

The Search for Clarity

Mitch Wand

August 24, 2009



Bloomberg

Northeastern University

Or,

How I learned to stop worrying and
love the λ -calculus

Searching for Clarity

- Most people have a limited tolerance for complexity
- Essential vs. incidental complexity
- My approach:
 - Find something I didn't understand
 - Simplify it until I did understand it
 - Find the essential problem
 - Explain it as simply as possible
 - Find an organizing principle
 - Use as little mathematical infrastructure as possible

Outline

- Some examples to aspire to
- Three stories about my early career
- Conclusions & Future Work...

Example: Newton's Laws

- An abstraction of physical reality
 - Mass, velocity, energy
- They are predictive laws
- They page in a whole set of techniques
 - Algebra, calculus, etc.

What are Newton's Laws for Computation?

- Question raised by Matthias and Olin, 3/07
- Surprised to find: I already knew them!
- Each of these
 - Introduces an abstraction of reality
 - Can be used to predict behavior of physical systems (within limits)
 - Leads to a set of techniques for use

First Law (Church's Law):

$$(\lambda x.M)N = M[N/x]$$

Second Law (von Neumann's Law):

$$(\sigma[l := v])(l') = \begin{cases} v & \text{if } l' = l \\ \sigma(l') & \text{otherwise} \end{cases}$$

Third Law (Hoare's Law):

$$\frac{(P \wedge B)\{S\}P}{P\{\mathbf{while } B \mathbf{ do } S\}(P \wedge \neg B)}$$

Fourth Law (Turing's Law):

$$TM_{\text{univ}}(m, n) = TM_m(n)$$

Another example

Programming
Languages

B. Wegbreit*
Editor

(CACM, March 1977)

Subgoal Induction

James H. Morris Jr. and Ben Wegbreit
Xerox Palo Alto Research Center

Subgoal induction presents a way of “thinking recursively”; i.e. assuming one’s ability to solve simpler problems and generating solutions to more complex problems.

Subgoal Induction

- Goal: prove partial correctness of a recursive function
- Define an input-output predicate
 - e.g., you might have
$$\phi(x; z) = [z^2 \leq x < (z + 1)^2]$$
or whatever.
 - This asserts that z is acceptable as a value for $F(x)$.
- For each branch, get a verification condition.

Example

```
(define (F x)
  (if (p x)
      (a x)
      (b (F (c x)))))
```

$$(\forall x)[p(x) \Rightarrow \phi(x; a(x))]$$

$$(\forall x, z)[\neg p(x) \wedge \phi(c(x); z) \Rightarrow \phi(x; b(z))]$$

Getting down to me...

Problem: Give a semantics for actors

- Why was this hard?
 - This was 1973-74
 - Before Sussman & Steele
 - We still didn't entirely trust metacircular interpreters
 - Denotational semantics was just starting
 - Operational semantics was unstructured
- Actors were about message-passing
- Message-passing was a complicated process
 - Everything was an actor, including messages
 - If you receive a message, how do you figure out what's in it?
 - You send it a message, of course!
 - Metacircular interpreter didn't help, since it relied on message-passing

Requirements Creep Ensued

- This rapidly morphed into finding a better general model for defining programming languages.
- Slogans:
 - “every programming language includes a semantic model of computation.”
 - “every (operational) semantics seems to include a programming language.”
 - “if your semantic model is so all-fired great, why not program in it directly?”
- I called this “programming in the model”

My proposal

- A “frame model” of computation
- Each frame consisted of
 - A continuation
 - A set of bindings
 - An accumulator (for passing values)
 - An action
- An action was a primop or a list of actions
 - Generic rules for dealing with lists of actions
 - Each primop was a function from frames to frames

JSBACH: A Semantics-Oriented Language

Every programming language entails a model of computation--the the language designer's idea of how his language works. On the other hand, each of the popular models of computation includes a programming language--the language in which the "transition function" is specified. If we had a truly perspicuous theory of computation, there would be no need for a conventional programming language to serve as an intermediary between our mental models and the semantic theory; we would instead program directly in terms of the model. Our "programming language" would then be the language of the model. A conventional programming language makes a choice of some constructs from the model; each such choice is a point of arbitrariness. Consequently, one way of designing an exceptionally "clean" programming language is to deal in terms of an explicit semantic theory of computation.

Submitted to 2nd POPL (1974)

The submitted summaries should be about 1000-2000 words long, should make clear the significance and originality of the proposed paper and should include comparisons with and references to relevant literature.

The deadline for submission of summaries is August 15, 1974. Authors will be notified of acceptance or rejection by September 30, 1974, and the accepted papers, typed on special forms, will be due at the above address by November 15, 1974.

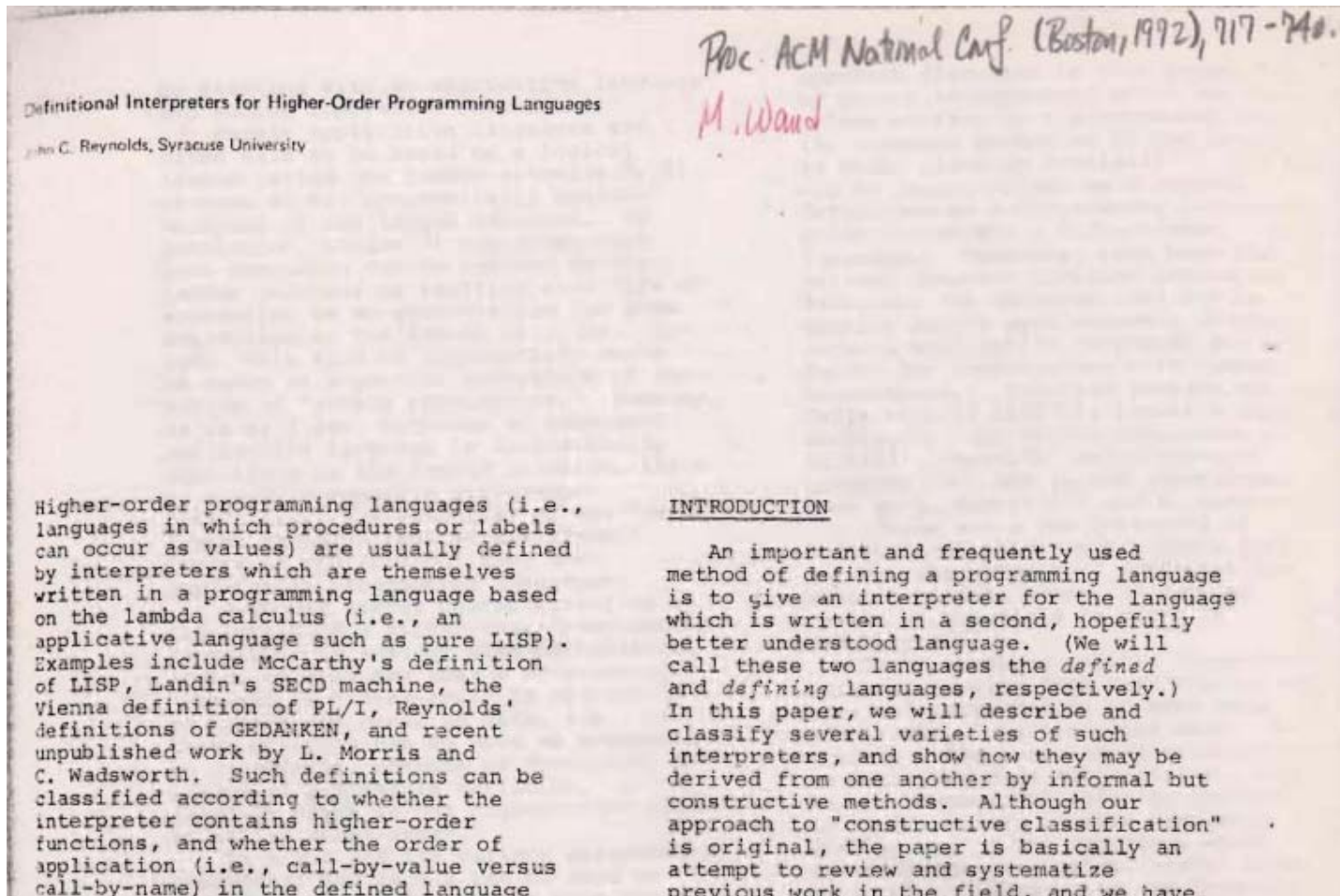
The program committee consists of Alfred V. Aho, Susan L. Graham, Carl Hewitt, M. Douglas McIlroy, James H. Morris and John C. Reynolds (Chairman).

Did NOT cite Reynolds "Definitional Interpreters for Higher-Order Programming Languages" (1972)

Rejection....

- 6/74 submitted to POPL 74
 - 7 pages, double-spaced
- Rejected...
 - But with encouragement from John Reynolds
- 12/74 submitted longer version to CACM
 - Still hadn't cited Reynolds 72
- 12/75 Rejected from CACM
 - Ref rpt:
 - “This paper adds nothing significant to the state of the art.”

Reynolds 72



Definitional Interpreters for Higher-Order Languages

- Introduced a recipe for building an interpreter:
 1. Start with interpreter using recursion, higher-order functions, whatever.
 2. Convert to CPS (“tail form”)
 3. Choose first-order representations for the higher-order functions (“defunctionalize”)
 4. (implicit) Convert to a flowchart-register machine [McCarthy 62]

So when did I read Reynolds 72?

- Sometime in early 1975 (Still before S&S 75)
- This put the last nail in the coffin for JSBACH
 - All the real action seemed to be in the “atomic actions” of the model
 - Reynolds 72 made it clear that the rest was unimportant, too.

December 1975: Lightning Strikes!

M Wand

MASSACHUSETTS INSTITUTE OF TECHNOLOGY
ARTIFICIAL INTELLIGENCE LABORATORY

AI Memo No.

December 1975

SCHEME

AN INTERPRETER FOR EXTENDED LAMBDA CALCULUS

by

Gerald Jay Sussman and Guy Lewis Steele Jr.

Abstract:

Inspired by ACTORS [Greif and Hewitt] [Smith and Hewitt], we have implemented an interpreter for a LISP-like language, SCHEME, based on the lambda calculus [Church], but extended for side effects, multiprocessing, and process synchronization. The purpose of this implementation is tutorial. We wish to:

(1) alleviate the confusion caused by Micro-PLANNER, CONNIVER, etc. by

1976: We play with Scheme

- Many tiny Scheme implementations in Lisp
- Studied recursion-removal, etc.

C O D A

(CEVAL x a) ≡ (+ #clink# [#exp# #env# #evl# #unl# a #clink#]) (DOEVAL x)

(NEVAL x n) ≡ (+ #env# n) (DOEVAL x)

(RETURNTO v c) ≡ (+ #clink# c) (RETURN v)

(RETURN v) ≡ (+ [] v) (cond
 if #clink# then (+ #exp# <1 #clink#>
 (+ #env# <2 #clink#>
 (+ #evl# <3 #clink#>
 (+ #unl# <4 #clink#>
 (+ #pc# <5 #clink#>
 (+ #clink# <6 #clink#>)
 else (error PROCESS-RAN-OUT #exp# FAIL-ACT))

(DOEVAL x) ≡ (+ #exp# x) (+ #pc# '(cond
 if (atom #exp#) then (RETURN (cond
 if (or (numberp #exp#) (primop #exp#)) then #exp#
 if (assq #exp# #env#) then <2 ASSQ>
 else (syneval #exp#))
 if (get <1 #exp#> QUICK) then (RETURN (eval GET))
 if (get <1 #exp#> SLOW) then <CEVAL (eval GET)>
 if (get <1 #exp#> MOVING) then <NEVAL (eval GET)>
 if (get <1 #exp#> MACRO) then <GET [#exp#]>
 else <DOEVLIS (cond
 if (same <1 (car #exp#)> λ) then [[(car #exp#)] (cdr #exp#)]
 else [[] #exp#])>)

(DOEVLIS e u) ≡ (+ #evl# e) (+ #unl# u) (+ #pc# '(cond
 if #unl# then (CEVAL (car #unl#) '(DOEVLIS (snoc #evl# []) (cdr #unl#)))
 if (atom (car #evl#)) then (RETURN <(car #evl#)(cdr #evl#)>)
 if (same <1 (car #evl#)> λ) then (NEVAL <3 (car #evl#)>
 (pairlis <2 (car #evl#)> (cdr #evl#) #env#))
 if (same <1 (car #evl#)> β) then (NEVAL <3<2 (car #evl#)>>
 (pairlis <2<2 (car #evl#)>> (cdr #evl#) <3 (car #evl#)>))
 if (same <1 (car #evl#)> δ) then (RETURNTO <2 #evl#> <2 (car #evl#)>)
 else (error BAD-FUNCTION-EVARGLIST #exp# FAIL-ACT))

L I S P

quote ≡quick <2 #exp#>
define ≡quick (setrl <2 #exp#> [β <3 #exp#> []])
λ ≡quick [β #exp# #env#]
if ≡slow [<2 #exp#> '(DOEVAL (cond if [] then <3 #exp#> else <4 #exp#>))]
evaluate ≡slow [<2 #exp#> '(DOEVAL [])]
catch ≡moving [<3 #exp#> (cons [<2 #exp#> [δ #clink#]] #env#)]
labels ≡moving [<3 #exp#> (nconc (hlabels (mapcar '(λ (d)
 [<1 d> [β <2 d> []])) <2 #exp#>)) #env#)]
(hlabels n) ≡ (mapc '(λ (vc) (change <3<2 vc>> n)) n) n
(EVAL x n) ≡ (+ #clink# [[] [] [] [] [] []]) (NEVAL x n)
 (repeat WHILE (eval #pc#))
 []

CODA: A language on a page
(Dan Friedman, early 1976)

Continuation-Based Program Transformation Strategies (1980)

- Idea: analyze the algebra of possible continuations for a given program
- Find clever representations for this algebra
 - Defunctionalization (Reynolds)

Example

```
(define (fact n)
  (define (fact-loop n k)
    (if (zero? n)
        (k 1)
        (fact-loop (- n 1) (lambda (v) (k (* n v))))))
  (fact-loop n (lambda (v) v)))
```

$k ::= (\text{lambda } (v) v) \mid (\text{lambda } (v) (k (* n v)))$

$k ::= (\text{lambda } (v) (* m v))$

Example, cont'd

```
k ::= (lambda (v) (* m v))
```

```
(lambda (v) (* m v)) => m
```

```
(lambda (v) v) => 1
```

```
(lambda (v) (k (* n v))) => (* k n)
```

```
(k v) => (* k v)
```

```
(define (fact n)
```

```
  (define (fact-loop n k)
```

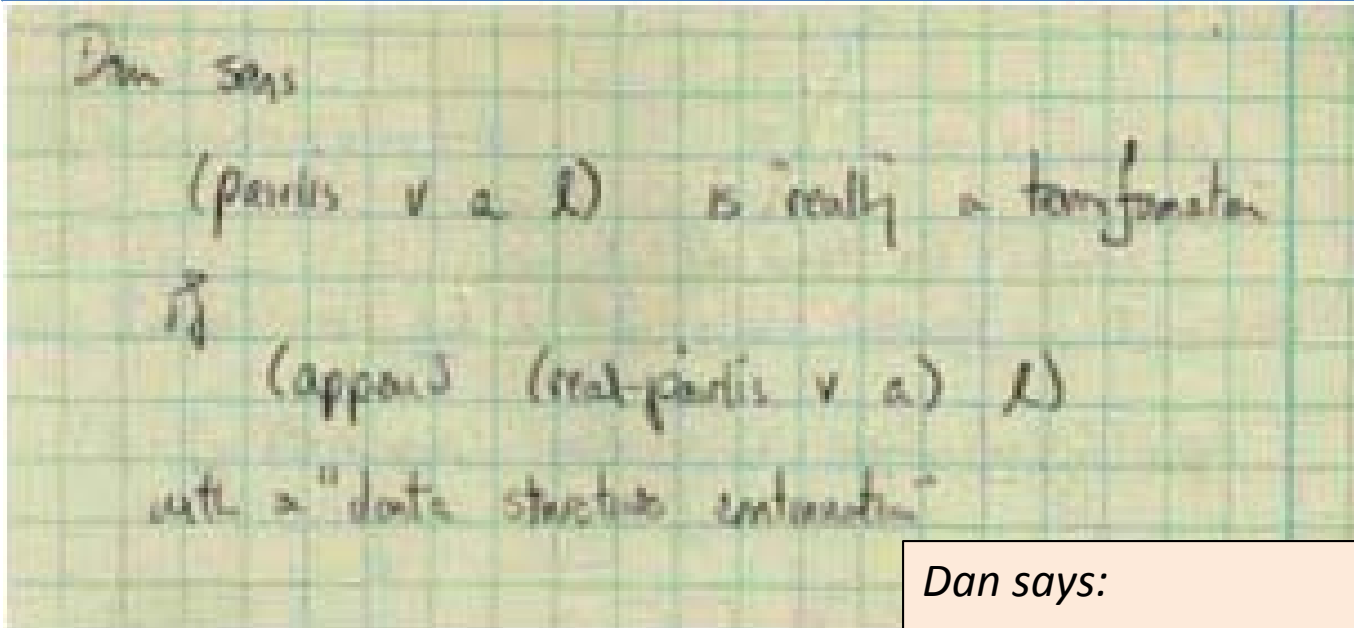
```
    (if (zero? n)
```

```
        k
```

```
        (fact-loop (- n 1) (* k n))))
```

```
  (fact-loop n 1))
```

Where did this come from?



9/22/76

- where did this come from? I don't know
- what did this mean? I didn't know

Dan says:

*(pairlis v a l) is "really" a transformation
of
(append (real-pairlis v a) l)
with a "data structure continuation"*

But it sounded like fun, so I set to work

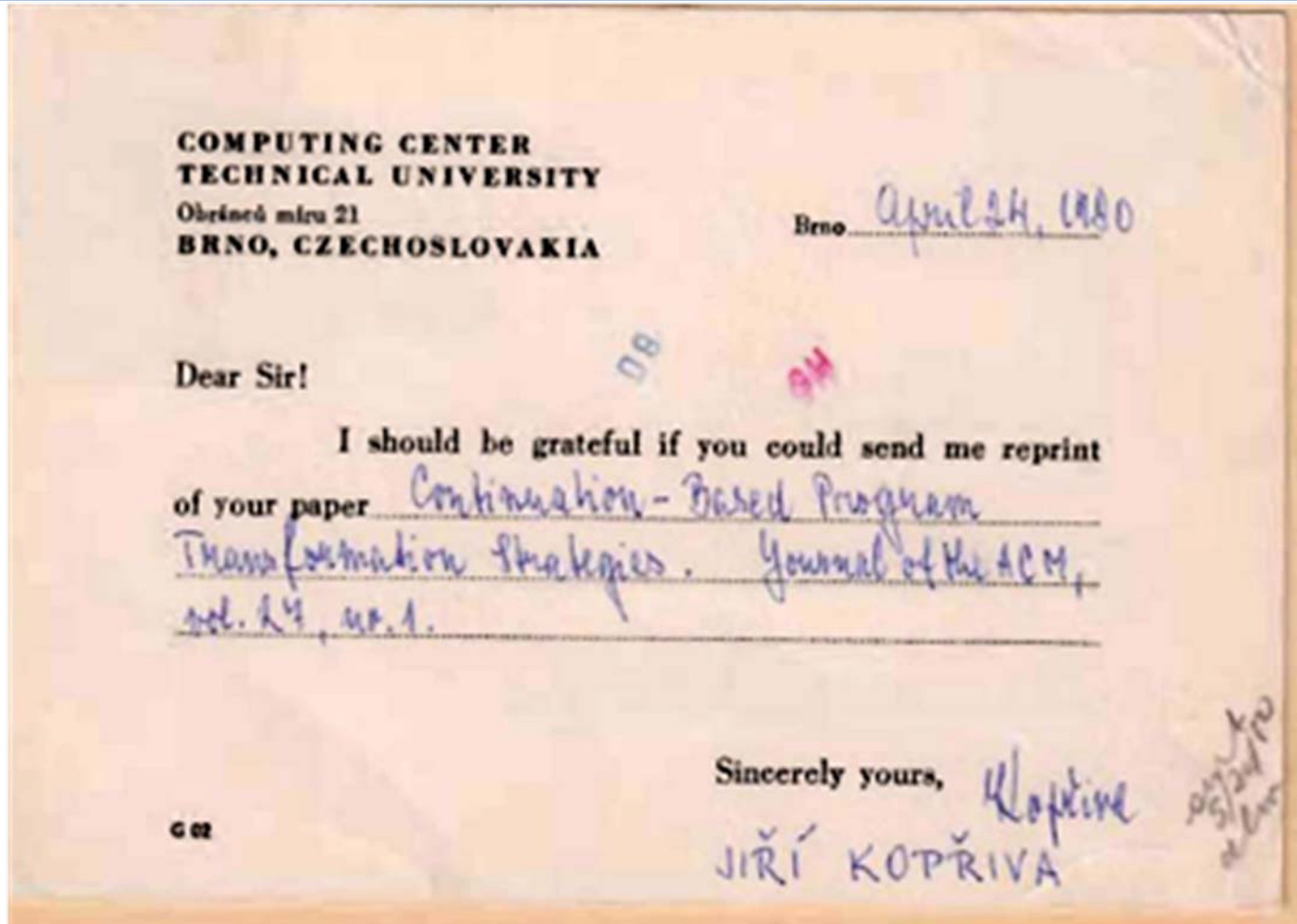
- 9/23-29 more calculations
- 10/2/76: The 91-function in iterative form
 - “single continuation builder; can replace w/ ctr”
 - Notation uses $F(x, \gamma)$, (send v (C1 γ)), like eventual paper.
- 11/27/76: outline of a possible paper, with slogans:
 - “Know thy continuation”
 - “There’s a fold in your future”
 - “Information flow patterns: passing info down as a distillation of the continuation.”
- 12/8/76:
 - “continuations are useful source of generalizations”
- 1-2/77: continued to refine the ideas



But getting it out took forever

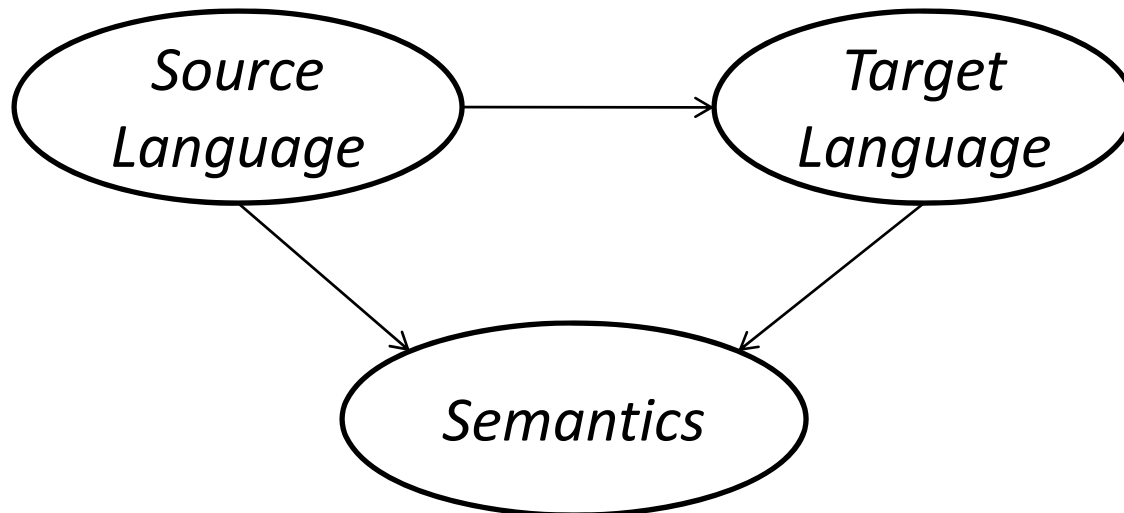
- 3/77 appeared as TR
- 6/77 submitted to *JACM*
- 11/77 accepted subject to revisions
- 1/78 revised TR finished
- 4/78 resubmitted to *JACM*
- 2/79 accepted
- Early 1980: actually appeared

Quaint Customs



Semantics-Directed Machine Architecture (1982)

- Problem: Why did compilers work?
- State of the art:
 - Start with semantics for source, target languages
 - Compiler is transformation from source language to target language
 - Would like compiler to preserve semantics



Sometimes this works

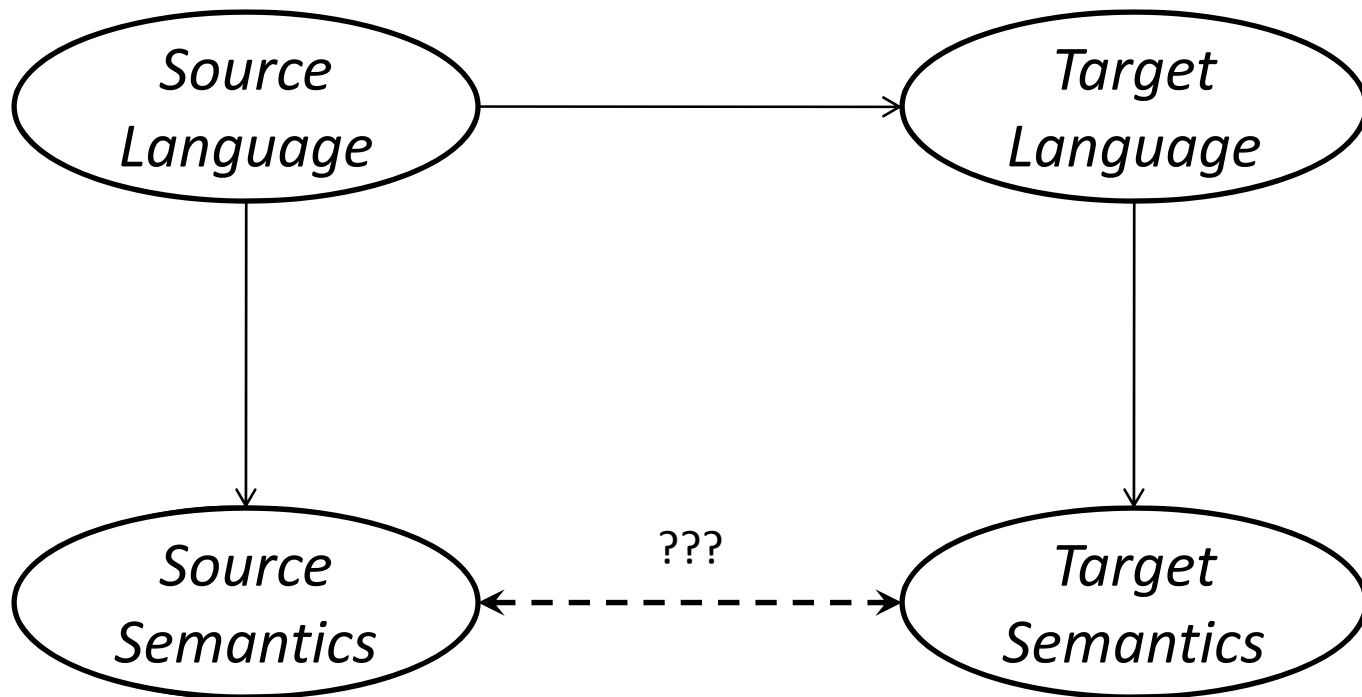
- Source language in direct semantics, target machine uses a stack.

$$\mathit{run}(\mathit{comp}[e], \zeta) = \mathcal{E}[e] :: \zeta$$

- Easy proofs (induction on source expression)
 - [McCarthy & Painter 1967]
- I wrote a compiler this way
 - But what about **CATCH** ?

General Case

- But usually more like:



How to connect source and target semantics?

- A function?
 - In which direction?
- A relation?
 - With what properties?
- Congruence Relations
 - Milne & Strachey
 - Stoy
 - Reynolds
 - Hairy inverse-limit constructions



A New Idea

- Use continuation semantics for both source and target semantics.
 - Connecting direct & continuation semantics was hard.
 - My source language (Scheme!) required continuation semantics.
- Choose clever representation of continuation semantics that would look like machine code

M 1/28/80

Consider:

$$\text{seq}_K(\alpha, \beta) \in K \rightarrow V^n \rightarrow C$$

$$= \lambda k. \lambda \vec{v}. \alpha(\beta k \vec{v})$$

[V → C]

$$\alpha: K \rightarrow C$$

$$\beta: K \rightarrow V^{n+1} \rightarrow C$$

$$K \rightarrow V^n \rightarrow V \rightarrow C$$

$$K \rightarrow V^n \rightarrow K$$

Then we get

$$\text{seq}(E[e_1 + e_2]) = \text{seq}_0(E[e_1], \text{seq}_1(E[e_2], \text{add}))$$

where $\text{add } kv, v_2 = k(v_1 + v_2)$

$$E[I] = \text{fetch}(I)$$

~~$$\text{fetch}(I) = \lambda I. \lambda$$~~

$$\text{fetch}(I) = \lambda k \sigma. k(\sigma(I)) \sigma$$

—

$$\mathcal{P} : Exp \rightarrow Int$$

$$\mathcal{E} : Exp \rightarrow [Int \rightarrow Int] \rightarrow Int$$

$$\mathcal{P}[e] = \mathcal{E}[e](\lambda v.v)$$

$$\mathcal{E}[n] = \lambda k.k(n)$$

$$\mathcal{E}[e_1 - e_2] = \lambda k.\mathcal{E}[e_1](\lambda v_1.\mathcal{E}[e_2](\lambda v_2.k(v_1 - v_2)))$$

$$B_k(\alpha, \beta)v_1 \dots v_k = \alpha(\beta v_1 \dots v_k)$$

$$const(n) = \lambda k.k(n)$$

$$sub = \lambda k v_1 v_2.k(v_1 - v_2)$$

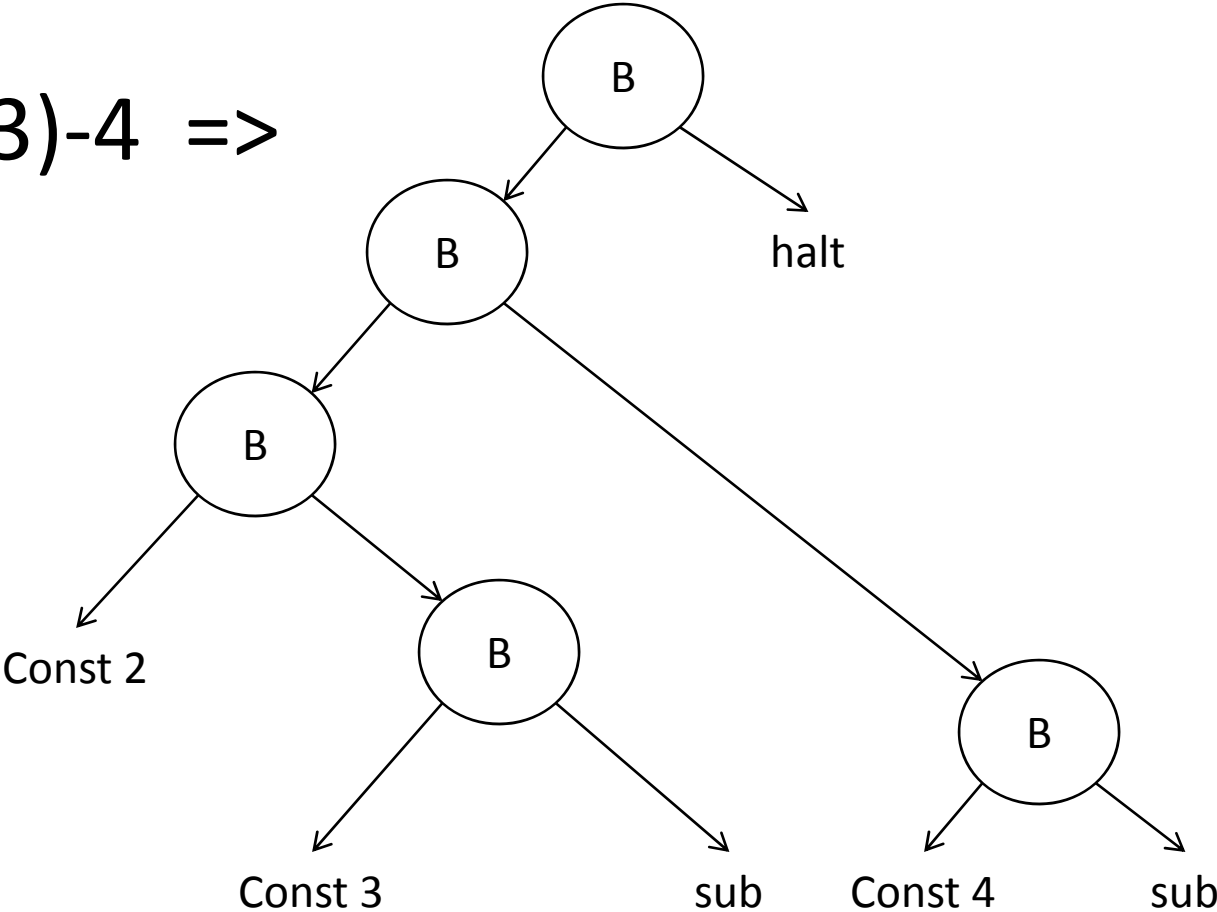
$$halt = \lambda v.v$$

$$\mathcal{P}[e] = B_0(\mathcal{E}[e], halt)$$

$$\mathcal{E}[n] = const(n)$$

$$\mathcal{E}[e_1 - e_2] = B_1(\mathcal{E}[e_1], B_2(\mathcal{E}[e_2], sub))$$

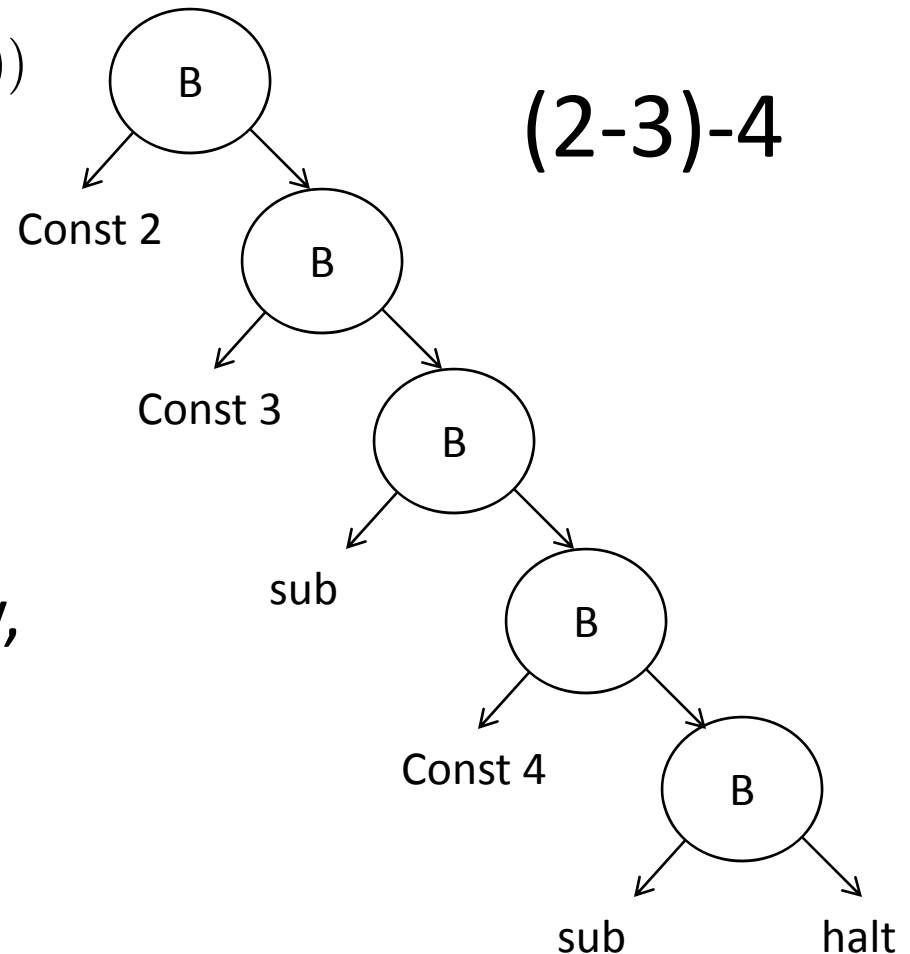
$(2-3)-4 \Rightarrow$



But the B's are associative

$$B_k(B_p(\alpha, \beta), \gamma) = B_{k+p-1}(\alpha, B_k(\beta, \gamma))$$

- Get a linear sequence of “machine instructions”
- “Correct by construction”
- Could do procedures and lexical addressing this way, too



1st Paper: Deriving Target Code as a Representation of Continuation Semantics

- Appeared as IU TR 94 (6/80)
- 8/80 submitted to Sue Graham for *TOPLAS*
- 2/81 rejected w/ encouragement
- Eventually appeared in *TOPLAS* 1982, with material from the 2nd paper...

2nd Paper: Different Advice on Structuring Compilers and Proving Them Correct

- Title taken from a series of papers (L. Morris, POPL 73, Thatcher, Wagner, Wright 1980)
- Did it again with a different language, different proof
- Retold the story in terms of Hoare's abstraction function:
 - from syntactic algebra (representations) to semantic algebra (values)
 - the “master commuting diagram”

The Master Commuting Diagram

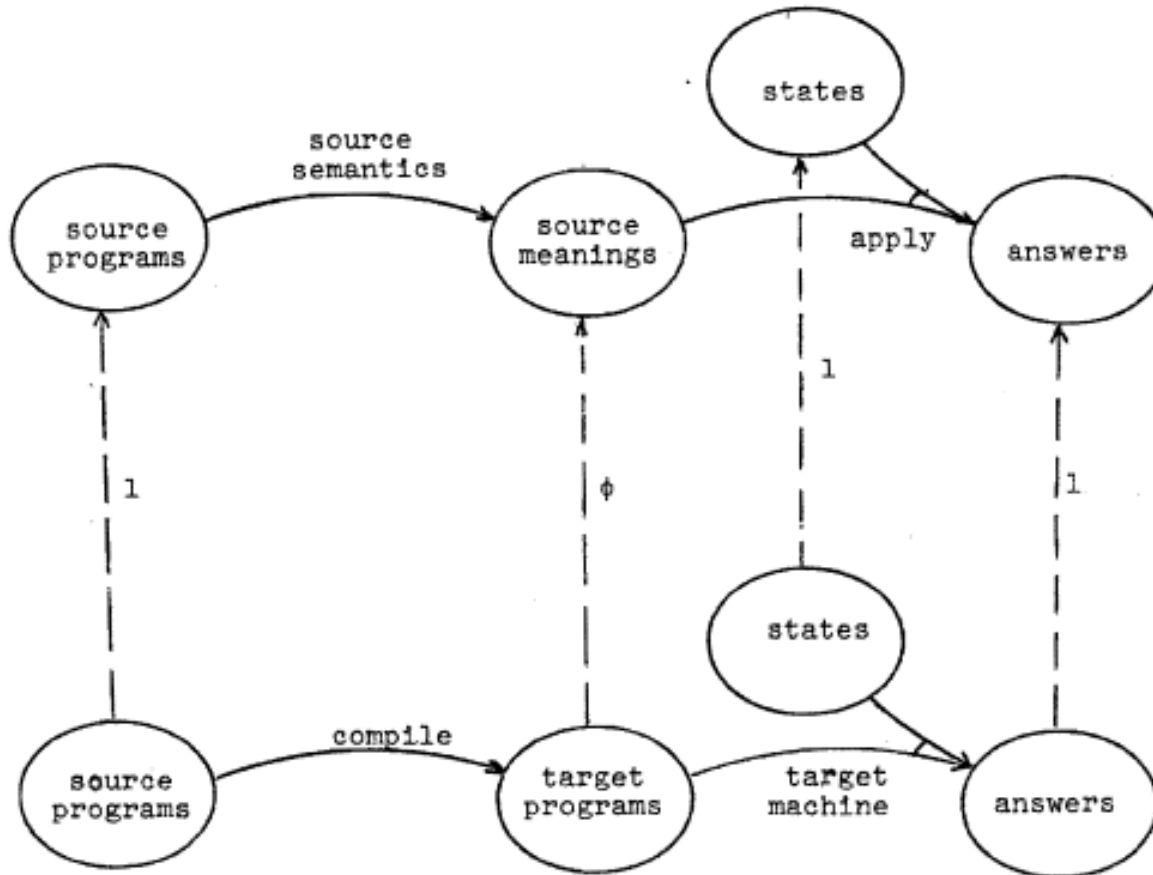


Figure 1 Master Commuting Diagram

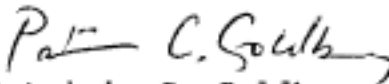
Submitted to POPL 81

Dear Prof. Wand

This is to inform you that "Different Advice on Structuring Compilers and Proving them Correct" has been rejected for presentation at 8th POPL. Only 21 out of 122 submissions were accepted. Many of the rejected papers were basically publishable, but not appropriate for POPL or not ready for publication without extensive revisions. Because the program committee must consider a large number of papers in a short time, it is not feasible to accept papers conditionally or to explain the reasons for rejection of individual papers.

Let me thank you again for your interest. I hope you have more success the next time you decide to submit something for a POPL meeting.

Sincerely,


Patricia C. Goldberg

9/21/80

Different Advice... (long version)

- Long version of POPL submission
- PCF-style proof of adequacy of operational semantics for the machine.
 - Easy because only first-order.
- Written 9/80 (IU TR 95)
- 12/80 submitted to Ravi Sethi for *The Science of Programming* (later became *Science of Computer Programming*)
- 5/81 accepted subject to revision
- Time passed...
- 9/82 withdrawn
 - Good ideas had all appeared in revised TR 94, POPL 82
 - Some bugs, fixable but techniques had become obsolete

I know the published version won't be this way, but I found it terribly annoying to have tables and figures at the end of each section. I mean really annoying! I have the back pages of each section ripped out (and scattered all over my office). (Slight exaggeration.)

3rd Paper: Semantics-Directed Machine Architecture

- 7/81: submitted to POPL 82
 - 10 pages– but double-spaced!
- Written in TROFF

New Ideas

- Connection to *reduction*
- Action of machine simulates reduction of the λ -term.
- Form of term becomes machine architecture
- All you need is syntax!!
 - “concrete semantics” (semantics by compositional translation into some “well-understood metalanguage”)
- Reduction is “eventually outermost”, so by general theorem it will find a normal form if there is one.
 - No longer had to worry about adequacy
 - Solves the problem of that pesky bottom arrow

It's a stack machine!

$code ::= halt \mid B_k(const(n), code) \mid B_k(sub, code)$

$config ::= code v_1 \dots v_k$

$halt v \rightarrow v$

$B_k(const(n), \beta v_1 \dots v_k) \rightarrow \beta v_1 \dots v_k n$

$B_{k+2}(sub, \beta v_1 \dots v_k w_1 w_2) \rightarrow \beta v_1 \dots v_k (w_2 - w_1)$

Accepted!!

After that, things got easier

- POPL 82: Semantics-Directed Machine Architecture
- POPL 83: Loops in Combinator-Based Compilers
- POPL 84: A Types-as-Sets Semantics for Milner-style Polymorphism
- POPL 85: Embedding Type Structure in Semantics
- POPL 86: Finding the Source of Type Errors
- POPL 87: Macro-by-Example: Deriving Syntactic Transformations from the Their Specifications
- POPL 88 Correctness of Static Data Flow Analysis in Continuation Semantics
- + LFP 84, 86, 88, 92

A pretty good
decade 😊

Conclusions and Future Work

- Some technical themes
 - Choosing the formalism to fit the problem
 - Not always category theory!
 - Not always lattices & cpo's
 - Learning to take advantage of the metalanguage
 - In the 70's, everybody said they were doing denotational semantics
 - But really they were just doing compositional translation into λ -calculus (the “well-understood metalanguage”)
 - Leave the hairy math to the mathematicians

Learning from experience

- Some personal themes
 - Learning how to tell a compelling story.
 - Learning when to try to tell the story better (or differently).
 - Learning when to give up and do something else.

Important topics for the next 5 years

- Macros
 - Slogan: Macros should be as familiar a tool in the programmer's toolkit as closures.
 - Goal: write a macros chapter for *EOPL*.

Important topics for the next 5 years

- Parallel and distributed programming
 - Multicore, etc.
 - Distributed algorithms
 - How to prove properties of the algorithms
 - How to implement them (& know that you've done it right)
 - How to program using them (& know that you've done it right)

Important topics for the next 5 years

- The problem is not in the code
 - Our code is remarkably robust
 - Programs deadlock, but they rarely crash
 - Problem is in the *interaction* between programs and external things
 - Other programs
 - The Real World: hardware, people, physical objects
 - The incidental complexity *is* the real complexity
 - How can our expertise help manage this?

Acknowledgements

- My family
- Larry Finkelstein, the administration, and my colleagues at NU CCIS (and at IU)
- National Science Foundation
- MIT Press
- MITRE
- DARPA
- Mozilla Corporation
- Microsoft Research

Acknowledgements

Boleslaw Cieselski
Will Clinger
Bruce Duba
Christopher Dutchyn
Matthias Felleisen
Robby Findler
Dan Friedman
Steven Ganz
David Gladstein
Joshua Guttman
Chris Haynes
David Herman
Gregor Kiczales
Eugene Kohlbecker
Stefan Kolbl
Vasileios Koutavas
Karl Lieberherr
Philippe Meunier
Albert Meyer

Margaret Montenyohl
Patrick O'Keefe
Dino Oliva
Johan Ovlinger
Jens Palsberg
John Ramsdell
Jonathan Rossie
Stuart Shapiro
Olin Shivers
Paul Steckler
Gregory Sullivan
Jerzy Tiuryn
Aaron Turon
Dale Vaillancourt
Dimitris Vardoulakis
Zheng-Yu Wang
Galen Williamson
David Wise

(not) The End (I hope!)