# POPL ミーティング
# (2006/11/29)

米澤研　M2　佐藤秀明

# 今回の内容

- Dynamic Heap Type Inference for Program Understanding and Debugging.
  - Marina Polishchuk, Ben Liblit and Chloe W. Schulze.
  - POPL 2007.
- データ構造を実行時に型推論
  - Subtyping のアイデアを応用
  - デバッガ拡張として実装
  - バグを型エラーとして発見

# メモリ破壊エラーのデバッグ手法

- 動的解析の限界
  - 変数 watch 機能は遅い
  - デバッグ用 printf() を挿入すべき位置が自明ではない
- 静的解析の限界
  - 実行時のメモリレイアウトに依存するバグは分からない

目的：型システムによるデバッグ支援

# Consistent Typing の提案

- ある時点でのメモリ状態を各 byte ごとに型付け
  - メモリ構造から型を特定するための制約を抽出
  - 制約問題が解けなかったらバグ

# データ構造の定義例

```
struct Point {
   double x, y;
};

struct Shape {
   char *name;
   FILE *fptr;
};

struct Part {
   struct Point center;
   struct Shape *shape;
   struct Assembly *owner;
};
```

```
struct PartNode {
   struct Part *part;
   struct PartNode *next;
};

struct Assembly {
   struct Point center;
   struct PartNode *nodes;
   struct Assembly *owner;
};
```

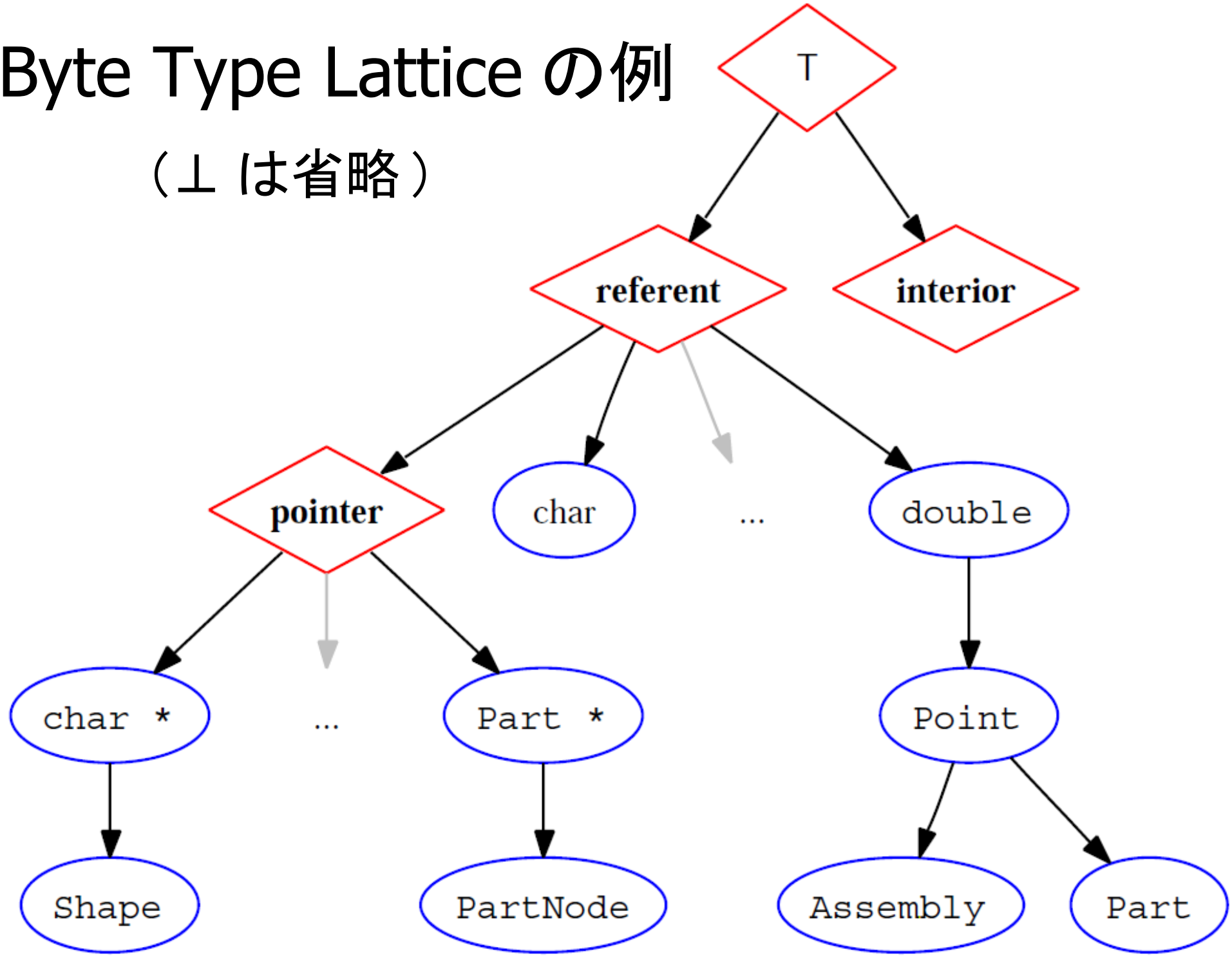
- 以降、この例に基づいて説明

# Byte Type Lattice

- Subtyping 関係を定義
  - 型付けに用いる制約のひとつ
- 特別な built-in types を用意
  - pointer: ポインタ型
  - referent: ポインタで指し示すことのできる型
    - 多倍長な値の先頭アドレスなど
  - interior: ポインタで指し示すことのできない型
    - 多倍長な値の先頭アドレス以外など
- ⊥ に型付けされたら型エラー

# Byte Type Lattice の例

（⊥ は省略）



# Compound Types

- struct はその第 1 フィールドと同様に扱える

$$T_1 \times \ldots \times T_k \ < \ T_1$$

- array もその第 1 要素と同様に扱える

$$T[n] \ < \ T$$

# Covariance/Contravariance

- Subtyping 関係は pointer に移ったら成り立たない

$$\tau_1 < \tau_2 \not\Rightarrow \tau_1\ \mathrm{ptr} < \tau_2\ \mathrm{ptr}$$

  - Array の場合も同様
- この関係を要求するプログラムはあやしい
  - キャストの使用
  - 型安全性を壊すその他の手段の使用
- 関数型についてもたぶん同様
  - companion technical paper が見つからない…

# Union Types

- いま使用されている型でタグ付け

$$T_1 + \ldots + T_k \rightarrow T_r(T_1 + \ldots + T_k)$$

- 適切な subtyping 関係を設定

$$T_1 + \ldots + T_k < \text{referent}$$

$$T_r(T_1 + \ldots + T_k) < T_1 + \ldots + T_k$$

$$T_r(T_1 + \ldots + T_k) < T_r$$

- 複数の supertype を持つのは union と⊥だけ

# 解析するプログラム例

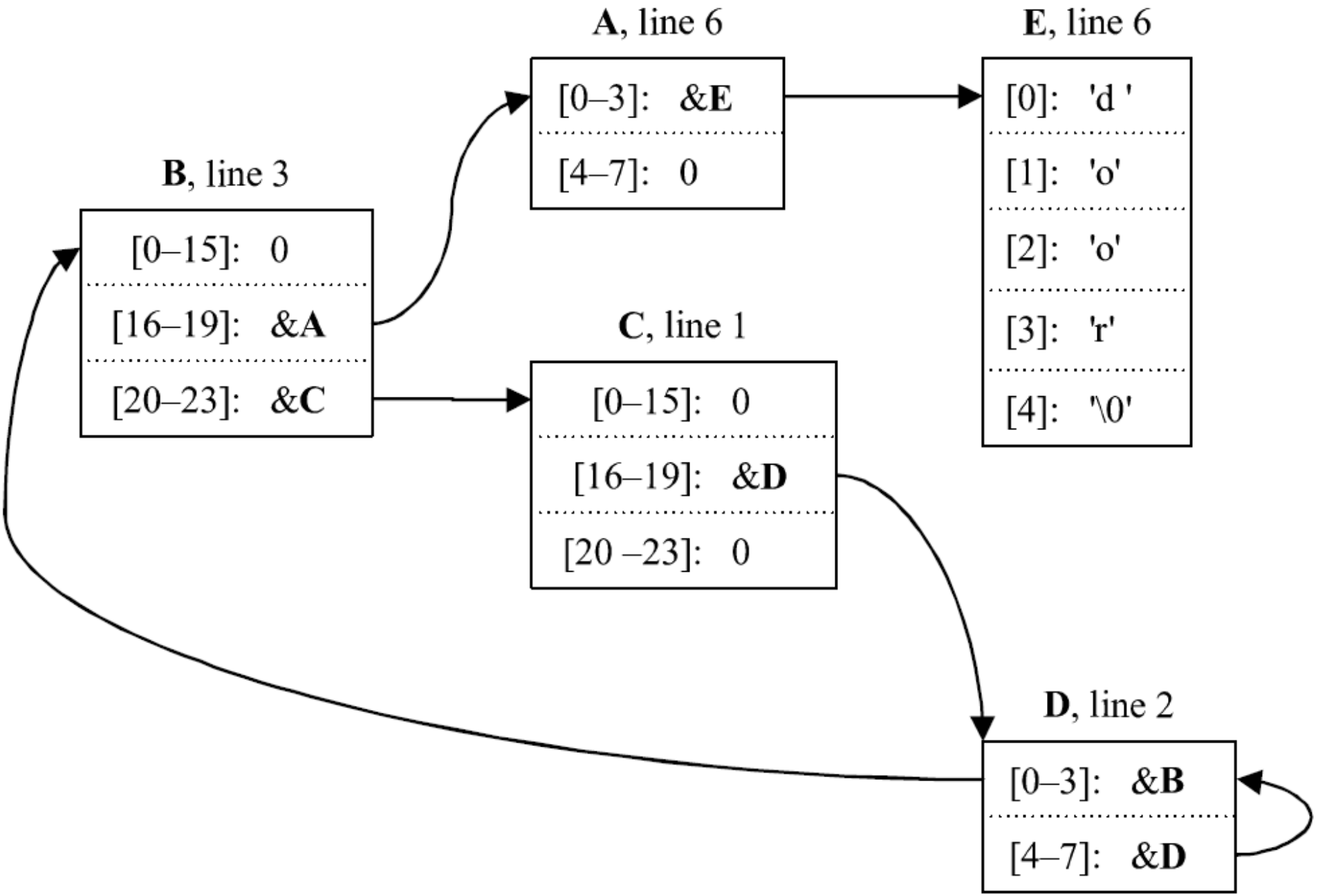
```
   Assembly *create_assembly() {
1     Assembly *assm =
        malloc(sizeof(Assembly));
2     PartNode *node =
        malloc(sizeof(PartNode));

3     node->part = malloc(sizeof(Part));
4     node->next = node;
5     assm->nodes = node;

      // build part's shape and set name
6     init_part(node->part, "door", assm);
      ...
7     return assm;
   }
```

注：malloc() はゼロクリア付き

# 作成されたデータ構造


A, line 6
[0–3]: &E
[4–7]: 0
E, line 6
[0]: 'd '
[1]: 'o'
[2]: 'o'
[3]: 'r'
[4]: '\0'
B, line 3
[0–15]: 0
[16–19]: &A
[20–23]: &C
C, line 1
[0–15]: 0
[16–19]: &D
[20 –23]: 0
D, line 2
[0–3]: &B
[4–7]: &D

# Consistency Constraints

- 型を特定するための５つの制約
  - Mandatory constraints
  - Filtering constraints
  - Size constraints
  - Type constraints
  - Debug constraints
- あり得る型の組合せを全探索

# (1) Mandatory Constraints

- valid な pointer ならその型は pointer

$$\text{validPointer}(\text{addr}) \Rightarrow \text{typeof}(\text{addr}) \leq \text{pointer}$$

- この情報から候補をある程度絞れる

| Block | Valid Pointers | Value-Consistent Types |
|---|---|---|
| **A** | $\mathbf{A}+0$ | `PartNode, Shape` |
| **B** | $\mathbf{B}+16, \mathbf{B}+20$ | `Part, Assembly` |
| **C** | $\mathbf{C}+16$ | `Part, Assembly` |
| **D** | $\mathbf{D}+0, \mathbf{D}+4$ | `PartNode, Shape` |

# (2) Filtering Constraints

- 格納される値やその扱われ方から型を決定
  - 整数 constant→enum
  - word-aligned な整数→関数ポインタ
  - 1 バイトごとにストア→ char array
- それ以外に有用な制約がない場合に有効

# (3) Size Constraints

- データ構造のサイズに関する制約

$$\text{typeof}(b) = \tau \Rightarrow \text{sizeof}(b) = \text{sizeof}(\tau)$$
$$\text{typeof}(b) = \tau[n] \Rightarrow \text{sizeof}(b) = n \times \text{sizeof}(\tau)$$

| Block | Size | Size-Consistent Types |
|---|---|---|
| **A** | 8 | `PartNode, Shape, char[8], ...` |
| **B** | 24 | `Part, Assembly, Shape[3], PartNode[3], float[6], ...` |
| **C** | 24 | `Part, Assembly, Shape[3], PartNode[3], float[6], ...` |
| **D** | 8 | `PartNode, Shape, int[2], ...` |
| **E** | 5 | `char[5]` |

# (4) Type Constraints

- subtyping に関する制約
  - (i)

$$\text{typeof}(addr) \le \text{pointer} \wedge (*addr) \in (\text{allocated block}) \Rightarrow \text{typeof}(*addr) < \text{referent}$$

  - (ii) interior への pointer を抑止

$$\text{typeof}(addr) \le \tau \in (\text{atomic type}) \Rightarrow \forall 1 \le i < \text{sizeof}(\tau).\ \text{typeof}(addr + i) = \text{interior}$$

  - (iii) dangling pointer を抑止

$$\forall block \in (\text{start of allocated block}).\ \text{typeof}(block) < \text{referent}$$

# (4) Type Constraints( 続き )

- (iv) ポインタの整合性

$$\mathrm{typeof}(addr) \leq \tau\ ptr \Rightarrow \mathrm{typeof}(*addr) \leq \tau$$

- (v) struct の整合性

$$\mathrm{typeof}(addr) \leq \tau_1 \times \ldots \times \tau_k \Rightarrow \forall 1<n\leq k.\ \mathrm{typeof}(addr + i_n) = \tau_n$$

- (vi) array の整合性

$$\mathrm{typeof}(addr) \leq \tau[n] \Rightarrow \forall 1\leq i<n.\ \mathrm{typeof}(addr + i * \mathrm{sizeof}(\tau)) = \tau$$

# (5) Debug Constraints

- デバッグ情報を利用
  - なくてもよい
- subtyping なしに厳密に型付け
  - int と指定されたら int に固定

# Heap Typing Algorithm

- 入力：
  - メモリのスナップショット
  - allocate された各ブロックの先頭アドレスとサイズ
  - プログラムで定義された型情報
  - in-scope な変数のデバッグ情報
- 出力：
  - allocate された全領域のバイトごとの型付け

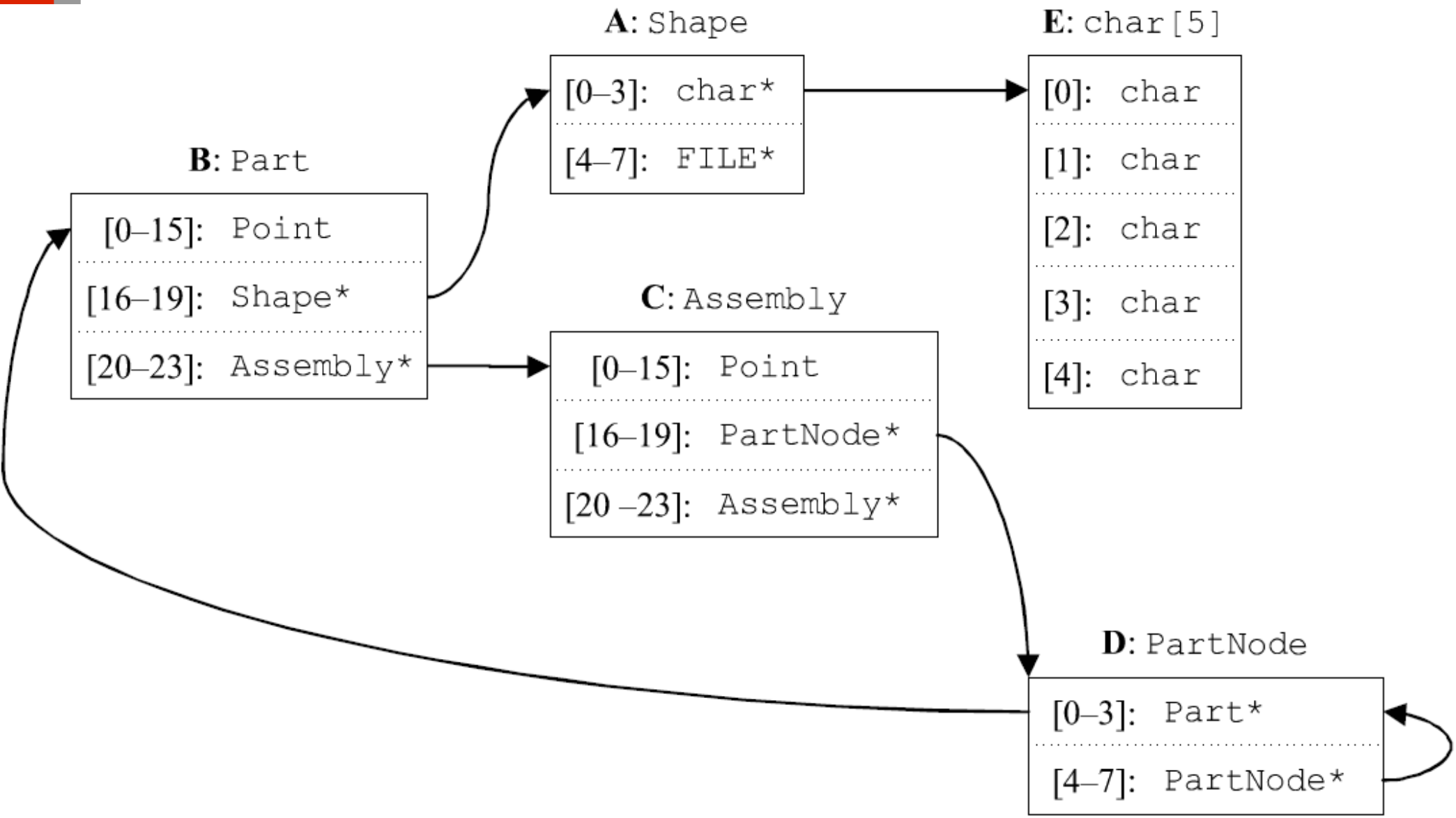
# アルゴリズムの手順

1. 型情報の初期状態を生成
   - mandatory 制約と type 制約 (i)(ii)(iii) を利用
   - pointer/interior/referent/T のどれかの型がつく
2. デバッグ情報を伝搬
   - type 制約 (iv)(v)(vi) を利用
   - この時点では最も一般的な型がついている
3. 可能な型付けの組合せを全探索
   - より具体的な型のほうから順に試していく
   - size 制約や subtyping 制約を満たすかどうかチェック

# 探索例

| Block | Type Considered | Outcome | Induced Constraints |
|---|---|---|---|
| **C** | `FILE` | size conflict: sizeof(**C**) < sizeof(`FILE`) | |
| **C** | `Part` | ✓ | **D** + 0 : `Shape` |
| **D** | `Shape` | type conflict at **D** + 0: *meet*(`Shape`,`FILE`) = ⊥ | |
| **C** | `Assembly` | ✓ | **D** + 0 : `PartNode` |
| **D** | `PartNode` | ✓ | **B** + 0 : `Part` |
| **A** | `PartNode` | size conflict: sizeof(**E**) < sizeof(`PartNode`) | |
| **A** | `Shape` | ✓ | **E** + 0 : `char` |
| **E** | `char[5]` | ✓ | |

# 型付け結果


A: Shape
[0–3]: char*
[4–7]: FILE*
E: char[5]
[0]: char
[1]: char
[2]: char
[3]: char
[4]: char
B: Part
[0–15]: Point
[16–19]: Shape*
[20–23]: Assembly*
C: Assembly
[0–15]: Point
[16–19]: PartNode*
[20 –23]: Assembly*
D: PartNode
[0–3]: Part*
[4–7]: PartNode*

# アルゴリズムの詳細

- デバッグ情報は最悪無視してもよい
  - 静的な型情報よりも動的なそれを重視
- 型の付かなかったブロックは解析対象から外す
  - 残りのブロックの型付けを最大限努力
- 全探索のコストは低い
  - ブロックの先頭アドレスにつく型の subtype のみ探索
- アルゴリズムの正しさは technical paper で
  - 型の inconsistency は全て検知

# 実装

- gdb と GNU libc を改造
  - 指定した expression の型を推論するコマンド
    - “whatsat”
  - メモリの型を可視化する機能
  - allocate されたブロックの先頭アドレスとサイズを管理
  - allocate するブロックのゼロクリア

# Case Study (1): Schedule2

- Schedule2: Siemens のジョブスケジューラ
  - malloc() 内でクラッシュ
- 型付けの結果：ある関数ポインタが型付け不能
  - デバッグ情報では関数型のはず
  - デバッグ情報を無視したらある構造体の型がついた
- 原因：構造体配列のオーバーラン
  - 関数ポインタを上書き
  - メモリ構造の可視化により確認
- 利点：バグを見つけたら直ちに原因調査に移れる
  - 再現性の低いバグに対処しやすい

# Schedule2 のメモリ可視化の例

```
? 0X804ABB8: 0x00000000
? 0X804ABBC: 0x00000000
                          (queue [4])
                            [0] = (queue)
D 0X804ABC0: 0x00000001       length = (int) 1
D 0X804ABC4: 0x0804b0d0       head = (process *) 0x804b0d0
                            [1] = (queue)
D 0X804ABC8: 0x00000001       length = (int) 1
D 0X804ABCC: 0x0804b0a8       head = (process *) 0x804b0a8
                            [2] = (queue)
D 0X804ABD0: 0x00000000       length = (int) 0
D 0X804ABD4: 0x00000000       head = (process *) 0x0
                            [3] = (queue)
D 0X804ABD8: 0x00000000       length = (int) 0
D 0X804ABDC: 0x00000000       head = (process *) 0x0
? 0X804ABE0: 0x00000002
D 0X804ABE4: 0x0804b008 (void *(*)()) 0x804b008
D 0X804ABE8: 0x00000000 (void (*)()) 0
D 0X804ABEC: 0x00749380 (void *(*)()) <realloc_hook_ini>
D 0X804ABF0: 0x007493d0 (void *(*)()) <memalign_hook_ini>
? 0X804ABF4: 0x0804b018
? 0X804ABF8: 0x00000000
```

# Case Study (2): Exif

- Exif: JPEG のメタデータ編集ツール
- バッファを正しく初期化しているかチェックした
  - 自動ゼロクリア機能は off にして調査
- 型付けの結果：誤った初期化操作を発見

```
entries = realloc(entries,
  sizeof (ExifEntry) * (count + 1));
```

  - 正しくは ExifEntry ではなく(ExifEntry *) であるべき
  - 動作には問題ないがメモリの無駄
  - 無駄な領域は未初期化値が入っているため型付け不能

# 関連研究

- Physical subtyping[Chandra et al.][Siff et al.]
  - メモリレイアウトにつく型を静的に解析
  - 我々はより単純なモデル上で動的に解析
- Conservative GC[Boehm et al.]
  - valid なポインタに見えるものはポインタとして扱う
  - 我々は型情報も計算できる
- Memory visualization[Zimmermann et al.]
  - デバッグのためのメモリ可視化
  - 我々はグローバルな制約もうまく扱える
- コンパイラの立場からのメモリ破壊解析 [ 多数 ]
  - 我々はデバッガとの親和性が高い

# まとめ

- C 言語のメモリ破壊を発見するためのデバッガ拡張
  - メモリレイアウトを動的に型推論
  - 実際にいくつかのバグを発見