DEPARTMENT OF COMPUTER SCIENCE
UNIVERSITY OF COPENHAGEN
D I K U

Bootstrapping Compiler Generators from Partial Evaluators

Robert Glück
University of Copenhagen

WG 2.11 Meeting
Halmstad Sweden
2012

Today's Plan

Part 1: Theory
- Brief review of partial evaluation
- The new bootstrapping technique

Part 2: Practice
- An online compiler generator for recursive Flowchart
- Experimental validation & operational properties

This talk reports:
- *Bootstrapping can be a viable alternative to the 3rd Futamura projection.*

Programs as Data Objects

Build programs that treat programs as data objects:
- Analyze, transform & generate programs
- Manipulate programs by means of programs

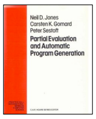
Three basic operations on programs: [Glück Klimov'94]

1. **Specialize**:	e.g. partial evaluation
2. **Invert**:	e.g. reversible computation
3. **Compose**:	e.g. deforestation, slicing

▲ Programs are semantically the most complex data structure in the computer!

Brief Review of Partial Evaluation

- **Partial evaluation**: technique to specialize programs.


Neil D. Jones
Carsten K. Gomard
Peter Sestoft
Partial Evaluation and Automatic Program Generation

- Partial evaluators were designed & implemented.
  Scheme, Prolog, ML, C, Fortran, Java, ...
- Literature: standard book [JonesGomardSestoft'93].
- Most intense research phase from mid 80ies to end 90ies.
- Cornerstone are the 3 Futamura projections [Futamura'71].

More formally: What is a Specializer?

Program specialization:

$$\mathbf{r} \;=\; [\mathbf{s}](p,x)$$
$$[\mathbf{r}]\ y \;=\; [p](x,y)$$

Terminology:
s ... specializer
r ... residual program

Characteristic equation:

$$\underbrace{[\ [\mathbf{s}](p,x)\ ]\ y}_{\textit{2 stages}} \;=\; \underbrace{[p](x,y)}_{\textit{1 stage}}$$

Note: specializer **s** is itself a two-argument program.

What is a Compiler Generator?

Program staging:

$$\mathbf{g} \;=\; [\mathbf{cog}]\ p$$
$$[\ [\mathbf{g}]\ x]\ y \;=\; [p](x,y)$$

Terminology:
cog... compiler generator
g ... generating extension
Ershov'77

Characteristic equation:

$$\underbrace{[\ [\ [\mathbf{cog}]\ p\ ]\ x\ ]\ y}_{\textit{3 stages}} \;=\; \underbrace{[p](x,y)}_{\textit{1 stage}}$$

Note: program p staged wrt. implicit division: x known before y.
cog is a program-generator generator.

New: Staging a Specializer

Characteristic equation:

[[[cog] p] x] y = [p](x,y) = out

Special case:

$\underbrace{[\,[\,[cog]\ s\,]\ s\,]\ s}_{\text{3 stages}} \quad = \quad \underbrace{[s](s,s)}_{\text{1 stage}} \quad = \quad cog'''$

bootstrapping 3rd Futamura projection **(double self-application)**

Klimov Romanenko'87 Glück Klimov'95, Glück'09	Futamura'71 Turchin'77, Ershov'78	theory
this talk	Jones et al.'85	practice

Full Bootstrapping

Summary:

$\underbrace{[\,[\,[cog]\ s\,]\ s\,]\ s}_{\textbf{bootstrapping}} \quad = \quad \underbrace{[s](s,s)}_{\text{3rd Futamura projection}} \quad = \quad cog'''$

Full bootstrapping:

1. cog' = [cog] s
2. cog'' = [cog'] s
3. cog''' = [cog''] s
4. cog''' = [cog'''] s **self-generation**

Partial Bootstrapping

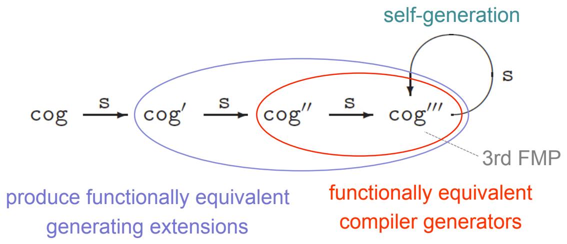
Two important properties:

1. Last two cog'' and cog''' **are functionally equivalent**:
   [cog''] = [cog''']
2. All three cog', cog'', cog''' **produce functionally equivalent generating extensions:**
   [[cog'] p] = [[cog''] p] = [[cog'''] p]

➔ It is not always necessary to perform a full bootstrap.

Q: Can we bootstrap compiler generators in 1 or 2 steps that are "good enough" for practical use ?

Properties of the Bootstrapping Technique



Glück'09

Bootstrapping vs. Futamura Projections

- **Futamura's technique**: "all-or-nothing": unless *double self-application* is successful, *no* compiler generator.
- **Bootstrapping**: can stop generation process at any step (1,2,3) *and* obtain a working compiler generator.

Three bootstrapping steps:

- **1 step**: specializer need *not* be self-applicable (e.g. online); source language need not be Turing-complete; an advantage for DSL (e.g. video device drivers);
- **2 steps**: no loss of transformation strength.
- **3 steps**: alternative to Futamura's technique [Futamura'71,'73].

How to Get Started?

2nd Part of Talk

How to get started?

Chicken-and-Egg Dilemma

Two ways to obtain the initial compiler generator:

1. Write cog by hand.
   [Beckman et al.'75, Holst Launchburg'91, Birkedal Welinder'94, ...]
2. Generate cog by specializer (3rd Futamura projection).
   Requires a self-applicable program specializer.
   [Futamura'71, Jones et al.'85, Romaneko'90, ...]

Ackermann Function in Flowchart

Division:
m=static *n=dynamic*

$$A(m,n) = \begin{cases} n+1 & \text{if } m = 0 \\ A(m-1,1) & \text{if } n = 0 \\ A(m-1, A(m,n-1)) & \text{otherwise} \end{cases}$$

Ershov'78

```
((m n) (ack)
 ((ack  (if (= m 0) done next))
  (next (if (= n 0) ack0 ack1))
  (done (return (+ n 1)))
  (ack0 (n := 1)
        (goto ack2))
  (ack1 (n := (- n 1))
        (n := (call ack m n))
        (goto ack2))
  (ack2 (m := (- m 1))
        (n := (call ack m n))   (n := (call ack m n))
        (return n)) ))
```

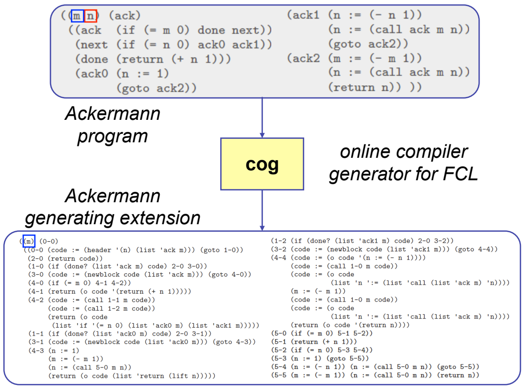
constant assignment: static n

polyvariant call

Three Block Generators

```
(1-0 (if (done? (list 'ack m) code) 2-0 3-0))
(3-0 (code := (newblock code (list 'ack m))) (goto 4-0))
(4-0 (if (= m 0) 4-1 4-2))
(4-1 (return (o code '(return (+ n 1)))))
(4-2 (code := (call 1-1 m code))
     (code := (call 1-2 m code))
     (return (o code (list 'if '(= n 0) (list 'ack0 m) (list 'ack1 m)))))
(1-1 (if (done? (list 'ack0 m) code) 2-0 3-1))
(3-1 (code := (newblock code (list 'ack0 m))) (goto 4-3))
(4-3 (n := 1)
     (m := (- m 1))
     (n := (call 5-0 m n))
     (return (o code (list 'return (lift n)))))
(1-2 (if (done? (list 'ack1 m) code) 2-0 3-2))
(3-2 (code := (newblock code (list 'ack1 m))) (goto 4-4))
(4-4 (code := (o code '(n := (- n 1))))
     (code := (call 1-0 m code))
     (code := (o code (list 'n ':= (list 'call (list 'ack m) 'n))))
     (m := (- m 1))
     (code := (call 1-0 m code))
     (code := (o code (list 'n ':= (list 'call (list 'ack m) 'n))))
     (return (o code '(return n))))
```

Generating a Generating Extension



```
((m n) (ack)                              (ack1 (n := (- n 1))
 ((ack  (if (= m 0) done next))                 (n := (call ack m n))
  (next (if (= n 0) ack0 ack1))                 (goto ack2))
  (done (return (+ n 1)))                  (ack2 (m := (- m 1))
  (ack0 (n := 1)                                 (n := (call ack m n))
        (goto ack2))                             (return n)) ))
```

Ackermann program

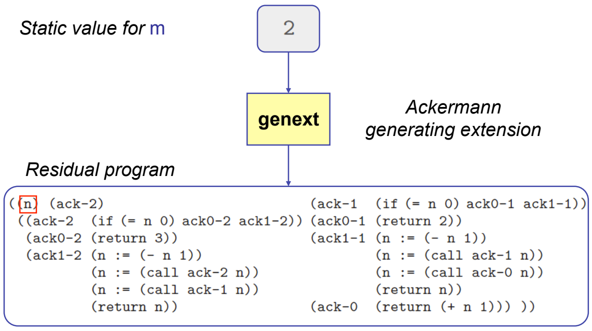
cog

online compiler generator for FCL

Ackermann generating extension

```
((m) (0-0)                                                 (1-2 (if (done? (list 'ack1 m) code) 2-0 3-2))
 ((0-0 (code := (header '(n) (list 'ack m))) (goto 1-0))   (3-2 (code := (newblock code (list 'ack1 m))) (goto 4-4))
  (2-0 (return code))                                      (4-4 (code := (o code '(n := (- n 1))))
  (1-0 (if (done? (list 'ack m) code) 2-0 3-0))                 (code := (call 1-0 m code))
  (3-0 (code := (newblock code (list 'ack m))) (goto 4-0))      (code := (o code
  (4-0 (if (= m 0) 4-1 4-2))                                          (list 'n ':= (list 'call (list 'ack m) 'n))))
  (4-1 (return (o code '(return (+ n 1)))))                     (m := (- m 1))
  (4-2 (code := (call 1-1 m code))                              (code := (call 1-0 m code))
       (code := (call 1-2 m code))                              (code := (o code
       (return (o code                                                (list 'n ':= (list 'call (list 'ack m) 'n))))
        (list 'if '(= n 0) (list 'ack0 m) (list 'ack1 m)))))     (return (o code '(return n))))
  (1-1 (if (done? (list 'ack0 m) code) 2-0 3-1))           (5-0 (if (= m 0) 5-1 5-2))
  (3-1 (code := (newblock code (list 'ack0 m))) (goto 4-3))(5-1 (return (+ n 1)))
  (4-3 (n := 1)                                            (5-2 (if (= n 0) 5-3 5-4))
       (m := (- m 1))                                      (5-3 (n := 1) (goto 5-5))
       (n := (call 5-0 m n))                               (5-4 (n := (- n 1)) (n := (call 5-0 m n)) (goto 5-5))
       (return (o code (list 'return (lift n)))))          (5-5 (m := (- m 1)) (n := (call 5-0 m n)) (return n))
```

Running the Generating Extension



Static value for m

2

genext

Ackermann generating extension

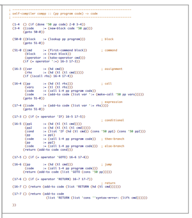
Residual program

```
((n) (ack-2)                              (ack-1  (if (= n 0) ack0-1 ack1-1))
 ((ack-2  (if (= n 0) ack0-2 ack1-2)) (ack0-1 (return 2))
  (ack0-2 (return 3))                     (ack1-1 (n := (- n 1))
  (ack1-2 (n := (- n 1))                          (n := (call ack-1 n))
          (n := (call ack-2 n))                   (n := (call ack-0 n))
          (n := (call ack-1 n))                   (return n))
          (return n))                     (ack-0  (return (+ n 1))) ))
```

Online Compiler Generator in FCL

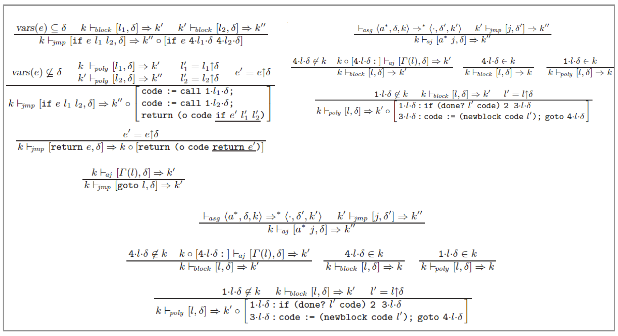
Compiler generator:

Self-compiler:



3 pages of pretty-printed Flowchart program text

Compiler Generator for Flowchart



$$\frac{\mathrm{vars}(e) \subseteq \delta \quad k \vdash_{block} [l_1, \delta] \Rightarrow k' \quad k' \vdash_{block} [l_2, \delta] \Rightarrow k''}{k \vdash_{jmp} [\texttt{if}\ e\ l_1\ l_2, \delta] \Rightarrow k'' \circ [\texttt{if}\ e\ 4{\cdot}l_1{\cdot}\delta\ 4{\cdot}l_2{\cdot}\delta]}$$

$$\frac{\mathrm{vars}(e) \not\subseteq \delta \quad \begin{array}{ll} k \vdash_{poly} [l_1, \delta] \Rightarrow k' & l_1' = l_1{\uparrow}\delta \\ k' \vdash_{poly} [l_2, \delta] \Rightarrow k'' & l_2' = l_2{\uparrow}\delta \end{array} \quad e' = e{\uparrow}\delta}{k \vdash_{jmp} [\texttt{if}\ e\ l_1\ l_2, \delta] \Rightarrow k'' \circ \left[\begin{array}{l} \texttt{code := call}\ 1{\cdot}l_1{\cdot}\delta\texttt{;} \\ \texttt{code := call}\ 1{\cdot}l_2{\cdot}\delta\texttt{;} \\ \texttt{return (o code}\ \underline{\texttt{if}\ e'\ l_1'\ l_2'}\texttt{)} \end{array}\right]}$$

$$\frac{e' = e{\uparrow}\delta}{k \vdash_{jmp} [\texttt{return}\ e, \delta] \Rightarrow k \circ [\texttt{return (o code}\ \underline{\texttt{return}\ e'}\texttt{)}]}$$

$$\frac{k \vdash_{aj} [\Gamma(l), \delta] \Rightarrow k'}{k \vdash_{jmp} [\texttt{goto}\ l, \delta] \Rightarrow k'}$$

$$\frac{\vdash_{asg} \langle a^*, \delta, k \rangle \Rightarrow^* \langle \cdot, \delta', k' \rangle \quad k' \vdash_{jmp} [j, \delta'] \Rightarrow k''}{k \vdash_{aj} [a^*\ j, \delta] \Rightarrow k''}$$

$$\frac{4{\cdot}l{\cdot}\delta \notin k \quad k \circ [4{\cdot}l{\cdot}\delta :] \vdash_{aj} [\Gamma(l), \delta] \Rightarrow k'}{k \vdash_{block} [l, \delta] \Rightarrow k'} \qquad \frac{4{\cdot}l{\cdot}\delta \in k}{k \vdash_{block} [l, \delta] \Rightarrow k} \qquad \frac{1{\cdot}l{\cdot}\delta \in k}{k \vdash_{poly} [l, \delta] \Rightarrow k}$$

$$\frac{1{\cdot}l{\cdot}\delta \notin k \quad k \vdash_{block} [l, \delta] \Rightarrow k' \quad l' = l{\uparrow}\delta}{k \vdash_{poly} [l, \delta] \Rightarrow k' \circ \left[\begin{array}{l} 1{\cdot}l{\cdot}\delta : \texttt{if (done?}\ l'\ \texttt{code) 2}\ 3{\cdot}l{\cdot}\delta \\ 3{\cdot}l{\cdot}\delta : \texttt{code := (newblock code}\ l'\texttt{); goto}\ 4{\cdot}l{\cdot}\delta \end{array}\right]}$$

$$\frac{\vdash_{asg} \langle a^*, \delta, k \rangle \Rightarrow^* \langle \cdot, \delta', k' \rangle \quad k' \vdash_{jmp} [j, \delta'] \Rightarrow k''}{k \vdash_{aj} [a^*\ j, \delta] \Rightarrow k''}$$

$$\frac{4{\cdot}l{\cdot}\delta \notin k \quad k \circ [4{\cdot}l{\cdot}\delta :] \vdash_{aj} [\Gamma(l), \delta] \Rightarrow k'}{k \vdash_{block} [l, \delta] \Rightarrow k'} \qquad \frac{4{\cdot}l{\cdot}\delta \in k}{k \vdash_{block} [l, \delta] \Rightarrow k} \qquad \frac{1{\cdot}l{\cdot}\delta \in k}{k \vdash_{poly} [l, \delta] \Rightarrow k}$$

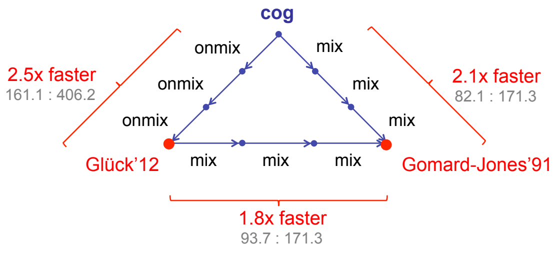
$$\frac{1{\cdot}l{\cdot}\delta \notin k \quad k \vdash_{block} [l, \delta] \Rightarrow k' \quad l' = l{\uparrow}\delta}{k \vdash_{poly} [l, \delta] \Rightarrow k' \circ \left[\begin{array}{l} 1{\cdot}l{\cdot}\delta : \texttt{if (done?}\ l'\ \texttt{code) 2}\ 3{\cdot}l{\cdot}\delta \\ 3{\cdot}l{\cdot}\delta : \texttt{code := (newblock code}\ l'\texttt{); goto}\ 4{\cdot}l{\cdot}\delta \end{array}\right]}$$

See paper for formal definition. Glück’12

Bootstrapping

Last Part of Talk

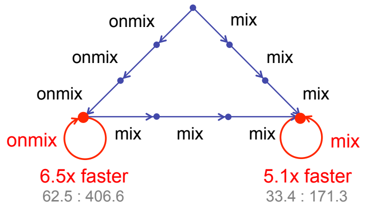
3-Step Bootstrapping



Experimental validation of bootstrapping:
Reproduces the Gomard-Jones mix-cog [1991], but faster.
Reproduces the onmix-cog [G’12], but faster.

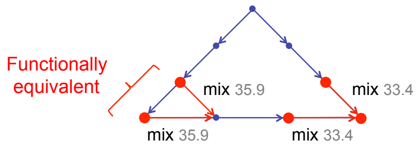
Run times: CPU+GC in ms

Self-Generation



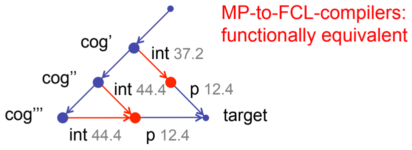
Partial correctness test: **Perfect reproduction.**
Time for self-generation also indicates efficiency.
Desirable: self-generation ≥ 3x fast than 3rd FMP.

2-Step Bootstrapping



All 2nd-step compiler generators practically “**good enough**”:
No compromise in terms of speed.
Size up to twice as large.

1-Step Bootstrapping



Are **1st-step** compiler generators “**good enough**” ?
Depends on initial cog: scenario w/advanced initial cog.
Advantage: no self-application of new specializer required.

MP-interpreter: Sestoft’86, Mogensen’88

Main Results

1. Standard PE is **strong enough** for bootstrapping.
2. Bootstrapping is a **viable alternative** to the 3.FMP.
3. 3-step bootstrapping produces the **exact same programs** and **can be faster** than 3.FMP.
4. 1 and 2-step can produce "**good enough**" compiler generators (not possible with 3.FMP).
5. Reproduced the 1991-Gomard-Jones cog, but faster.

References

Bootstrapping compiler generators:

- Glück R., Bootstrapping compiler generators from partial evaluators. Clarke E.M., et al. (eds.), Perspectives of System Informatics. Proceedings. LNCS 7162, 2012.

Self-applicable online partial evaluation:

- Glück R., A self-applicable online partial evaluator for recursive flowchart languages. Software - Practice and Experience, 42(6), 2012.

Self-generating specializers:

- Glück R., Self-generating program specializers. Information Processing Letters, 110(17), 2010.