First-Class Implementations

Climbing Up the Semantic Tower — At Runtime

François-René Rideau, *TUNES Project*

LambdaConf, 2018-06-05

http://fare.tunes.org/files/cs/fci-lc2018.pdf

Based on my PhD thesis (completed in 2017, not defended)

Take Home Points

Reason about Implementations: Category Theory!

Practical Protocol Extraction: First-Class Impl.

Principled Applications: Migration, etc.

Runtime Reflection *and* Static Semantics

Advancement Status

Writing of PhD thesis completed after 20 years!

In my copious spare time: building proof-of-concept.

TODO: Get language implementers on board.

4- PROFIT!

Advancement Status

Writing of PhD thesis completed after 20 years!

In my copious spare time: building proof-of-concept.

TODO: Get language implementers on board.

4- PROFIT!

Point of View and R&D programme

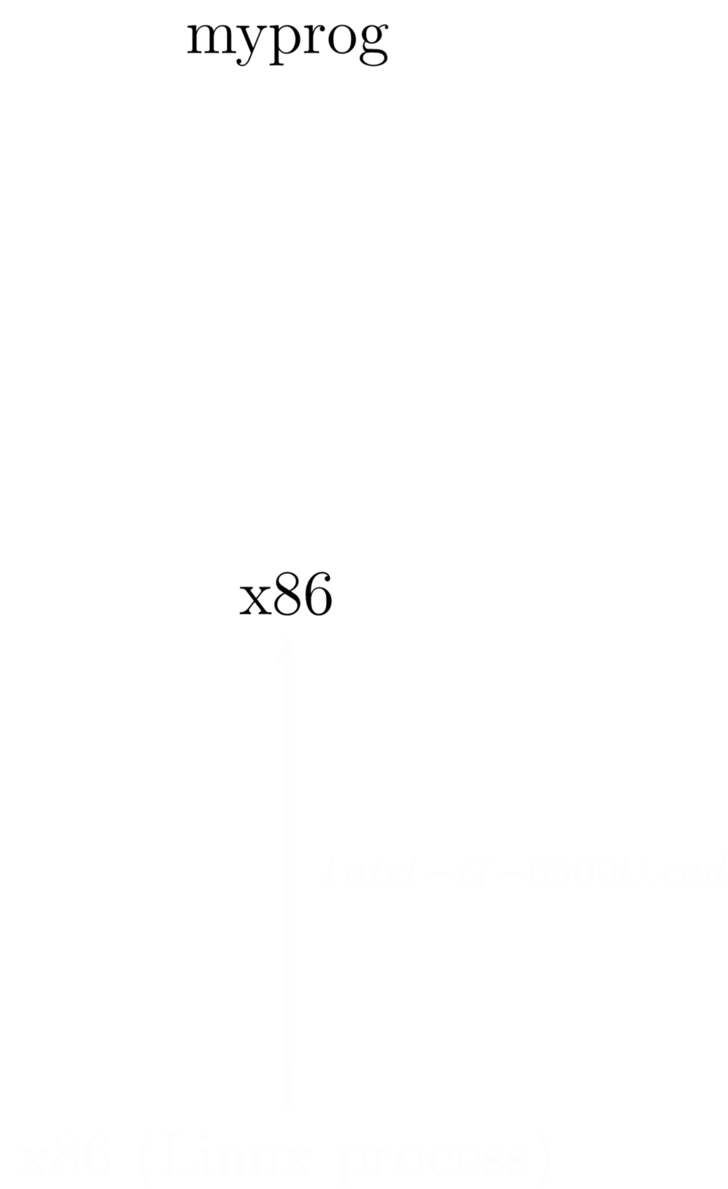
I. Formalizing Implementations

I.1 A Universal Framework

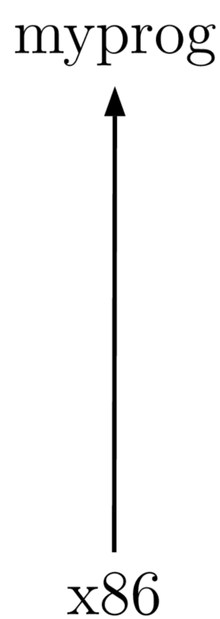
Implementations, informally

You want a program

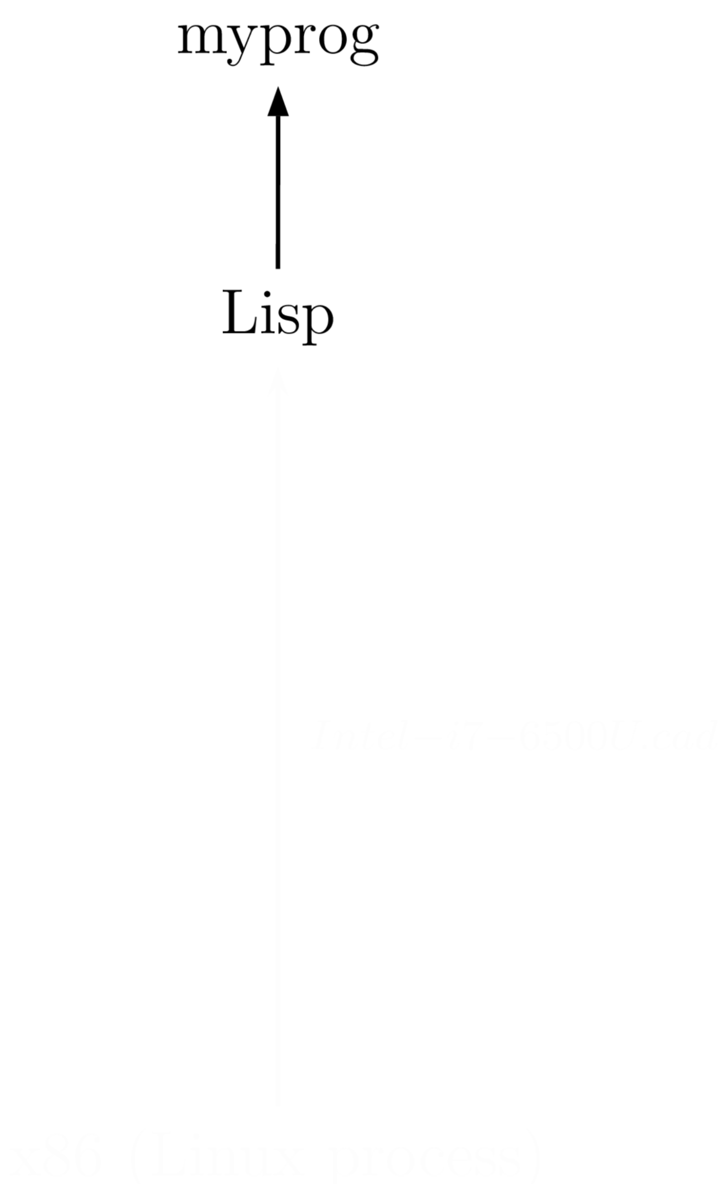
You have a PC



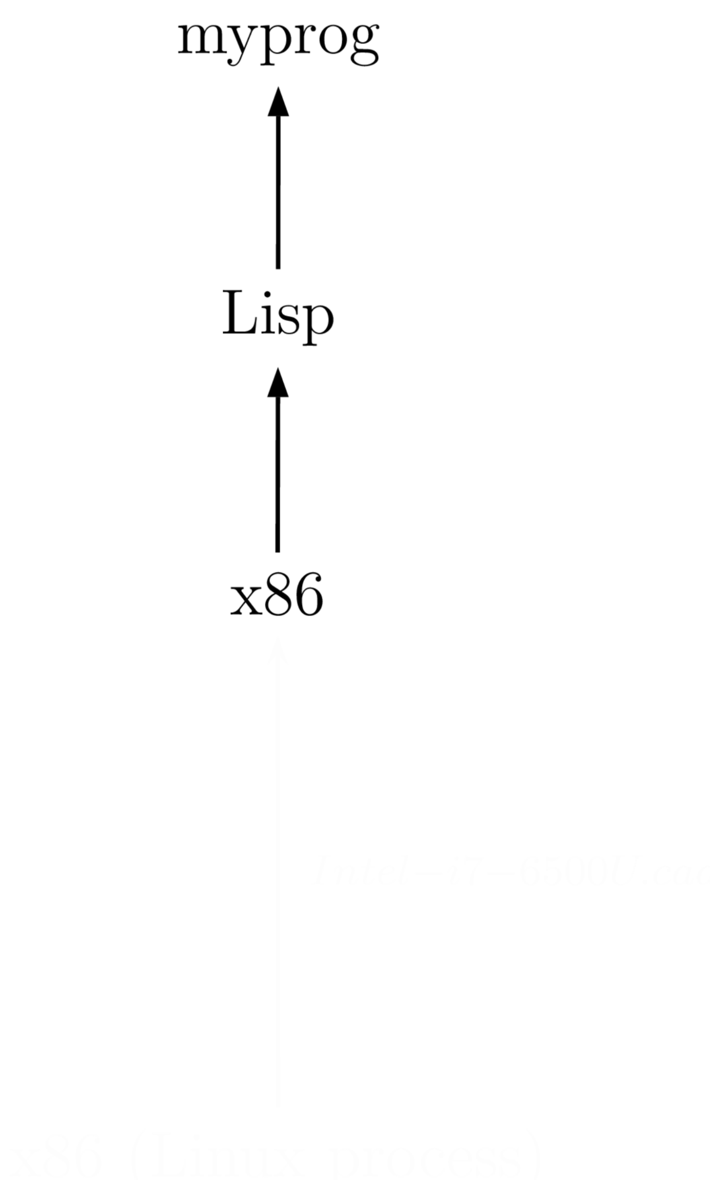
You write an implementation



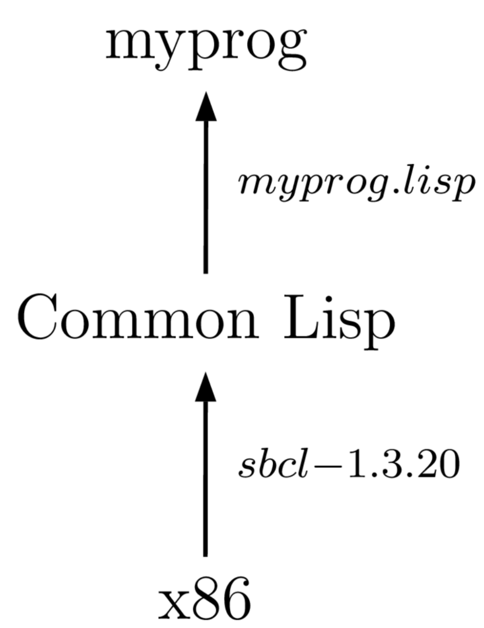
In the best possible language



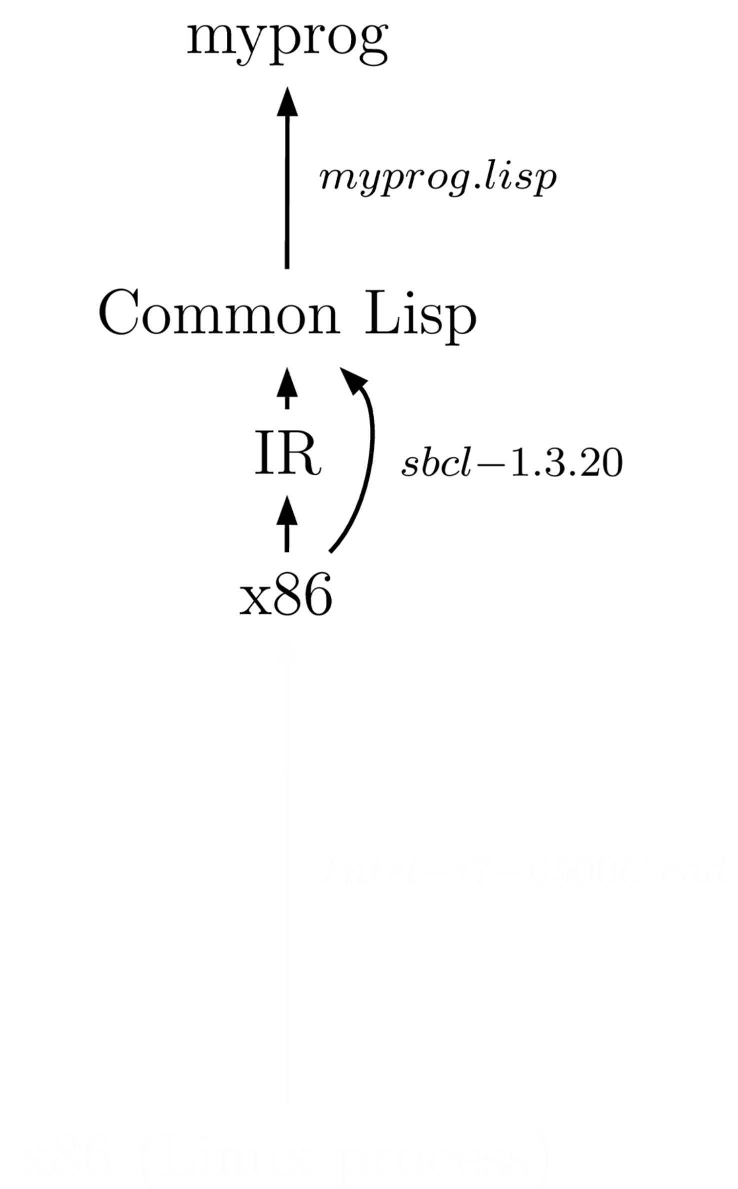
The language itself has an implementation



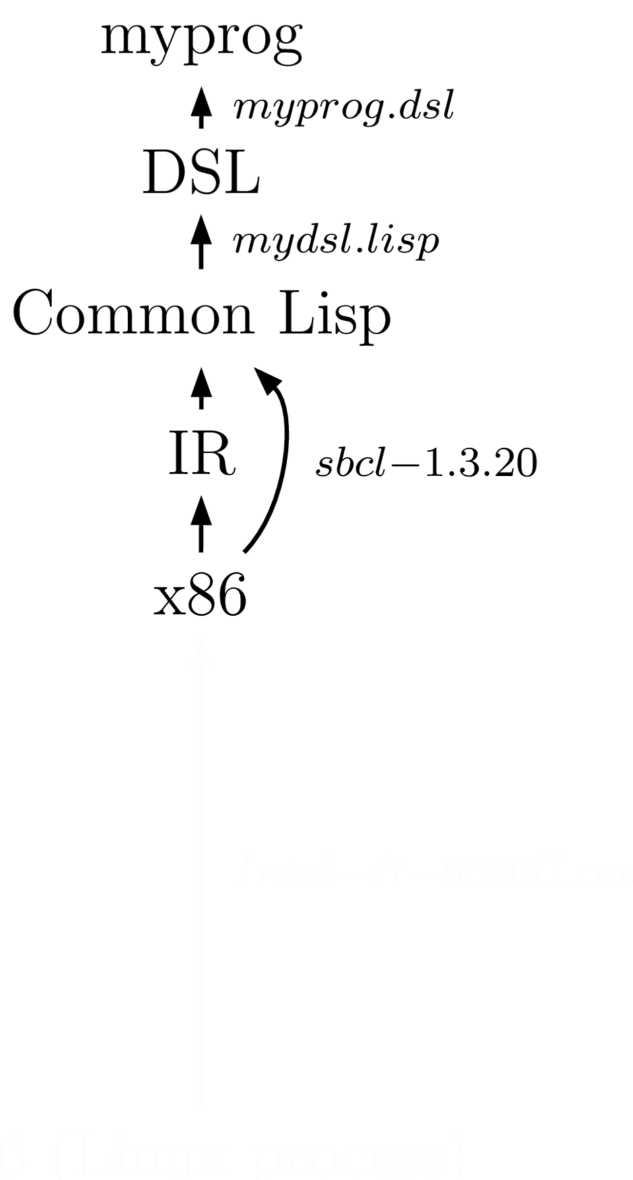
Specific dialects, implementations, versions...



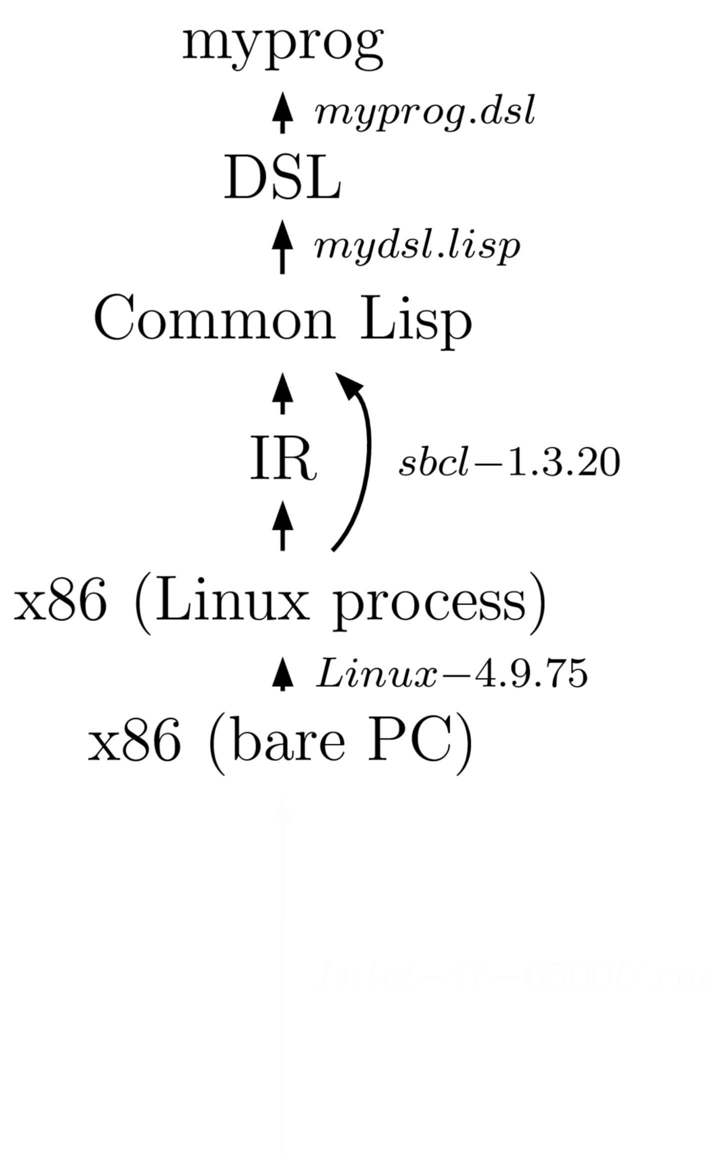
Compiling is hard, use an IR...



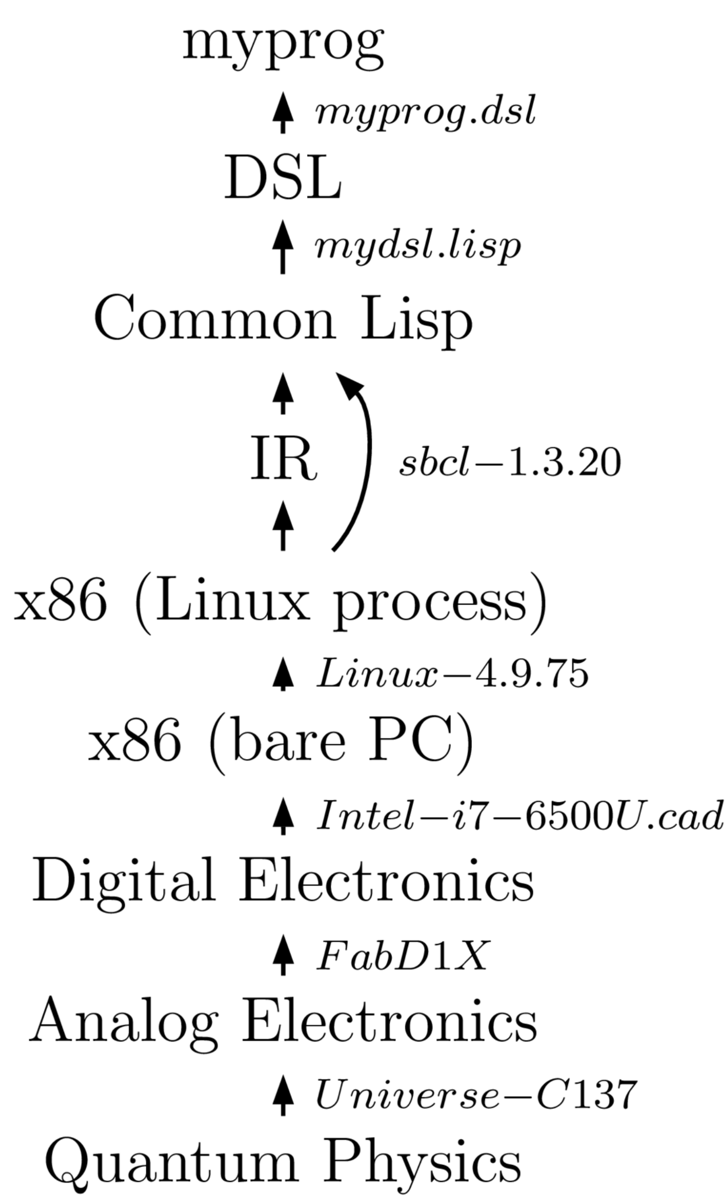
Programming is hard, use a DSL...



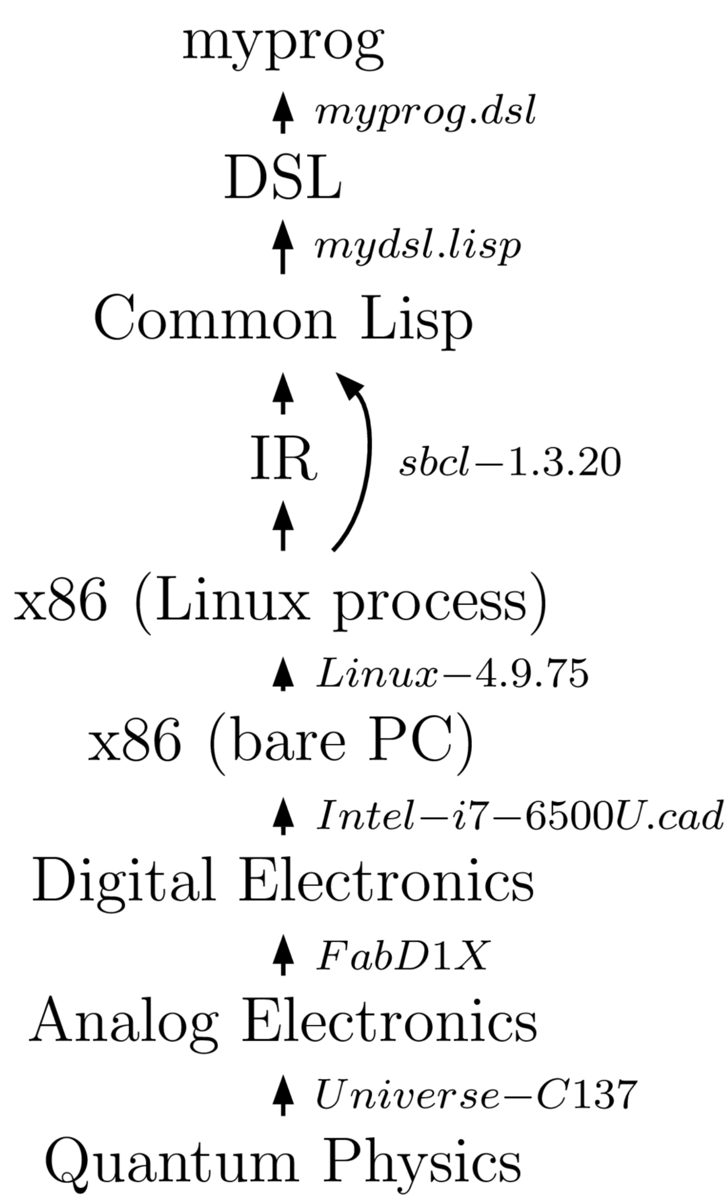
What do you mean, x86?



There is no bottom!



Always finer divisions



Implementations, informally

Implementation: *relating* two computations

Specific implementations: SBCL 1.3.20…

Holding together *Towers* of computations

Reasoning: correctness, other useful properties

Formalization Challenges

First, we must formalize computations

- Few are adequately formalized

- Incompatible formalisms, *to unify*

Then, we must formalize implementations

- What suitable relations between computations?

- What composable properties for these relations?

Existing Semantic Formalisms to Unify

Operational Semantics (Small Step)

Operational Semantics (Big Step)

Labeled Transition Systems

Term Rewriting, Rewrite Logic

Modal Logic, Hoare Logic, Refinement

Partial Order

Abstract State Machines

Denotational Semantics reducing to the above

Denotational Semantics with equational theory

Category Theory

Universal: graphs, preorders, labeled transitions...

Simple core: nodes, arrows, structure preservation

Unlimited abstraction: always higher categories

Structural theorems "for free"

Types, Curry-Howard Isomorphism

Seeking the essential: no incidental punning

Categorical Notation

$$X \xrightarrow[C]{\phi} Y$$

$$x \xmapsto[\phi]{} y$$

$$X \rightarrowtail Y$$

$$X \dashrightarrow Y$$

Computation as Categories

Nodes: states of the computation

Arrows: transitions between states, traces

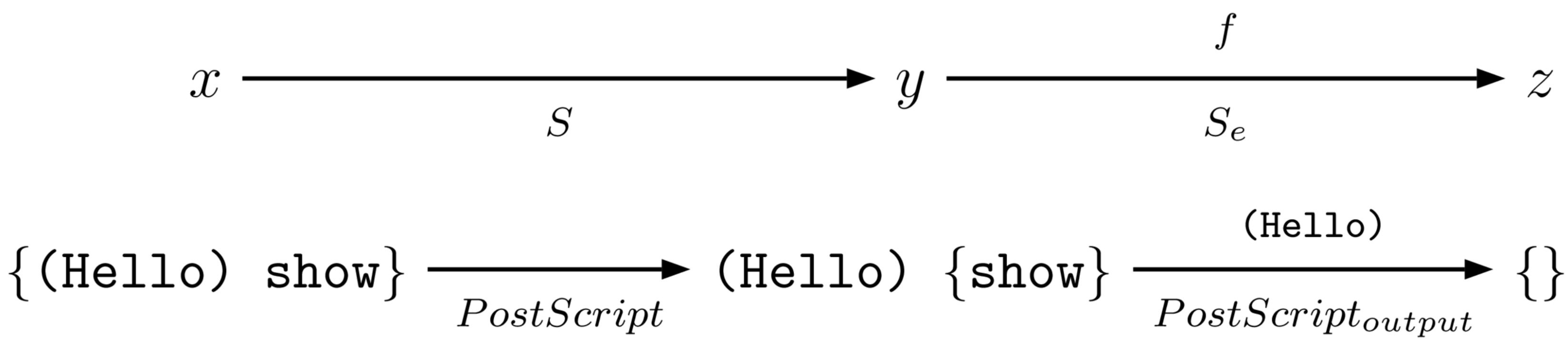
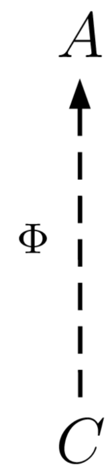


Figure conventions:

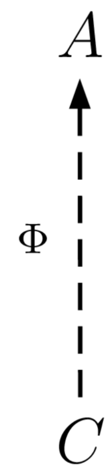
- Computation progresses left to right

- Effect label above, category (subset) below

(Abstract) Interpretation



(Concrete) Implementation



Concrete Implementation vs Abstract Interpretation

Dynamic (Runtime) vs Static (Compile-time)

Operational Semantics vs Denotational Semantics

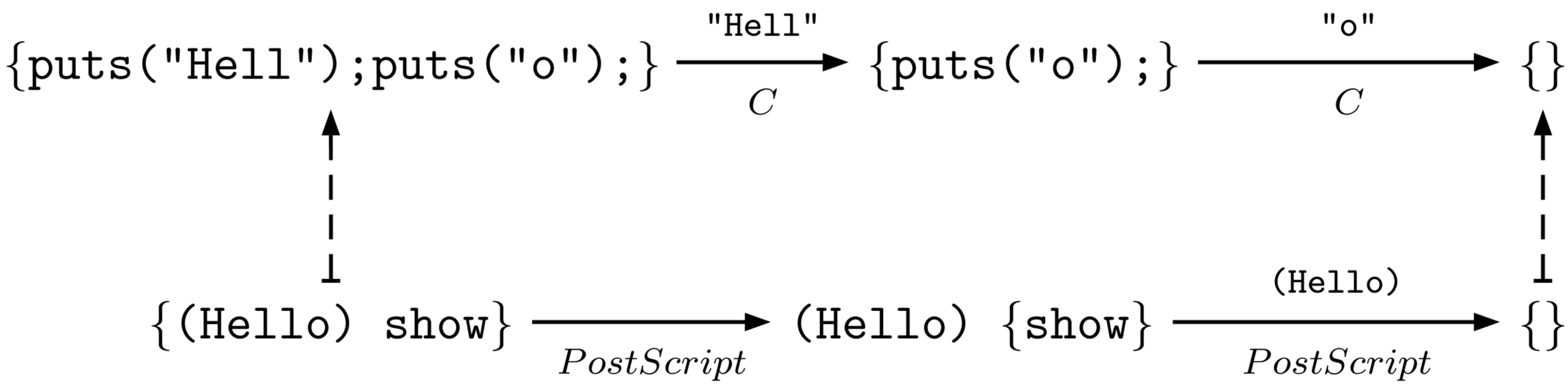
Downward (concrete) vs Upward (abstract)

Co-functorial vs Functorial

Noisy vs lossy

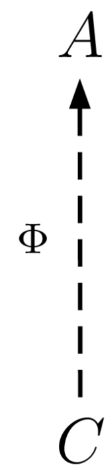
Non-deterministic vs deterministic

Partiality

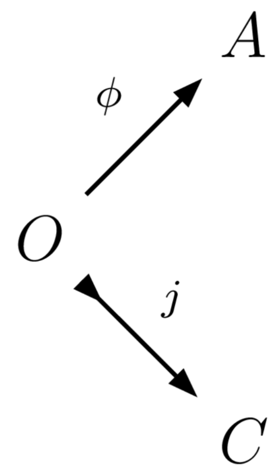


{puts("Hell");puts("o");}
"Hell"
C
{puts("o");}
"o"
C
{}
{(Hello) show}
PostScript
(Hello) {show}
(Hello)
PostScript
{}

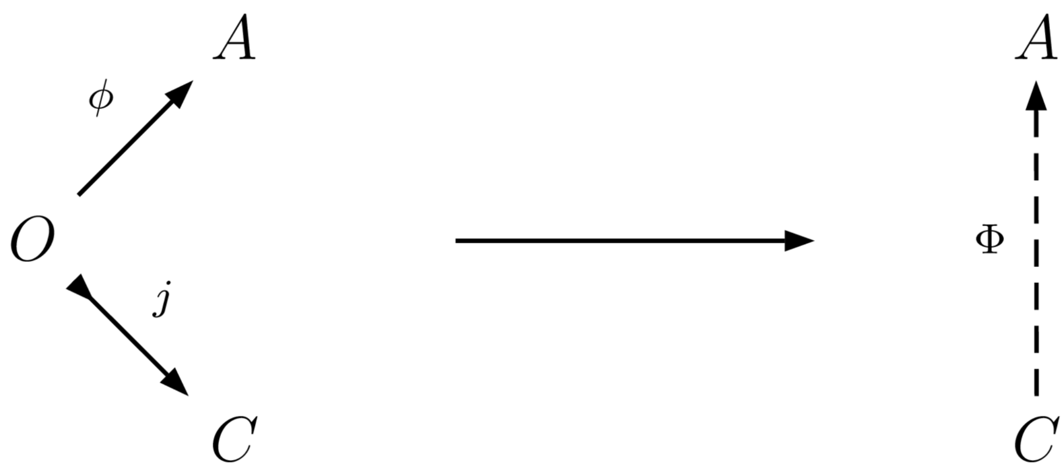
Partial Functions (1)



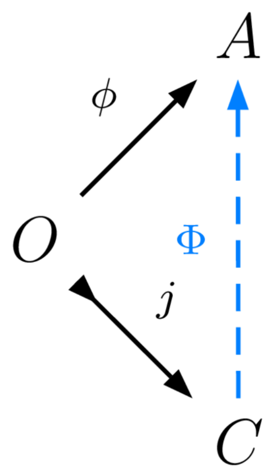
Partial Functions (2)



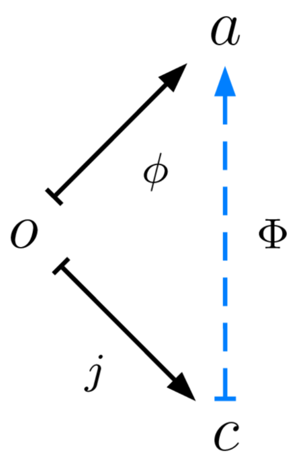
Partial Functions (3)



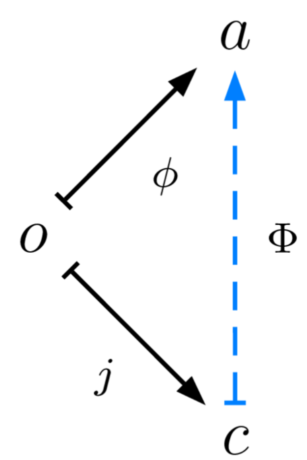
Deduction



Observable State



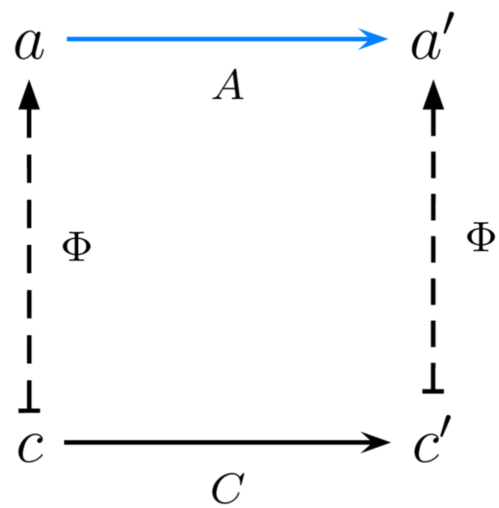
Observable State



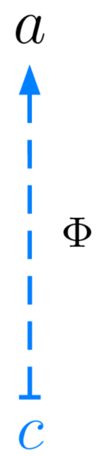
$$o = c$$

I.2 Properties of Implementations

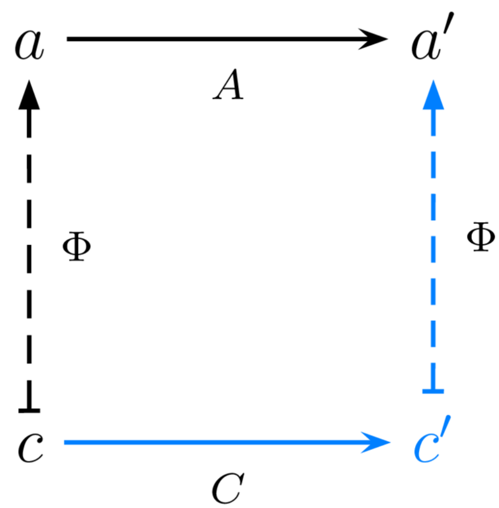
Soundness



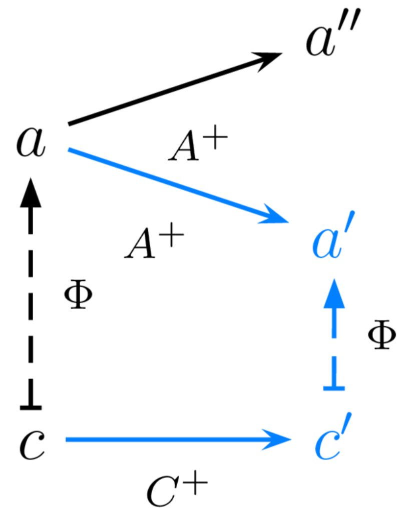
Totality



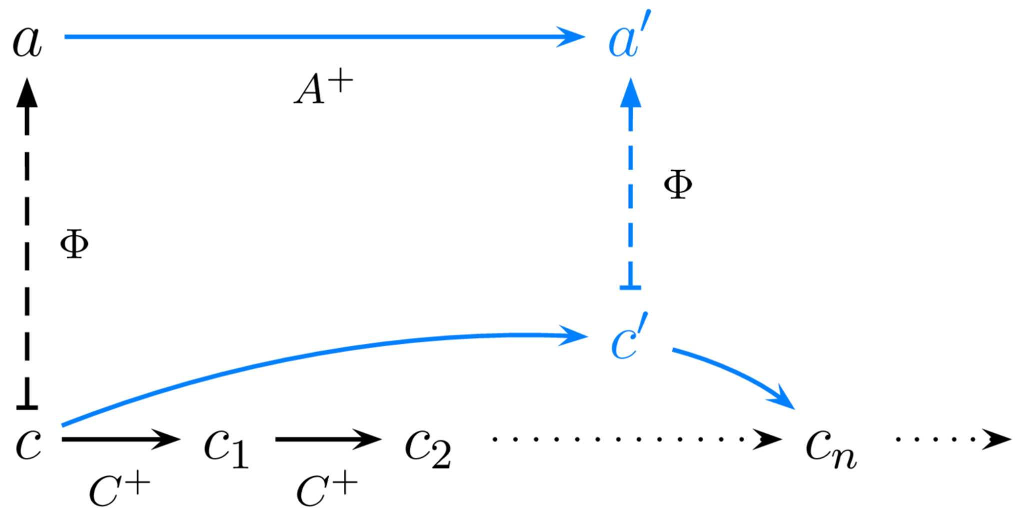
Completeness



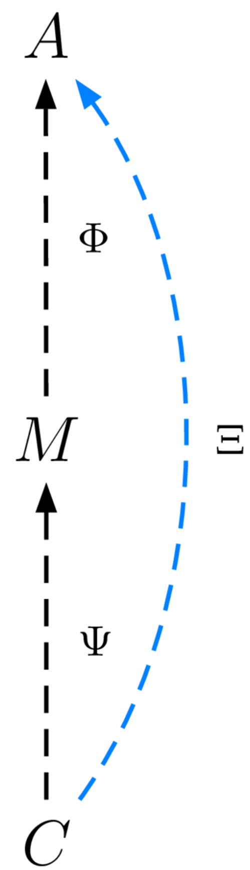
Advance Preservation



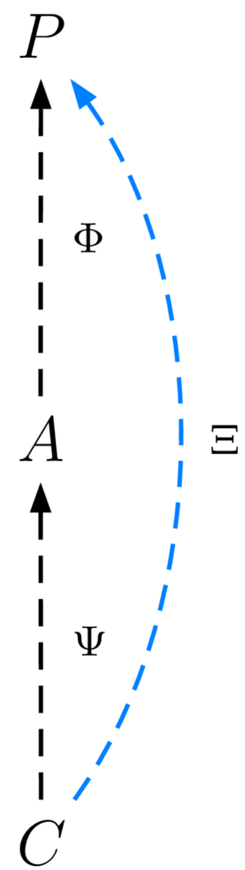
Liveness



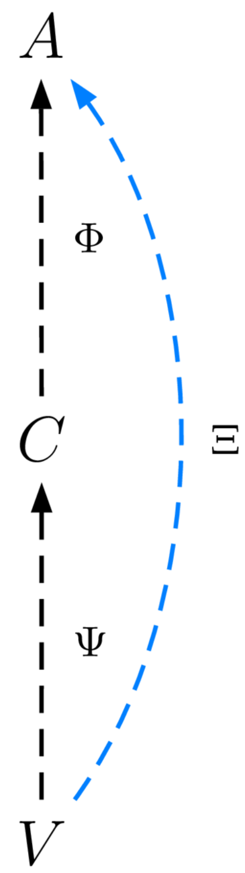
Composability



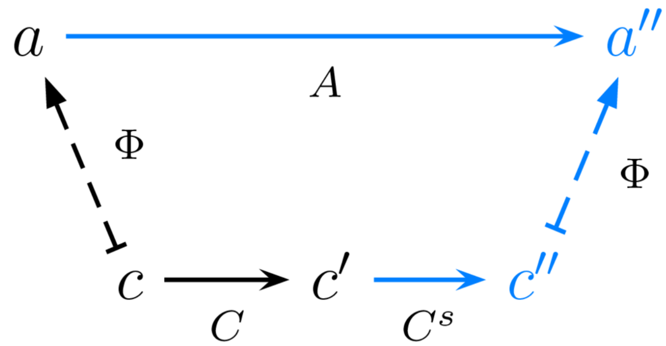
Composability



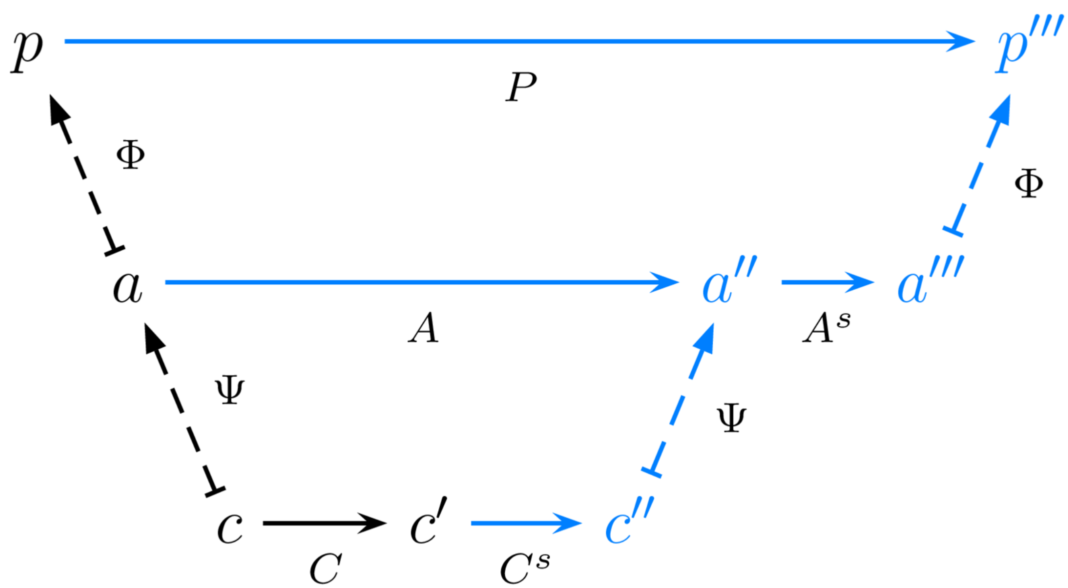
Composability



Observability (aka PCLSRing)

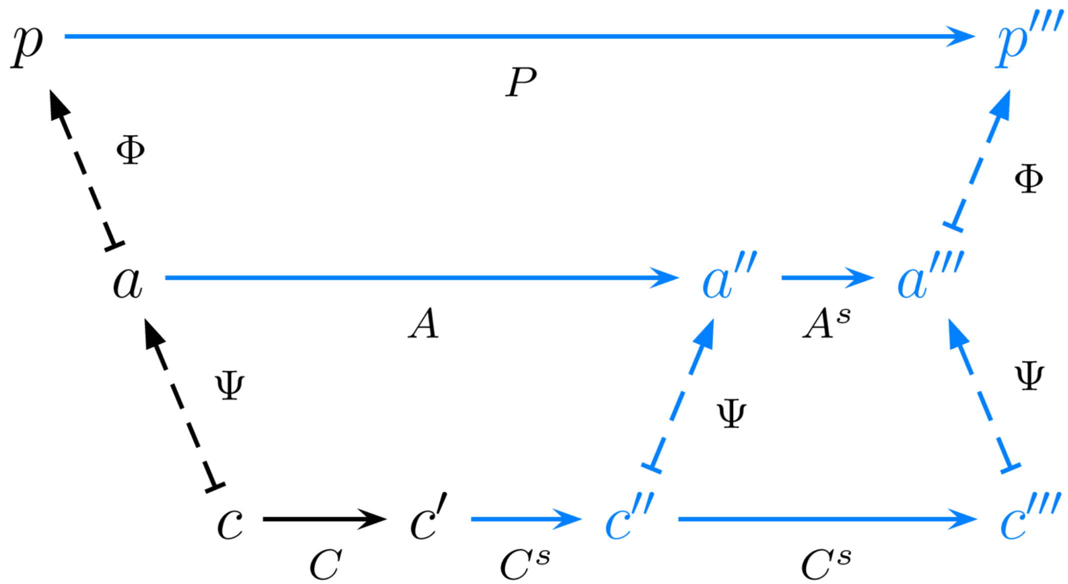


Observability (aka PCLSRing)



... not composable!

Observability + Completeness



Composable!

II. First-class Implementations

II.1 Protocol Extraction

Protocol: Categories (in Agda)

```
record Category … : Set … where …
  field
    Obj : Set …
    _⇒_ : Rel Obj …
    id : ∀ {A} → (A ⇒ A)
    _∘_ : ∀ {A B C} → (B ⇒ C) → (A ⇒ B) → (A ⇒ C)
  …
```

Showing fields with computational content

Many more fields for logical specification

Protocol: Categories (in Haskell)

```
class Cat s where
  type Arr s :: *
  dom :: (Arr s) → s
  cod :: (Arr s) → s
  idArr :: s → (Arr s)
  composeArr :: (Arr s) → (Arr s) ⇢ (Arr s)
```

Pure total functions: →

Effectful functions: ⇢ (partial, non-det...)

Protocol: Operational Semantics

```
class (Cat s) ⇒ OpSem s where
  run :: s ⇢ Arr s
  done :: s → Bool
  advance :: s ⇢ Arr s
  eval :: s ⇢ Arr s
```

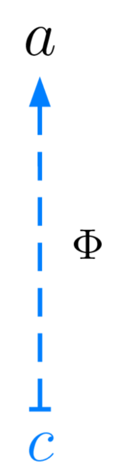
Protocol: Implementation

```
class Impl a c where
  interpret :: c ⇢ a
  interpretArr :: (Arr c) ⇢ (Arr a)
```

So far, a (partial) functor from `c` to `a`

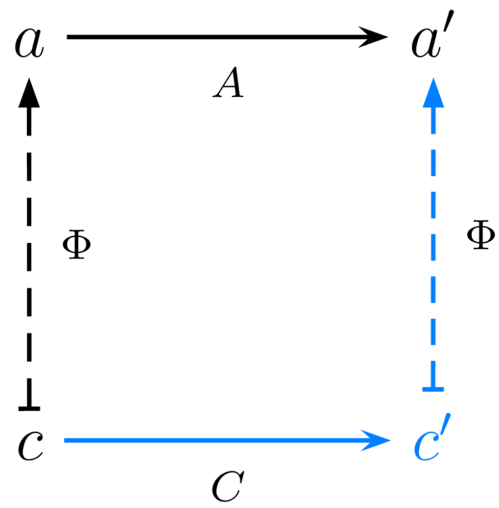
Arr = pirate sound = functorial map

Protocol: Totality



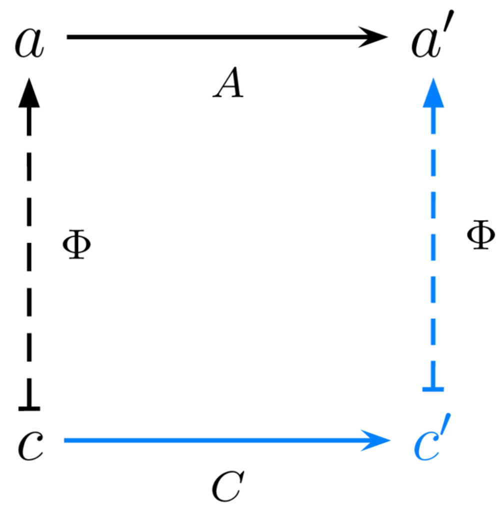

```
implement :: a ⇢ c
```

Protocol: Completeness




```
implementArr :: c → (Arr a) ⇢ (Arr c)
```

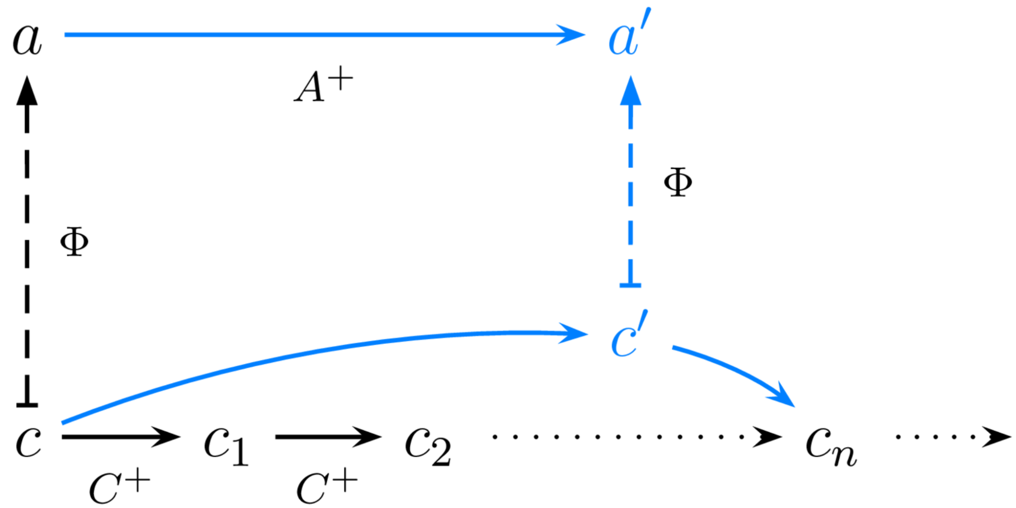
Protocol: Completeness (with Dependent Types)




```
implementArr :: c → (Arr a) ⇢ (Arr c)
```

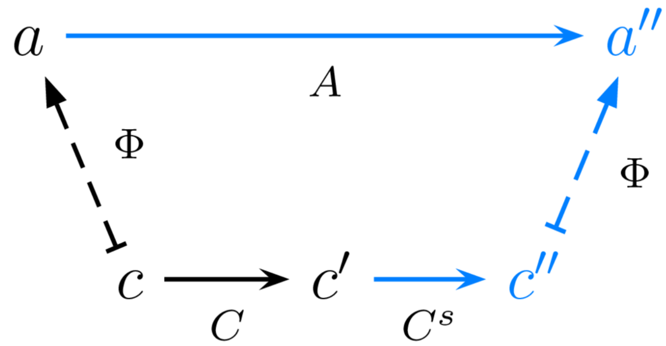
```
implement⇒ : ∀ (c : C.o) {a a' : A.o}
(f : C.⇒ a a') {Φ.o c a} → ∃(λ {c' : C.o} →
∃(λ (g : C.⇒ c c') → Φ.⇒ g f))
```

Protocol: Liveness




```
advanceInterpretation :: c ⇢ Arr c
```

Protocol: Observability (PCLSRing)




```
safePoint :: c ⇢ Arr c

safeArrow :: Arr c ⇢ Arr c
```

Reified vs Reflected Evaluation

Reified:

```
eval :: s ⇢ Arr s
```

Only effect is non-determinism

Reflected:

```
eval! :: s ⇢ s
```

Arbitrary side-effects

Runnable vs Observable Protocols

Reflection:

```
perform :: s ⇢ m
performArr :: (Arr s) → m ⇢ m
```

first-class semantics runnable as machine state

Reification:

```
simulate :: m ⇢ s
simulateArr :: m → (m ⇢ m) ⇢ Arr s
```

machine state observable as first-class semantics

Lifting Reflection and Reification Protocols

If you can `implement` `a` with `c`:

```
a.perform anod = anod & implement & c.perform

a.performArr aarr m =
  ((m & c.simulate & implementArr) aarr & c.performArr) m

a.simulate state = state & c.simulate & interpret

a.simulateArr m change =
  change & c.simulateArr m & safeArrow & interpretArr
```

Lifting Evaluation Protocols

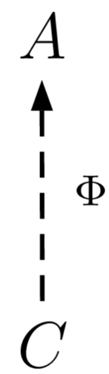
If the implementation is live, observable:

```
a.run anod =
  anod & implement & c.run & safeArrow & interpretArr

a.advance anod =
  anod & implement & advanceInterpretation & interpretArr
```

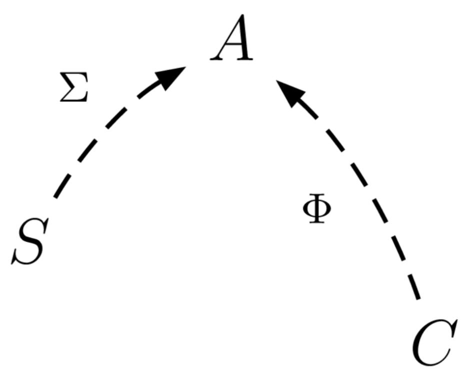
II.2 The Semantic Tower

Compilation (1)



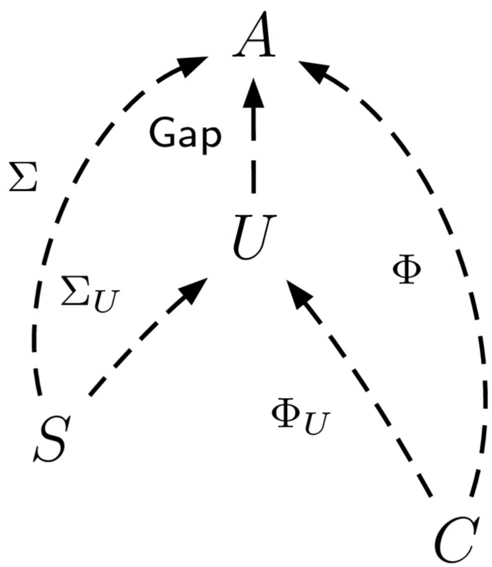

```
implement :: (Impl a c) ⇒ a ⇝ c
```

Compilation (2)



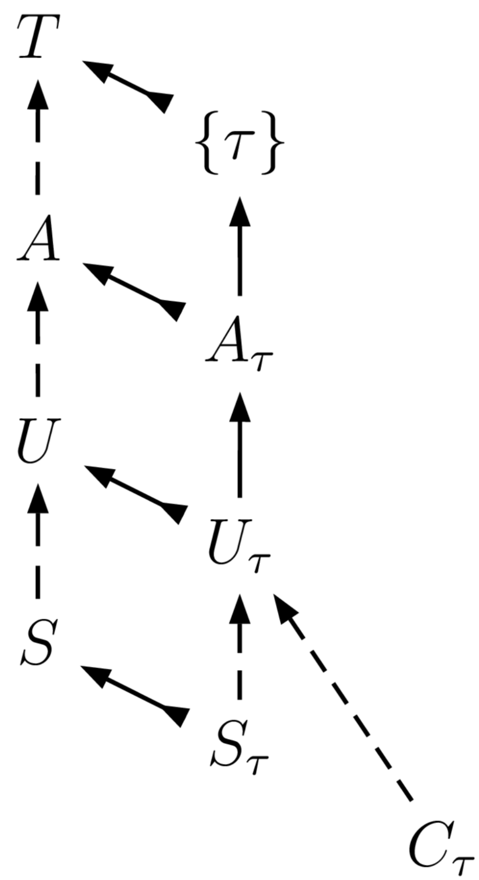

```
interpret :: (Impl a s) ⇒ s ⇢ a
implement :: (Impl a c) ⇒ a ⇢ c
```

Compilation (3)



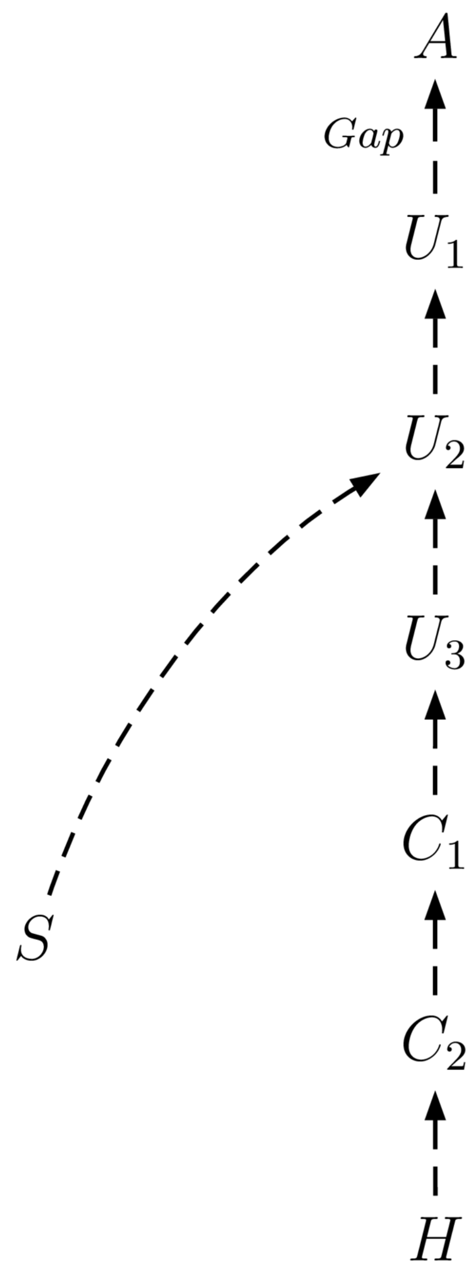

```
u :: OpSem -- specify up to what rewrites
interpret :: (Impl u s) ⇒ s ⇢ u
implement :: (Impl u c) ⇒ u ⇢ c
```

Static Type Systems

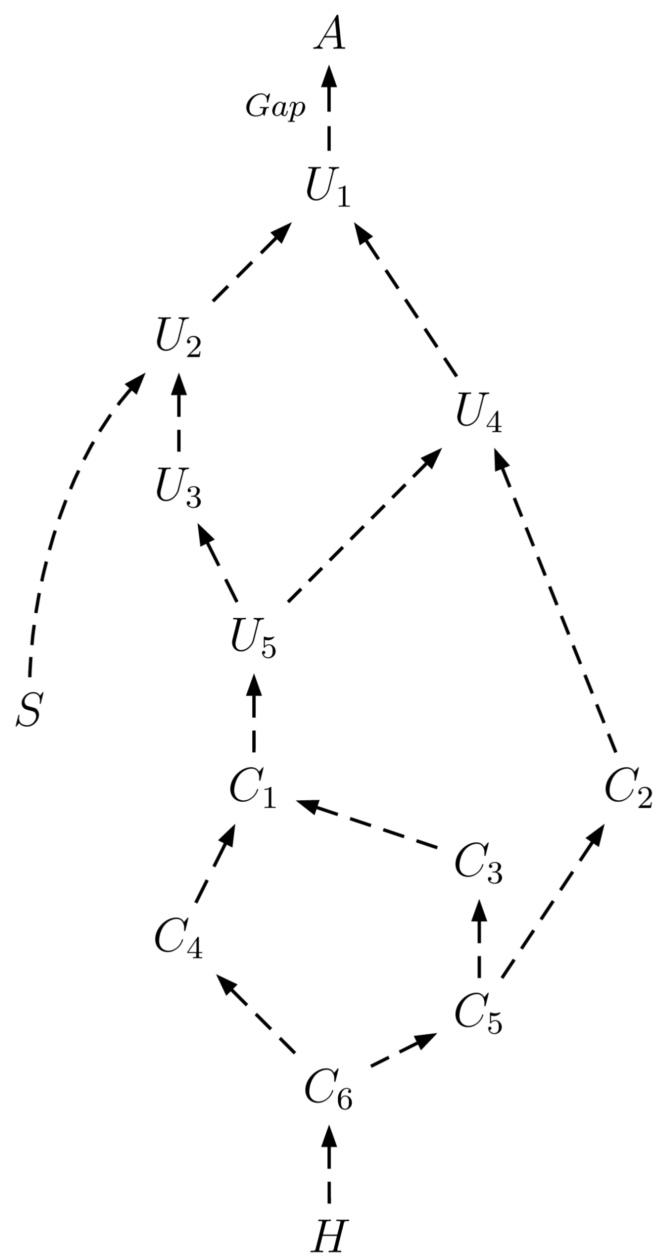


Subject reduction: *T* contains no exomorphisms

Semantic Tower



The Tower is not Linear



More reinterpretations...

Aspect-Oriented Programming

Erlang-style Fault Tolerance

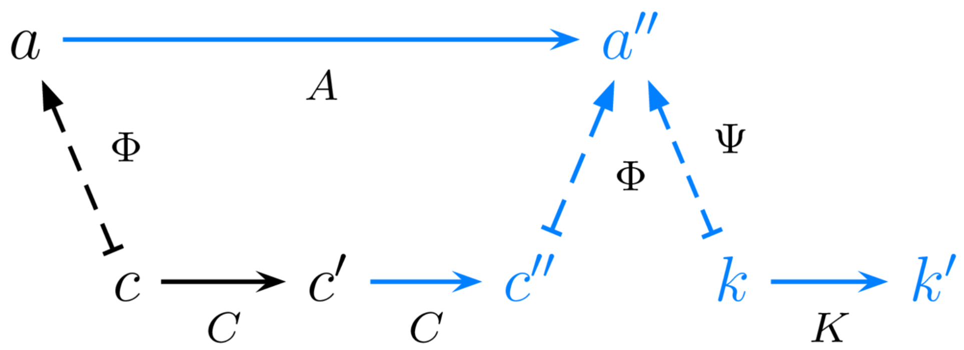
Refactoring

Developing

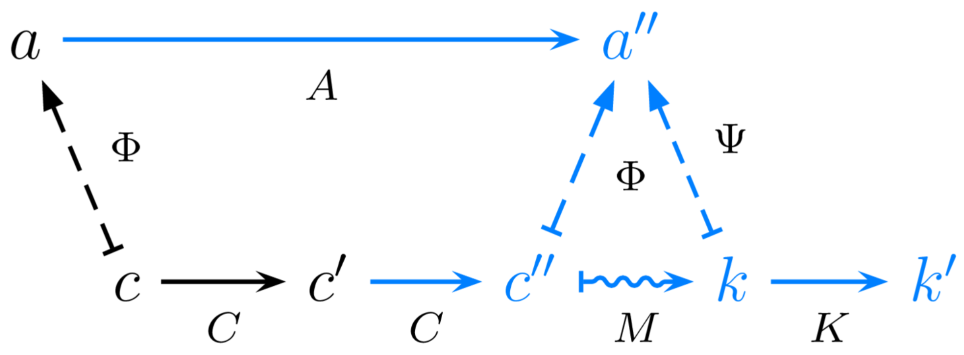
III. Principled Reflection

III.1 Migration

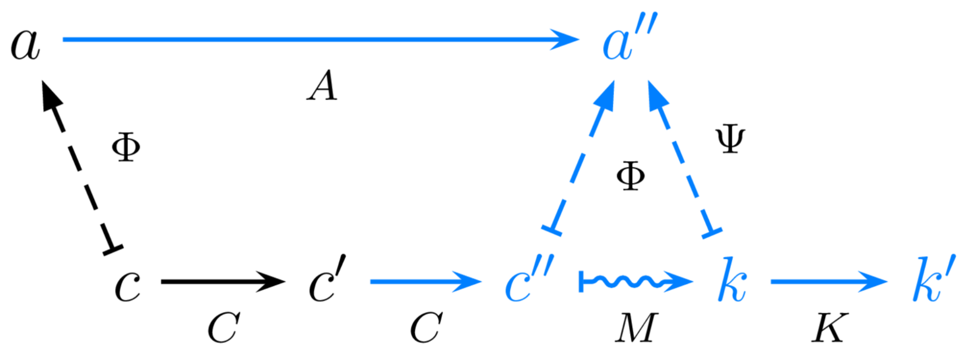
Migration



Migration (Optimized)

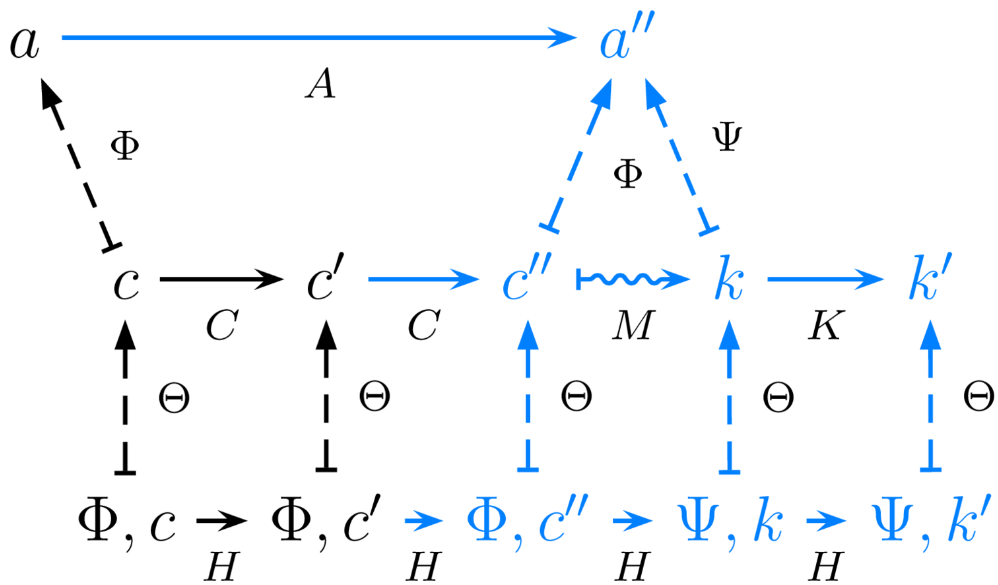


Migration (Optimized)

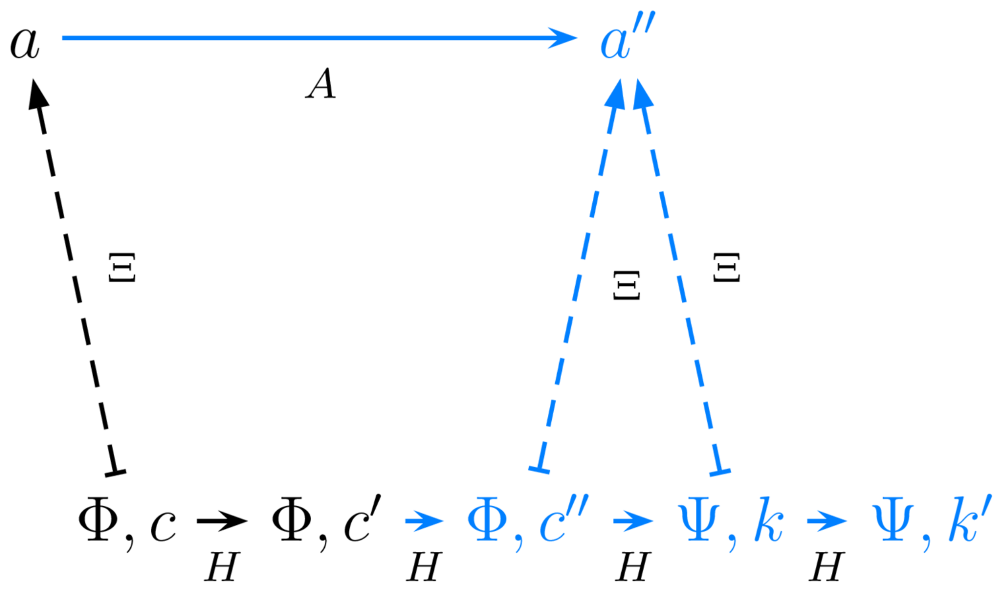


Does the second line break typing?

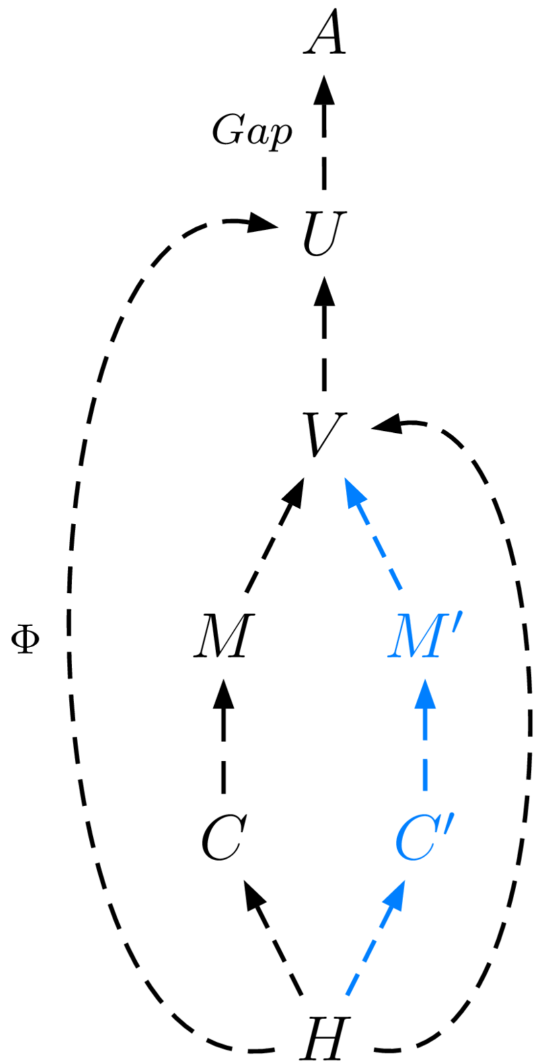
Migration (Implemented)



Migration (Factored out)



Migration Tower



When your hammer is Migration...

Process Migration

Garbage Collection

Zero Copy Routing

Dynamic Configuration

JIT Compilation

etc.

Fruitful change in Perspective

Correctness

Runtime Optimization

Retroactivity

Composability

Predictable Cost-Reduction

Requirement: Full Abstraction

Computations have a clear opaque bottom:

- It's perfectly clear what the bottom is

- The bottom is totally opaque

Indeed, what's below can change at runtime!

Alternatively, include what's "below" in the spec

Needed: explicit language or system support

Migration Control

Internal: automatic change in representation

External: parameters under user control

One man’s internal is another man’s external…

Need an Architecture for migration control

III.2 Natural Transformations of Implementations

Instrumentation

Tracing, Logging, Stepping, Profiling

Omniscient debugging, Comparative Debugging

Code and Data Coverage

Resource Accounting, Access Control

Parallelization, Optimistic Evaluation

Orthogonal persistence

Virtualization

Optimizations

Natural Transformation

Twist: *dual* of nat. transf. on *dual* of (partial) funct.

Automatic Instrumentation

Universal transformations

Composable transformations

Amenable to formal reasoning

Open problem, but promiseful approach

IV. Reflective Architecture

IV.1 Runtime Architecture

Runtime Architecture

Development Platform (Emacs, IDE, ...)

User Interface Shell

Operating System

Distributed and Virtualized Application Management

Every Program has a Semantic Tower

Semantics on top + Turtles all the way to the bottom

Top specified by User, bottom controlled by System

For the PLs your build, those you use

Static or dynamic control

Every Tower has its Controller

Runtime Meta-program, Shared (or not)

Virtualization: control effects, connect I/O

Reflective Tower of Meta-programs

New meta dimension: Puppeteers all the way back!

Implicit I/O

```
Input :: tag -> IO indata
Output :: tag -> outdata -> IO ()
```

Effects handled by the controller

Virtualization of effects at language level

Dynamically reconfigurable

IV.2 Architectural Benefits

Performance: Dynamic Global Optimization

When configuration changes, migrate

Optimize the current configuration

Minimize encoding, Zero copy

Skip unobserved computations

Simplicity: Separate program and metaprogram

Example: File selector, UI, etc.

Evolve, Distribute, Share, Configure separately

Separate Capabilities, Semantics

Robustness, Security: Smaller Attack Surface

Not Just a Library

Semantic separation vs inclusion

Bound at Runtime vs Fixed at Compile-/Load- time

Different scopes and capabilities

Different control flow

Different Social Architecture

New dimension of modularity

Deliver components, not applications

No more fixed bottom, fine-grained virtualization

Orthogonally address “Non-functional requirements”

Pay aspect specialists for components

Conclusion

Related Works and Opportunities

Formal Methods for proving program correctness

Open Implementation, AOP...

Many hacks for GC, Migration, Persistence...

Virtualization, distribution...

Common Theme

Programming in the Large, not in the Small

Software Architecture that Scales

Semantics matter

Dimensions of Modularity beyond the usual

The Take Home Points

Reason about Implementations: Category Theory!

Observability: Key neglected concept — safe points

Practical Protocol Extraction: First-Class Impl.

Explore the Semantic Tower — at runtime!

Principled Applications: Migration, etc.

Natural Transformations generalize Instrumentation

Runtime Reflection *and* Static Semantics

Price: Full Abstraction, Observability, Interpretation

Challenge

Put First-class Implementations in your platform

Factor your software into meta-levels

Develop Generic Tooling, Reflective Architecture

Enjoy simplification, robustness, security

The Meta-Story

My contribution is mostly not technical.

It is more ambitious:

The Meta-Story

My contribution is mostly not technical.

It is more ambitious:

A change of point of view about computing

The Meta-Story

My contribution is mostly not technical.

It is more ambitious:

A change of point of view about computing

The essence of FP: relating abstract and concrete

My blog: *Houyhnhnm Computing*

`https://ngnghm.github.io/`

Ancient: *TUNES Project*

`https://tunes.org/`