

# Bootstrapping Compiler Generators from Partial Evaluators

Robert Glück  
University of Copenhagen

IFIP 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.](#)

2

## 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!

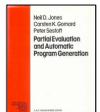
3

## Brief Review of Partial Evaluation

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



- 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].



4

## More formally: What is a Specializer?

Program specialization:

$$r = [s](p,x)$$

$$[r] y = [p](x,y)$$

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

Characteristic equation:

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

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

5

## What is a Compiler Generator?

Program staging:

$$g = [\text{cog}] p$$

$$[[g] x] y = [p](x,y)$$

Terminology:  
cog ... compiler generator  
g ... generating extension

Ershov'77

Characteristic equation:

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

Note: program **p** staged wrt. implicit [division](#): **x** known before **y**. cog is a program-generator generator.

6

## New: Staging a Specializer

Characteristic equation:

$$[[[\text{cog}] p] x] y = [p](x,y) = \text{out}$$

Special case:

$$[[[\text{cog}] s] s] s = [s](s,s) = \text{cog}'''$$

3 stages                      1 stage

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

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

7

## Full Bootstrapping

Summary:

$$[[[\text{cog}] s] s] s = [s](s,s) = \text{cog}'''$$

**bootstrapping**    3rd Futamura projection

**Full bootstrapping:**

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

8

## Partial Bootstrapping

Two important properties:

1. Last two cog'' and cog''' are functionally equivalent:

$$[\text{cog}''] p = [\text{cog}'''] p$$

2. All three cog', cog'', cog''' produce functionally equivalent generating extensions:

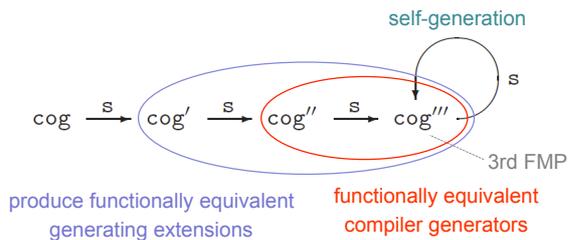
$$[[\text{cog}'] p] = [[\text{cog}''] p] = [[\text{cog}'''] p]$$

→ 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 ?

10

## Properties of the Bootstrapping Technique



Glück'09 11

## 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].

12

## How to Get Started?

2nd Part of Talk

13

## 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, ...]

14

## Ackermann Function in Flowchart

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

```

((m n) (ack)
  ((ack (if (= m 0) done next))
    (next (if (= n 0) ack0 ack1))
    (done (return (+ n 1))))
  (ack0 (n := 1) constant assignment: static n
    (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) )) polyvariant call
  
```

Ershov'78

15

## Three Block Generators

```

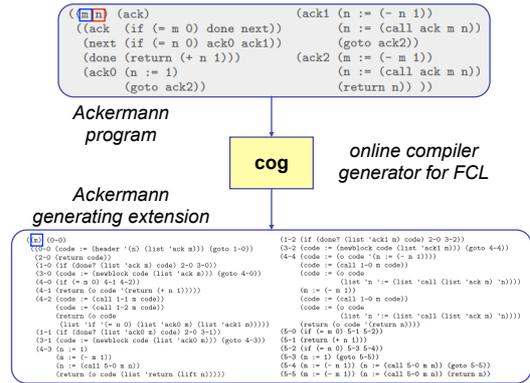
(1-0 (if (done? (list 'ack m) code) 2-0 3-0))
(3-0 (code := (newblock code (list 'ack m))) (goto 4-0))
(4-0 (if (= m 0) 4-1 4-2))
(4-1 (return (o code '(return (+ n 1)))))
(4-2 (code := (call 1-1 m code))
  (code := (call 1-2 m code))
  (return (o code (list 'if '(= n 0) (list 'ack0 m) (list 'ack1 m)))))

(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)))))

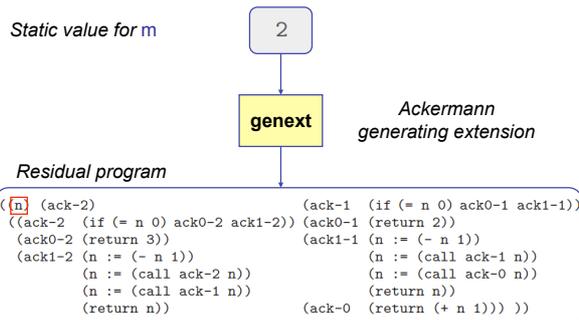
(1-2 (if (done? (list 'ack1 m) code) 2-0 3-2))
(3-2 (code := (newblock code (list 'ack1 m))) (goto 4-4))
(4-4 (code := (o code '(n := (- n 1))))
  (code := (call 1-0 m code))
  (code := (o code (list 'n := (list 'call (list 'ack m) 'n))))
  (m := (- m 1))
  (code := (call 1-0 m code))
  (code := (o code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

16

## Generating a Generating Extension



## Running the Generating Extension



## Online Compiler Generator in FCL



3 pages of pretty-printed Flowchart program text

21



## 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.

37

## 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.

38