On the Mechanics of Program-Generator Generators

Robert Glück
University of Copenhagen

NII Shonan Meeting
Staging and HPC
Japan 2014

Today's Plan

Part 1: Conceptually
- Brief review of generating extensions
- Staging programs into generating extensions

Part 2: Construction
- MetaScheme and multi-level generating extensions
- A compiler generator for recursive Flowchart
- Advanced: bootstrapping a DSL-compiler generator

Generating Extension

Program with two arguments:

out = [p](x,y)

Generating extension of program p:

res = [**gen**] x
[res] y = [p](x,y)

Terminology: **gen** is a *generating extension*
Ershov'77

Characteristic equation:

[[**gen**] x] y = [p](x,y)   correctness: functionally equivalent

2 stages   ***1 stage***

gen: program p staged wrt. division: x known before y

Where does **gen** come from?

Staging area: early, late?

$gen_{Scala, MetaOCaml, Scheme ...}$ = [ ] p   handwrite **gen**

PE area: **This talk:**

$gen_{Scala, MetaOCaml, Scheme ...}$ = [**cog**] p   automate task

Terminology:
cog ... *compiler generator* for historical reasons (p=interpreter)
also called *program-generator generator*

Where does **cog** come from?

"Cogen approach":   early, late?   **This talk:**

cog = [ ] spec   handwrite **cog**

Futamura projections (two options):   automate task:

cog = [spec](spec,spec) 3^{rd}: self-apply spec
cog = [cog'] spec 4^{th}: stage spec

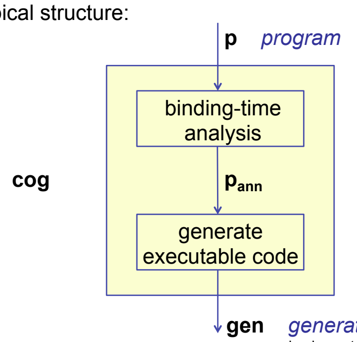
Terminology:
spec ... *program specializer* (e.g. *partial evaluator*)

Just a Game with Symbols?

2nd Part of Talk

Approach: Handwrite **cog**

Typical structure:



staging of computation

annotated program, staging information

e.g. add admin. code, code optimizers

generating extension
implemented in C, ML, Scala, MetaScheme, MetaOCaml ...

Two Examples of Handwritten **cog**

1. **Multi-level compiler generator** (monovariant, offline): [Glück,Jørgensen'95]
   source language: **Scheme**
   target language: **MetaScheme**

2. **Two-level compiler generator** (polyvariant, online): [Glück'12]
   source = target language: **Recursive Flowchart**
   an imperative language w/goto, blocks, lists

MetaScheme

$p ::= d_1 \ldots d_m$

$d ::= (\texttt{define}\ (f\ x_1 \ldots x_n)\ e)$

$e ::= c \mid x \mid (\underline{\texttt{if}}_t\ e_1\ e_2\ e_3)$
$\quad \mid (\underline{\texttt{lambda}}_t\ (x_1 \ldots x_n)\ e) \mid (e_0\ \underline{@}_t\ e_1 \ldots e_n) \mid (\underline{\texttt{let}}_t\ ((x\ e_1))\ e_2)$
$\quad \mid (f\ e_1 \ldots e_n) \mid (\underline{op}_t\ e_1 \ldots e_n) \mid (\underline{\texttt{lift}}_t^s\ e)$

$t = 0$: evaluate op as usual (e.g. by Scheme implementation)
$t > 0$: interpret op as code-generating operation
lift: coerce (time t) value into (time t+s) value

MetaScheme together with multi-level typing rules
is a statically-typed multi-level programming language.
[Glück,Jørgensen'95,'96,'97,'99]

From Program to Generating Extension

```
(define (iprod n v w)
  (if (> n 0)
      (+
       (* (ref n v)
          (ref n w))
       (iprod (- n 1) v w))
      0))
```

Program in Scheme:
Inner product of two
n-dimensional vectors v, w

auto-staged by cog (n:0, v:1, w:2)

```
(define (iprod3 n v w)
  (if (> n 0)
      (_ '+ 2
       (_ '* 2
        (_ 'lift 1 1
         (_ 'ref 1 (lift 1 n) v))
         (_ 'ref 2 (lift 2 n) w))
       (iprod3 (- n 1) v w))
      (lift 2 0)))
```

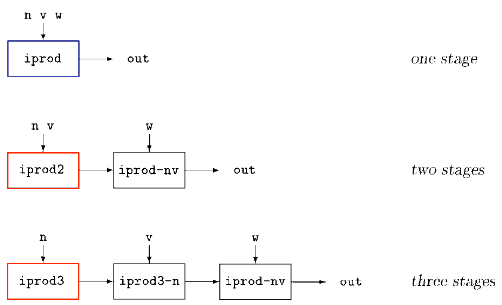
```
(define (_ op t . es)
  (if (= t 1)
      `(,op . ,es)
      `(_ (QUOTE ,op) ,(- t 1) . ,es)))

(define (lift s e)
  (if (= s 1)
      `(QUOTE ,e)
      `(LIFT ,(- s 1) (QUOTE ,e))))
```

3-level Generating Extension (MetaScheme concrete syntax)

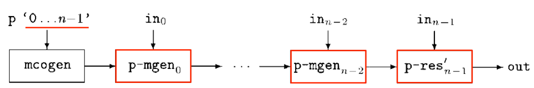
Library (can use peephole opt., algebraic simpl., etc.)

Computing the Inner Product in Stages



auto-staged by cog: iprod2, iprod3 from iprod.
computation performed in 1, 2, 3 stages.

General: Multi-Level Staging



$$\left.\begin{aligned}
\texttt{p-mgen}_0 &= [\![\texttt{mcogen}]\!]_L\ \texttt{p}\ `0\cdots n{-}1\text{'} \\
\texttt{p-mgen}_1 &= [\![\texttt{p-mgen}_0]\!]_L\ \texttt{in}_0 \\
&\ \vdots \\
\texttt{p-mgen}_{n-2} &= [\![\texttt{p-mgen}_{n-3}]\!]_L\ \texttt{in}_{n-3} \\
\texttt{p-res}'_{n-1} &= [\![\texttt{p-mgen}_{n-2}]\!]_L\ \texttt{in}_{n-2} \\
\texttt{out} &= [\![\texttt{p-res}'_{n-1}]\!]_L\ \texttt{in}_{n-1}
\end{aligned}\right\}\ n\ \textit{stages}$$

$$[\![\texttt{p}]\!]_L\ \texttt{in}_0 \cdots \texttt{in}_{n-1} = [\![\cdots[\![[\![\texttt{mcogen}]\!]_L\ \texttt{p}\ `0\cdots n{-}1\text{'}]\!]_L\ \texttt{in}_0 \cdots]\!]_L\ \texttt{in}_{n-1}$$

Generation pipeline: "offline" (order '0...n-1' fixed at start),
"online" (order decided on-the-fly)
[Glück,Jørgensen'97]

Multi-Level Binding-Time Analysis

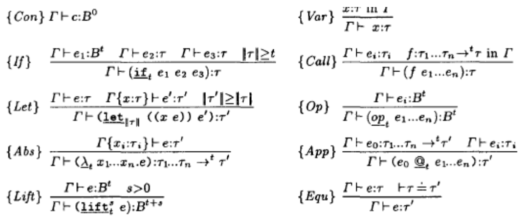


Fig. 7. Typing rules for well-annotated multi-level programs (i ranges over $0 \le i \le n$).

Task of MBTA: given program p and bt-time values (0,...,*n*-1), find a **consistent staging** which is - in some sense - the **best**.

[Glück,Jørgensen'96]

Two Examples of Handwritten **cog**

✔ **1. Multi-level compiler generator** (monovariant, offline):
source language: **Scheme** [Glück,Jørgensen'95]
target language: **MetaScheme**

Next:

2. Two-level compiler generator (polyvariant, online):
source = target language: **Recursive Flowchart** [Glück'12]

Ackermann Function in Flowchart

Initial 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]

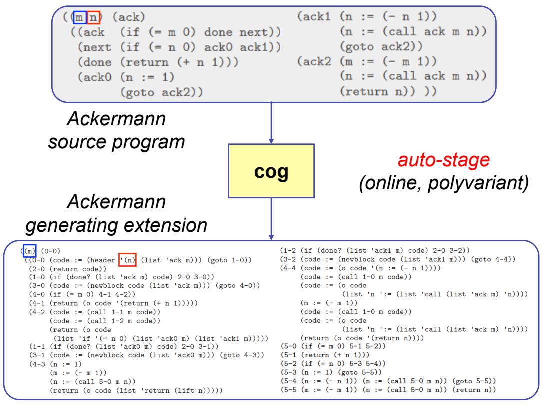
```
((m n) (ack)
 ((ack  (if (= m 0) done next))
  (next (if (= n 0) ack0 ack1))
  (done (return (+ n 1)))
  (ack0 (n := 1)          constant assigned: n static
        (goto ack2))
  (ack1 (n := (- n 1))
        (n := (call ack m n))
        (goto ack2))
  (ack2 (m := (- m 1))
        (n := (call ack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


Ackermann Generating Extension

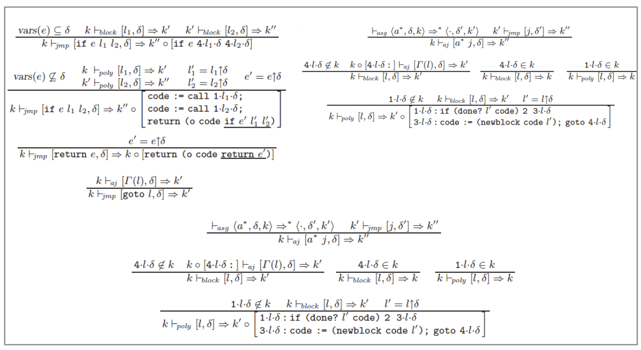
add administrative code

```
(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))       static: copy from source program
(4-1 (return (o code '(return (+ n 1)))))
(4-2 (code := (call 1-1 m code))
     (code := (call 1-2 m code))  dynamic: turn into code generator
     (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



cog for Recursive Flowchart



See paper for definition of compiler generator.
[Glück'12]

More Examples of Handwritten **cog**

✔ **1. Multi-level compiler generator** (monovariant, offline):
source language: **Scheme**
target language: **MetaScheme**

✔ **2. Two-level compiler generator** (polyvariant, online):
source = target language: **Recursive Flowchart**

3. More handwritten cog-systems:

ML-cog [Birkedal,Welinder'94]
C-Mix II [Andersen'94]
PGG, ... [Thiemann'96,'99]

New to Ershov's Generating Extensions

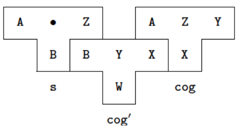
[Ershov'77]

Program **1-stage** *computation*	*Generating extension* **2-stage** *computation*
[interpreter] (pgm, data)	= [[compiler] pgm] data
[parser] (grm, text)	= [[parser-gen] grm] text
[spec] (p, x)	= [[cog] p] x

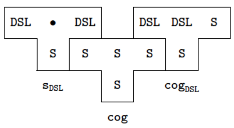
The generating extension of a specializer is a compiler generator.

Advanced: Bootstrapping **cog** by **cog'**

4th Futamura Projection (general case):



Generating cog_{DSL} for a domain-specific language DSL:



cog involves **4 languages** (general case): [Glück '09]
source language A, implementation language X, target language Y, target language Z of the generating extension (produced by cog).

References

Multi-level compiler generator, MetaScheme:

- Glück R., Jørgensen J., Efficient multi-level generating extensions for program specialization. Hermenegildo M., Swierstra S.D. (eds.), PLILP. Proceedings. LNCS 982, 1995.
- Glück R., Jørgensen J., Multi-level specialization (extended abstract). Hatcliff J., et al. (eds.), Partial Evaluation. LNCS 1706, 1999.

Two-level compiler generator, bootstrapping:

- Glück R., Is there a fourth Futamura projection? In: PEPM. Proceedings. 2009.
- Glück R., Bootstrapping compiler generators from partial evaluators. Clarke E.M., et al. (eds.), Perspectives of System Informatics. Proceedings. LNCS 7162, 2012.
- Glück R., A self-applicable online partial evaluator for recursive flowchart languages. Software - Practice and Experience, 42(6), 2012.

... and references therein.