Concatenative Programming

From Ivory to Metal

Jon Purdy

- *Why Concatenative Programming Matters* (2012)
- Spaceport (2012–2013)
  Compiler engineering
- Facebook (2013–2014)
  Site integrity infrastructure (Haxl)
- *There Is No Fork: An Abstraction for Efficient, Concurrent, and Concise Data Access* (ICFP 2014)
- Xamarin/Microsoft (2014–2017)
  Mono runtime (performance, GC)

What I Want in a Programming Language

- Prioritize reading & modifying code over writing it
- Be expressive—syntax closely mirroring high-level semantics
- Encourage “good” code (reusable, refactorable, testable, &c.)
- “Make me do what I want anyway”
- Have an “obvious” efficient mapping to real hardware (C)
- Be small—easy to understand & implement tools for
- Be a good citizen—FFI, embedding
- Don’t “assume you’re the world”

Notable Concatenative Programming Languages

- **Forth** (1970)
  Chuck Moore
- **PostScript** (1982)
  Warnock, Geschke, & Paxton
- **Joy** (2001)
  Manfred von Thun
- **Factor** (2003)
  Slava Pestov &al.
- **Cat** (2006)
  Christopher Diggins
- **Kitten** (2011)
  Jon Purdy
- **Popr** (2012)
  Dustin DeWeese
- …

History

Three Formal Systems of Computation

- Lambda Calculus (1930s)
  Alonzo Church
- Turing Machine (1930s)
  Alan Turing
- Recursive Functions (1930s)
  Kurt Gödel

Church's Lambdas

e	::=	x	Variables
	|	$\lambda x.\ e$	Functions
	|	$e_1\ e_2$	Applications

$\lambda x.M[x]$	$\Rightarrow$	$\lambda y.M[y]$	α-conversion
$(\lambda x.M)E$	$\Rightarrow$	$M[E/x]$	β-reduction

$\lambda x.x \cong \lambda y.y$

$\lambda x.(\lambda y.x) \cong \lambda y.(\lambda z.y)$

$$\begin{aligned}
&(\lambda x.\lambda y.\lambda z.xz(yz))(\lambda x.\lambda y.x)(\lambda x.\lambda y.x)\\
&\cong (\lambda y.\lambda z.(\lambda x.\lambda y.x)z(yz))(\lambda x.\lambda y.x)\\
&\cong \lambda z.(\lambda x.\lambda y.x)z((\lambda x.\lambda y.x)z)\\
&\cong \lambda z.(\lambda x.\lambda y.x)z((\lambda x.\lambda y.x)z)\\
&\cong \lambda z.z
\end{aligned}$$

Turing's Machines

$M = \langle Q, \Gamma, b, \Sigma, \delta, q_0, F \rangle$

Q	Set of states
Γ	Alphabet of symbols
$b \in \Gamma$	Blank symbol
$\Sigma \subseteq \Gamma \setminus \{b\}$	Input symbols
$q_0 \in Q, F \subseteq Q$	Initial & final states
δ	State transition function

$\delta : (Q \setminus F) \times \Gamma \to Q \times \Gamma \times \{L, R\}$

- Begin with initial state & tape
- Repeat:
    - If final state, then halt
    - Apply transition function
    - Modify tape
    - Move left or right

Gödel's Functions

$f(x_1, x_2, \ldots, x_k) = n$	Constant
$S(x) = x + 1$	Successor
$P_i^k(x_1, x_2, \ldots, x_k) = x_i$	Projection
$f \circ g$	Composition
$\rho(f, g)$	Primitive recursion
$\mu(f)$	Minimization

~~Three~~ Four Formal Systems of Computation

- Lambda Calculus (1930s)
  Alonzo Church
- Turing Machine (1930s)
  Alan Turing
- Recursive Functions (1930s)
  Kurt Gödel
- **Combinatory Logic (1950s)**
  Moses Schönfinkel, Haskell Curry

Combinatory Logic (SKI, BCKW)

Just combinators and applications!

$Sxyz = xz(yz)$	Application
$S = \lambda x.\lambda y.\lambda z.xz(yz)$	“Starling”
$Kxy = x$	Constant
$K = \lambda x.\lambda y.x$	“Kestrel”
$Ix = x$	Identity
$I = \lambda x.x$	“Idiot”

$Bxyz = x(yz)$	Compose
$Cxyz = xzy$	Flip
$Kxy = x$	Constant
$Wxy = xyy$	Duplicate

$SKKx = Kx(Kx) = x$

M	$= SII$	$= \lambda x.xx$
L	$= CBM$	$= \lambda f.\lambda x.f(xx)$
Y	$= SLL$	$= \lambda f.(\lambda x.f(xx))(\lambda x.f(xx))$

What is concatenative programming?

Turing machines → imperative
Lambda calculus → functional
Combinatory logic →* concatenative

“A **concatenative programming language** is a **point-free** computer programming language in which **all expressions denote functions**, and the **juxtaposition** of expressions denotes function **composition**.”

— Wikipedia,
Concatenative Programming Language

“...a **point-free** computer programming language...”

Point-Free Programming

```
find . -name '*.txt'
 | awk '{print length($1),$1}'
 | sort -rn
 | head
```

```
hist ∷ String → [(Char, Int)]
hist = map (head &&& length)
  . group . sort
  . filter (not . isSpace)
```

```
define hist (List<Char>
  → List<Pair<Char, Int>>):

  { is_space not } filter
  sort group
  { \head \length both_to
    pair } map
```

Point-Free (Pointless, Tacit) Programming

- Programming: dataflow style using combinators to avoid references to variables or arguments
- Topology/geometry: abstract reasoning about spaces & regions without reference to any specific set of "points"
- Variables are "`goto` for data": unstructured, *sometimes* needed, but structured programming is a better default
- "Name code, not data"

Value-Level Programming

Can Programming Be Liberated from the Von Neumann Style? (1977) John Backus

CPU & memory connected by “von Neumann bottleneck” via primitive “word-at-a-time” style; programming languages reflect that

```
int inner_product
(int n, int a[], int b[])
{
  int p = 0;
  for (int i = 0; i < n; ++i)
    p += a[i] * b[i];
  return p;
}
```

Value-Level Programming

```
int inner_product
(int n, int a[], int b[])
{
  int p = 0;
  for (int i = 0; i < n; ++i)
    p += a[i] * b[i];
  return p;
}
```

```
n=3; a={1, 2, 3}; b={6, 5, 4};
p ←  0;
i ←  0;
p ←  0 + 1 * 6 = 6;
i ←  0 + 1     = 1;
p ←  6 + 2 * 5 = 16;
i ←  1 + 1     = 2;
p ← 16 + 3 * 4 = 28;
28
```

Value-Level Programming

- Semantics & state closely coupled: values depend on all previous states
- Too low-level:
    - Compiler *infers* structure to optimize (e.g. vectorization)
    - Programmer mentally executes program or steps through it in a debugger
- No high-level combining forms: everything built from primitives
- No useful algebraic properties:
    - Can’t easily factor out subexpressions without writing “wrapper” code
    - Can’t reason about subparts of programs without context (state, history)

FP

Def InnerProd ≡
(Insert +) ∘ (ApplyToAll ×) ∘ Transpose

Def InnerProd ≡
(/ +) ∘ (α ×) ∘ Trans

```
innerProd ∷ Num a ⇒ [[a]] → a
innerProd = sum
  . map product
  . transpose
```

FP

Def InnerProd ≡
(Insert +) ∘ (ApplyToAll ×) ∘
Transpose

Def InnerProd ≡
(/ +) ∘ (α ×) ∘ Trans

InnerProd:⟨⟨1, 2, 3⟩, ⟨6, 5, 4⟩⟩
((/ +) ∘ (α ×) ∘ Trans):⟨⟨1,2,3⟩, ⟨6,5,4⟩⟩
(/ +):((α ×):(Trans:⟨⟨1,2,3⟩, ⟨6,5,4⟩⟩))
(/ +):((α ×):⟨⟨1,6⟩, ⟨2,5⟩, ⟨3,4⟩⟩)
(/ +):(⟨×:⟨1,6⟩, ×:⟨2,5⟩, ×:⟨3,4⟩⟩)
(/ +):⟨6,10,12⟩
+:⟨6, +:⟨10,12⟩⟩
+:⟨6,22⟩
28

Function-Level Programming

- Stateless: values have no dependencies over time; all data dependencies are explicit
- High-level:
  - Expresses intent
  - Compiler *knows* structure
  - Programmer reasons about large conceptual units

- Made by *only* combining forms
- Useful algebraic properties
- Easily factor out subexpressions:
  Def SumProd ≡ (+ /) ◦ (α ×)
  Def ProdTrans ≡ (α ×) ◦ Trans
- Subprograms are all pure functions—all context explicit

J

```
innerProd =: +/@:(*/"1@:|:)

innerProd >1 2 3; 6 5 4

(+/ @: (*/"1 @: |:))
  >1 2 3; 6 5 4

+/ (*/"1 (|: >1 2 3; 6 5 4))

+/ (*/"1 >1 6; 2 5; 3 4)

+/ 6 10 12

28
```

J

You *can* give verbose names to things:

```
sum       =: +/
of        =: @:
products  =: */"1
transpose =: |:

innerProduct =:
  sum of products of transpose
```

(J programmers don’t.)

Function-Level Programming: Summary

- Primitive pure functions
- Combining forms: combinators, HoFs, “forks” & “hooks”
- Semantics defined by *rewriting*, *not* state transitions
- Enables purely algebraic reasoning about programs (“plug & chug”)
- Reuse mathematical intuitions from non-programming education
- Simple factoring of subprograms: “extract method” is cut & paste

~~Three~~ ~~Four~~ Five Formal Systems of Computation

- Lambda Calculus (1930s)
  Alonzo Church
- Turing Machine (1930s)
  Alan Turing
- Recursive Functions (1930s)
  Kurt Gödel
- Combinatory Logic (1950s)
  Moses Schönfinkel, Haskell Curry
- **Concatenative Calculus (~2000s)**
  Manfred von Thun, Brent Kirby

Concatenative Calculus

The Theory of Concatenative Combinators (2002) Brent Kirby

E	::=	C	Combinator
	|	$[\, E\,]$	Quotation
	|	$E_1\ E_2$	Composition $(E_2 \circ E_1)$

$[\, A\,]$ dup	$= [\, A\,]\ [\, A\,]$
$[\, A\,]\ [\, B\,]$ swap	$= [\, B\,]\ [\, A\,]$
$[\, A\,]$ drop	$=$
$[\, A\,]$ quote	$= [\, [\, A\,]\,]$
$[\, A\,]\ [\, B\,]$ cat	$= [\, A\ B\,]$
$[\, A\,]$ call	$= A$

Concatenative Calculus

{ dup, swap, drop, quote, cat, call } is Turing-complete!

Smaller basis:

$[\,B\,]\,[\,A\,]\ \mathrm{k}$	$= A$
$[\,B\,]\,[\,A\,]\ \mathrm{cake}$	$= [\,[\,B\,]\,A\,]\,[\,A\,[\,B\,]\,]$
$[\,B\,]\,[\,A\,]\ \mathrm{cons}$	$= [\,[\,B\,]\,A\,]$
$[\,B\,]\,[\,A\,]\ \mathrm{take}$	$= [\,A\,[\,B\,]\,]$

Combinatory Logic (BCKW)

- *B* — apply functions
- *C* — reorder values
- *K* — delete values
- *W* — duplicate values

$Bkab = k(ab)$	Compose/apply
$Ckab = kba$	Flip
$Kka = k$	Constant
$Wka = kaa$	Duplicate

Connection to logic: substructure!

- *W* — contraction
- *C* — exchange
- *K* — weakening

Combinatory Logic

- BCKW = SKI
  - *S* = *B*(*BW*)(*BBC*)
  - *K* = *K*
  - *I* = *WK*
- SKI → LC (expand combinators)
- LC → SKI (abstraction algorithm)
- { *B*, *C*, *K*, *W* } = LC

- BI = ordered + linear
  "Exactly once, in order"
  (Works in any category!)
- BCI = linear
  "Exactly once"
- BCKI = affine
  "At most once"
- BCWI = relevant
  "At least once"

Substructural Type Systems

- Rust, ATS, Clean, Haskell (soon)
- Rust (affine): if a mutable reference exists, it must be unique—eliminate data races & synchronization overhead
- Avoid garbage collection: precisely track lifetimes of objects to make memory usage deterministic (predictable perf.)
- Reason about any resource: memory, file handles, locks, sockets…
- Enforce protocols: “consume” objects that are no longer valid
- Prevent invalid state transitions
- Reversible computing
- Quantum computing

Substructural Rules in Concatenative Calculus

[*A*] dup *k*	= [*A*] [*A*] *k*
W*ka*	= *kaa*
[*A*] [*B*] swap *k*	= [*B*] [*A*] *k*
C*kab*	= *kba*
[*A*] drop *k*	= *k*
K*ka*	= *k*

- Continuations are no longer scary or confusing
- “Current continuation” (call/cc) is simply the remainder of the program
- Saving a continuation is as easy as saving the stacks and instruction pointer

Concatenative Calculus
$\approx$ Combinatory Logic
$+$ Continuation-Passing Style

“...all expressions denote **functions** [...] juxtaposition...denotes function **composition.**”

Stacks

- Composition is the main way to build programs, but what are we composing functions *of*?
- We need a convenient data structure to store the program state and allow passing multiple values between functions
- Most concatenative languages use a heterogeneous *stack*, separate from the call stack, accessible to the programmer
- Other models proposed; stack is convenient & efficient in practice

“Everything is ~~an object~~ ~~a list~~ a function”

Literals (“nouns”) take stack & return it with corresponding value on top.

`2` : $\forall s.\ s \to s \times \mathbb{Z}$
`"hello"` : $\forall s.\ s \to s \times \text{string}$

Operators & functions (“verbs”) pop inputs from & push outputs to stack.

`(+)` : $\forall s.\ s \times \mathbb{Z} \times \mathbb{Z} \to s \times \mathbb{Z}$
`(±)` : $\forall s.\ s \times \mathbb{Z} \times \mathbb{Z} \to s \times \mathbb{Z} \times \mathbb{Z}$

Term `2` is a function, pushes *value* `2`. `2 3 +` is a function, equal to `5`. Can be split into `2 3` and `+` or `2` and `3 +`.

Higher-order functions (“adverbs”) take functions (“quotations”).

```
["ay", "bee", "cee"]
{ "bo" (+) say } each
// aybo beebo ceebo
```

Forth

```
: SQ ( n -- n^2 ) DUP * ;
2 SQ
```

Imperative or pure? Both!

```
2 SQ ⇒ 2 DUP * ⇒ 2 2 * ⇒ 4
2 ⇒ 2 2 ⇒ 4
```

```
: READ  (     -- str ) ... ;
: EVAL  ( str -- val ) ... ;
: PRINT ( val --     ) ... ;
: LOOP  (     --     )
  READ EVAL PRINT LOOP ;
: REPL LOOP ;
```

Stack Shuffling

```
: MAX 2DUP < IF SWAP THEN DROP ;
```

```
5 3     MAX
5 3     2DUP < IF SWAP THEN DROP
5 3 5 3 < IF SWAP THEN DROP
5 3 0   IF SWAP THEN DROP

5 3     DROP
5
```

```
3 5     MAX
3 5     2DUP < IF SWAP THEN DROP
3 5 3 5 < IF SWAP THEN DROP
3 5 1   IF SWAP THEN DROP
3 5     SWAP DROP
5 3     DROP
5
```

Local Variables

Can be more readable to drop from function to value level with local variables.

```
dup2 (<) if { swap } drop
```

```
→ x, y;
if (x < y) { y } else { x }
```

Locals are simply lambda expressions in disguise—composing instead of applying. “Lambda” is decoupled into “anonymous function” and “variable binding”.

Remember: `f g` $= g \circ f = \lambda s.\ g\ (f\ s)$

```
f (→ x; g)
```

$= \lambda s.\ (\lambda x.\ g\ (\mathrm{snd}\ s))\ (\mathrm{fst}\ s)$

Translation to Lambdas

Simple translation from concatenative terms to lambda terms:

```
(a b)'     = λs. b' (a' s)
[ a ]'     = λs. pair (λt. a' t) s     [strict]
           = λs. pair a' s             [lazy]

dup'       = λs. pair (fst s) s
swap'      = λs. pair (fst (snd s)) (pair (fst s) (snd (snd s)))
…
```

A Spoonful of Sugar

Having the option to write operators infix makes it easier to copy & tweak math expressions from other languages, even if it breaks concatenativity.

Same goes for control flow: people are accustomed to **`if`**…**`elif`**…**`else`** and can choose a combinator form if they want its specific advantages.

```
(1 + 2) * (3 + 4)
1 2 (+) 3 4 (+) (*)
```

```
b neg
  + (b ^ 2 - 4 * a * c) sqrt
  / (2 * a)
```

```
b neg b 2 (^) 4 a (*) c (*)
(-) sqrt (+) 2 a (*) (/)
```

Without local variables? Have fun.

Close mapping from syntax to semantics

- Data flow order matches program order: things happen the way you write them
- Syntax monoid: concatenation and empty program; semantic monoid: function composition and identity function on stacks
- Monoid homomorphism from syntax to semantics, preserving identity and joining operation
- Not an isomorphism: multiple input programs can map to the same semantics
- Programs compose! The meaning of the concatenation of two programs is the composition of their meanings
- Can be concatenative at the lexical level (Forth, Factor) or the term level (Kitten)

Factor(ing)

concatenative.org wiki

“C”:

```
var price =
  customer.orders[0].price;
```

Factor:

```
orders>> first price>>
```

Factor(ing)

concatenative.org wiki

```
var orders =
  (customer == null ? null
    : customer.orders);
var order =
  (orders == null ? null
    : orders[0]);
var price =
  (order == null ? null
    : order.price);

dup [ orders>> ] when
dup [ first ] when
dup [ price>> ] when
```

Factor(ing)

concatenative.org wiki

```
dup [ orders>> ] when
dup [ first ] when
dup [ price>> ] when

MACRO: maybe ( quots -- )
  [ '[ dup _ when ] ] map
  [ ] join ;

{ [ orders>> ] [ first ]
  [ price>> ] } maybe
```

Value Propositions of Concatenative Programming

- Pure functions are a good default unit of behavior
- Function composition is a good default means of combining behaviors
- Juxtaposition is a convenient notation for composition
- Having a simple language with a strong mathematical foundation makes it easier to develop tooling and reason about code

Implementation

How do we make this efficient?

- Forth: typically threaded code to support dynamic behavior
- Stack is *reified* in memory for flexibility, but dynamic effects (?DUP, PICK) are frowned upon anyway
- If you have enough arity & type information, you can do ordinary native compilation

```
define ite<R..., S...>
(R..., (R... → S...), (R... →
S...),
  Bool → S...):
  not if { swap } drop call

/* → f, t, x; if (x) { f }
else { t } call */

{"good"} {"oh no"} (1 < 2) ite
```

Implementation

Implementation of Stack-based Languages on Register Machines (1996) M. Anton Ertl

- Spectrum of representations
- Represent the stack in memory
- Cache top value in a register (huge win for code size & perf.)
- Cache multiple values
- FSM of possible registers in calls

- Conversion to SSA/SSI/CPS
  - Program is post-order flattened data flow graph
  - No dynamic stack ops
  - Must know arity of functions / generate specializations
  - Uses standard register allocation techniques
  - Stack shuffling becomes `mov` or no-op

Linear Lisp

Linear Logic and Permutation Stacks—The Forth Shall Be First (1993) Henry Baker

- Variables are consumed when used; copies must be explicit
- Can be compiled efficiently to a stack machine architecture
- Reduce Von Neumann bottleneck

“A…stack cache utilizes its space on the chip & memory bandwidth better than a register bank of the same capacity [...] A linear stack machine should be even more efficient [...] *all* of the data held in the stack cache is live data and is not just tying up space.”

Linear Lisp

Linear Logic and Permutation Stacks—The Forth Shall Be First (1993) Henry Baker

- “Most people describe the top several positions of the Forth stack as ‘locations’, but it is more productive to think of them as ‘busses’, since no addressing is required to read from them at all--the ALU is directly connected to these busses.”
- “...one can conceive of multiple arithmetic operations being performed simultaneously on a number of the top items of the ‘stack’...in parallel”

Linear Lisp

Linear Logic and Permutation Stacks—The Forth Shall Be First (1993) Henry Baker

- Because call rate is so high, and functions are small, you can use the call stack to store not return addresses, but functions themselves
- A “call” copies the contents of a function onto the return stack (queue) and proceeds
- Can be implemented with a cyclic shift register—small loops are just repeated shifts of this register, no branch prediction required

Value Representation: Boxing?

- Pros: uniform representation, generic functions are easy—no need to generate specializations
- Cons: performance overhead of indirections; need RC or GC
- With *no* types or *full* static types, most things can be unboxed
- Small arrays: put elements directly on the stack; size is known
- Closures: copy captured variables onto stack w/ function pointer; invoking closure is just pop+jump
- Otherwise: COW/RC

Static Typing

State of Type Systems in Concatenative Programming

- Most concatenative languages are dynamically typed (Joy, Factor, PostScript) or untyped (Forth)
- There have been a handful of Forths with simple type checkers
- Cat was the first concatenative language with static types based on Hindley–Milner; now defunct
- Nobody else was working on a statically typed one, so I started working on Kitten (2011)

“Simply Aritied” Languages

Type Inference for Stack Languages (2017) Rob Kleffner

Approach used in some static Forths: each function has *m* inputs and *n* outputs

```
dup  : a -- a a
swap : a b -- b a
drop : a --
```

Problem: no stack polymorphism

$\texttt{call}_{1,1}$: `a ( a -- b ) -- b`
$\texttt{call}_{1,2}$: `a ( a -- b c ) -- b c`
$\texttt{call}_{2,1}$: `a b ( a b -- c ) -- c`
`...`

Typing with Tuples

Stack represented as a product type (tuple); “rest of stack” is polymorphic.

- dup : $\forall sa.\ s \times a \to s \times a \times a$
- swap : $\forall sab.\ s \times a \times b \to s \times b \times a$
- drop : $\forall sa.\ s \times a \to s$
- call : $\forall st.\ s \times (s \to t) \to t$
  Modus ponens: given a state & proof (closure) it implies a new state, can get to the new state

Types can get unwieldy—add syntactic sugar to make it usable.

```
define map<S…, A, B>
(S…, List<A>,
  <T…>(T…, A → T… → B)
  → S…, List<B>)
```

```
define map<A, B>
(List<A>, (A → B) → List<B>)
```

Challenges with Stack Polymorphism

- All functions are polymorphic wrt. the part of the stack they don’t touch; higher-order functions are higher-rank; recursion is polymorphic
- *Complete and Easy Bidirectional Type Checking for Higher-Rank Polymorphism* Joshua Dunfield, Neel Krishnaswami

E.g., functional argument to `map` must be applied on different stack states.

map : $\forall sab.$
$(s \times \text{List } a \times (s \times a \to s \times b)$
$\to s \times \text{List } b)$

map : $\forall sab.$
$(s \times \text{List } a \times \boldsymbol{\forall t.}\ (\boldsymbol{t} \times a \to \boldsymbol{t} \times b)$
$\to s \times \text{List } b)$

Challenges with Stack Polymorphism

- Higher-order functions can be polymorphic over the stack—need to generate specializations based on arity (and calling convention)

E.g., functional argument to `dip` may have an arbitrary (but known) effect.

dip : $\forall sta.\ (s \times a \times (s \to t) \to t \times a)$

```
{ drop } dip
swap drop
```

```
{ "meow" } dip
"meow" swap
```

Representing Effects

- Can’t “do” anything with only pure functions; should we throw up our hands and have an impure language? (Forth, Factor, Cat, &al.)
- Haskell uses monads: represent actions as values, build them with pure functions; under the hood, compile to imperative code
- Problem: monads don’t compose—can’t (always, easily) mix effects
- Solution: **algebraic effects**

Effect Types (Permissions) in Kitten

```
define newline (-> +IO):
  "\n" print

define print_or_fail
(Bool -> +IO +Fail):
    -> x;
    if (x):
        "good" print
    else:
        "bad" fail
```

If `f` needs +A and `g` needs +B, `f  g` needs +A +B or +B +A (commutative)

Effect Types ("Permissions") in Kitten

Inspired by Koka (2012) Daan Leijen

Compositional: a function has the effects of the functions it calls.

Polymorphic: a higher-order function has the effect of its argument:

```
map<A, B, +P> (List<A>,
 (A → B +P) → List<B> +P)
```

- Effects: enforce what a function is allowed to do (e.g. I/O, unsafe)
- Coeffects: enforce constraints on the environment where a function is called (e.g. platform)
- RAII: "handler" that discharges a permission (e.g. locking)
- Optimizations: functions can be reordered iff their permissions are commutative

Finally…

Summary

- Simple, elegant foundation
- Surprising connections to deep areas of computer science
- Admits efficient implementation both in theory and in practice
- Amenable to programming “exotic” machines (stack archs, reversible/quantum computers)
- Easy to reason about, modify, & refactor programs; easy to write good tooling with confidence
- Naturally supports static types and effect typing

Questions?

Bonus: Metaprogramming

Forth style: “compiling” vs. “interpreting” words (or mixed, depending on `STATE`). Factor uses this with its “macros” and “parsing words”.

Treat preceding terms as stack, evaluating code at compile time to construct new terms:

```
"%s: %d" #printf
Term → Term
List<Char>, Int32 → +IO
```

Bonus: Arrows

Concatenative programming is closely related to the “arrows” of John Hughes for describing static data flow graphs.

```
(f *** g) (x, y) = (f x, g y)
x y \f \g both
```

```
(f &&& g) x = (f x, g x)
x \f \g both_to
```

```
both<A, B, C, D> // ***
(A, B, (A → C), (B → D) → C,
D)
```

```
both_to<A, B, C> // &&&
(A, (A → B), (A → C) → B, C)
```

```
dip<S…, T…, A> // first
(S…, A, (S… → T…) → T…, A)
```