ATTAPL 輪講
5.1 - 5.3
(2006/11/15)

米澤研 M2 佐藤秀明

PCC: Proof-Carrying Code

- 怪しげなコードの安全性を検証する手段
  - 「安全」の定義はコードの利用者側で設定可能
    - 型安全性など
- コードの作成者側がコードの補足説明を添付
  - 利用者側での安全性検証が容易

PCC の実装いろいろ

- JavaVM/.NET verifier
  - 型宣言情報を付加
- Typed Assembly Language (前章)
  - loop invariant を付加
- Foundational PCC [Appel, 2001]
  - 安全であることの証明を (ほぼ) まるごと付加
- Touchstone PCC [Necula, 1998]
  - Foundational PCC よりコード作成者の負担小
  - 大きなプログラムにも適用可能

以降、Touchstone PCC について説明

Touchstone PCC の概念図

- Figure 5-1 (p179) を参照

- Agent: 対象コード
  - Executable content: コード本体
  - Checking support: コードの補足説明
- Host system: コード実行環境
  - VCGen: 検証すべき条件 (VC: verification condition) をコード本体から抽出
  - Checker: VC が成り立つことを検証
  - Safety policy: 利用者側が定義した安全性基準

PCC のコード例

- OCaml で表現すると

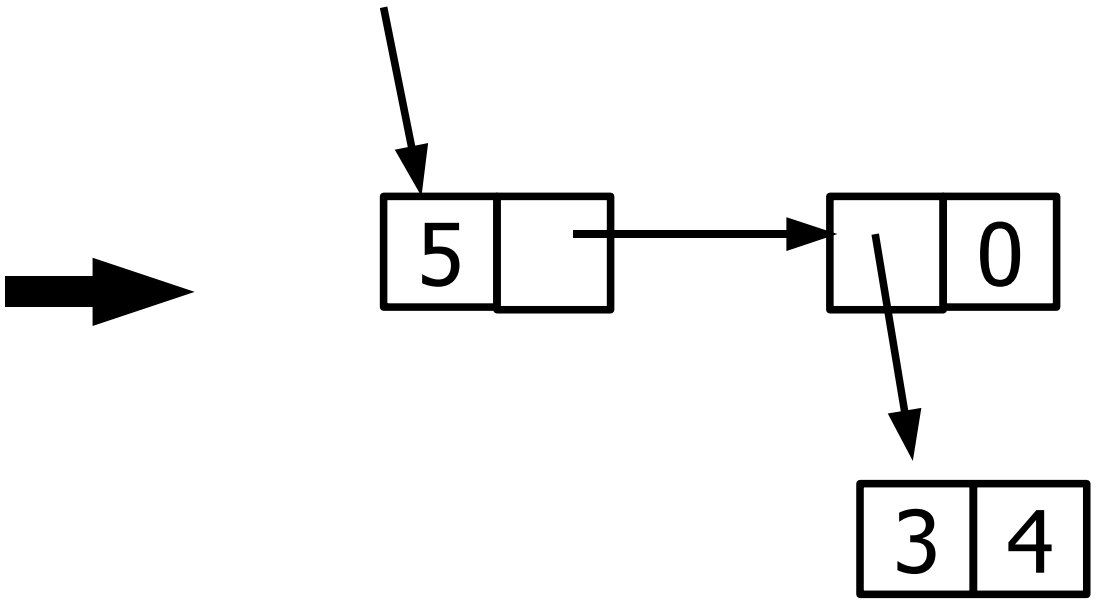
```
type maybepair = Int of int | Pair of int * int
let rec sum(acc : int, x : maybepair list) =
  match x with
  | nil -> acc
  | (Int i) :: tail -> sum(acc + i, tail)
  | (Pair (l, r)) :: tail -> sum (acc + l + r, tail)
```

- これをアセンブリに落とす
  - が、その前に…

maybepair のメモリ上表現

- Int x は 2x+1 (奇数) で表そう
- Pair(x, y) はペア構造への偶数値ポインタで表そう

[Int 2; Pair (3, 4)]



sum のコンパイル結果

- Figure 5-4 (p182) を参照
  - 12 行目は「$r_{acc} := r_{acc} + r_s$」の間違い

- 補足：アセンブリの syntax

$r_x := e$	代入
$r_x :=$ Mem[e]	ロード
Mem[e'] := e	ストア
jump L	ジャンプ
if e jump L	条件ジャンプ
return	リターン

e は算術演算、論理演算、定数、レジスタなど

Safety Policy の方向づけ

- ここでは単純な型安全性を保証することにしよう
  - ポインタはリストまたはペア構造を指す
  - リストセルの第 1 要素は
    奇数値か Pair への偶数値ポインタ
  - リストセルの第 2 要素は 0 か次のセルへのポインタ
  - ペアの要素については特に条件なし

これらのポリシーをどう Formalize するか？

ユーザによる safety policy の設定

a)VC(verification conditions) の syntax を拡張
- 検証すべき条件を表現する論理式

b)各関数の preconditions/postconditions を定義
- 対象コードの実行直前 / 直後で成り立つはずの条件
- 実行者側と対象コードとのインターフェースを規定

c)VC を導出するための proof rules を拡張
- Checker が使用

VC の Built-in Syntax

Formulas $F ::= \text{true} \mid F_1 \wedge F_2 \mid F_1 \vee F_2$
$\mid F_1 \Rightarrow F_2 \mid \forall x.F \mid \exists x.F$
$\mid \text{addr } E_a \mid E_1 = E_2 \mid E_1 \neq E_2$
$\mid f\ E_1 \ldots E_n$

Expressions $E ::= x \mid \text{sel } E_m\ E_a \mid \text{upd } E_m\ E_a\ E_v$
$\mid f\ E_1 \ldots E_n$

- addr E_a: E_a は valid な address
- sel E_m E_a: メモリ状態 E_m におけるアドレス E_a の内容
- upd E_m E_a E_v: E_m において E_a に E_v をストア後の新状態

(a) VC のユーザ定義 Syntax

- 以下の Syntax を追加
  - 型を表現するしくみを導入

Word types	W ::= int | ptr {S} | list W | {x | F(x)}
Structure types	S ::= W | W;S
Formulas	F ::= ... | E:W | listinv E_m

  - {x | F(x)}: Formula F をみたす値の集合
  - Structure types: 複数 words のサイズを持ちうる型
  - listinv E_m: 任意のポインタの指す先は適切な型の値
    - 詳細は後述

maybepair list の型表現

- 型付けできる条件を適切に表現可能

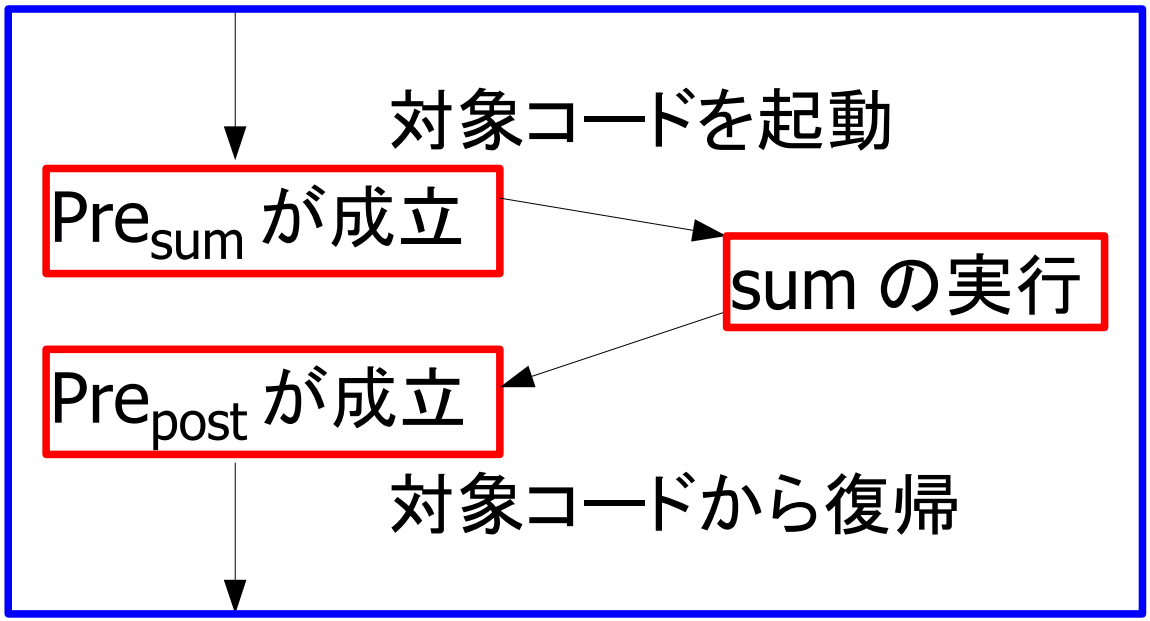
x : maybepair list

x : list {y | (even y) ⇒ y : ptr {int; int}}

  - 以下、maybepair list を mp_list と表記

(b) Preconditions/Postconditions

- Pre: コード実行直前に成り立つと仮定できる条件
  - 例 : Pre_{sum} = r_x: mp_list ∧ listinv r_M
    - r_M: メモリ全体を表す疑似レジスタ
- Post: コード実行直後に成り立つ必要のある条件
  - 例 : $Post_{sum}$ = listinv r_M



(c) Proof Rules

- VC を導出するための規則
  - built-in ルールは Figure 5-5 (p186) を参照
  - ユーザ定義のルール拡張は Figure 5-6 (p187) を参照

- ユーザ定義ルールで規定されるもの：
  - 型付け規則
  - listinv の成立条件：ポインタと指す先との型整合性
    - (UPD)
  - addr の成立条件：ポインタ型を持つこと
    - (PTRADDR)

Exercise (1)

- Exercise 5.2.5 (p186): array の導入
  - 先頭要素は array のサイズ
  - 各要素が word type/structure の両場合でどうなるか？

低級言語検証の難しさ

- 高級言語 :syntax-directed な解析が可能
  - 適用すべき typing rule が構文に応じて一意に定まる
- 低級言語 : 粒度が細かすぎて無理
  - 複数命令をまとめて考慮する必要性
  - しかし命令列の順序交換が可能なため困難
    - おまけにレジスタの型は実行中頻繁に変わる…

```
match x with
  _ :: t -> e
```

$r_t := r_x$

$r_t := r_t + 4$

if $r_x = 0$ jump LNil

$r_t := Mem[r_t]$

…

様々な解決方法

- 抽象度の高い命令を採用 (Java Bytecode Verifier)
  - 問題：激しい最適化ができない
- 散在する各命令の中間結果情報を管理 (TAL)
  - 問題：いくつかの命令はやはり抽象化されている

各命令の中間結果情報を後々まで記憶する必要性
↓
全ての情報が集まってから検証をやればいいのでは？
↓
VCGen と Checker の分業体制

Symbolic Evaluation による VC 抽出

١. 初期 symbolic state を用意
   - 各レジスタに適当な初期値を設定

٢. コードを 1 命令ずつ解釈実行
   - 命令実行とともに symbolic state を更新
   - 現在成り立つ条件を assumption として管理
     - 分岐条件などからわかる
   - VC を適宜出力
     - ユーザ定義ポリシーにかかわる命令実行の際

- 利点：命令列の順序入れ換えへの耐性

Symbolic Evaluation の例

$\sigma=\{r_t=t, r_x=x, r_M=m\}$, A

$r_t := r_x$ symbolic state assumption

$\sigma=\{r_t=x, r_x=x, r_M=m\}$, A

$r_t := r_t + 4$

$\sigma=\{r_t=x+4, r_x=x, r_M=m\}$, A

if $r_x = 0$ jump LNil

$\sigma=\{r_t=x+4, r_x=x, r_M=m\}$, A $\wedge$ $x\neq0$

$r_t := Mem[r_t]$

$\sigma=\{r_t=sel\ m\ (x+4), r_x=x, r_M=m\}$, A $\wedge$ $x\neq0$

...

LNil: $\sigma=\{r_t=x+4, r_x=x, r_M=m\}$, A $\wedge$ $x=0$

VC

$(r_t=x \wedge r_x=x+4 \wedge r_m=m \wedge A \wedge x\neq0) \Rightarrow addr(r_t)$

コード作成者による Annotation

- コード本体だけではよくわからない
  - 低級言語の解析≒リバースエンジニアリング
  - 保守的に検証せざるを得ない
- コード作成者が invariant 情報を annotate
  - コード作成者が手で書く / コンパイラが自動生成
  - 例 : ループのラベルの位置に loop invariant を付加

    Loop: INV = r_X : mp_list $\wedge$ listinv r_M
  - annotation を全面的に信用するわけではない

VCGen の実行手順

1. 各 invariant をコードに付加
   - precondition (関数開始)
   - postcondition (関数終了)
   - annotation
2. コードを invariant の出現位置で分割
3. 各分割ごとに symbolic evaluation を実行

sum (p182) の冒頭に付加された invariants

1	sum:	INV r_x : mp_list $\wedge$ listinv r_M
2	Loop:	INV r_x : mp_list $\wedge$ listinv r_M
3		if rx $\neq$ 0 jump LCons

VCGen 上の Symbolic Evaluation

$SE(i,\sigma)$ =

$SE(i+1,\sigma[r \leftarrow \sigma(e)])$	(r := e)
$\sigma(e) \Rightarrow SE(L,\sigma) \wedge not(\sigma(e)) \Rightarrow SE(i+1,\sigma)$	(if e jump L)
$addr\ \sigma(a) \wedge SE(i+1,\sigma[r \leftarrow \sigma(sel\ r_M\ a)])$	(r := Mem[a])
$addr\ \sigma(a) \wedge SE(i+1,\sigma[r_M \leftarrow \sigma(upd\ r_M\ a\ e)])$	(Mem[a] := e)
$\sigma(Post)$	(return)
$\sigma(I)$	(INV I)

$VC = \sigma_0($ 開始位置の invariant $) \Rightarrow SE(i+1,\sigma_0)$

where $\sigma_0=\{r_1=x_1, \ldots, r_n=x_n\}$

i: プログラムカウンタ

σ: symbolic state

σ(e): e 中のレジスタの出現を σ でマップ

VCGen の実行例

- Figure 5-7 (p196) を参照
  - sum (p182) を symbolic evaluation
  - 下線部は assumptions
    - 有効期間をインデントレベルで表現
  - 箱囲みの部分は verification conditions

Checker による VC 導出

- 6 行目

$$x_1{:}mp\_list \wedge listinv\ m_1 \wedge x_1 \neq 0 \Rightarrow addr\ x_1$$

$$\dfrac{\dfrac{x_1{:}mp\_list \quad x_1 \neq 0}{x_1{:}ptr\{maybepair;mp\_list\}}\ (CONS)}{addr\ x_1}\ (PTRADDR)$$

- 11 行目

$$x_1{:}mp\_list \wedge listinv\ m_1 \wedge x_1 \neq 0 \wedge even(sel\ m_1\ x_1) \Rightarrow addr\ x_1$$

  - 導出は Figure 5-8 (p197) を参照

Exercise (2)

- Exercise 5.3.3 (p197): 2 行目の invariant を導出

$$x_1 : mp\_list \wedge listinv\ m_1 \wedge x_1 \neq 0 \wedge even(sel\ m_1\ x_1) \Rightarrow sel\ m_1\ (x_1 + 4) : mp\_list$$

- Exercise 5.3.4 (p197): 冗長な導出を抑止
  - “sel m1 x1 : ptr {int}” の導出が複数回実行される
    - 11 行目と 13 行目の VC チェック
  - 1 回に抑えるにはどんな invariant を付加すればよい？

実装の話

- 利点：Checker は決定的に動作可能
  - 低級言語の曖昧さとは無縁
- 欠点：VCGen と Checker を本当に分離すると大変
  - VC の総サイズが爆発
  - 対策：VC が発生したら直ちに検証して破棄

Exercise (3)

- Exercise 5.3.5 (p198): call/ret のサポート
  - 適当な calling convention を設定
- Exercise 5.3.6 (p198): 到達不可能地点の認識
  - 返らない関数 call の直後に特別な annotation を付加
  - annotation を完全には信頼しない検証は可能か？
- Exercise 5.3.7 (p198): 間接ジャンプのサポート
  - ジャンプ先ラベルの候補を annotate
- Exercise 5.3.8 (p199): スタックフレームのサポート
  - 各要素を疑似レジスタ列として扱えるよう制限すると楽

まとめ

- Proof-Carrying Code
  - コード本体 + コードに関する補足説明
  - コードが利用者側の定義したポリシーを満たすか検証
  - コードの走査作業と検証作業を分離