- This slide was
  - a material for the "Reading PLDI Papers (PLDIr)" study group
  - written by Kazuhiro Inaba (www.kmonos.net), under my own understanding of the papers published at PLDI
    - So, it may include many mistakes etc
- For your correct understanding, please consult the original paper and/or the authors' presentation slide!

k.inaba (稲葉 一浩), reading:

PLDIr #14
Jul 26, 2010

paper written by K. R. M. Leino A. P.-Heffter Y. Zhou
(PLDI 2002)

Using Data Groups to Specify and Check Side Effects

Using Data Groups to ***Specify*** and Check Side Effects

概要

- オブジェクト指向プログラムの副作用解析をやりたい
  - ここでは、副作用 ≒ フィールドの破壊的書換
- 全自動でやるのは
  - 無理
  - modular でない
    - ライブラリも含め対象アプリの全ソースコードがないと不可能。現実的でない。
- ということで、プログラマによるアノテーションに頼る
  - どういうアノテーションをもらうのが良いか？

基本的なアイデア

- 対象言語
  - “OOLONG” という多分この論文のためだけに設計された、仮想の副作用解析専用言語
  - untyped
  - class とかの概念もない
  - object とその field access があるだけ
    - JavaScript (-prototype) や Lua みたいなの
- アノテーション
  - 関数を modifies で修飾

```
proc push(stk, e) modifies stk.contents
impl push(stk, e) { stk.contents << e }
```

むずかしい点

- （class があった方が説明しやすいので Java風の謎言語で説明）

```
interface Stack          { void push(e); }
void useStack(Stack s) { s.push(“hoge”); }
```

  - Q: このコードはどういう副作用をおこす？
  - A: Stack の実装によるので、上のinterface宣言だけでは解析のしようがない

```
class ArrayStack implements Stack {
   Object[] contents;
   void push(e) modifies this.contents
     { contents << e; }
}
```

むずかしい点＆解決策

- かといって、interfaceのアノテーションに、実装詳細を書くわけにも…

```
interface Stack {
    void push(e) modifies this.contents;
}
```

- そりゅーしょん：グループ宣言

```
interface Stack {
    group impl_data;
    void push(e) modifies imple_data;
}
```

```
class ArrayStack implements Stack {
    field contents in impl_data;
    void push(e) { contents.add(e); }
}
```

フォーマルに言うと

Decl ::= **group** *Id* [**in** *IdList*]
| **field** *Id* [**in** *IdList*] (**maps** *Id* **into** *IdList*)*
| **proc** *Id* "(" *IdList* ")" [**modifies** *ExprList*]
| **impl** *Id* "(" *IdList* ")" "{" *Cmd* "}"

Figure 0: The grammar of the language oolong.

- "グループ" を宣言できる
  - グループには階層構造を定義できる
- 各 "フィールド" はどのグループに属すか宣言できる
  - 同時に複数のグループに所属できる
  - maps : メンバオブジェクトのフィールドをグループ宣言

```
class java.util.ArrayList { Object[] __data; … }
class ArrayStack implements Stack {
 { ArrayList elems;
   field elems maps __data to impl_data; }
```

さらに色々とややこしい問題

- 実は未だあまり解決してない
  - 上半分をみただけではassert 通りそうに見える
  - でもダメ
- Pivot Uniqueness
- Owner Exclusion
  - “return vec;” みたいなのはOolongでは禁止

```
interface Stack {
    group contents
    void push(e)
                modifies this.contents;
    Object hoge();
}
void q( Stack s ) {
  vd = s.hoge().__data;
  s.push(42);
  assert( vd == s.vec.__data );
}

// NO
class HogeStack implements Stack {
    field vec maps __data to contents;
    void push(e)  { vec.__data << e; }
    Object hoge() { return vec; }
}
```

アノテーションを使って

- これ

```
field c      field d      field f      group g
proc p(t) modifies t.c.d.g
proc q(u) modifies u.g
impl p(t) {
  assume t ≠ null ;
  var y in
    y := t.f ; q(t.c.d) ; assert y = t.f
  end }
```

を

- こんな論理式に変換する導出規則を定義

$$
\begin{array}{l}
UBP \wedge BP \wedge ownExcl(t, t.c.d.g, \$_0) \wedge alive(\$_0, t) \wedge \\
\$ = \$_0 \wedge t \neq \mathbf{null} \Rightarrow \\
\quad (\forall y :: (\forall u :: u = \$(\$(t.c).d) \Rightarrow \\
\qquad mod(u.g, t.c.d.g, \$_0) \wedge ownExcl(u, u.g, \$) \wedge \\
\qquad (\forall \$' :: \\
\qquad\quad (\forall X :: alive(\$, X) \Rightarrow alive(\$', X)) \wedge \\
\qquad\quad (\forall X, F :: \$(X.F) = \$'(X.F) \vee mod(X.F, u.g, \$)) \\
\qquad\quad \Rightarrow \$(t.f) = \$'(t.f)\ )))
\end{array}
$$

- あとは論理式のChekcerで検査する(のだと思う)

k.inaba (稲葉 一浩), reading:

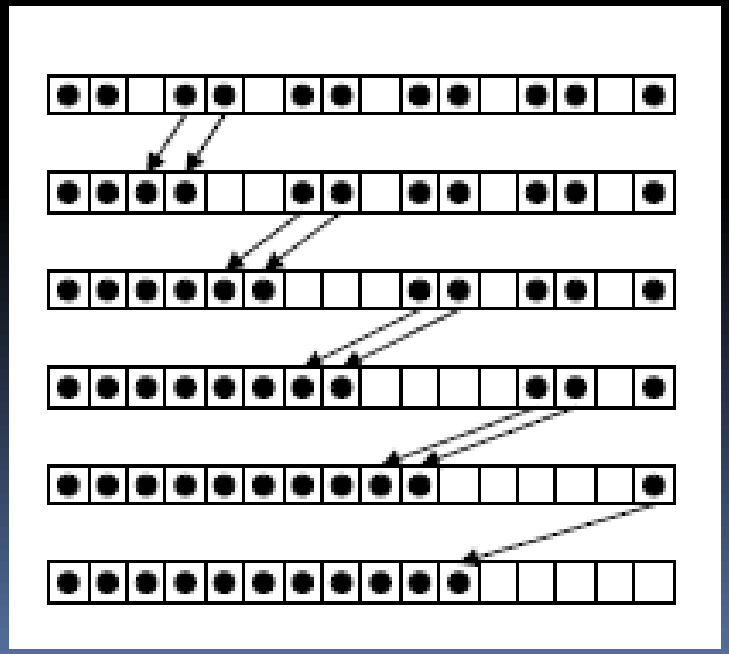
PLDIr #14
Jul 26, 2010

paper written by A. S.-Lezama R. Rabbah, R. Bodik, and K. Ebcioglu (PLDI 2005)

Progamming by Sketching for Bit-Streaming Programs

問題

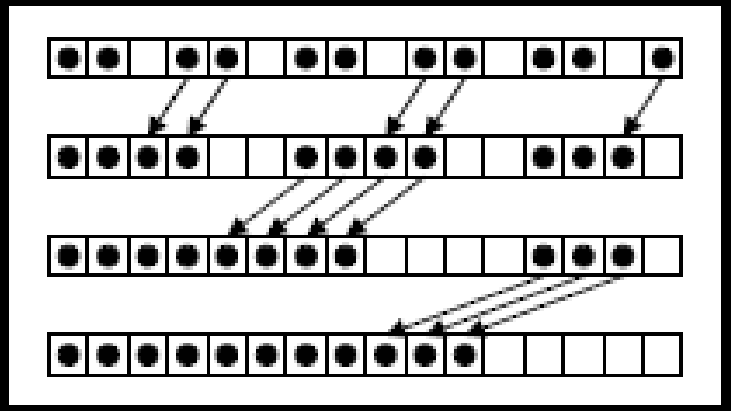
- wビットの整数を、
  3の倍数番目のビットを消して
  左に詰めて下さい
  - 単純な実装 ： w/3 回のシフト演算等
    （右はこの研究の提案言語で書いたコード）



```
bit->bit filter DropThird {
  work push 2 pop 3 {
    for (int i=0; i<3; ++i) {
      bit x = peek(i)
      if (i<2) push(x);
      pop();
    }
  }
}
```

速い実装

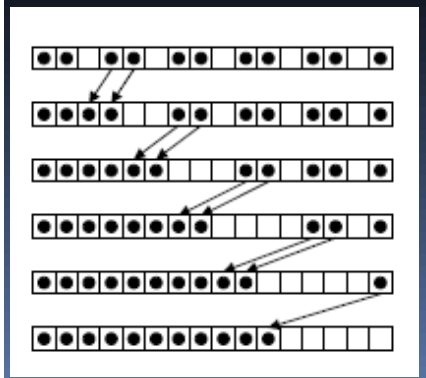
- なんか並列っぽく、まとめてシフト
  - $\log_2 w/3$ 回のシフト等

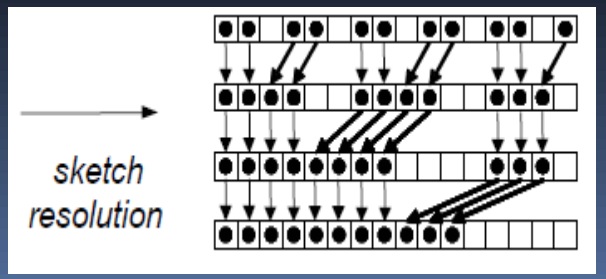


- 「なんか並列っぽく、まとめて」まで思いついたとしても、この格好いい実装をきちんと正しく作るのは結構大変
  - 各ステップで何ビットずつまとめればいい？
  - 各ステップでどのくらいシフトすれば？
  - 動かしちゃいけない部分はどこだろう？

提案言語 StreamBit

- 「遅いけど正しい実装」
  と
- 「だいたいこんな感じでまとめてシフトを何回かやればできる巧い実装があるはず！」というスケッチを与えると
  高速な実装を自動合成





どんな感じに書くか（16bitの例）

```
bit->bit filter SketchedStage1 {
  work push 16 pop 16 {
    while(*) {
      switch (*) {
      case CopyOneBit:  push(pop()); break;
      case ShiftBits:
         pop(); while (*) push(pop()); push(0);
      }
    }
  }
}
```

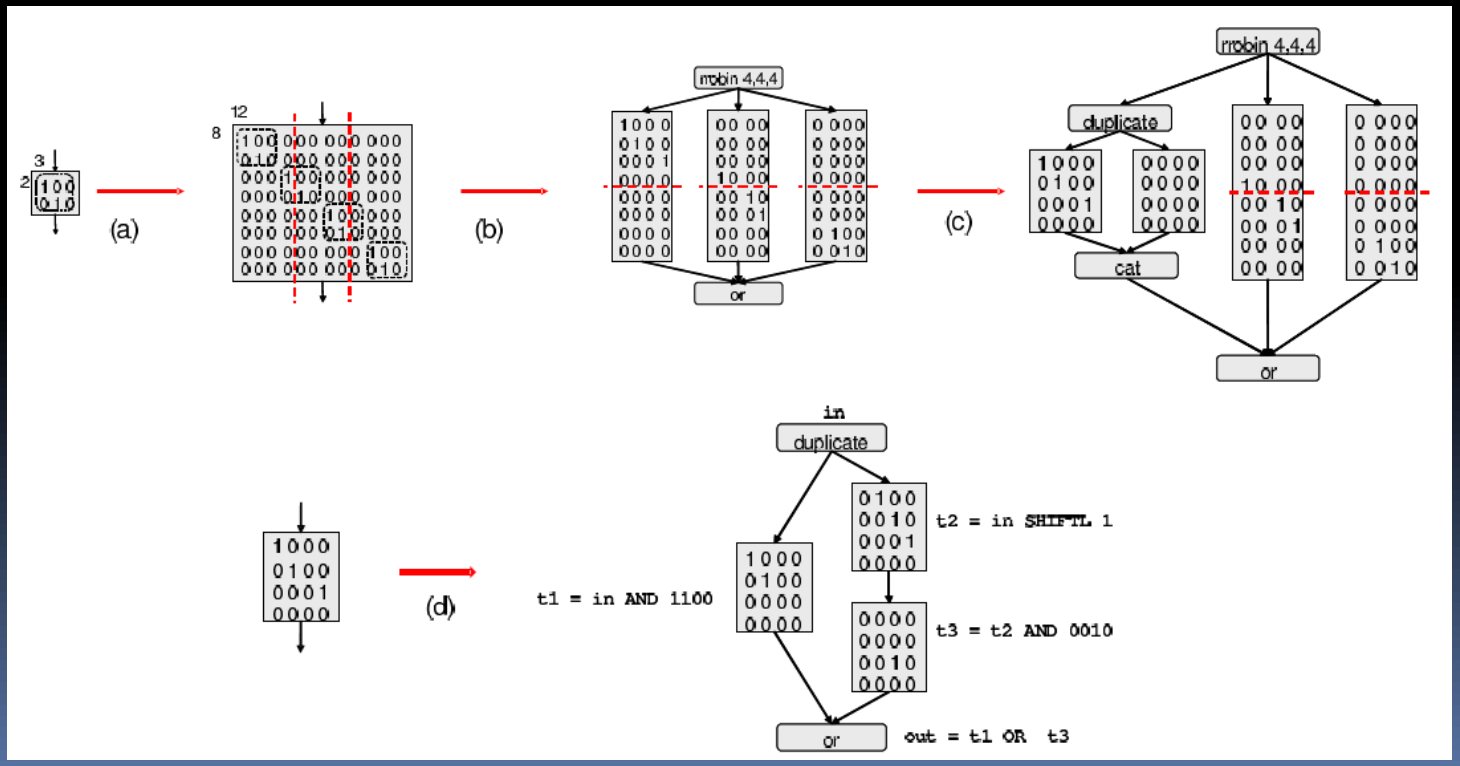
細かいことは
わからん(*)けど
ビットによって
コピーしたりシフト
したりする！

まとめてシフトする
幅はわからん(*)けど
なんか適当にまとめて
ビットずらすはず！

```
bit->bit pipeline LogShifter {
   add SketchedStage1
   add SketchedStage1
   add SketchedStage1
}
```

16bitなら、
3回もやれば
できるはず！

- と書くと、*をうまく埋めて完全な実装を作ってくれるそうです。
  - 実装は探索でみつける（+ 様々なヒューリスティクス）
  - push/pop による記述は、AND, OR, XOR, LSHIFT, RSHIFT の組み合わせにうまくコンパイラが展開してくれるらしい



評価

- こんな計算をするストリーム暗号を

$$
\begin{aligned}
X_0 &= IP(M) \\
X_{i+1}^f &= X_i^b \oplus F(X_i^f, K_i) \\
X_{i+1}^b &= X_i^f \\
F(X_i, K_i) &= L(K_i \oplus P_i(X_i))
\end{aligned}
$$

- いろんな人に実装してもらう
  - かかった時間
  - できた実装の速度
  - などで比較（C言語と勝負）
- 論文のグラフ参照

その後

- APLAS 2009 Invited Talk
  - ビット以外でも色々できるように発展しているらしい

```
1: #define LHS {| tmp | (l | nl).(h | t)(.next)? |}
2: #define LOC {| LHS | null |}
3: #define COMP {| LOC ( == | != ) LOC |}

    list reverseEfficient(list l){
4:      list nl = new list();
5:      node tmp = null;
6:      bit c = COMP;
7:      while(c){
8:          repeat(??)
9:              if( COMP ){ LHS = LOC; }
10         c = COMP;
        }
    }
```

リストの逆転なんて、なんかループして終了条件チェックして適当に代入を繰り返してればどうにかなるのでは！！！！