- 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 the following paper:

PLDIr #4 :: Dec 2, 2009
paper written by A. Heydon, R. Levin, and Yuan Yu

Caching Function Calls Using Precise Dependencies

普通のメモ化は precise でない

```
function f(x, y, z)
 { return heavy( if x>0 then y else z ) }
```

普通のメモ化

```
memo4f = new HashMap
function f(x, y, z)
 { if memo4f.has(x,y,z) then
     v = memo_f[x,y,z]
   else
     v = heavy( if x>0 then y else z )
     memo4f[x,y,z] = v
   return v }
```

f(1, 2, 3) の結果は
f(1, 2, 7) の時にも
使えるはず！
もったいない！

この論文の Contribution

```
function f(x, y, z)
 { return heavy( if x>0 then y else z ) }
```

という定義を見たら

- x＞0 のときは (x, y) をキーに記憶
- x≦0 のときは (x, z) をキーに記憶

のように、自動で細かく結果をキャッシュ！

そんな関数型言語を実装したそうです。

あぷりけーしょん

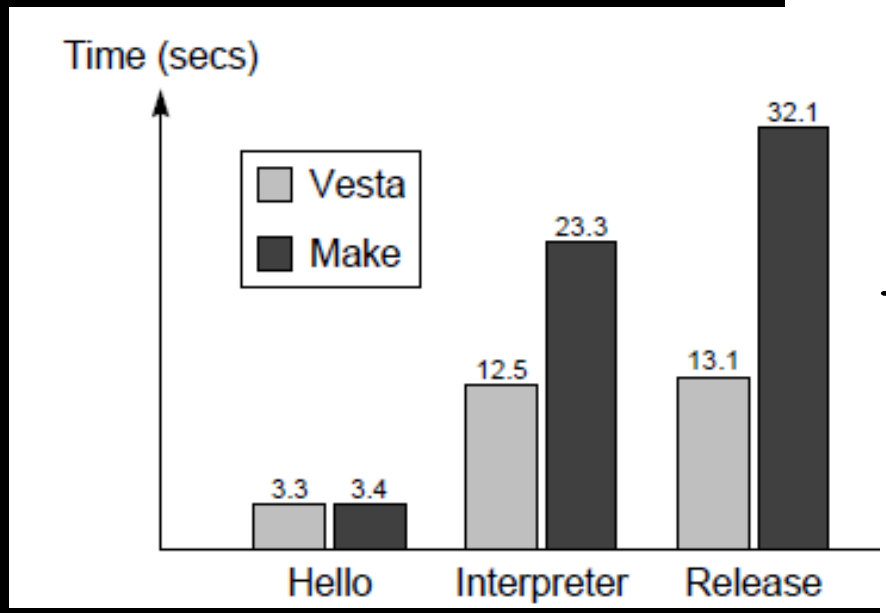
- ソースコード管理システム“Vesta”
  - Compaq で使われてるらしい
    - http://sf.net/projects/vesta/　http://www.vestasys.org/
  - Make と CVS を合わせたようなもの
  - ビルドルールをこの言語で書く→Makeっぽく動く

```
function build(opt, env) {
  obj = compile(“foo.o”, [foo.c=env/foo.c])
  return link(“foo”, obj, opt, env)
}
```

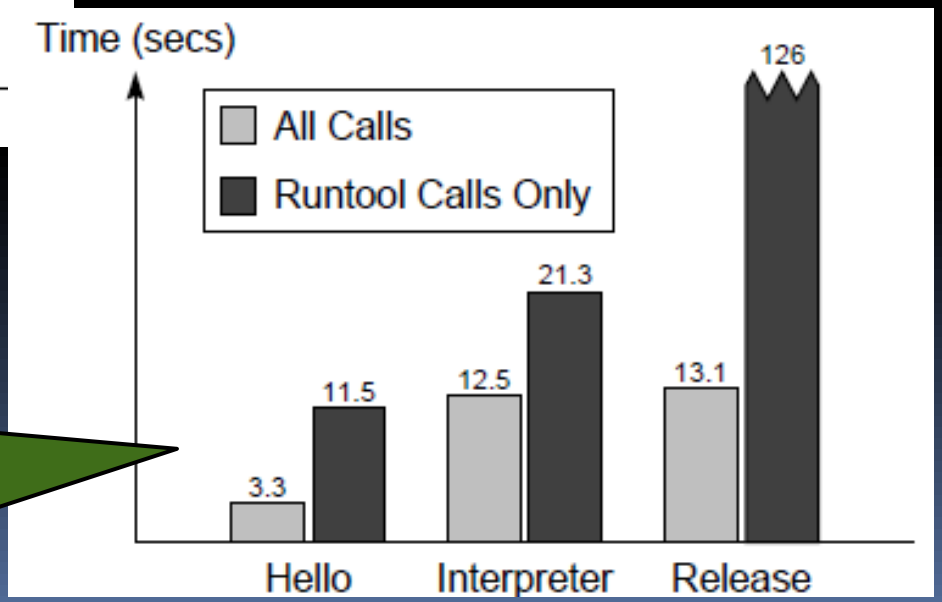
※ envはファイルシステム全体を表すツリー
(all your filesystem are belong to Vesta!)
※ opt と env/foo.c をキーにbuild関数をメモ化

実験結果
1ファイル変更→rebuild

Test	Source Lines	Source Files	Runtool Calls
Hello	10	1	2
Interpreter	53,304	103	117
Release	119,602	255	333



Make より速い



コンパイラ等の呼出以外のキャッシュも効いてる

実装

- 実行時に動的に依存関係を計算（§４）
  - eval(式, 変数束縛)
    → (値, どの変数にどう依存してたか情報)
- 依存性の表現
  - 例: `{ V:x/foo/bar, X:y/buz }`
    - 変数xのフィールドfooのフィールドbarの値
    - と、変数yのフィールドbuzの存在性に依存
  - V, X の他に D, T, L, E がある
- これを使って適切にキャッシュ（§３, 詳細略）

実装

$$e = \underline{\text{let}}\ x = [r = [s = y], t = z]\ \underline{\text{in}}\ x/r/s$$

- D(e, c, p) =
  式eを、環境cでの評価結果にパスpでアクセスした値は何に依存してるか

$$\begin{aligned}
& D(e, c, V : \epsilon) \\
\equiv\ & \{\ \text{let rule}\ \} \\
& \emptyset \cup D([r = [s = y], t = z], c, V : r/s) \\
\equiv\ & \{\ \text{binding constructor rule}\ \} \\
& D([s = y], c, V : s) \\
\equiv\ & \{\ \text{binding constructor rule}\ \} \\
& D(y, c, V : \epsilon) \\
\equiv\ & \{\ \text{variable rule}\ \} \\
& \{\ V : y\ \}
\end{aligned}$$

例

を、割と常識的な規則で再帰的に計算

- 例: D(if e1 then e2 else e3, c, p)

$$= \begin{cases} D(e1,c,\varepsilon) \cup D(e2,c,p) & \text{when } e1 \rightarrow \text{true} \\ D(e1,c,\varepsilon) \cup D(e3,c,p) & \text{when } e2 \rightarrow \text{false} \end{cases}$$

まとめ

- 依存関係を細かく見てくれるメモ化
- インタプリタが実行時に、
  eval のついでに依存関係も計算する実装
- Make っぽいものの実現に利用

k.inaba (稲葉 一浩), reading the following paper:

paper written by N. Ramsey and S. Peyton Jones

A Single Intermediate Language That Supports Multiple Implementations of Exceptions

[論文の内容]

C--での例外サポート機能の解説

- C--（http://cminusminus.org/ ）
  - 独自言語　→　Cのソースに変換　→　機械語
    - という実装の言語処理系はよくあるけど、Cは必ずしもこの目的に最適とは言えない
  - C--は、この目的に特化して作られたC風言語
- この論文は「C--に、例外の実装に便利な機能をいれたよー」というもの
  - 長距離ジャンプ機能＋制御フローの明示
  - 要はちゃんとしたsetjmp / longjmp

基本プリミティブは継続（continuation）

```
f(bits32 x) {
  bits32 y;
  …
continuation k(y):
  …
}
```

- 継続といっても、単にスタック上の位置を覚えてるだけ
  - one-shot
  - 関数 f の終了後にkを呼び出すと未定義

◆ CPS/Stack Cutting

- 継続を明示的に持ち回る
- スタックを一気に削ってジャンプ

```
g(x, k) also cuts to k;
```

```
cut to k(42);
```

◆ Stack Unwinding

- 処理系定義のランタイム関数を呼び出してハンドラ登録
- 後はランタイムが好きにする

```
g(x) also unwinds to k;
```

◆ Multiple return

```
g(x) also returns to k;
```

```
return<0/1> 42; //kに飛ぶ
return<1/1> 42; //正常
```