．

$$
\begin{aligned}
s\ (statement)\ ::= & \ \texttt{skip} \mid e_1 = e_2 \mid f(\tilde{e}) \mid t = f(\tilde{e}) \mid s_1;\ s_2 \\
& \mid \texttt{free}(e) \mid t = \texttt{malloc}(e) \\
& \mid \texttt{assert}(e_1,\ e_2) \mid \texttt{assert\_null}(e) \\
& \mid \texttt{if}\ e\ \texttt{then}\ s_1\ \texttt{else}\ s_2 \mid \texttt{switch}(e)\ sw \\
& \mid \texttt{loop}(s_1)\ s_2 \mid \texttt{break} \mid \texttt{continue} \\
& \mid \texttt{return}(e) \\
sw\ (switch\ cases)\ ::= & \ \texttt{default}:\ s \\
& \mid \texttt{case}\ n:\ s;\ sw
\end{aligned}
$$

図 9: 文

$e_1 = e_2$ は代入で，$e_1$ の評価結果に，$e_2$ の評価結果を代入する．第 2 章では，*value types* にポインタ型しかなかったため，ポインタの参照先の更新しかなかったが，この型システムでは `int` や `float` などの変数の代入も行うことができる．

$f(\tilde{x})$ は関数呼び出しで，関数 $f$ を引数 $\tilde{x}$ で呼ぶ．$\texttt{temp} = f(\tilde{x})$ は，関数を呼んだ後，更にその関数の返り値を temp に代入する．第 2 章では，関数に返り値が定義されていなかったが，Clight では返り値のある関数が定義されているので，この文が定義されている．$\texttt{free}(e)$ は，$e$ の参照先を解放する．$e$ を参照するので，$e$ はポインタ型でないといけない．$\texttt{temp} = \texttt{malloc}(e)$ は，新しいメモリ領域を確保して，それを temp に代入する．assert と assert_null は，型チェックのための注釈に相当する命令である．$\texttt{assert}(e_1, e_2)$ は $e_1$ と $e_2$ が値の等しいポインタであることをチェックする文である．この命令は型システムでは，$e_1$ と $e_2$ の所有権を分配するという機能を持っている．$\texttt{assert\_null}(e)$ は $e$ の評価結果が null であることをチェックする命令である．この命令は型システムでは，$e$ がその後任意の型をもつことができるということを表す．free や malloc，assert は構文上は関数呼び出しと同じだが，これらの命令は，所有権に関して特別な操作を行うため，別の命令として定義してある．

$\texttt{if}\ e\ \texttt{then}\ s_1\ \texttt{else}\ s_2$ は，$e$ の評価結果が 0 でない場合 $s_1$ を実行し，0 の場合 $s_2$ を実行する．$\texttt{switch}(e)\ sw$ は，$e$ の評価結果に応じて，$sw$ の中の $s$ を実行する．$sw$ 中の各 $s$ は，break で終わらなければならない．$\texttt{loop}(s_1)s_2$ は，$s_1; s_2$ を繰り返し実行する．$s_1$ が skip の場合，while 文に対応し，それ以外の場合，for 文に対応する．break は，それ以降の命令を実行せずに，ブロックを抜ける．continue は，それ以降の命令を実行せずに，次のループを実行する．Clight では，while や for は全て無限ループに変換され，ループの脱出には，if 文と break 文を使う．

最後に，プログラムについて定義する．元の Clight のプログラムでは，大域変数が定義されている，つまり $Promgrams$ の定義の中に $dcl$ が含まれているが，本システムでは扱っていないため省略している．

**定義 10 (プログラム)**
Clight のプログラムを図 10 のように定義する．

変数宣言 $dcl$ は，変数の型と変数の名前の組のリストになっている．内部関数 $F$ は，プログラム内で定義されている関数で，関数の返り値 $t$，関数名 $id$，引数 $dcl_1$，局所変数 $dcl_2$，関数本体 $s$ の組になっている．3.2 節で述べたように，Clight ではブロック内などで変数宣言はできず，全て関数の先頭で宣言されるため，$dcl_2$ のように予め局所変数を全て取ってくることができる．外部関数 $Fe$

$$
\begin{aligned}
P\ (Programs) &::= Fd\ list;\ \texttt{main} = id \\
dcl\ (declarations) &::= (t,\ id)\ list \\
Fd\ (Function\ definitions) &::= F \mid Fe \\
F\ (Internal\ functions) &::= (t,\ id,\ dcl_1,\ dcl_2,\ s) \\
Fe\ (External\ functions) &::= (\texttt{extern},\ t,\ id,\ dcl)
\end{aligned}
$$

図 10: プログラム

は，プログラム外で定義されている関数で，$Fd$ は，内部関数と外部関数のどちらかである．プログラム $P$ は，この $Fd$ のリストと，プログラム実行時に最初に呼ばれる関数 `main` で定義されている．

上記の式 (図 8) と文 (図 9) に対して，型付け規則を与える．基本的には，第 2 章の図 4 を拡張する形で与える．

まず，式についての型付け規則を定義する．式が型付け可能か判断する型判断は，$\Gamma \vdash^{w/r} e : t$ の形をしている．型環境 $\Gamma$ のもとで $e$ を評価すると $e$ に $t$ という型がつくということを表している．$w/r$ は，$e$ に対して書き込みが行われているかどうかを区別するためのラベルである．$w$ の時は変数に対して書き込みが行われているという意味である．$r$ の時，変数の読み込みが行われているという意味である．

**定義 11 (式)**
式の型付け規則を図 11 のように定義する．

EVAR と ETEMP はそれぞれ変数と一時変数に関する型付け規則で，型環境内に登録されている型をそのまま式の型とすればよい．ECONST は定数に関する型付け規則で，定数には `int` 型と `float` 型とがあるが，今回のメモリリーク検出には必要ない情報なので，全て `void` 型としている．EUNOP と EBINOP は演算に関する型付け規則である．$unop$ と $binop$ を関数とみなして，引数の型があっていることを確かめている．

EDEREF_R と EDEREF_W はそれぞれポインタの参照に関する型付け規則である．EDEREF_R は右辺値の場合で，ポインタの参照先から値の読み込みを行っているので，0 より大きい所有権が必要になる．EDEREF_W は左辺値の場合で，ポインタの参照先に書き込みを行っているので，1 の所有権が必要になる．また，

$$\frac{}{\Gamma,\ x : t \vdash^{w/r} x : t} \quad \text{(EVAR)}$$

$$\frac{}{\Gamma,\ \texttt{temp} : t \vdash^{w/r} \texttt{temp} : t} \quad \text{(ETEMP)}$$

$$\frac{}{\Gamma \vdash^{w/r} \texttt{const} : \texttt{void}} \quad \text{(ECONST)}$$

$$\frac{\Gamma \vdash^{w/r} e : t \qquad unop : t \to t'}{\Gamma \vdash^{w/r} unop\ e : t'} \quad \text{(EUNOP)}$$

$$\frac{\Gamma \vdash^{w/r} e_1 : t \qquad \Gamma \vdash^{w/r} e_2 : t' \qquad binop : t \to t' \to t''}{\Gamma \vdash^{w/r} e_1\ binop\ e_2 : t''} \quad \text{(EBINOP)}$$

$$\frac{0 < o}{\Gamma,\ x : \texttt{pointer}(t,\ o) \vdash^{r} *x : t} \quad \text{(EDEREF\_R)}$$

$$\frac{o = 1}{\Gamma,\ x : \texttt{pointer}(t,\ o) \vdash^{w} *x : t} \quad \text{(EDEREF\_W)}$$

$$\frac{fl(id) = t}{\Gamma,\ x : \texttt{struct}(id,\ fl) \vdash^{w/r} x.f : t} \quad \text{(EFIELD)}$$

図 11: 式の型付け規則

それぞれポインタの参照に関する型付け規則なので，$x$ は pointer 型でなければならない．

EFIELD はフィールドアクセスに関する型付け規則である．$fl$ をフィールド名から型への写像とすると，$fl(id) = t$ でアクセスしようとしているフィールド名がきちんと定義されているかものかどうかを確認している．フィールドアクセスに関する型付け規則なので，$x$ は struct 型でなければならない．

続いて，文の型付け規則を定義する．文が型付け可能か判断する型判断は，$\Theta;\ B;\ C;\ \Gamma \vdash s \Rightarrow \Gamma'$ の形をしている．$\Theta$ は関数環境で，関数名から関数の型への写像である．$\Gamma$ は型環境で，変数名から変数の型への写像である．$B$ は ***break* 環境**で，break が呼ばれた際に使用する．定義は型環境と同じで，変数名から変数の型への写像である．break が実行された際に，ループを抜けた後の型環境を記録しておくために使う．$C$ は ***continue* 環境**で，break 環境と同様に，定義は型環境と同じである．continue が実行された際に，次のループを実行するために，ループ実行前の型環境を記録しておくために使う．関数環境 $\Theta$，break 環境 $B$，continue 環境 $C$，型環境 $\Gamma$ のもとで，文 $s$ を実行すると，実行後の型環境が $\Gamma'$ になるという意味である．

**定義 12 (文)**

文の型付け規則を図 12 のように定義する.

$$\frac{}{\Theta;\ B;\ C;\ \Gamma \vdash \texttt{skip} \Rightarrow \Gamma'} \quad \text{(Sskip)}$$

$$\frac{\Gamma\ \vdash^w e_1 : t_1 \qquad \Gamma\ \vdash^r e_2 : t_2 \qquad \Gamma'\ \vdash^w e_1 : t_1' \qquad \Gamma'\ \vdash^r e_2 : t_2' \qquad \texttt{empty}(t_1) \qquad t_2 = t_1' + t_2'}{\Theta;\ B;\ C;\ \Gamma \vdash e_1 = e_2 \Rightarrow \Gamma'} \quad \text{(Sassign)}$$

$$\frac{\Theta(f) = \tilde{t} \to \tilde{t}' \qquad \texttt{empty}(t_0) \qquad \texttt{return}(f) = t_1}{\Theta;\ B;\ C;\ \Gamma,\ \texttt{temp} : t_0,\ \tilde{x} : \tilde{\tau} \vdash \texttt{temp} = f(\tilde{x}) \Rightarrow \Gamma,\ t : t_1,\ \tilde{x} : \tilde{\tau}'} \quad \text{(Scall\_set)}$$

$$\frac{\Theta(f) = \tilde{t} \to \tilde{t}'}{\Theta;\ B;\ C;\ \Gamma,\ \tilde{x} : \tilde{t} \vdash f(\tilde{x}) \Rightarrow \Gamma,\ \tilde{x} : \tilde{t}'} \quad \text{(Scall)}$$

$$\frac{o = 1 \qquad o' = 0 \qquad \texttt{empty}(t)}{\Theta;\ B;\ C;\ \Gamma,\ x : \texttt{Tpointer}(t,\ o) \vdash \texttt{free}(x) \Rightarrow \Gamma,\ x : \texttt{Tpointer}(t,\ o')} \quad \text{(Sfree)}$$

$$\frac{o = 0 \qquad o' = 1 \qquad \texttt{empty}(t)}{\Theta;\ B;\ C;\ \Gamma,\ \texttt{temp} : \texttt{Tpointer}(t, o)\ \vdash \texttt{temp} = \texttt{malloc}(e) \Rightarrow \Gamma,\ \texttt{temp} : \texttt{Tpointer}(t, o')} \quad \text{(Smalloc)}$$

$$\frac{\Gamma\ \vdash^r e_1 : t_1 \qquad \Gamma\ \vdash^r e_2 : t_2 \qquad \Gamma'\ \vdash^r e_1 : t_1' \qquad \Gamma'\ \vdash^r e_2 : t_2' \qquad t_1 + t_2 = t_3 + t_4}{\Theta;\ B;\ C;\ \Gamma \vdash \texttt{assert}(e_1,\ e_2) \Rightarrow \Gamma'} \quad \text{(Sassert)}$$

$$\frac{\Gamma \vdash^r e : t \qquad \Gamma' \vdash^r e : t'}{\Theta;\ B;\ C;\ \Gamma \vdash \texttt{assert\_null}(e) \Rightarrow \Gamma'} \quad \text{(Snull)}$$

$$\frac{\Theta;\ B;\ C;\ \Gamma \vdash s_1 \Rightarrow \Gamma'' \qquad \Theta;\ B;\ C;\ \Gamma'' \vdash s_2 \Rightarrow \Gamma'}{\Theta;\ B;\ C;\ \Gamma \vdash s_1;\ s_2 \Rightarrow \Gamma'} \quad \text{(Sseq)}$$

$$\frac{\Theta;\ B;\ C:\ \Gamma \vdash s_1 \Rightarrow \Gamma_1 \qquad \Theta;\ B;\ C;\ \Gamma \vdash s_2 \Rightarrow \Gamma_2 \qquad \Gamma_1 = \Gamma_2}{\Theta;\ B;\ C;\ \Gamma \vdash \texttt{if}(e)\ \texttt{then}\ s_1\ \texttt{else}\ s_2 \Rightarrow \Gamma_2} \quad \text{(Sif)}$$

$$\frac{\Theta;\ B;\ C;\ \Gamma \vdash s_1 \Rightarrow \Gamma_1 \qquad \ldots \qquad \Theta;\ C;\ \Gamma \vdash s_m \Rightarrow \Gamma_m \qquad \Gamma' = \Gamma_1 = \Gamma_2 = \cdots = \Gamma_m}{\Theta;\ B;\ C;\ \Gamma \vdash \texttt{switch}(e)\ \texttt{case}\ n_1 :\ s_1;\ \texttt{case}\ n_2 :\ s_2;\ \ldots\ \texttt{case}\ n_m :\ s_m; \Rightarrow \Gamma'} \quad \text{(SSWITCH)}$$

$$\frac{\Gamma';\ \Gamma;\ \Gamma \vdash s_1 \Rightarrow \Gamma_1 \qquad \Gamma';\ \Gamma;\ \Gamma_1 \vdash s_2 \Rightarrow \Gamma_2 \qquad \Gamma_2 = \Gamma}{\Theta;\ B;\ C;\ \Gamma \vdash \texttt{loop}(s_1)\ s_2 \Rightarrow \Gamma'} \quad \text{(SLOOP)}$$

$$\frac{}{\Theta;\ \Gamma;\ C;\ \Gamma \vdash \texttt{break} \Rightarrow \Gamma'} \quad \text{(SBREAK)}$$

$$\frac{}{\Theta;\ B;\ \Gamma;\ \Gamma \vdash \texttt{continue} \Rightarrow \Gamma'} \quad \text{(SCONTINUE)}$$

$$\frac{\Gamma \vdash e : t \qquad \Gamma' \vdash e : t' \qquad t = \texttt{return}(f) + t'}{\Theta;\ B;\ C;\ \Gamma \vdash \texttt{return}\ e \Rightarrow \Gamma} \quad \text{(SRETURN)}$$

図 12: 文の型付け規則

SCALL_SET は，関数の返り値を変数に代入する際の規則である．第 2 章の関数は返り値がなかったため，この型付け規則も定義されていなかった．変数に代入するため，その変数の所有権は全て 0 でなければならない．また代入後には，その変数の型と関数の返り値の型とが等しくなる．

SNULL は，assert_null の規則である．assert_null$(e)$ は，$e$ がヌルポインタであることを表すための命令で，この型システムでは，第 2 章の型システムと同様，ヌルポインタは任意の型をもつことができる．そのため，assert_null$(e)$ 実行後，$e$ は任意の型を持つことができる．

SSWITCH は，switch 文の規則で，SIF と同様，それぞれの分岐に関して型がつくかどうか調べる．それぞれの分岐終了後の型環境は，全て等しくなければならない．

SLOOP は，loop 文の規則である．$s_1$ が skip の時，while に対応し，それ以外の場合 for に対応する．まず break 環境が $\Gamma'$，continue 環境が $\Gamma$，型環境が $\Gamma$ の元で，$s_1$ に型がつくかどうか調べる．break はそれ以降の命令を実行せず，ループを抜けるという命令なので，$s_1$ で break が呼ばれるとループを抜け，実行後の環境になるように，break 環境に実行後の環境 $\Gamma'$ を入れる．continue は，それ以降の命令を実行せず，次のループを実行するという命令なので，$s_1$

で continue が呼ばれると次のループを実行できるように continue 環境にループ実行前の環境である $\Gamma$ を入れる．次に，break 環境が $\Gamma'$，continue 環境が $\Gamma$，型環境が $s_1$ 実行後の型環境 $\Gamma_1$ のもとで $s_2$ に型がつくかどうか調べる．これも $s_1$ の処理と同様である．$s_2$ を実行後，また $s_1$ を実行するので，$s_2$ の実行後の型環境 $\Gamma_2$ と，$s_1$ の実行前の型環境 $\Gamma$ は等しくなければならない．

SBREAK は，break 文の規則である．break 環境と型環境が等しい状態で，break を実行すると実行後の型環境は，任意の型環境になる．

SCONTINUE は，continue 文の規則である．break の規則と同様に，continue 環境と型環境が等しい状態で，continue を実行すると実行後の型環境は，任意の型環境になる．

SRETURN は，return 文の規則である．return($f$) は，関数 $f$ の返り値の型を表している．return 実行前の $e$ の型についている所有権を，関数の返り値と実行後の $e$ の型とに分配している．

**例 3 (型付け規則)**

図 13 のプログラムを例に挙げる．なお，x に関しては，所有権を持っていないため，省略する．

まず p を型環境に追加する．変数宣言時は全ての所有権が 0 でなければならないため，p の型は Tpointer(int, OConst(0)) である．7 行目から while が実行され，8 行目で malloc が呼ばれているため，所有権 1 が与えられ，p の型が Tpointer(int, Oconst(1)) になる．次に break が実行されているため，SBREAK から，break が実行される前の環境と break 環境が等しくなる．break 環境は，SLOOP から，実行後の型環境になっているため，ループを抜けた 19 行目の時点での p の型は，Tpointer(int, OCosnt(1)) になる．break を実行した後は任意の型環境になるため，13 行目の時点での p の型を Tpointer(int, Oconst(1)) とすれば，14 行目の free(p) が実行できるようになる．free(p) を実行すると所有権が 0 になるため，15 行目時点で p の型が Tpointer(int, Oconst(0)) になる．これが，ループ実行前の型環境と等しいため，while に正しく型がついていることがわかる．while 実行後の 20 行目の free(p) も p に所有権 1 が与えられているため実行ができる．実行後の p の型が Tpointer(int, Oconst(0)) になる．関数の終了時点で，関数内で宣言されているポインタ p の所有権が 0 になっているため，このプログラム全体に正しく型がついていることがわかる．

```
1  int main() {
2      int x;
3      int *p;
4
5      x = 0;
6      /* p: Tpointer(int, Oconst(0)) */
7      while (1) {
8          p = malloc(sizeof (int));
9          /* p: Tpointer(int, Oconst(1)) */
10         if (x > 5) {
11             break;
12         }
13         /* p: Tpointer(int, Oconst(1)) */
14         free(p);
15         /* p: Tpointer(int, Oconst(0)) */
16         x = x + 1;
17     }
18
19     /* p: Tpointer(int, Oconst(1)) */
20     free(p);
21     /* p: Tpointer(int, Oconst(0)) */
22     return 0;
23 }
```

図 13: 型付け規則の例

# 第4章　検証器の実装

この章では，第3章で定義した型システムに基いた検証器の実装について述べる．

## 4.1　概要

検証器の実装にあたっては，3.1節で説明したCompCertを拡張する形で実装した．本研究では，下図のConstraint Generator, Constraint Reducer, Type-error Slicerを実装した．

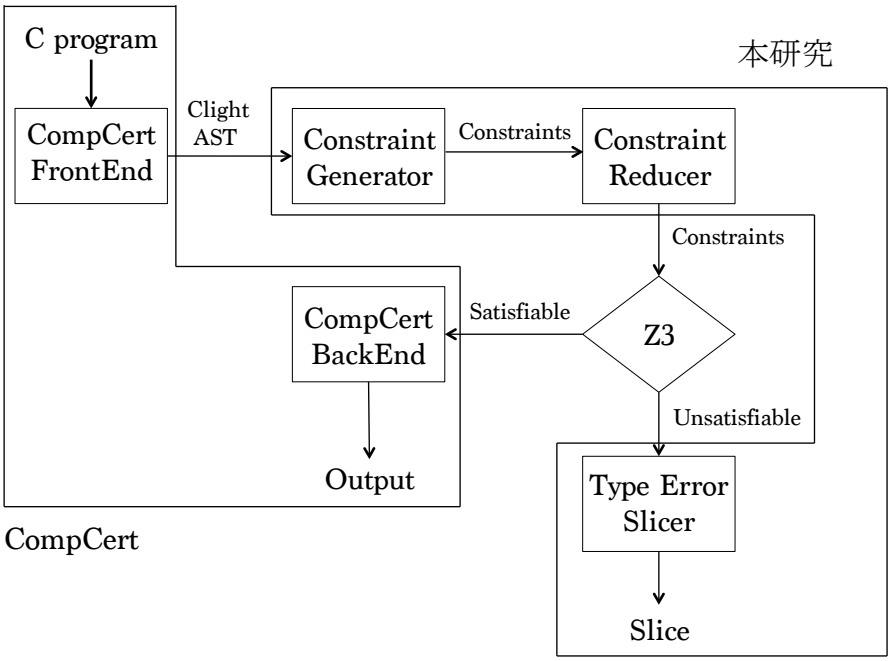


図14: 実装の全体図

CompCertは，C言語のプログラムを受け取るとまず中間言語であるClightに変換する．検証器はそのClightの抽象構文木を入力として，型付け規則(図12)に従い制約式を生成する．そしてその制約式をSMTソルバが解くことのできる形に変換した後，SMTソルバに渡す．制約式が充足可能であればプログラムに型がつき，メモリ操作に関する誤りが無いことがわかるため，Clihgt ASTをそのままCompCertに返す．充足不能の場合，SMTソルバはunsat coreを返す．

unsat core とは，充足不能の原因となっている制約式の部分集合である．型エラースライサーは，unsat core を受け取り，unsat core に含まれている制約式を生成した命令の行番号を，スライスとして出力する．

## 4.2 制約生成

この節では，制約生成 (Generate) について説明する．まず，扱う制約式について定義する．

**定義 13 (制約式)**
制約式を図 15 のように定義する．

$$
\begin{aligned}
c\,(constraints) \;::=\; & \texttt{Le}(o_1,\, o_2,\, loc) \\
& \mid \texttt{Lt}(o_1,\, o_2,\, loc) \\
& \mid \texttt{Eq}(o_1,\, o_2,\, loc) \\
& \mid \texttt{Empty}(id,\, n,\, loc) \\
& \mid \texttt{Teq}((id,\, n)\; list,\; (id,\, n)\; list,\; loc)
\end{aligned}
$$

図 15: 制約式

各制約式の $loc$ は，どの命令から生成された制約式なのかを記録しておくための情報である．制約式が充足不能となった場合，unsat core に含まれている制約式がどの命令から生成されたものなのかを調べ，その命令の行番号を抽出するために使用する．

`Le` は $o_1 \leq o_2$，`Lt` は $o_1 < o_2$，`Eq` は $o_1 = o_2$ にそれぞれ対応している．`Empty` は，構造体の名前を表す $id$，同じ名前でも違う所有権をもつ構造体を区別する識別子 $n$ を受け取る．構造体中の所有権が全て 0 であるという制約式を表している．`Teq` は，構造体同士の所有権が等しいという制約式を表している．各引数がリストになっているのは，型の定義で新しく追加した `Tplus` を引数として取れるようにするためである．これにより，構造体を足し算したもの同士が等しいという制約式も表すことができる．

制約生成の流れは，以下の通りである．入力として Clight プログラムの抽象構文木を受け取る．図 10 で定義したように，プログラムは関数定義のリストに

なっている

1. 関数定義から関数環境を生成する
2. 1 の操作を各関数定義に適用する
3. 引数，局所変数を型環境に追加する
4. 関数本体に対して所有権に関する制約式を生成する
5. 3, 4 を各関数定義に適用する

まず，関数定義から関数環境を生成する．関数環境は，関数名から関数の型への写像で定義されている．関数の型 function は，引数の型 $t_1\ list$ と，関数実行後の引数の型 $t_2\ list$，返り値の型 $t$ の組で定義されている．返り値の型 $t$ と引数の型の $t_1\ list$ は，関数定義に使われている返り値の型 $t$ と引数のリスト $dcl_1$ をそのまま使えば良い．この際，引数の型にポインタ型が含まれている場合，フレッシュな所有権変数を割り当てる．また，構造体が含まれている場合も，フレッシュな識別子を割り当てる．そして，実行前の引数の型にフレッシュな所有権変数と構造体にフレッシュな識別子を割り当てたものを関数実行後の引数の型とする．この操作を各関数定義に対して行う．

**例 4 (関数環境の生成)**

例えば,

```
struct list {
    struct list *next;
    int n
};

void free_list(struct list *l)
```

のように，構造体と関数が定義されている場合，free_list には,

```
function(
  struct(list,
         [(list, comp_ptr(list, 0, Ovar(0)));
          (list, int)]),
  struct(list,
         [(list, comp_ptr(list, 1, Ovar(1)));
         (list, int)]),
```

void)
のような型がつく．1 引数目の Tstruct が free_list の引数の型，2 引数目の Tstruct が free_list を実行した後の引数の型，3 引数目の void が free_list の返り値の型になっている．このように，関数実行後の引数の型は，実行前の引数の型内に含まれる識別子にフレッシュなものを割り当てた型にする．そして，(free_list, function(...)) を関数環境に追加する．

関数環境の生成後，関数本体について制約式を生成する．まず，関数定義の引数 $dcl_1$，局所変数 $dcl_2$ から型環境を生成する．型環境は，変数名から変数の型への写像で定義されているので，$dcl_1$, $dcl_2$ をそのまま使えば良い．この際，関数環境の時と同様に，型内に所有権や構造体の識別子が含まれている場合は，他と被らないように新しい識別子を割り当てる．また，関数本体実行前の局所変数の所有権は全て 0 でないといけないという制約式をこの段階で生成する．

この型環境を最初の型環境として，関数本体 $s$ に対して，図 12 に従い所有権に関する制約式を生成する．基本的には，型付け規則の前提部分を制約式にしていく．例えば，前提部分の $o = 0$ などは，そのまま Eq($o$, OConst($0$)) とすればよい．

empty($t$) は，型 $t$ 内の所有権が全て 0 であるということを表している．そこで，型 $t$ 内の所有権が全て 0 であるという制約式を生成する関数 empty_type を定義する．int や float など，所有権が割り当てられていない型は，全ての所有権が 0 になっているとみなして良い．pointer($t$, $o$) は，Eq($o$, Oconst($0$)) という制約式を生成した後，$t$ に対して再帰的に empty_type を適用していく．struct($id$, $fl$) は，$fl$ の各要素に対して empty_type を適用する．comp_ptr($id, n, o$) は，pointer と同様に Eq($o$, Oconst($0$)) という制約式を生成した後，参照先に対して所有権が 0 であるという制約式を生成する．しかし，comp_ptr は，再帰を含む構造体内での自己参照をしているポインタのため，empty_type をそのまま適用していくと，無限ループに陥ってしまう．そこで，図 15 で定義した Empty($id, n, loc$) で，comp_ptr の参照先の所有権が全て 0 である，ということを表す．

$t_1 + t_2$ は，型 $t_1$ と型 $t_2$ を足した型を表している．型同士の足し算は，所有権同士の足し算として定義されている．そこで，型 $t_1$ 内の所有権と型 $t_2$ 内の所有権を足す関数 add_type を定義する．基本的には，同じ型同士しか足し算は行うことができない．所有権が割り当てられていない int などに関しては，empty_type

と同様に何もせず，そのまま $t_1$ を返す．pointer$(t'_1,\ o_1)$ と pointer$(t'_2,\ o_2)$ に関しては，$t'_1$ と $t'_2$ に対して再帰的に add_type を適用し，その返り値を $t'$ とすると，pointer$(t',\ \mathtt{Oplus}(o_1,\ o_2))$ を返す．struct 同士の足し算に関しては，まず構造体の名前が一緒かどうかを確認する．同じ場合は，フィールドリストの各要素に対して add_type を適用する．名前が違う場合は，足し算を行うことはできないので，エラーを返す．comp_ptr 同士の足し算に関しても，まず同じ構造体内のポインタであるかどうかを確認する．また，empty_type と同様に comp_ptr の参照先に関して再帰的に add_type を実行すると，無限ループに陥ってしまうため，Tplus を使い，comp_ptr 同士の足し算を表す．

更に，comp_ptr はポインタ型のため，comp_ptr$(id,\ n,\ o_1)$ と pointer$(t,\ o_2)$ の足し算を定義することができる．この場合は，comp_ptr を構造体を参照しているポインタと読み替える．まず，comp_ptr$(id,\ n,\ o_1)$ を一回展開する．展開するには，構造体の名前 $id$ と構造体内の所有権を区別する識別子 $n$ を使う．まず $id$ と $n$ の組で，今までに展開されているかどうかを調べる．展開されている場合は，以前展開された構造体をそのまま使う．展開されていない場合，構造体内の pointer と comp_ptr の所有権と識別子をフレッシュなものに書き換えた新たな構造体をつくり，その構造体を参照先にする．この時，$id$ と $n$ の組が新しく作った構造体を参照しているという情報を記録しておく．これにより，$id$ と $n$ の組は常に同じ構造体に展開されることになる．

$t_1 = t_2$ は，型 $t_1$ と型 $t_2$ が等しいという条件を表している．型同士の等しさは，所有権同士の等しさで定義されている．そこで，型 $t_1$ 内の所有権と型 $t_2$ 内の所有権が等しいという制約式を生成する関数 eq_type を定義する．定義は，empty_type や add_type と同様にすればよい．所有権が割り当てられていない型に関しては，何もしない．pointer 同士に関しては，再帰的に eq_type を適用する．comp_ptr 同士に関しては，再帰的に適用すると無限ループに陥るため，Teq を使い，等しさを表す．comp_ptr と pointer に関しては，comp_ptr を構造体を参照しているポインタとみなす．comp_ptr を展開する際の処理も同様である．

**例 5 (制約生成)**

構造体の定義は例 4 と同じとする．

型環境が

```
  l: pointer(struct(list,
                    [(list, comp_ptr(list, 0, Ovar(0)));
                     (list, int)]),
                    Ovar(1))
```

のもとで,

```
  free(l)
```

を実行すると，実行後の型環境が

```
  l: pointer(struct(list,
                    [(list, comp_ptr(list, 0, Ovar(0)));
                     (list, int)])struct(list,
                    [(list, comp_ptr(list, 0, Ovar(0)));
                     (list, int)]),
                    Ovar(2))
```

になり，制約式として

```
  Eq(Ovar(1), Oconst(1), loc);
  empty_type(struct(list,
                    [(list, comp_ptr(list, 0, Ovar(0)));
                     (list, int)]));
  Eq(Ovar(2), Oconst(0), loc);
```

が生成される．更に empty_type からは,

```
  Eq(Ovar(0), Oconst(0), loc);
  Empty(list, 0, loc);
```

が生成される．loc には，free(l) のプログラム中での行番号が入る．

関数本体に対して制約式の生成が終わると，関数本体実行後の型環境が得られる．この型環境内に含まれる引数の型は，関数環境内に追加されている関数実行後の引数の型と一致しなければならない．また，関数本体実行後には関数の局所変数の所有権は全て 0 になっていなければならない．この 2 つの条件に関しても制約式を生成する．この操作を各関数定義に対して適用する．

以上の操作によって得られた制約式を集めたものが制約式の集合となる．この制約式の集合を SMT ソルバで解く．Le，Lt，Eq は不等式の形をしているため，そのまま SMT ソルバで解くことができるが，Empty，Teq はこのままでは

SMT ソルバは解くことができない．そのため，SMT ソルバに解ける形に変換する必要がある．

## 4.3 制約式変換

この節では，前節で生成した制約式を SMT ソルバに解ける形に変換する手法について述べる．制約式 Empty と Teq はそのままでは SMT ソルバは解くことができないため，不等式の形まで変換する必要がある．制約式の変換は，前処理と実際に変換する段階にわけられる．

再帰のある構造体に関する制約式は前節で述べたように，ポインタ型と同様に再帰的に展開して生成するという方法で生成すると，無限に展開されてしまう．そのため，どこかで展開をとめる必要がある．先行研究では，テンプレート型 $(\mu\alpha.\,\mathtt{ref}_{f1})\,\mathtt{ref}_{f0}$ を使用して，展開回数を制限していた．この型は，ある展開回数までは別々の所有権が割り当てられているが，それ以降は同じ所有権が割り当てられているとみなすという近似をしていることになる．しかし，先行研究で実装された検証器では，十分に多い回数展開を行って制約式を生成するだけで，それ以降は同じ所有権が割り当てられているという部分は実装していなかった．

本研究では，前処理の段階でプログラム中で何回か展開した先のメモリ領域に対して操作を行っている場合，その回数分は展開を行う．そして，変換する段階でそれ以降に関しては同じ所有権が割り当てられているとみなすという処理を行う．これにより，より理論に基づいた実装になっている．

### 4.3.1 前処理

前処理の段階では，制約生成の段階で既に展開されたことのある comp_ptr を展開する．制約生成の，add_type や eq_type 内で，$\mathtt{comp\_ptr}(id,\ n,\ o)$ と $\mathtt{pointer}(t,\ o)$ との間の足し算や型の等しさなどを求めると，$\mathtt{comp\_ptr}(id,\ n,\ o)$ は一回展開され，$id$ と $n$ の組がどの構造体に展開されるかという情報は記録される．Empty や Teq 内の $id$ と $n$ の組が制約生成の段階で記録されているかどうかを調べ，記録されている場合は展開先の構造体の $\mathtt{comp\_ptr}(id,\ n',\ o')$ を使い，書き換えるということを行う．この操作を $id$ と $n'$ の組に関しても再帰的に適用していくことで，制約生成の段階で展開された回数分，つまりプログラム中で展開された回数分は，構造体を展開するというということを実現している．

**例 6 (前処理: Empty)**
構造体の定義は例 4 と同じである．制約式

```
Empty(list, 1, loc);
```

が生成され，制約生成の段階で，

```
(list, 1) -> struct(list,
                    [(list, comp_ptr(list, 2, Ovar(2)));
                     (list, int)])
(list, 2) -> struct(list
                    [(list, comp_ptr(list, 3, Ovar(3)));
                     (list, int)])
```

という情報が記録されていた場合，前処理によって制約式は

```
Empty(list, 3, loc);
Eq(Ovar(2), Oconst(0), loc);
Eq(Ovar(3), Oconst(0), loc);
```

という制約式に書き換えられる．

Teq の場合も同様に展開を行うのだが，Teq の引数はリストになっているため，複数の $id$ と $n$ の組が出現することがある．この場合は，リスト内の全ての $id$ と $n$ の組が制約生成の段階で，展開されている場合のみ書き換えを行い，リスト内の要素の一部だけ記録されている場合は展開を行わない．

### 4.3.2 制約の不等式への変換

前処理の段階で，制約生成の段階で記録された $id$ と $n$ の情報を使って構造体を展開し制約式を書き換えたため，変換の段階では記録されていない $id$ と $n$ に関しての処理を行う．

具体的には，前処理が終わった段階で記録されていない $id$ と $n$ の組は全て，自分自身と同じ型の構造体を参照しているとみなす．そしてその情報を使い，前処理と同様に制約式を書き換えていく．制約式が書き換え前と書き換え後とで変化しなくなった時点で，書き換えをやめる．この操作が終わると，Empty と Teq からは新しい制約式が生成されることはないため，制約式から Empty と Teq を消去してもよい．消去後の制約式には，Lt，Le，Eq しか入っていないため，SMT ソルバで解ける形になっている．

**例 7 (変換: Teq)**

構造体の定義は例 4 と同じである．制約式

```
Teq([(list, 1)], [(list, 2); (list, 3)])
```

が生成され，前処理の段階で

```
(list, 1) -> struct(list,
                    [(list, comp_ptr(list, 4, Ovar(4)));
                     (list, int)])
```

という情報が記録されてい場合，(list, 2) と (list, 3) に関して情報が記録されていないため，前処理の段階ではこの制約式の書き換えは行われない．変換の段階では，まず (list, 2) と (list, 3) に関して自分自身を参照しているという情報を追加する．

```
(list, 1) -> struct(list,
                    [(list, comp_ptr(list, 4, Ovar(4)));
                     (list, int)])
(list, 2) -> struct(list,
                    [(list, comp_ptr(list, 2, Ovar(5)));
                     (list, int)])
(list, 3) -> struct(list,
                    [(list, comp_ptr(list, 3, Ovar(6)));
                     (list, int)])
(list, 4) -> struct(list,
                    [(list, comp_ptr(list, 4, Ovar(4)));
                     (list, int)])
```

この情報を使い制約式を書き換えると

```
Teq([(list, 4)], [(list, 2); (list, 3)]);
Eq(Ovar(4), Oplus(Ovar(5), Ovar(6)));
```

になる．そして，Teq からはもう新しい制約式は生成されないため Teq は消去され，

```
Eq(Ovar(4), Oplus(Ovar(5), Ovar(6)));
```

だけになる．

## 4.4 制約解消

制約式変換を終えた制約式は，SMT ソルバで解ける形になっている．その制約式を SMT ソルバで解くことで，制約式が充足可能かどうか判定することができる．今回の実装では SMT ソルバには，Z3 [6] を使用した．ソルバが制約を充足可能と判定した場合は，プログラムに型がつき，メモリ操作に関する誤りが発生していないことがわかる．ソルバが充足不能と判定した場合は，同時に unsat core を返す．unsat core とは，制約式の充足不能性の原因となっている制約式の部分集合である．unsat の場合は，プログラム中のどこかでメモリ操作に関する誤りがあり，その箇所が型エラーの原因となっている．その箇所を特定するのは，小さなプログラムでは容易だが，大きなプログラムだと困難である．そこで，unsat core を用いて型エラーの原因となっている命令の集合をプログラム中から切り出して，表示する．この情報は，プログラマが型エラーの原因を特定するのに役立つと考えられる．

## 4.5 型エラースライサー

型エラースライサーは，制約式の集合と unsat core を入力として，型エラーの原因に関係している命令の行番号をスライスとして出力する．unsat core には制約式の充足不能性の原因，つまり型エラーの原因となっている制約式が含まれている．そこで，unsat core に含まれている各制約式がそれぞれどの命令から生成されたものなのかを調べる．制約式には，その制約式を生成した命令の行番号 $loc$ が含まれているので，unsat core に含まれている各制約式の $loc$ をまとめたものをスライスとすればよい．

今回ソルバとして使用した Z3 は，unsat core を制約式の名前で返す．そのため，Z3 に制約式を渡す際に，各制約式に名前をつけ，どの制約式にどういう名前をつけたのかを記録しておくことで，Z3 が unsat core として返してきた制約式の名前から制約式を特定し，$loc$ を調べることができる．

**例 8 (型エラースライサー)**

プログラム 1 (付録参照) を例に挙げる．このプログラムは，再帰を含む構造体 `list` を使って単方向リストの操作を行っているプログラムである．`make_list` で単方向リストの各要素に新しいメモリ領域を割り当て，`free_all_list` でそれら全てを解放する．しかし，`free_all_list` 内で，`free` を実行し忘れてい

るため，メモリリークが発生している．

このプログラムから制約式を生成し，Z3 で解くと

```
unsat
(c_70 c_67 c_66 c_64 c_54 c_52
 c_49 c_48 c_32 c_31 c_8 c_3 c_7)
```

を返す．メモリリークが発生しているため制約式は充足不能である．また充足不能のため，unsat core を一緒に返している．unsat core の各要素は，制約式の名前である．型エラースライサーは，これらの名前がついた制約式を生成した命令を探しそれの行番号をスライスとして出力する．

```
[11; 18; 30; 32; 39; 40; 43]
```

30 行目の malloc で新しいメモリ領域を割当て，32 行目の return でその変数を返り値としているため，39 行目で make_list を呼び出しその返り値を代入している変数 l には，所有権が与えられているはずである．しかし，40 行目で free_all_list を呼び出しているにもかかわらず，43 行目で，関数の終了時には関数の局所変数の所有権，つまり l 内の所有権は全て 0 になっていないといけないという条件をみたすことができていない．更に 18 行目の free_all_list もスライスに含まれていることから，free_all_list の実装が間違っているのではないかと，プログラマは推測することができる．

# 第5章　予備実験

この章では，本研究で実装した検証器に対して行った予備実験について述べる．

表 1: 実験結果

| Program | LOC | Time_g | SIZE_g | Time_r | SIZE_r | Time_s |
|---|---|---|---|---|---|---|
| sl_app | 55 | 0.165 | 200 | 0.748 | 193 | 34.0 |
| sl_free | 44 | 0.112 | 114 | 0.505 | 110 | 26.5 |
| sl_merge | 60 | 0.280 | 250 | 1.395 | 237 | 37.5 |
| sl_mut | 60 | 0.140 | 149 | 0.495 | 145 | 30.9 |
| sl_reverse | 63 | 0.173 | 222 | 0.901 | 205 | 33.2 |
| sl_search | 60 | 0.189 | 189 | 0.654 | 176 | 36.8 |

表 1 は実験結果をまとめたものである．各列の意味は以下のとおりである．

- LOC: プログラムの行数
- Time_g: 制約生成にかかった時間 (msec)
- SIZE_g: 生成された制約式の数
- Time_r: 制約変換にかかった時間 (msec)
- SIZE_g: 変換後の制約式の数
- Time_s: 制約解消にかかった時間 (msec)

実験に用いた計算環境は CPU: Intel Core i7 1.7 GHz，MEM:8GB，OS:OSX 10.9.5 である．使用したソフトの各バージョンは，CompCert: 2.4， OCaml: 4.02.1， Z3: 4.3.2 である．

いずれのプログラムも，単方向リストを操作するプログラムで，末永らが先行研究で実装した検証器に対して行った予備実験で使われたプログラムと同等のものである．各プログラムには，assert_null($p$) という注釈を手動で適当な箇所に挿入してある．これは，第 3 章で述べたように，ポインタの参照先を null とみなして，任意の型をもてるようにするための命令である．

sl_app は，リストを 2 つ生成した後，片方のリストをもう片方のリストの最後尾に繋げたリストを生成し，それを繋げたリストを生成し，そのリストを解放する．sl_free は，単純にリストを生成しそれを解放する．sl_merge は，リストを 2 つ生成した後，それらのリストの先頭から要素をランダムに選んで繋げたリストを生成し，そのリストを解放する．sl_mut は，sl_free と同じ操作を行っているが，相互再帰関数を使って解放を行っている．sl_reverse は，リスト

を生成した後，そのリストの順番を逆にし，そのリストを解放する．sl_search は，リストを生成した後，リストから要素を一つ取り出してその要素を除いたリストを生成し，取り出した要素とそれを除いたリストを解放する．

いずれのプログラムもサイズは大きくないが，非常に高速に検出することができている．生成された制約式が変換によって数が減っているのは，変換の際に生成された制約式の中に，同じ制約式があるとそれを消す操作を行っているためである．

また，各プログラムに対してメモリリークが発生するように，free 命令を消すなどの書き換えを行い，実験を行った．結果として，全てのプログラムに対して検証器は充足不能と判定し，エラーを正しく検出していることを確認した．

# 第6章　関連研究

本研究では，末永らが提案したメモリリークを静的に検出する型システム [2] を制御文で拡張しした．また，その拡張した型システムに基いて検証器を実装した．検証器の実装においては，Leory らによって実装された C 言語のコンパイラ CompCert [4, 7] を拡張する形で実装した．また型システムの拡張も，CompCert で使われている中間言語 Clight [3] を元にしている．

CompCert を使用した他の研究としては [8] などがある．従来の CompCert は他のモジュール内の関数の呼び出しの正しさはは保証しない．そこで，この研究は他のモジュール内の関数の呼び出しなどの正しさも保証する Composite CompCert を提案している．

本研究で，SMT ソルバとして使用した Z3 を利用した研究として [9] などがある．これは，Z3 を利用して，モデル検査の分野で使用される補間を求める手法を提案している．

型エラースライサーは，静的型付けの行われている関数型言語に対して，型エラーが発生した際に，型エラーに関係しているプログラム地点をスライスとして出力するツールである [10, 11]．更に，型エラースライサーは本来，関数型言語を対象としていたが，近年 Java などの言語に対しても研究が行われている [12]．一般の静的型付け言語において，型推論はプログラムを抽象構文木に変換した後，各ノードに対して型付け規則に基いて型がつくかどうか確認していき，失敗した場合，失敗したノードを 1 つだけ型エラーの原因として出力する．しかし，型エラーの原因が 1 つしか挙げられないため，プログラマは型エラーの原因を正確に特定することができない．そこで型エラースライサーは，各ノードから生成された制約式にラベルをつけ，制約式全体が充足不能とわかった場合，充足不能となっている最小の制約式の集合を抽出し，その集合に含まれている制約式のラベルからプログラム地点を特定し，スライスとして出力するということをしている．本研究の型エラースライサーもこの考えに基づいて実装をしている．第 4 章で述べたように，制約式の充足不能の原因を抽出する際に，SMT ソルバが返す unsat core を用いている．また，これまでの型エラースライサーは単純型を対象としていたが，本研究では，所有権型を対象としている．[13, 14] では，並行プログラミングにおけるレースの原因やデッドロックの原因を型エラースライサーを用いて特定しているが，制約式の充足不能の原

因特定に unsat core は用いられていない.

他のメモリリークの検証器は，Xie らによる SAT ソルバを用いた [15] や，Cherem らによるグラフを用いた [16] などがある．しかし，両方ともループを健全な形で扱うことができない.

Hoare 論理を拡張して，領域の確保や解放，領域へのアクセスなどメモリに関する操作をを扱えるようにした体系として Separation Logic [17] が提案されている．この Separation Logic に基いて，メモリリークの検出やエイリアス解析などを行う手法として Shape analysis [18–20] という手法がある．しかし，これらの手法は検出に非常に時間がかってしまう.

# 第7章　おわりに

本研究では，末永らが提案したメモリリークを静的に検出するための型システムを制御文で拡張し，その拡張した型システムに基いて検証器を実装することで，検証器の信頼度を高めた．また，型エラースライサーを組み込むことで，検証器の利便性も高めた．更に，予備実験を行いいくつかのプログラムに対して，メモリリークを正しく検出できることを確かめた．

今後の課題として，まず扱える構文を増やすことが挙げられる．第3章で述べたように，配列型，キャストやアドレス演算子などの式，goto や label などの文，大域変数は，本研究で拡張した型システムではまだ扱っていない．goto や label などは，break や continue の考えを応用すれば扱えるようになると考えられる．また，拡張した型システムの健全性の証明を行っていない．元の型システムでは，プログラムに型がつけば，プログラム中でメモリリークが起きてないことが保証されていたが，今回拡張した制御文などの構文に関しては，その性質の証明ができていない．更に，より大きなプログラムに対して実験を行うことが挙げられる．今回の予備実験ではいずれも小さなプログラムに対してのみ実験を行ったので，より大きなプログラムでもメモリリークを正しく検出できるか，どの程度早く検出できるかなどを確かめる必要がある．同時に，型エラースライサーが型エラーの原因の特定にどの程度有効なのかを確認し，改良していく必要がある．

# 謝辞

はじめに，本研究に際し，テーマの決定から，検証器の実装，本論文の執筆に至るまで，日頃から多大なる御指導を賜りました末永幸平准教授に深く感謝致します．また，本研究に有益なご助言を下さいました五十嵐淳教授，馬谷誠二助教に御礼を申し上げます．更に，内見を引き受けてくださり，本研究にコメントを下さいました高瀬英希助教に感謝申し上げます．最後に，日頃の雑談に付き合ってもらうだけでなく，本研究に様々な指摘まで下さった五十嵐・末永研究室の皆様に感謝致します．

# 参考文献


[1] Kernighan, B. W. and Ritchie, D.: *The C Programming Language*, Prentice-Hall (1978).

[2] Suenaga, K. and Kobayashi, N.: Fractional Ownerships for Safe Memory Deallocation, *Programming Languages and Systems, 7th Asian Symposium, APLAS 2009, Seoul, Korea, December 14-16, 2009. Proceedings* (Hu, Z.(ed.)), Lecture Notes in Computer Science, Vol. 5904, Springer, pp. 128–143 (2009).

[3] Blazy, S. and Leroy, X.: Mechanized Semantics for the Clight Subset of the C Language, *J. Autom. Reasoning*, Vol. 43, No. 3, pp. 263–288 (2009).

[4] Leroy, X.: Formal verification of a realistic compiler, *Commun. ACM*, Vol. 52, No. 7, pp. 107–115 (2009).

[5] Bertot, Y. and Castéran, P.: *Interactive Theorem Proving and Program Development - Coq'Art: The Calculus of Inductive Constructions*, Texts in Theoretical Computer Science. An EATCS Series, Springer (2004).

[6] de Moura, L. M. and Bjørner, N.: Z3: An Efficient SMT Solver, *Tools and Algorithms for the Construction and Analysis of Systems, 14th International Conference, TACAS 2008, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2008, Budapest, Hungary, March 29-April 6, 2008. Proceedings* (Ramakrishnan, C. R. and Rehof, J.(eds.)), Lecture Notes in Computer Science, Vol. 4963, Springer, pp. 337–340 (2008).

[7] Krebbers, R., Leroy, X. and Wiedijk, F.: Formal C Semantics: CompCert and the C Standard, *Interactive Theorem Proving - 5th International Conference, ITP 2014, Held as Part of the Vienna Summer of Logic, VSL 2014, Vienna, Austria, July 14-17, 2014. Proceedings* (Klein, G. and Gamboa, R.(eds.)), Lecture Notes in Computer Science, Vol. 8558, Springer, pp. 543–548 (2014).

[8] Stewart, G., Beringer, L., Cuellar, S. and Appel, A. W.: Compositional CompCert, *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2015, Mumbai, India, January 15-17, 2015* (Rajamani, S. K. and Walker, D.(eds.)), ACM, pp. 275–287 (2015).

[9] McMillan, K. L.: Interpolants from Z3 proofs, *International Conference on*

*Formal Methods in Computer-Aided Design, FMCAD '11, Austin, TX, USA, October 30 - November 02, 2011* (Bjesse, P. and Slobodová, A.(eds.)), FMCAD Inc., pp. 19–27 (2011).

[10] Haack, C. and Wells, J. B.: Type Error Slicing in Implicitly Typed Higher-Order Languages, *Programming Languages and Systems, 12th European Symposium on Programming, ESOP 2003, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2003, Warsaw, Poland, April 7-11, 2003, Proceedings* (Degano, P.(ed.)), Lecture Notes in Computer Science, Vol. 2618, Springer, pp. 284–301 (2003).

[11] Dinesh, T. B. and Tip, F.: A Slicing-Based Approach for Locating Type Errors, *Proceedings of the Conference on Domain-Specific Languages, DSL'97, Santa Barbara, California, USA, October 15-17, 1997* (Ramming, C.(ed.)), USENIX (1997).

[12] Boustani, N. E. and Hage, J.: Improving type error messages for generic java, *Proceedings of the 2009 ACM SIGPLAN Symposium on Partial Evaluation and Semantics-based Program Manipulation, PEPM 2009, Savannah, GA, USA, January 19-20, 2009* (Puebla, G. and Vidal, G.(eds.)), ACM, pp. 131–140 (2009).

[13] 飯村枝里, 小林直樹, 末永幸平: 型に基づくレース解析とその結果表示のためのスライシング, プログラミングおよびプログラミング言語ワークショップ (2007).

[14] 飯村枝里, 小林直樹, 末永幸平: 型エラースライシングによるデッドロックの原因特定, 情報処理学会論文誌プログラミング (PRO), Vol. 1, No. 2, pp. 71–84 (2008).

[15] Xie, Y. and Aiken, A.: Context- and path-sensitive memory leak detection, *ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pp. 115–125 (2005).

[16] Cherem, S., Princehouse, L. and Rugina, R.: Practical memory leak detection using guarded value-flow analysis, *ACM SIGPLAN*, pp. 480–491 (2007).

[17] Reynolds, J. C.: Separation Logic: A Logic for Shared Mutable Data Structures, *17th IEEE Symposium on Logic in Computer Science (LICS 2002), 22-25 July 2002, Copenhagen, Denmark, Proceedings*, IEEE Computer Society,

pp. 55–74 (2002).

[18] Berdine, J., Calcagno, C., Cook, B., Distefano, D., O'Hearn, P. W., Wies, T. and Yang, H.: Shape Analysis for Composite Data Structures, *Computer Aided Verification, 19th International Conference, CAV 2007, Berlin, Germany, July 3-7, 2007, Proceedings* (Damm, W. and Hermanns, H.(eds.)), Lecture Notes in Computer Science, Vol. 4590, Springer, pp. 178–192 (2007).

[19] Distefano, D., O'Hearn, P. W. and Yang, H.: A Local Shape Analysis Based on Separation Logic, *Tools and Algorithms for the Construction and Analysis of Systems, 12th International Conference, TACAS 2006 Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2006, Vienna, Austria, March 25 - April 2, 2006, Proceedings* (Hermanns, H. and Palsberg, J.(eds.)), Lecture Notes in Computer Science, Vol. 3920, Springer, pp. 287–302 (2006).

[20] Guo, B., Vachharajani, N. and August, D. I.: Shape analysis with inductive recursion synthesis, *Proceedings of the ACM SIGPLAN 2007 Conference on Programming Language Design and Implementation, San Diego, California, USA, June 10-13, 2007* (Ferrante, J. and McKinley, K. S.(eds.)), ACM, pp. 256–265 (2007).

# 付録

## A.1 プログラム例

```
1  struct list{
2      struct list *next;
3      int e;
4  };
5
6  struct list *malloc(unsigned int size);
7  void free(void*);
8  void assert_null(void*);
9
10 void free_all_list (struct list *l) {
11     struct list *p;
12
13     if (l == '\0') {
14         assert_null(l);
15         return;
16     } else {
17         p = l->next;
18         free_all_list(p);
19         // free(l);
20     }
21 }
22
23 struct list *make_list(unsigned int n) {
24     struct list *ret;
25
26     if (n == 0) {
27         assert_null (ret);
28         return ret;
29     } else {
```

```
30          ret = malloc(sizeof (struct list));
31          ret->next = make_list(n-1);
32          return ret;
33      }
34  }
35
36  int main() {
37      struct list *l;
38
39      l = make_list(3);
40      free_all_list(l);
41
42      return 0;
43  }
```

Program 1: 再帰を含む構造体